OSOS DOCUMENTATION

Integrate & Extend

A section of the OSOS Documentation and Help Center

English edition

Platform release 1.10
Published 23 November 2026

Guides · How-to

Contents

Integrate & extend — start here 5

Guides

Packaging & distribution

Packaging & distribution 6
The Package Format 22
Package Resource Schemas 30
The dependency graph 38
Generating a package 44
The import API 49
Schema versioning 59
Module JSON transfer 63
Packages, Marketplace & the In-App Store 66
Functions: run your own code 87
Files: storage rules 95
Criteria and field-path dialects 99
Calling the HTTP API 104
Authentication & OAuth 116
Real-time: live events and calls 122
Webhooks & integrations 126
Grafana Dashboards 130
Data Model & Events 136
Run the platform from source (local development) 152

OSOS as a Backend

OSOS as a Backend 164
Backend Quickstart 168
Choosing a Credential 173
Token Lifecycle 176
Tenancy and Identity 179
Reading Data 183
Writing Data 187
Working with Files 193
Subscribing to Events 198
Long-Running Work 201
Errors and Retries 206
Generating a client 231
Environments and promotion 235
The security boundary 238
Workflow model and API reference 241
Data views API reference 265
Permissions API 283

Endpoint reference

Endpoint reference 293

Comments Service

Comment — comments-service 296
Reaction — comments-service 299

Data Service

Access — data-service 301
BackupAdmin — data-service 308

CounterWidget — data-service 311
Counters — data-service 313
Dashboard — data-service 314
DataCollections — data-service 316
DataEntities — data-service 324
DataEntityAggregate — data-service 331
DataEntityBlocks — data-service 333
DataEntityCategories — data-service 335
DataEntityFields — data-service 338
DataEntries — data-service 341
DataEntryTemplateExport — data-service 361
DataModule — data-service 362
DataQuery — data-service 365
DataRelation — data-service 369
DataViews — data-service 377
Delegations — data-service 391
EntityEntriesExport — data-service 392
Expression — data-service 393
GenericView — data-service 394
Groups — data-service 398
Icons — data-service 399
ImportExport — data-service 400
Job — data-service 402
Locks — data-service 403
OrgConfig — data-service 405
QuickFilterGroup — data-service 406
ReportEntry — data-service 408
Request — data-service 412
SensitiveFieldGrants — data-service 413
SideBar — data-service 416
Spaces — data-service 417
StatisticQuery — data-service 420
Stores — data-service 424
SystemButton — data-service 426
SystemLink — data-service 430
SystemObject — data-service 431
SystemSettings — data-service 432
TagGroup — data-service 433
Tags — data-service 435
Templates — data-service 436
UiSettings — data-service 437
User — data-service 441
UserPreferences — data-service 444
Version — data-service 445
ViewApiDocs — data-service 446
ViewEntriesExport — data-service 447
Workflow — data-service 448

Functions Service

Functions — functions-service 449

Notification Api

Notification — notification-api 450
NotificationTemplate — notification-api 451
PushNotificationsTokens — notification-api 452

Reaction Service
Announcement — reaction-service 454
DataReaction — reaction-service 457
ReactionLog — reaction-service 461
SystemReaction — reaction-service 463
WhatsAppTemplates — reaction-service 465

Timeline Service
Timeline — timeline-service 466

Workflow Service
Workflow — workflow-service 469
Requests & services (API and internals) 473
Account security and step-up (internals) 480
Entity model and API reference 484
Pages, sidebar and share links API reference 510
Reaction model and API reference 521
Buttons API reference 557
Portal sign-in and spaces (internals) 566
Query model and API reference 571
Relations, references and mapped fields: model and API 579
Users API & identity accounts 589
Aggregate and grouping reference 595
Export templates API reference 603
Groups, org units & membership 612
Portal shell: home page, sidebar and share links (API) 620
Virtual entities: model, API and internals 624
Delegation API & enforcement 638
History and timeline API reference 644
Notifications and delivery (API and internals) 655
Report model and API reference 660
Auto-increment sequences and counter widgets: internals and API 668
Preview mode internals 675
Data stores: model, routing and API 681
Summary and dashboard API reference 688
Workflow participation reference 701
Comments and mentions reference 707
Sensitive and encrypted fields: model, API and enforcement 712
Module model and API reference 740
Field configuration reference 750
Internal field types 766

MCP Server for AI Agents
MCP Server for AI Agents 769
MCP Quickstart 771
MCP Tools Reference 773
Records and Lists: Behaviour Reference 776

How-to
Use an API token 782
Integration FAQ 785

Integrate & extend — start here

This section is for **integrators and professional developers**: people who connect other systems to OSOS or extend it with code. Here the technical detail is the point — endpoints, payloads, credentials, error codes.

Building an application in the designer instead? That is Build apps [Build Apps › Build apps — start here], and it needs no code.

Your first hour

1. **Get a credential.** Decide which kind your integration needs in Choosing a credential, then create it (an application user and its API token are created by a tenant administrator — see API tokens).
2. **Make your first call.** Follow the Backend quickstart: read records, then create one.
3. **Learn the contract.** Reading data, Writing data and Errors & retries cover what every integration needs.

Find what you need

I want to…	Start with
Call the platform from another system	OSOS as a backend · HTTP API
Authenticate users or services	Authentication & OAuth · Token lifecycle
React to changes in another system	Webhooks & integrations · Subscribing to events · Real-time
Run my own code inside a process	Functions (FaaS)
Upload and download files	Files · File storage & attachments
Ship applications between environments	Environments & promotion · Packaging & distribution
Let an AI agent work with the platform	MCP server for AI agents
Chart platform data in Grafana	Grafana datasource
Look up an endpoint	Endpoint reference
Run the platform from source	Local development

Technical companions to the builder and admin guides

Pages for builders and administrators describe screens and outcomes; the API and engine behaviour behind them lives here: Requests & services · Permissions API · Workflow model & API · Data model & events · Criteria & field-path dialects.

Packaging & distribution

This page is for **integrators and engineers** who drive packaging through the API, build tooling around packages, or need to know exactly what the export and import engine does. It explains how the engine fits together, the rules it enforces, and how it resolves dependencies, and it points to the reference pages for the archive format, the import wire contract and the marketplace.

Doing this in the UI?

Exporting, importing, versioning, publishing and installing from the **Packages** screens is covered for builders in Export, import and install apps [Build Apps].

Packaging moves a set of built platform resources — modules, entities, views, queries, relations, reactions, reports, pages (generic views), system buttons, collections, statistic queries, counter widgets, functions, announcements — from one environment (or space) to another, along with everything they depend on. A package is a single **ZIP** that bundles the selected resources with the entry-level data, shared data-store rules and template/binary files they need. It is the mechanism behind the usual **development** → **staging** → **production** promotion path.

Availability

Since 1.8. The ZIP-based export/import engine shipped in the 1.8 milestone (its first commits are on the 1.7 pre-release tags; mapping profiles on the 1.8 pre-release tags). **Since 1.10**, named, versioned packages, the review inbox, the marketplace connection, installations and the in-app store are built on top of it — see Packages, Marketplace & the In-App Store. Individual features are dated in their sections.

Where to find what

You need	Page
The engine, its access rules and how it resolves dependencies	This page
Exactly which files a package ZIP contains, every manifest field, the import-time limits and validator	The package format
The JSON shape inside each resource file	Package resource schemas
The edges in `dependencies.json` and how they drive remapping and ordering	The dependency graph
Every import-session endpoint, the state machine, the session document and concurrency rules	The import API
Which `schemaVersion` values exist and which environments accept them	Schema versioning
Producing or transforming a package programmatically	Generating a package
Packages, versions, publish requests, the marketplace, installs and updates	Packages, Marketplace & the In-App Store
The older single-file module/entity JSON transfer	Module JSON transfer

How the engine fits together

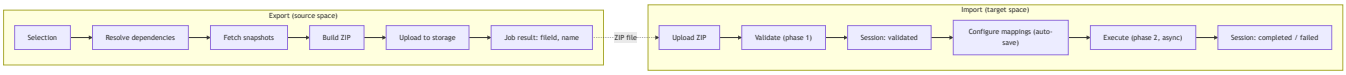
Packaging is split across two services, both LoopBack 4 and both Auth2-guarded:

Service	Base path	Role
`packaging-service`	`/api/packaging`	Export pipeline, import validation/execution, sessions, overrides, mapping profiles
`dependency-service`	`/api/dependency`	Directed dependency graph; supplies edges, dependents-tree, topological order

A package is produced by an **export job** and consumed by an **import session**. Both long-running operations run asynchronously on **BullMQ** (Redis-backed) workers: the HTTP endpoints enqueue work and return a job or session id that you poll for status.

Every packaging controller is annotated `@authenticate` with the Auth2 strategy, and every endpoint requires an `x-space-id` request header — a missing header answers `400 x-space-id header is required`.

Two-phase lifecycle



Exporting a package

An export takes a selection of root resources, resolves what they depend on, and writes the ZIP. In the web app this is the **Export resources** page under **Packages › Build** (see the builder guide [Build Apps › Export, import and install apps]); through the API it is the endpoints below.

> **Tip:** Always run the export **analysis** first and resolve external-dependency warnings before shipping a package. A resource whose hard dependency is missing from the package will fail to import cleanly.

What the package contains

Every package carries a **manifest** describing the source (schema version, source environment, source space, source version) and a **topologically sorted** set of resources — parents before children. The exporter follows the **hard** dependency edges of your selection so the package is self-contained; **soft** (optional) references are recorded but not followed. Entry-level references, shared data-store rules, and the template/widget/page binary files your resources need are collected alongside. The archive itself is specified in The package format.

Export API

All export endpoints live under `/api/packaging/v1/export`. They share the `ExportRequest` body and the `x-space-id` guard.

Method	Path	Purpose	Returns
`POST`	`/v1/export/preview`	Accurate resource counts after the real fetch+filter pipeline	`ExportPreviewResponse`
`POST`	`/v1/export/analysis`	Merged preview + external-dependency analysis in one round-trip	`AnalyzeExportResponse`
`POST`	`/v1/export`	Enqueue an export job and, `Since 1.10`, write its export record	`{ jobId, exportId }`
`GET`	`/v1/export/{jobId}/status`	Poll job state + result	`ExportStatusResponse`

`GET /status` enforces that the caller's space owns the job (`403 Access denied to this export job` otherwise) and, `Since 1.10`, that the caller is the user who started it or a system administrator — anyone else gets `404`, as if the job did not exist. On success the status result carries the storage `fileId` and generated `name`; the downloadable ZIP lives in **storage-service** as a temporary file. The filename is the export's English name, else its Arabic name, sanitised to `[a-zA-Z0-9_-]`; when that leaves nothing (an Arabic-only name, for instance) it falls back to `<NAMESPACE>-package-<ISO-timestamp>.zip`.

The preview returns counts by type and entity; the analysis adds the **external-dependency analysis** — everything the selection references *outside* itself, grouped by module, with unresolved external dependencies surfaced as warnings.

ExportRequest fields

Field	Type	Notes
`resources`	`ResourceRef[]`	**Required** (≥1, else `400 At least one resource is required`). Each ref is `{type, id, withChildren?}`.
`excludes`	`ResourceRef[]`	Specific `(type,id)` pairs to drop from the resolved set.
`excludeTypes`	`ResourceType[]`	Whole resource types to drop from the resolved set.
`description`	`string`	Free text, stored in `manifest.description`.
`name`	`ILocalizedValue` (`{en?, ar?}`)	`Since 1.10` **required by** `POST /v1/export`: `400 EXPORT_NAME_REQUIRED` when neither language has text (preview and analysis ignore it). It names the export's origin and the ZIP file (see above). On 1.8 it was an optional plain string.
`maxClosureRounds`	`number`	Depth of transitive **entry-reference** closure. Default `5`; `0` disables (direct edges only). Higher = deeper nested-entry capture at the cost of more API calls and a larger package.

`ResourceRef.withChildren` triggers "smart resolution": for `DataModule` and `DataEntity` the service expands the `MEMBERSHIP` dependents tree from dependency-service and pulls in the children (user-scoped and external children excluded). In the web app, ticking a module or entity node sends it with `withChildren`.

A plain export carries no README `Since 1.10`. `POST /v1/export` has no `readme` field. The README belongs to a package and its versions (edited on the package page and written into each version's ZIP), and the import side still reads and keeps a README it finds in a file. The request-body limit of `POST /v1/export` is **30 MiB** (`413 PAYLOAD_TOO_LARGE` beyond that).

Export records `Since 1.10`

Every `POST /v1/export` writes an **export record** before it queues the job, so the Build page's **Exports** tab can list past exports, reopen one, and start the same export again (the export page opens with `?from=<exportId>`). The response is `{jobId, exportId}`, where the job id is `export-<exportId>`; poll `GET /v1/export/{jobId}/status` as usual.

Method	Path	Purpose
`GET`	`/v1/exports`	The space's export records, newest first, with `total`. Paged by `skip` (default `0`) and `limit` (default `50`, at most `100`).
`GET`	`/v1/exports/{id}`	One record **with its selection** (`resources`, `excludes`, `excludeTypes`), to start the same export again. Another space's id is `404 EXPORT_NOT_FOUND`.
`DELETE`	`/v1/exports/{id}`	Deletes the **record** only.

All three are open to anyone who could start an export (a Developer somewhere — see Export access). A record carries `id`, `name`, `description?`, `resourceCount` (the selection as sent), `packagedCount?` (what the package held once dependencies were resolved), `status` (`running`, `completed` or `failed`), `createdBy`, `createdAt`, `completedAt?`, `fileId?`, `size?` and `failedReason?`. A `running` record whose job has already ended is brought up to date when it is read; one whose job can no longer be found after 60 seconds becomes `failed` with `EXPORT_JOB_LOST`.

Deleting a record does not delete the ZIP. The file stays in storage, and a job that is still running still finishes. Conversely, a plain export's ZIP is a temporary file, so a record's `fileId` can point at a file that has already expired.

Export access `Since 1.10`

Exporting is a **developer** action, not an administrator one. `POST /v1/export`, `/v1/export/preview` and `/v1/export/analysis` are authorised against the roots in `resources`:

- a **system grant of Developer or higher** may export any roots;
- a system grant of **None** refuses, whatever the resource-level grants say;
- otherwise **every** root needs Developer or higher on the module or entity that owns it — a module by its module grant; an entity by its entity grant, else its module's; a collection by its entity (only when the collection and the entity are in the same module); a page (generic view) by the module it belongs to. Any other root type, or an empty list, needs the system grant.

The export page and its resource tree mirror this: they open for module developers and offer only the roots the user can develop.

The export itself refuses resources that belong to an **installed package** with `403 EXPORT_INSTALLED_RESOURCE` (`details.resourceIds`, `packageId`): an installed template is the publisher's to ship, not yours. See installed resources.

What gets exported (and what doesn't)

`EXPORTABLE_TYPES` — only these types are written to `resources/` in the ZIP:

`DataEntity`, `DataView`, `DataQuery`, `DataRelation`, `DataCollection`, `DataReaction`, `GenericView`, `DataModule`, `SystemButton`, `StatisticQuery`, `Report`, `Function`, `CounterWidget`, `Announcement`, `DataEntityAggregate` (`Since 1.10`), `QuickFilterGroup` and `TagGroup` (both `Since 1.10`).

The two group types travel as entity children. On import their `id` and `dataEntity` are never written (the entity comes from the target path), `normalizedName` is the identity used to match an existing group, and an update changes only `name` and `queries` — a tag group's `color` cannot change on update either.

`isNonExportableResource` filters out, even when selected:

- **Archived** entities (`state === Archived`) — and their whole nested subtree cascades out.
- **User-scoped** resources.
- **Implicit** `DataEntity` (unless `implicitOptions.allowEdit`) and any **implicit** `DataReaction`.

Core identity entities — `USER`, `GROUP`, `ORGUNIT`, `WORKING_HOURS` (`CORE_ENTITY_IDS`) — are never written as resources; they exist independently in every environment and are handled as references (see Value references).

Export pipeline internals

`PackageExportService.executeExport` runs, in order:

1. **Resolve selections** — expand `withChildren`, apply `excludes` / `excludeTypes`, dedupe by `(type,id)`.

2. **Fetch + filter** — batch-fetch full JSON snapshots per type from data-service (projection `value,dataEntity,state` for entries), key each by a unique `normalizedName` (collision-suffixed with the id), then apply `isNonExportableResource` and the archived-subtree cascade.
3. **Clear entity-collection items** — non-ADT `DataCollection` snapshots are exported with `items: []` (they reference `DataEntry` ids not in the package); ADT collections keep their embedded scalar items.
4. `Since dev` **Strip function secrets** from every packaged `Function` — see below.
5. **Collect cross-cutting payloads** — entry references (value-level refs to core/entry UUIDs, with transitive closure), shared stores (`storeRules` with `type === 2`), and template/binary file references.
6. **Strip** server-managed fields (`SERVER_MANAGED_KEYS`, e.g. `creationInfo` / `updateInfo`) and `executionPlan` from views — import re-derives them.
7. **Build manifest + index**, topologically sort resources (Kahn's algorithm over hard edges), **stream** the ZIP via `archiver` (zlib level 6), and **upload** to storage-service. Streaming keeps worker memory bounded (~16 MB highwater) so multi-hundred-MB exports do not run out of memory.

Function secrets stay in their environment `Since dev`

On the development line a function can hold encrypted **secrets** and has its own **identity** (an application user); see Function secrets and per-function identity. Packaging never moves a secret's value or a function's identity between environments, because the ciphertext is bound to the source environment's keys.

Step	What happens to a `Function`
Export	Each secret row keeps its `key` (and any other field) but its `value` becomes `null`; the system-managed `OSOS_API_TOKEN` row and the `identity` reference are removed. This runs before the export collects entry references, so the identity cannot pull the function's user record into the package. A function that is only *referenced* by an exported resource is stripped the same way.
Content hash	The hash covers the secret **names, in order**, and nothing the target sets for itself, so an installed function still hashes equal to its package however its secrets are filled in later.
Install (create)	The packaged value is stripped again, so a package exported before this rule existed — which may still carry the source's ciphertext, token row and identity — cannot plant them in the target.
Update of an installed function	The target's own state is carried over: its identity, its `OSOS_API_TOKEN` row, and the stored value of every secret whose key the package still declares. Secrets the package no longer declares are dropped; newly declared names arrive empty. The update always reads the target first, and fails rather than writing blind if that read fails.

Nothing in packaging requires a secret to be set: a freshly installed function holds the declared names with empty values until an administrator of the target fills them in.

The package (ZIP) format

The archive layout, every manifest field, `index.json`, `dependencies.json`, the mapped-field sidecar, the published-version files, the import-time safety limits and the strict validator are specified in The package format. Which `schemaVersion` values exist, which environments accept them, and how validation treats `sourceVersion` and per-resource `specificationVersion`, are in Schema versioning. Format versioning is independent of the platform release, and older packages remain importable by design.

Importing a package

Importing is a session-based flow in three stages. Each import creates an **import session** that tracks progress and can be retried or cancelled. The wire contract for every stage — bodies, responses, status gates and the concurrency token

— is The import API; this section explains what the engine does at each stage. In the web app the flow is **Packages › Installed › Install from file** (see the builder guide [Build Apps › Export, import and install apps]).

A ZIP whose manifest carries a `package` block is a published package version and is imported as an **install** — see Installing.

Stage 1 — Validate

`POST /v1/import/validate` takes `{fileId, fileName?, importStrategy?}`, downloads the ZIP from storage-service, and runs `ValidationService.validate`, which:

1. Extracts and validates the ZIP and checks `schemaVersion`.
2. Classifies each exportable resource against the target space, resolving the correct **scope** (entity-scoped, module-scoped, or global — see `resolveResourceScope`) and matching by `normalizedName` (batched per type, with entity/module-filtered re-queries and an id-based fallback). Result per resource: `NEW` or `EXISTING` (`ValidationStatus`).
3. Generates default resource mappings: `NEW → CREATE`; `EXISTING → SKIP` (or `OVERWRITE` when the strategy is `OVERWRITE_ALL`).
4. Enriches with dependency metadata (`dependsOn`, `requiredBy`, module ownership, `cascadeSkipCount`), detects **unmapped dependencies** (hard external refs not in the package, and `Since dev` page-widget references to something outside it), and detects **value references**.
5. Persists an `ImportSession` (status `validated`, or `failed`/non-proceedable when there are blocking errors) and returns `{sessionId, validationReport, updateIdentifier}`.

The `ValidationReport` (`resources`, `errors`, `warnings`, `canProceed`, `unmappedDependencies?`, `availableTargets?`, and `Since 1.10` `mappedFields?`) lets you review everything before any write. Validation is read-only: nothing is written to the target yet. Identity collisions and unmapped dependencies are surfaced here for your attention.

After a successful validation the web app reloads the new session to show the server-resolved value references. If that reload fails, it still opens the review step with the report it already holds, and says that the value references could not be loaded. `Since 1.8`

Stage 2 — Configure

Configuration decides how each resource lands in the target. It **auto-saves** with optimistic locking, so two people editing the same session cannot silently overwrite each other. `PATCH /…/config` is deliberately relaxed — every section is optional and hard constraints are deferred to execute. It merges resource mappings, value-reference choices and unmapped-dependency targets, prunes stale value references, re-runs the `AUTO_CREATE` nested-reference guard, and returns a `dependencyImpact` / `mappingSummary`. `Since 1.10` the same save carries `mappedFieldDecisions` and answers with the recomputed `mappedFields` and `canProceed` (see Mapped fields).

Per-resource action (`MappingAction`). The wire values are `CREATE`, `OVERWRITE` and `SKIP`; the web app presents them as four outcomes:

Outcome	On the wire	Meaning
Create	`CREATE`	Create the resource new in the target, with a new id.
Map (reuse)	`SKIP` with a `targetId` on an `EXISTING` resource	Leave the existing target resource untouched; the id mapping is still registered, so references from other imported resources resolve to it.
Overwrite	`OVERWRITE`	Replace the existing target resource with the packaged version.

Outcome	On the wire	Meaning
Skip	`SKIP`	Do not import this resource. For a `NEW` resource, its hard dependents are cascade-skipped.

You can also:

- Apply a **bulk strategy** across all mappings via `PATCH /…/strategy` — **Manual**, **Merge new only** (`MERGE_NEW_ONLY`) or **Overwrite all** (`OVERWRITE_ALL`) — preserving user-chosen module remaps.
- **Accept exact-match suggestions** (`PATCH /…/suggestions/accept-exact`) to bulk-convert every unactioned exact-confidence principal suggestion into a `REMAP` to its suggested target.
- **Remap principals** — map users, groups and org units from source to target. Mapping profiles remember these between a given source and target so the next import lines them up automatically.
- **Decide what each mapped field copies from** when the import cannot work it out (see Mapped fields).
- Optionally **hand-edit a resource's payload** (a payload override) before import.

Stage 3 — Execute

`POST /…/execute` first checks that every unmapped dependency, mapped field and value reference the run needs is resolved, and `Since 1.10` that the tenant has room for every entry the run will create; the full gate sequence and its error codes are in The import API. It then atomically transitions `validated → importing` (a double-execute guard), enqueues a BullMQ import job, and persists `importJobId`.

The engine imports resources in dependency order, performing **id remapping** so each source identifier is translated to the correct target identifier as resources are created, and handles the hard cases automatically — cascade-skip, cycle-breaking, soft-dependency nulling, value-reference handling and a retry pass (see Dependency resolution and remapping).

- **Retry** (`POST /…/retry`) re-enqueues, resetting failed and cascade-skipped rows to pending and re-attempting only what did not land. Executing a session already in `failed` / `completed` delegates to retry. Prefer retry over re-importing from scratch.
- **Cancel** (`POST /…/cancel`) stops a queued or running import; resources already written remain. `Since 1.10` a cancel is **final** — start a new import.

Per-resource payload overrides

Between validate and execute, a caller may replace a single resource's JSON for the next execution; the archive stays untouched. In the web app this is **Edit JSON** on a resource row, which opens an embedded code editor with syntax highlighting. The endpoints, validation rules and status gates are in The import API; when to use them rather than transform the ZIP is in Generating a package.

Import access and execution guards `Since 1.10`

- **Import is space-admin only.** Validate, configure, execute, retry, cancel, payload overrides and mapping profiles all require a system-level administrator grant (or a service call). Import writes skip per-resource grants in data-service, so the gate is at the door. Export is the opposite — Developer on the exported roots.
- **Value references need an explicit action.** Execute answers `400 VALUE_REFERENCE_ACTION_UNSET` while a value reference the import will write has no action and no safe default, rather than writing a source id into the target.
- **No silent principal creation.** An **Auto-create** of a reference whose identity — a user's email or username, for example — already matches an entry in the target is refused for that reference

(`VALUE_REFERENCE_IDENTITY_COLLISION`); and principals you did not choose are never auto-created.

- **Archived targets are not "Existing".** A package resource that would match an archived target is a blocking `TARGET_ARCHIVED` error, instead of overwriting an archived record; a retry re-checks.
- **Installed resources are not overwritten.** A plain import that would write a resource owned by a package installation is refused with `409 TARGET_INSTALLED`; fork the installation first (Fork and uninstall).
- **Entity indexes are replaced, and impossible ones dropped.** An entity update always carries its indexes, so an import **replaces** the target entity's custom indexes with the packaged ones (the server-managed index status is never exported or written). A custom index whose field path the target cannot index — the path is missing, has a shape data-service cannot index, or names a language the target does not have — is dropped and reported in the result's `importWarnings` (`Dropped index "<name>": field path <path> …`) instead of failing the whole entity.
- **Mapped fields need a decision.** A mapped field whose source the import cannot find is a blocking validation error until the installer chooses what it copies from, or chooses not to copy (see Mapped fields).
- **The entries limit is checked before the run.** Before an install or a retry starts, packaging asks whether the tenant has room for **all** the entries the run will create, and refuses with `403 ENTRY_CAP_EXCEEDED` ("Archive or delete entries, or upgrade the plan, then install again") when it has not. Nothing is written. If the check itself fails for another reason, that is logged and the run goes on, because every individual create is still checked.
- **Files that could not be transferred are dropped, not fatal.** A resource still carrying a file id from the source tenant — because the export could not download the file, or the import could not upload it — has that file removed (cleared, dropped from an array, or the thumbnail removed) and an **import warning** recorded. A file id that already resolves in the target storage, as in a same-tenant import, is kept.
- **A page whose widget points outside the package still installs.** A statistic-query widget on a page whose data entry was not mapped is left **unconfigured** — the same state the widget editor produces — instead of failing the page with `UNRESOLVED_PAYLOAD_REFERENCES`. Navigation nodes and dashboards still fail explicitly.

Nulled principal references leave arrays and fail closed `Since 1.4`. When the import clears a value reference that sits inside a reference array — a service's authenticated audience, say — the entry is *removed* from the array rather than left as a `null` slot. And when that empties both the users and the groups of a view's redirection rule, the rule becomes `Restricted`, matching nobody, instead of widening to every authenticated user. The rule applies on every supported line (1.4, 1.6, 1.8 and later), and a data-service migration repairs rules that already held a `null`.

Dependency resolution and remapping

The hardest part of import is turning source-environment UUIDs into valid target-environment references. The engine pre-computes a target identifier for each resource it will create, so references can be rewritten even before the referenced resource physically exists — this is what allows the import to run in a single dependency-ordered pass. Dependencies come in two strengths: **hard** dependencies must exist for a resource to be valid (a view depends on its entity); they are followed on export and gate the import. **Soft** dependencies are optional, recorded but not followed on export, and nulled on import if they cannot be resolved. The edge format the engine reads is documented in The dependency graph.

Value references `Since 1.8`

A *value reference* is an entry-level UUID embedded in a resource field that points at an entry in a core or other entity (for example a reaction referencing a specific USER or ORGUNIT entry, or a DataCollection item). `ValueReferenceItemModel` records the `entryId`, `entityType`, occurrences (every path where it appears), whether it `existsInTarget`, any identity-match candidate (`existing`), and the chosen `ValueReferenceAction`:

Action	Effect at import
`KEEP`	Reuse the same entry (valid when it exists in target).
`REMAP`	Point the reference at a chosen target entry (`remapTargetId`).
`NULL`	Clear the reference.
`AUTO_CREATE`	Recreate the entry in the target from the packaged `entryData` (parent-first for nested entries).

`AUTO_CREATE` entries are only created when at least one *referencing resource is actually written* (CREATE/OVERWRITE); references consumed only by skipped or mapped resources are marked `skipped` ("not needed"). GROUP entries are created in their concrete source entity (`sourceEntityType` , e.g. `ROLE` , `ORGUNIT` , `USERS_GROUP`).

The packaged `entryData` (and any `sourceEntryData`) of a value reference is returned intact by `GET /v1/import/{sessionId}` and is never stripped from the session. That payload is what makes **Auto-create** an option at all: the wizard offers the action, renders the inline entry-payload editor, and computes nested-reference badges from it. A reference that arrives with no packaged payload can only be kept, remapped, or cleared.

Targets must be live. The entry index deliberately retains archived entries, so existence alone is not enough:

- A `REMAP` target must be **live**: a configuration save that points a reference at a non-`Active` entry is rejected with `400` , matching what the wizard's own entry picker offers. This catches a target archived *after* the picker listed it — or after a stored mapping profile last confirmed it. An entry whose lifecycle state is not reported at all stays permitted.
- The automatic `KEEP` (from `existsInTarget`) is held to the same bar: an archived index hit is treated as absent, so the reference falls through to the profile and natural-key stages instead of being kept onto an archived, unresolvable target.

Mapped fields: the installer decides `Since 1.10`

A **mapped field** mirrors a value from a parent entity — an order line showing its order's customer, say. It records *which* field on the parent it copies by that field's identifier, and identifiers do not survive the move between environments: the parent entity is usually matched onto an entity that already exists in the target, so the parent keeps the target's own identifiers and the packaged one names nothing there. Import first repairs this by matching on the field's name instead, which is stable across environments. The name travels in the `field-mapping-hints.json` sidecar that schema 1.2 added; without it, import can only read the name off the parent entity's own copy inside the package.

When the name match fails, **the installer decides** what the field copies from — *copy from* a field on the target parent, or *don't copy* — and the run does not start until every such field is decided. Only fields of the same type, with compatible relation targets and (for two-way mappings) matching required-ness are offered, and the list may be empty. A decision belongs to the target parent it was made for: change what the parent maps to and a decision that no longer fits is discarded. A store install or update that runs without the wizard cannot decide for you: an undecided mapped field comes back among the choices you must make on the import page.

The per-field `state` values, the candidate rules, the decision payload and the execute-time refusals (`MAPPED_FIELD_SOURCE_UNDECIDED` , `MAPPED_FIELD_SOURCE_UNAVAILABLE`) are specified in The import API.

External dependencies, explained `Since 1.10`

A package that refers to something it does not carry — typically an entity another module owns — lists it as an **external dependency**, and also says **why**. Each such item (in a published version's manifest, and in the unmapped dependencies an installer sees) carries `usedBy` — what in the package uses it: a field that **references** it, a mapped field that **copies** one of its fields (naming the parent field copied), or some **other** use with no field, such as a view naming

the entity — and `requiredFields`, the fields the target entity must offer for those copies to work. The publish view and the install wizard both show this ("Used by …", "*field* copies its *source* field"), so an installer can pick the right entity to map the dependency to before deciding the copied fields above.

When publishing refuses an identifier left inside a written value (`PUBLISH_DEPENDENCY_REFERENCE_UNSUPPORTED`), the problem names the resource — type, id and name — and, for a field assignment such as a reaction's, the field.

Page widgets pointing outside the package `Since dev`

A page widget's reference (a data-view widget's source or target, and every other widget option the dependency graph reads) is a **soft** edge. On the 1.10 line, publishing declares only hard external targets, and the import nulls such a soft reference without asking, so the widget installs broken. On the development line such references can be mapped:

- **Publish** also declares a soft external target in the version's external dependencies when one of its uses is a **page-widget reference**: the using resource is a page (`GenericView`), the reference sits under one of its widgets (`components[n]` or `groups[n]`, but not under a widget's visibility settings), and the target is not a module. It is declared **without becoming a package dependency**.
- **Install validation** offers a mapping for every external reference that is hard **or** such a page-widget reference. The row in `unmappedDependencies` carries `hard: false` when every use is a page-widget reference (no `hard` field means hard). The page's own dependency on it stays soft, so cascade-skip and impact counts do not change. A soft target that is scoped to an entity (a view, say) gets no name-based suggestion, because the reference does not say which entity it belongs to and such names are unique only inside one; a target that exists under the same id in this environment is still suggested.
- **Mapped**, the widget points at the chosen resource. **Left unmapped**, the widget's reference is nulled. The import page tags the row *Optional* (its tooltip: "Only page widgets use this. If you leave it unmapped, those widgets are cleared and the import continues.") and does not hold **Execute** back for it. A widget's reference to a module is never offered, and other soft page references — sharing, domain, visibility principals — are unchanged.

> ⚠ **Server-side execute check not yet aligned**
>
> At the development-line head this was checked against (backend `1.11.0-alpha.4`), the server's own execute check and the store's list of unresolved choices do not read `hard` yet: an unmapped optional row whose page the import will write is still refused at execute (`400`, "All unmapped dependencies must be resolved before execution") and, for a store install, is still returned as an unresolved choice. Until that is aligned, map the row to proceed.

Required workflow-step owners `Since 1.8`

A workflow step whose owner is **Fixed** must have a primary owner: data-service rejects an entity whose `workflows[n].steps[n].owner.source.value` is empty. Packaging enforces the same rule on the way in — a value reference whose action would **clear** that path (`NULL`) on a resource the import is actually going to write (`CREATE` or `OVERWRITE`) is blocked. A step's `fallbackOwner[n].source.value` is deliberately excluded, because a fallback owner is optional.

The guard fires at three points, and each evaluates the **effective** state after your change rather than the change in isolation:

Where	What you see
Validation (Phase 1)	A blocking `REQUIRED_OWNER_REFERENCE_CLEARED` error in the validation report, with `canProceed: false`.
Saving the configuration	The auto-save is rejected with `400`. A mappings-only save is caught too, when it re-activates a resource whose owner was cleared earlier.
Applying a bulk strategy	A bulk flip from **Skip** to **Create**/**Overwrite** that re-exposes a cleared owner is rejected the same way.

To clear the block, give the owner a real target: **Remap** the reference to a principal that exists in the target space, or **Auto-create** it from the packaged payload. **Keep** and **Clear** are not options once the guard has fired. Flipping the owning resource back to **Skip** also lifts it — the guard only considers resources the import will write, so the now-moot error is dropped from the report and `canProceed` is recomputed.

Re-pointing a value reference `Since 1.8`

Two rules govern what the import wizard sends when a reference's target changes. Both matter most when a stored mapping profile has already pre-filled the row:

- **A chosen target wins over the row's default.** Picking a concrete target is sent as an explicit remap even when `REMAP` is already the row's default action. A reference restored from a stored profile reloads with both its action *and* its default set to `REMAP`, so re-pointing only the target must still be sent, or the stale profile target would be kept.
- **Per-occurrence targets survive the save.** When one principal appears at several paths (the same fallback owner on several workflow steps, say), the target can be overridden for a single occurrence. Choosing a target for one occurrence also records the row's action on that occurrence, so the override is complete enough to be sent; clearing the target back to the row's default drops the override and the occurrence re-inherits from the row. Re-targeting a single occurrence additionally promotes a **Skip**ped owning resource to **Overwrite**, since a reference change only takes effect if the resource is actually written.

A reference whose decision came from a stored mapping profile is labelled as such in the wizard — a **Profile** pill in place of the natural-key-match pill — and carries a *Saved mapping unavailable* warning pill when the remembered target no longer resolves.

A shared principal remapped per resource `Since 1.8`

When the **same** principal — one user, group, or org unit — is referenced from more than one **resource** in the same import (a workflow step owner on an entity, say, and the user a view is restricted to), each resource's copy of that reference is retargetable independently. The wizard recognises that a principal renders under more than one resource row, and lets each resource's group read and write only its own occurrences. A principal confined to a single resource keeps using the row-level decision, which is also what a stored mapping profile remembers.

The row-level default is kept when one group diverges. An occurrence-level choice is only ever a *partial* override layered on the principal's row-level decision, and the server rejects an item-level remap that carries no target of its own. So the first target chosen for a shared principal also **establishes the row's default**; a later group for the same principal then diverges *against* that default instead of replacing it. That is what makes a choice made during **Configure** survive through to **Execute**, including for a principal that had no row-level target yet.

Auto-create on a principal that spans resources always applies at the row level rather than as a per-occurrence override — auto-created entries are keyed by the principal itself, not by where it happens to appear.

Cascade skip, cycles, and soft-dependency nulling

- **Cascade skip** — if a hard dependency is skipped or fails, all its hard dependents are cascade-skipped (`cascade_skipped`) before any writes.
- **Cycle breaking** — hard-dependency cycles between `DataEntity`s are detected (SCC) and broken: a member is stubbed on CREATE, then PATCHed with real cross-cycle refs once every member exists.
- **Soft-dependency nulling** — an unresolvable soft dependency's path is set to `null` rather than left as a broken source UUID (with type-specific exceptions, e.g. `dataSource.query` is left intact so data-service can surface a clearer error). `Since dev` a page widget's reference to something outside the package can be mapped instead.
- **Retry queue** — resources failing with a "missing target dependency" code (`DATA_QUERY_NOT_EXIST`, `DATA_ENTITY_NOT_EXIST`, `DATA_VIEW_NOT_EXIST`, `DATA_COLLECTION_NOT_EXIST`, `RELATION_NOT_EXIST`) are retried once after the main loop, after implicit-child discovery has populated the id map.

Topological ordering

Both export and import re-sort resources with `topologicalSort` (Kahn's algorithm over hard edges) so parents are created before children (e.g. `DataEntity` before its `DataView`). Import re-sorts even if the package was older or unsorted.

The dependency-service graph

`dependency-service` (`/api/dependency`) owns the dependency graph and is queried by packaging during export (outbound closure, dependents-tree for `withChildren`) and consumed at import via `dependencies.json`. Stored edges carry `fromType/fromId`, `toType/toId`, `kind` (`MEMBERSHIP` or `REFERENCE` — the only two `DependencyKind` values), `hard` (blocks target deletion), `path`, `resolvedVia` (`DIRECT_ID` / `NORMALIZED_NAME` / `REFERENCE_PROPERTY` / `PIPELINE_STAGE`) and `resourceVersion` (SHA-256 of the snapshot, for idempotent rebuilds). The wire shape of an edge is in The dependency graph. The read endpoints:

Method	Path	Purpose
`GET`	`/v1/dependencies/{type}/{id}`	Outgoing edges (what does this depend on?)
`GET`	`/v1/dependents/{type}/{id}`	Incoming edges (what depends on this?)
`GET`	`/v1/validate-delete/{type}/{id}`	Hard vs soft dependents (safe-delete)
`GET`	`/v1/impact/{type}/{id}`	Transitive impact analysis
`GET`	`/v1/external-dependents/{type}/{id}`	`Since 1.10`. What *outside* a module or entity references it or anything it owns — the list the module-delete, entity-delete and uninstall dialogs show

Edges are extracted by a declarative schema DSL and rebuilt event-driven (RabbitMQ lifecycle events → BullMQ workers).

Stored mapping profiles `Since 1.8`

After a successful import, confirmed **principal** (USER/GROUP/ORGUNIT) mappings are persisted as a `MappingProfile` keyed by `(sourceEnvironmentId, sourceSpaceId, targetSpaceId)`. A later import from the same source pre-applies those decisions before natural-key matching, so repeat promotions inherit prior choices.

What is remembered:

- **At most one decision per source principal.** A principal remapped per resource whose occurrences disagree is not persisted as a `REMAP` (the divergent choice would otherwise be re-applied to every occurrence next time). A successful `AUTO_CREATE` or an item-level `NULL` clear is unaffected, since neither is ever an occurrence-promoted value.
- **A confirmed `REMAP` is remembered** `Since 1.10`. An empty per-occurrence override counts as no override, so it does not make a principal look divergent; the principal you remapped is pre-applied on the next import from the same source. (On 1.8, saving the configuration stored every unset override as an explicit empty value, so in practice only auto-created principals reached the profile.)

Admin API (`mapping-profile.controller.ts`, space-admin gated, scoped to `x-space-id`):

Method	Path	Purpose
`GET`	`/v1/admin/mapping-profiles/count`	Count profiles for the space
`GET`	`/v1/admin/mapping-profiles`	List profile summaries
`DELETE`	`/v1/admin/mapping-profiles/{profileId}`	Delete a whole profile
`DELETE`	`/v1/admin/mapping-profiles/{profileId}/entries/{sourcePrincipalId}`	Delete one remembered entry

`MappingProfileEntry`: `sourcePrincipalId`, `action` (`remap` default / `clear`), `targetPrincipalId?`, `entityType`, `matchedBy?`, `confidence?`, `updatedAt`. Deletes take effect on the next import session. A cross-space or missing profile id returns `404` (never leaks existence in another space).

Before a remembered `targetPrincipalId` is re-applied, OSOS checks that the principal is still `Active` — liveness, not mere presence in the index. A remembered entry whose target has since been **archived** is treated as stale exactly like a deleted one: it is not applied as a live remap, the reference falls through to natural-key matching, and the row is flagged (`profileTargetMissing`) with a `PROFILE_TARGET_MISSING` warning so the wizard can mark it visibly. Unlike a deleted target, an archived target is **not pruned** from the stored profile — archiving is reversible, so the remembered decision survives it and takes effect again once the principal is restored. Only entries whose target has disappeared from the index entirely are dropped.

Collections in packaging

`DataCollection` is an exportable type but needs special handling:

- **Non-ADT (entity-backed) collections** are exported with `items: []` — their items are `DataEntry` UUIDs not carried in the package, so keeping them would create dangling references. **ADT collections** (whose `dataEntity` is a primitive ADT constant like `ADTString`) keep their embedded scalar items.
- Collections are **scoped**: entity-scoped when `dataEntity` is a real entity UUID, **global** when it is an ADT constant. Scope drives the existing-resource match at validation and the ZIP lookup key.
- Global/ADT collections have **no module membership edge** to climb, so during export dependency-analysis they are surfaced under a synthetic "Global" group rather than dropped.
- Collection **items** referenced by other resources are treated as value references with `entityResourceType = DataCollection`; existence checks and `AUTO_CREATE` use the DataCollection APIs instead of the entry-index APIs.

The import state machine

An import session moves through **validating** → **validated** → **importing** → **completed** (or **failed**). It can be executed only once — concurrent execute/retry/cancel attempts are resolved atomically — a session stuck in **importing** after a

worker crash is recovered to **failed** so a retry can unblock it, and a retry re-attempts only the failed and cascade-skipped resources. The full transition table and the recovery window are in The import API.

The packaging screens `Since 1.10`

The **Packages** menu holds four screens: **Store**, **Installed**, **Build** and **Reviews**. How a builder uses them is in Export, import and install apps [Build Apps]; their routes and visibility rules are in the marketplace page. The behaviours below are the ones that matter when you build on, test, or support those screens.

Build page and package page

- **Build** (`/admin/packages/build`) has two tabs: *Packages* (sortable by any column; a package's **Delete** stays visible but disabled, with the reason, while the package has versions) and *Exports* (`/admin/packages/build/exports` , the export records). Old `/admin/packages/mine/...` addresses redirect to `/build/...` .
- **The package page** (`/admin/packages/build/{packageId}`) edits the name and description in place (Enter or Ctrl/Cmd+Enter saves, Esc cancels). **Download**, in the header beside *View in store*, exports the package's modules as a file and is shown to developers while the package still has a module that exists. **Unpublish**, **Republish** and, for a package without versions, **Delete** sit in the ⋯ menu.
- **Contents** (the default tab) is one list of what the next version will hold — *Modules*, *Referenced records*, *Pages* and *Depends on* — each row saying what changed since the newest version, with an arrow that opens the changed items. A deleted module or page says what to do in its row. *Referenced records* are the records the package's settings point to (a workflow's owner group, say), of which the package carries a copy.
- **README** closes the Contents list and is marked when it changed. Markdown pasted into the README editor as plain text — headings, code fences, quotes, tables, bold, links, inline code, or two or more list lines — is converted to formatted text as one undo step. `Since 1.10` (from 1.10.2) the section says, before a version exists, that the next version keeps a copy of this README — or warns to add one first, once the README is known to be missing. `Since dev` a pasted web image is copied into the package. Both are detailed on the marketplace page.
- **Concurrent edits** `Since dev` . The page sends the package's update token with every draft and README save, so a save made from an older tab, or over someone else's newer save, is refused ("This was changed in another tab or by someone else since you opened it…") instead of overwriting it — see Stale saves are refused.
- **Versions.** *Create version* is a button on this tab; the dialog closes once the version exists, because sending it for review is a separate decision. A version the store never received can be **deleted**; one that was sent for review or is installed somewhere is refused (`VERSION_SENT` , `VERSION_INSTALLED`). Deleting a version removes its **record** — the ZIP stays in storage — and its number is never reused. *Compare versions* also shows when the README was added, removed or updated. There is no *Deprecate* action (it only ever marked the version in this environment); *Undo deprecation* stays for versions already deprecated.
- **History sidebar.** Created, edited (name, description, modules and pages added or removed, dependency ranges), README edited, versions created, deprecated, undeprecated and deleted, plus the review events — sent, approved, returned, store page listed or rejected.
- **Live reload.** The page reloads when the platform pushes a `packages/changed` event for the open package (a review decision, for example), and again when the browser tab regains focus.
- **Global pages.** *Pages* lists the package's global pages — pages that belong to no module or entity. A package may hold only pages. A page that later moves into a module or entity cannot be published (`PUBLISH_PAGE_NOT_GLOBAL`), and neither can one that was installed or forked from another package.

Installed list and installation page

- **Installed list** (`/admin/packages/installed`): a *Status* select (Installed, Failed, Uninstalled) and a *Source* select beside the search and sort; both start empty, so every status and source shows. *Installed* includes failed and running installs, so a failed install appears under both *Installed* and *Failed*. A row with a live installation opens its installation page; an import record that has a live installation redirects there too.
- **Installation page** (`/admin/packages/installed/installations/{installationId}`): source, origin, export time and result come from the newest import record behind the installation, with its README. *History* lists every install run (a failed run says so), plus fork and uninstall.
- **Failed installs** offer retry from the import session while it exists, or uploading the same package file again, which takes the installation over. When a compatible newer version is available, *Update* is offered instead: it overwrites what the failed run created and installs the rest. A failed store install, or a stopped store update, also offers **Report to *publisher*** `Since 1.10` (from 1.10.1) — see Reporting a failed install or update.
- **Fork** asks for the package's name to confirm, and is refused while other packages depend on the installation or an update is running. A forked installation's resources show "Forked, so changes to these resources are no longer checked". The page can render a *Deleted* tag for a resource removed after the fork, but the API reports such a resource as *missing*, so that tag does not appear yet.
- **Uninstall** is offered on an active installation and on a forked one (the dialog warns that your own changes to forked resources are deleted with them). The button stays disabled until every link from **outside** the package is removed; the server enforces the same rule with `409 UNINSTALL_BLOCKED_BY_OUTSIDE_LINKS` , so a call that skips the dialog is refused too. A stopped uninstall offers *Retry*.

Installed views and pages in their editors

A view or page installed by a package is **locked** by it. The settings, help and content panes are inert, and the save shortcut and **Done** are refused like the header Save; publishing or hiding a view stays available from the entity's views list. Name, description and icon can still be changed (the editor shows the package's value beside yours, your label is kept on every update, and an implicit view's name stays read-only); the server rules are on Data views [Build Apps › Data Views & Faces] and Pages and widgets [Build Apps › Pages & Widgets]. Relation overrides stay usable (they are saved on the parent view), and script editors are read-only but selectable, so their code can still be read and copied.

Operational limits and safety

Guard	Value	Where
Max decompressed package	200 MB	`MAX_DECOMPRESSED_SIZE`
Max ZIP entries	10 000	`MAX_ZIP_ENTRIES`
Path-traversal / unsafe entries	rejected	`isSafeEntryName` , `SAFE_PATH_COMPONENT`
Persisted session error message	capped 2 000 chars	`MAX_ERROR_MESSAGE_LENGTH`
Session TTL	7 days	`import-session.model.ts`
Stuck-import recovery	15 minutes in `importing`	`IMPORT_STUCK_THRESHOLD_MS`
Export/import concurrency	BullMQ workers (Redis)	`EXPORT_QUEUE_NAME` / `IMPORT_QUEUE_NAME`
Cancel-check cadence	every 5 processed resources	`CANCEL_CHECK_INTERVAL`

Implicit-child settle/poll timings are environment-overridable (`PKG_IMPLICIT_SETTLE_MS` , `PKG_IMPLICIT_POLL_ATTEMPTS` , `PKG_IMPLICIT_POLL_DELAY_MS`) — used so import can wait for data-service to spawn implicit views and queries asynchronously after an entity CREATE.

Limitations and gotchas

- **Hard edges only.** Only hard references are guaranteed to travel with a package; soft references may be nulled on import — review the export analysis warnings and re-point them afterwards if needed. `Since dev` a page widget's reference outside the package is offered for mapping at install (Page widgets pointing outside the package).
- **Function secrets do not travel** `Since dev`. A packaged function carries its secret names only; set the values in each target environment (Function secrets stay in their environment).
- **Principal mapping rarely lines up by itself** across environments — use mapping profiles and exact-match suggestions to map users and groups.
- **Cascade-skip on failure.** One failed hard dependency skips its dependents. Fix the root cause and **retry**.
- **Templates need their fields.** A template exported without its entity (or whose referenced fields or tokens are missing in the target) will fail to render after import.
- **Newer packages into older targets.** The engine tolerates some version drift (for example, it synthesizes implicit child resources an older target does not auto-create), but a package built on a newer schema version than the target accepts is refused with an incompatible-schema error.
- `ValidationStatus` **is only** `NEW` **or** `EXISTING`**.** A `CONFLICT` value is mentioned in a docstring and the web app has a *Conflict* label, but the current validation path never produces it — identity collisions are surfaced separately in the validation report.
- **Field-level wire DTOs** are defined in the `@osos/shared` contracts and the controller spec builders; these pages document the verified fields and shapes, not every optional property of every wire DTO. packaging-service and dependency-service publish no OpenAPI description, so they do not appear in the generated endpoint reference.

Distributing ready-made extensions

Packaging is how *your* app resources travel between environments. OSOS also ships **ready-made front-end plugin packages** (custom plugins) that provide additional display components. These are distributed as an **internal, in-app dynamic-component registry**: each plugin package declares components (widgets/templates) that are lazy-loaded and registered internally so the app resolves them by name at runtime. **Plugins are internal-only — there is no remote/federated loading** — they ship bundled with the app build and are not fetched from an external source at runtime, and are added through the Generic View builder. This is distinct from the ZIP packaging flow above, which distributes environment resources rather than bundled code.

> **Note:** The sample plugin scaffolds found in the plugins library are **empty shells with no actual functionality** — they are starting points for building your own plugin, not working components.

Related but distinct: module/entity JSON transfer

`data-service` also has an older, separate mechanism that exports and imports **modules and entities as one JSON file**. It is not the ZIP packaging system above and shares no code, format or endpoints with it; use packaging for a full app. Its format, endpoints and limits are in Module JSON transfer.

The Package Format

Packaging & Distribution documents the **workflow** — export jobs, import sessions, mappings, execution. This page documents the **artifact** those workflows produce and consume: exactly which files a package ZIP contains, what every manifest field means, and what the import boundary accepts or rejects. It exists for the reader who wants to inspect, transform, or generate a package rather than click through the export screen. Two companion pages carry the detail this one deliberately hands off: the payload shape inside each resource file is specified in Package Resource Schemas, and the supported ways to produce a package are compared in Generating a Package. The packaging engine this contract belongs to shipped in the **1.8** line (`Since 1.8`); on an older deployment these endpoints and the export UI are absent.

One thing is worth stating before any detail, because it is the question that usually brings a reader here: **a OSOS package is not a single JSON file.** It is a multi-file ZIP whose parts must agree with one another — the index must name every resource file, the dependency edges must point at real paths inside the payloads, and the manifest must describe both.

Archive layout

The archive is built and parsed by the packaging service's ZIP utilities. Fourteen kinds of top-level entry may appear — four of them `Since 1.10`; anything else fails validation (see the strict validator below):

```
manifest.json              required
index.json                 required
dependencies.json          written by every export; required at import
config.json                echo of the export request
entry-references.json      optional
store-references.json      optional
file-references.json       optional
field-mapping-hints.json   optional, schema 1.2 and later
hashes.json                published versions only, schema 1.3 and later
package-references.json    published versions only, schema 1.3 and later
README.md                  optional, schema 1.4
readme/{image}             optional, schema 1.4
files/{fileId}.bin         one per file reference
resources/{Type}/{normalizedName}.json
```

Entry	Presence	Contents
`manifest.json`	required	Package metadata — see below.
`index.json`	required	The ordered resource list and the id-to-filename map — see below.
`dependencies.json`	required at import	The serialized dependency edges — see below. Its absence fails extraction with `Invalid package: missing dependencies.json`.
`config.json`	written by every export	An echo of the original export request: `resources`, plus `excludes`, `excludeTypes`, `name` and `description` when they were set. Informational — import does not act on it.
`entry-references.json`	optional	`PackageEntryReference[]` — value-level entry references (specific users, groups, collection items) with the packaged `entryData` that makes auto-creation possible in the target.
`store-references.json`	optional	`PackageStoreReference[]` — snapshots of shared (cross-space) stores referenced by exported entities.
`file-references.json`	optional	`PackageFileReference[]` — `{fileId, fileName, mimeType, size}` metadata for template/binary attachments.

Entry	Presence	Contents
`field-mapping-hints.json` `Since 1.10`	optional	The mapped-field sidecar — the **only** structural addition in schema `1.2`. Written only when non-empty, and its presence is what makes the package declare `1.2`. See the sidecar's own section.
`hashes.json` `Since 1.10`	published versions	Canonical content hashes per resource and the package's `rootHash`. See published-version files.
`package-references.json` `Since 1.10`	published versions	Every reference from this package into another package. See published-version files.
`README.md`, `readme/{image}` `Since 1.10`	optional	The package README and the images it links to — the addition of schema `1.4`. See published-version files.
`files/{fileId}.bin`	one per file reference	The binary payload itself. Extraction fails with `missing binary for file reference` if a listed reference has no matching `.bin` entry.
`resources/{Type}/{normalizedName}.json`	the payload	One JSON snapshot per exported resource, grouped in one directory per type.

One more name, `unmapped-dependencies.json`, sits on the validator's allow-list but is not written by the current exporter — treat it as reserved.

Every JSON member is written minified (`JSON.stringify` with no spacing). The importer requires only valid JSON, not any particular formatting, so a transformed package may be pretty-printed freely.

manifest.json

The manifest is the first file read at import. Its authoritative definition is the shared `PackageManifest` contract; the export side writes it with `buildPackageManifest`.

Field	Required	Meaning
`schemaVersion`	yes	Package **format** version. The writer stamps the lowest version that describes the package (`pickSchemaVersion`): `1.4` when the package carries a README, else `1.3` when it carries a `package` block, else `1.2` when `field-mapping-hints.json` is present, else `1.1` — the three newer values are all `Since 1.10`. A 1.10+ importer accepts `1.0` to `1.4`; a pre-1.10 one accepts only `1.0` and `1.1` (`SUPPORTED_SCHEMA_VERSIONS`). Anything else is a blocking `INCOMPATIBLE_SCHEMA` validation error. The compatibility rules have their own page.
`createdAt`	yes	ISO 8601 export timestamp. Informational only.
`resourceCount`	yes	The number of resources actually packaged — the length of `index.json`'s `orderedResources`, after export-side filtering.
`resourceTypes`	yes	The distinct resource types that have at least one file under `resources/`. Only exportable types ever appear here.
`sourceEnvironment`	no	Legacy free-text environment name (the deployment's `NAMESPACE`). Superseded by `sourceEnvironmentId` but still written.
`sourceEnvironmentId`	no	Stable source-environment id, from the `ENVIRONMENT_ID` platform config. Omitted when that config is unset.
`sourceSpaceId`	no	The space the package was exported from.
`sourceVersion`	no	Application version at export time. Informational — compared only to warn (`NEWER_SOURCE_VERSION`), never to block.

Field	Required	Meaning
`description`	no	Free text supplied at export.
`package` `Since 1.10`	no	The published-version identity block — present only in a published package version, and what turns an import into an install. See published-version files.
`origin` `Since 1.10`	no	Where the file came from: `exportId`, `name`, `environment {id, name}`, `space {id, name}`, `exportedAt`, `exportedBy {id, name}`. Its members are nullable only on files exported before origins existed. The importer records it per resource, which is how the target can say where a resource came from.
`readme` `Since 1.10`	no	`{path: 'README.md', images[]}` — the README's location and its listed images (schema `1.4`).
`contents` `Since 1.10`	no	A summary for display: per-type `counts`, `modules[]` with their entities and field counts, an optional `other` bucket, `pages[]` (the package's global pages, `{id?, name}`), and `truncated` when the 100-module / 500-entity / 100-page cap is hit. Each module, entity (with its `moduleId`) and page also records its source-environment `id`, so a reviewer can be linked to the originals; versions built before the ids were recorded carry names only.

`sourceEnvironmentId` and `sourceSpaceId` matter more than they look: together with the target space they form the key of a persisted **mapping profile**, the mechanism that lets a repeat import from the same source pre-apply previously confirmed principal mappings. A hand-built manifest that omits them still imports, but no stored profile can ever match it — every user, group, and org-unit mapping starts from scratch on every import.

The strict validator type-checks only the four required fields; the optional fields are read where they are used.

index.json

`PackageIndex` has three members:

Member	Type	Meaning
`orderedResources`	`ResourceRef[]`	Every packaged resource as `{type, id, …}`, **topologically sorted** — parents before children.
`nameMap`	`Record<id, filename>`	Resource id → the base name of its file under `resources/{Type}/`. **This is the authoritative filename key.**
`displayNameMap`	`Record<id, localizedname>`	Resource id → `{en, ar}` display name, for the import wizard. Optional.

`nameMap` is authoritative for a reason a producer must respect: when two resources of the same type slugify to the same `normalizedName`, the exporter suffixes the second file name with the resource id (`projects` → `projects_<id>`). The resource's own `normalizedName` field is therefore only the *raw* slug, shared by every same-named sibling. The importer (`resolvePackagedResource`) resolves files through `nameMap` first and id-verifies any hit reached through a raw slug, precisely so that a name collision cannot silently return a same-named-but-different resource. A generator that derives filenames from anything other than its own `nameMap` recreates the bug that logic exists to prevent.

dependencies.json

A serialized `DependencyEdge[]` — the outbound dependency closure of the exported selection, fetched from the dependency graph at export time. The shape:

Field	Meaning
`from`, `to`	`ResourceRef`s — the depending resource and its dependency.

Field	Meaning
`path`	The JSON path inside the `from` resource's snapshot where the reference lives. This is what tells the importer **where to rewrite an id**.
`resolvedVia`	How the reference was found: `DIRECT_ID`, `NORMALIZED_NAME`, `REFERENCE_PROPERTY`, or `PIPELINE_STAGE`.
`hard`	Whether the dependency is structural. Hard edges gate import ordering and cascade-skips; soft edges may be nulled when unresolvable.
`kind`	`MEMBERSHIP` (child belongs to parent) or `REFERENCE` (cross-reference) — the only two values of the `DependencyKind` enum.
`fromActive`, `fromArchived`, `metadata`	Status flags and extra context.

The edges do double duty: they drive **id remapping** (via `path`) and the **topological sort** that orders the import. A producer that adds resources without matching edges gets them imported in arbitrary order with no reference rewriting — the resource lands, but everything it points at still carries source-environment ids. Edge semantics, the graph API, and how edges are extracted are covered in The Dependency Graph.

field-mapping-hints.json `Since 1.10`

This sidecar is the **entire** structural difference between schema `1.1` and schema `1.2`. No resource JSON gained a property, no manifest field was added or removed, and no existing entry changed shape: `1.2` adds one optional top-level entry and nothing else.

A **mapped field** is a `DataEntityField` carrying `mapping.enabled` — a field that mirrors a value from a parent entity. Its `mapping` block holds two field UUIDs: `referenceFieldId`, a field on *this* entity that points at the parent, and `sourceFieldId`, a field on the *parent* entity whose value is mirrored (plus an `updateMode`: `Local` `0`, `Never` `1`, `ParentOnly` `2`, `TwoWay` `3`).

`referenceFieldId` survives an import for free, because it is intra-entity and data-service reconciles fields by the payload's own `id`. `sourceFieldId` does not. The parent entity is usually **mapped** onto a pre-existing target entity rather than freshly written, so it keeps the target environment's own field UUIDs and the packaged `sourceFieldId` matches nothing there. The join key that does survive the environment boundary is the field's `fullNormalizedName` — `<normalizedName>` for a top-level field, `<block>.<field>` for a field on a block's internal entity.

Before `1.2` nothing carried that name across the boundary, so a mapping whose parent snapshot was absent from the package was **dropped silently while the import reported success**. The sidecar closes that gap. It maps `` `${entityId}:${fieldId}` `` — source-environment ids — to:

```
interface FieldMappingHint {
  sourceEntityId: string;                // source-env id of the mapped field's PARENT entity
  sourceFieldFullNormalizedName: string;  // flattened name of the source field on that parent
}
```

Two rules govern the entry as an archive member. It is **written only when non-empty**, which is what keeps a package with no mapped fields at schema `1.1` and therefore importable into an environment that has not been upgraded. And it is **read optionally**: its absence is not an error at any supported version, because the resolver falls back to the parent entity's own in-package snapshot — the path a `1.0` or `1.1` package has always taken.

The hints are regenerated on every export rather than persisted on the entity, deliberately, so they cannot drift when a field is renamed. When a source field still cannot be found in the target, the installer is asked what the field copies from — a decision the run will not start without — as documented with the import API.

Published-version files `Since 1.10`

A package version **published** from the Packages area — see Packages, Marketplace & the In-App Store — is the same ZIP with four additions. A plain export carries none of them.

`manifest.package` (`ManifestPackageInfo`):

Member	Meaning
`id`	The permanent package id, `pkg_` + a 26-character ULID.
`version`	Strict semver.
`displayName`, `description?`	Localized.
`releaseNotes`	Localized; required.
`minPlatformVersion`	The lowest platform version that may install it.
`builtOn?`	The authoring environment's platform version.
`license`	Always `'free'`.
`rootHash`	The package's canonical hash (from `hashes.json`).
`dependencies[]`	`{packageId, range, displayName?}` — other packages this one needs, as semver ranges.
`externalDependencies[]`	`{type, sourceId, name, requiredBy[]}` — things outside any package that it references. `Since 1.10` each item also explains itself: `displayName?`, `usedBy[]` (`{resourceId, resourceType, resourceName?, fieldName?, fieldLabel?, kind, copiedField?}`, with `kind` one of `reference`, `copiesField` or `other`) and `requiredFields[]` (`{name, label?, dataType?}`, the fields the target entity must offer for the package's copies to work). `Since dev` it may also list a soft target that a page widget shows (a view of an entity the package leaves out): such an item is mappable at install but is not a package dependency, and the installer sees it as an optional row (`hard: false`) — see Page widgets pointing outside the package.

On import, a `package` block selects **install mode**, which requires every member except `description` and `builtOn` to be well formed (`INSTALL_MANIFEST_INVALID` otherwise). A `package.id` that is not a valid `pkg_` id — a file published before package ids existed — is refused with `422 PACKAGE_FORMAT_OUTDATED`.

`hashes.json` — the `algorithm`, the `rootHash`, an optional `readme` hash (included in `rootHash` when set) and, per resource key, its `type`, `name`, `hash`, optional per-element hashes and, `Since 1.10`, an optional `parentKey` — the resource key of the packaged resource it belongs to (a view's entity, an entity's module), recorded outside the hash so adding it changed no `rootHash`, and absent from versions published before parents were recorded. The `algorithm` is `sha256/osos-canonical-v1`, `-v2` or `-v3`: new versions are written as **v3**, and a reader must accept all three. v2 leaves out the keys a client may change on an installed resource (see Package Resource Schemas); v3 additionally hashes a referenced record that the import matches by identity — a user, group, domain or space — by that identity alone, so a profile edit of a referenced user is not a change to the package. `Since dev` a `Function`'s hash covers its secret **names** only, under the same algorithm version (see Function secrets). Each resource is hashed as canonical JSON with server-managed and environment-specific keys left out and ids replaced by package-relative keys, so the same content hashes identically in any environment. That is what lets an installation report whether an installed resource has `changed`.

`package-references.json` — every reference from this package into another package: `resourceKey`, `path`, `packageId`, `targetResourceKey`. A reference that does not resolve in the installing space is

`INSTALL_REFERENCE_UNRESOLVED`.

`README.md` and `readme/` — the README's Markdown and the images it may link to, listed in `manifest.readme`. Only a **published version** is written with a README; a plain export takes none, though the importer still reads and keeps one it finds. They are validated on import against these limits:

Limit	Value
Markdown	256 KiB
Each image	2 MiB; PNG, JPEG, WebP or GIF
Images	at most 50, 20 MiB in total

A README that is missing, has the wrong path or type, exceeds a limit, or does not match its hash is reported as a README problem. All four additions are on the strict validator's allow-list; a pre-1.10 service turns a package declaring `1.3` or `1.4` away with `INCOMPATIBLE_SCHEMA` before it ever reaches them.

The resource files

Each packaged resource is one JSON file at `resources/{Type}/{name}.json`, where `{name}` comes from `nameMap`. The per-type payload schemas are the subject of Package Resource Schemas; what belongs here is what is true of the files as *archive members*:

- **Server-managed keys are absent.** Export deletes `_id`, `_resolved`, `createdAt`, `updatedAt`, `createdBy`, `updatedBy`, `creationInfo`, `updateInfo`, `stateHistory`, `updateIdentifier`, `spaceId`, `space`, `version`, and `__v` from every snapshot (`SERVER_MANAGED_KEYS`), and the import parsers strip them again defensively before writing. Do not emit them; do not expect them.
- **`id` survives.** It is the source-environment UUID, kept so the importer can id-verify filename lookups and key the remap table.
- **`executionPlan` is stripped** from `DataView` and `GenericView` snapshots — it is server-derived and regenerated in the target.
- **Entity-backed `DataCollection` snapshots carry `items: []`** — their items are entry UUIDs that do not travel in the package. ADT (primitive-typed) collections keep their embedded scalar items.
- **A `Function` carries secret names, never values** `Since dev`. Its `value.secrets[]` rows keep their `key` and any other field, but every `value` is `null`; the `OSOS_API_TOKEN` row and the `identity` reference are removed. See Function secrets stay in their environment.

Seventeen exportable types — and the eighteenth

`EXPORTABLE_TYPES` is the complete list of types that may appear under `resources/` — seventeen of them:

`DataEntity`, `DataView`, `DataQuery`, `DataRelation`, `DataCollection`, `DataReaction`, `GenericView`, `DataModule`, `SystemButton`, `StatisticQuery`, `Report`, `Function`, `CounterWidget`, `Announcement`, `DataEntityAggregate`, `QuickFilterGroup`, `TagGroup`.

The last three are new — `Since 1.10`. `QuickFilterGroup` and `TagGroup` are the groups an entity's quick filters and tags are organised into; they travel as entity children, and their payload shapes are not specified here (export one and read it). `DataEntityAggregate`, the third, is the resource behind the entity's **Aggregator** list, it travels as an entity child under `resources/DataEntityAggregate/`, and its payload shape is documented with the resource schemas. A package produced before it existed simply has no such directory.

The `ResourceType` enum those values come from has **eighteen** members. The extra one is `Dashboard`, and the mismatch is deliberate — the two lists answer different questions:

- `ResourceType` is the **dependency graph's** node vocabulary; its doc comment reads "Resource types tracked by the dependency graph."
- `EXPORTABLE_TYPES` is the **packaging subset** of that vocabulary.

A dashboard is an **entry-based** resource: `dependency-service` maps it from the `DASHBOARD` core entity, i.e. a dashboard is a data *entry* of a system entity, not a standalone resource document. It participates in the graph — packaging even walks `Dashboard.widgets[*]` paths when remapping relation composites — but it is never written to `resources/`. If a dashboard travels at all, it travels as an entry reference.

Identity types are excluded for a different reason: the `USER`, `GROUP`, `ORGUNIT`, and `WORKING_HOURS` core entities (`CORE_ENTITY_IDS`) exist independently in every environment, so references to their entries are carried in `entry-references.json` and remapped at import rather than packaged as resources.

Which *instances* of the seventeen types are dropped even when selected — archived entities and their subtrees, user-scoped resources, implicit resources — is export-pipeline behaviour and is documented with the export workflow.

Safety limits at import

`extractPackageZip` enforces hard limits before reading any content, in this order:

Guard	Value	On violation
Maximum ZIP entries	10 000 (`MAX_ZIP_ENTRIES`)	Extraction fails.
Path traversal — `..` anywhere, a leading `/`, or any backslash in an entry name	— (`isSafeEntryName`)	Extraction fails.
Maximum decompressed size, summed across entries	200 MB (`MAX_DECOMPRESSED_SIZE`)	Extraction fails.
Path components under `resources/`	must match `^[a-zA-Z0-9_.-]+$` (`SAFE_PATH_COMPONENT`)	The entry is **silently skipped**, not rejected.
`resources/` entries not exactly three path segments deep	—	Silently skipped.

The last two rows deserve emphasis, because they behave differently from the rest. A resource file whose type directory or file name contains a space, a colon, or a non-Latin character does not fail the upload — it is quietly ignored during extraction, and the failure surfaces later and less legibly, as a resource the import cannot find (and a cascade-skip of everything that hard-depends on it). This is the most common failure mode for a generator that derives filenames from human-visible names instead of from `nameMap`, which only ever contains safe slugs.

The strict validator

After the limits pass, `validatePackageStrict` runs at the extraction boundary so everything downstream can assume a well-formed archive. It checks three things — required entries, manifest shape, and the top-level allow-list — and reports failures as a typed `PackageFormatError` whose `detail.category` is one of four values:

Category	Raised when
`MISSING_ENTRY`	`manifest.json` or `index.json` is absent.

Category	Raised when
`INVALID_JSON`	`manifest.json` exists but does not parse as JSON.
`SCHEMA_VIOLATION`	The manifest parses but fails a shape check: it is not an object, `schemaVersion` or `createdAt` is not a string, `resourceCount` is not a number, or `resourceTypes` is not an array. The failing path is reported in `detail.schemaPath`.
`UNKNOWN_ENTRY`	Any top-level entry outside the allow-list — the eleven named JSON files, `README.md`, and the `resources/`, `files/` and `readme/` prefixes. A stray file in the archive root is treated as tampering or format drift and fails fast. This is also what a pre-`1.2` service *would* say about `field-mapping-hints.json`, which is exactly why a package carrying the sidecar declares `1.2` and is turned away with the more legible `INCOMPATIBLE_SCHEMA` instead.

The error carries `detail.entryName` / `detail.schemaPath` so a producer can handle failures programmatically instead of pattern-matching messages. Expect to meet all four categories while iterating on a generator — that is what they are for.

Note the asymmetry: `dependencies.json` is *not* on the required list of the strict validator, but extraction still fails without it, with a plain (untyped) error. In practice all three root JSON files are mandatory.

Producing or transforming a package

Three production paths exist, compared honestly — including what each costs — in Generating a Package. The short version: start from a real export whenever possible, because it is the only path that begins from an artifact guaranteed to satisfy everything on this page.

For per-environment parameterisation, you usually do not need to touch the ZIP at all. Within an import session, the payload-override endpoints (`GET` / `PATCH` / `DELETE` on `/v1/import/{sessionId}/resource/{sourceType}/{sourceId}/payload`) substitute a single resource's payload at execution time — the archive stays untouched, and the wire shapes (`GetResourcePayloadResponse`, `SaveResourcePayloadRequest`) are part of the shared packaging contracts. The session workflow around them is documented in The Import API.

Package Resource Schemas

> **Availability**
>
> **Since 1.8.**

The Package Format specifies the archive container. This page specifies what goes **inside** `resources/{Type}/{normalizedName}.json` — the payload shape per resource type, taken from the model classes that define them in the backend source. It is the piece a reader needs in order to transform or generate a package payload, and the piece that is otherwise only discoverable by reading an export. The packaging engine this contract belongs to shipped in the **1.8** line; on an older deployment these endpoints and the export UI are absent.

What this page covers — and what it does not

This is a **tiered** reference, and it does not cover all seventeen exportable types. The six Tier-1 types below are what a typical module package is made of; documenting all seventeen at once would mean documenting the platform's entire metadata model, and doing six precisely is more useful than doing seventeen shallowly.

Type	Status
`DataModule`	**Documented below**
`DataEntity`	**Documented below**
`DataView`	**Documented below**
`DataQuery`	**Documented below**
`DataReaction`	**Documented below** — top-level shape; the action tree at outline depth
`DataEntityAggregate`	**Documented below** — new this cycle (`Since 1.10`)
`DataRelation` , `DataCollection` , `GenericView` , `SystemButton` , `StatisticQuery` , `Report` , `Function` , `CounterWidget` , `Announcement`	**Not documented.** Export a package containing one and read the emitted JSON — the same procedure that verified this page. Two facts are verified: `Since dev` a `Function` 's `value.secrets[]` rows carry the secret `key` with a `null` `value` , and it has no `identity` and no `OSOS_API_TOKEN` row (Function secrets); and `Since dev` a `GenericView` (page) widget that points at a view outside the package is a mappable soft reference (Page widgets pointing outside the package).
`QuickFilterGroup` , `TagGroup` (both `Since 1.10`)	**Not documented.** Only their import rules are verified: `id` and `dataEntity` are never written, `normalizedName` is the match identity, an update changes only `name` and `queries` (and never a tag group's `color`). Their graph edges are in The dependency graph.

Do not infer a Tier-2 shape from the six documented here; the types differ more than their names suggest.

Invariants across every packaged resource

Six things hold for all of the types below, and knowing them first makes the per-type tables much shorter to read.

- **`id` is present and is the source-environment UUID.** Every model extends an `Identified` base (`services/data-service/src/models/identified.model.ts`) that defaults `id` to a UUID v4. The importer uses it to verify filename lookups and to key the id-remap table — a payload without it cannot be identity-checked.

- `specificationVersion` **is the per-resource format version**, distinct from the package's `schemaVersion`. Every Tier-1 model declares it required with a default of `1`. A resource whose version a parser does not support produces an `UNSUPPORTED_SPEC_VERSION` warning at validation, not a hard failure.
- `name` **is a localized object, never a bare string** — a `LocalizedValue` of the form `{"en": …, "ar": …}`. `description`, where present, is a `NullableLocalizedValue`. A generator emitting a plain string here produces a payload data-service will reject.
- `normalizedName` **is the stable machine identifier.** It is validated against `^[a-zA-Z_][a-zA-Z0-9_$]{2,127}$` (3–128 characters, `NormalizedNameValidation` in `packages/osos-shared/src/utils/normalize-name.ts`), and the platform's own slugifier lowercases and replaces every other character with `_`. It is unique-indexed on `DataModule` and `DataEntity`, but a plain validated string on `DataView`, `DataQuery`, and `DataReaction`. It is also **not** necessarily the resource's filename in the ZIP — that is `nameMap`'s job (see the package format).
- **Enums serialize as numbers, not names.** `type`, `state`, `visibility`, and their kin are numeric enums (`packages/osos-shared/src/models/enums.ts`): a packaged entity carries `"state": 1`, never `"state": "Active"`.
- **Server-managed keys are absent.** `creationInfo`, `updateInfo`, `stateHistory`, `updateIdentifier`, `version`, `createdAt`, `updatedAt`, `createdBy`, `updatedBy`, `spaceId`, `space`, `_id`, `_resolved`, and `__v` are deleted at export and stripped again by the import parsers (`SERVER_MANAGED_KEYS`, `services/packaging-service/src/resource-parsers/parser-utils.ts`). The per-type tables below still list the ones each model declares — marked *stripped* — so that a reader diffing a payload against the model source is not surprised by the gap.

One filtering rule bears repeating from the export pipeline because it shapes what can appear in a package at all: archived entities (with their nested subtrees), user-scoped resources (any resource whose `relatedTo.user` carries a `dataEntry`), implicit `DataEntity` without `implicitOptions.allowEdit`, and implicit `DataReaction` are never exported (`isNonExportableResource`, `services/packaging-service/src/libs/resource-utils.ts`).

DataModule

The smallest Tier-1 shape (`services/data-service/src/models/data-module/data-module.model.ts`) — a module is a named, iconed container whose membership is expressed by its children pointing at it.

Property	Type	Notes
`specificationVersion`	`1`	Required; literal `1`.
`normalizedName`	`string`	Required, unique-indexed, pattern-validated.
`name`	`LocalizedValue`	Required.
`description`	`NullableLocalizedValue`	Optional, nullable.
`icon`	`Icon`	Required.
`isImplicit`	`boolean`	Required.
`sharing`	`ApplicationSharing`	Required — the sharing level.
`creationInfo`, `updateInfo`	`ActionInfo`	Declared on the model; *stripped* from packages.

Note what is **absent**: there is no `entities` array. Membership runs the other way — each `DataEntity` carries a `module` property pointing at its module. This is why exporting a module `withChildren` consults the dependency graph's membership edges rather than reading anything off the module document itself.

DataEntity

The most consequential Tier-1 shape, and the largest: **40 declared properties** (`services/data-service/src/models/data-entity/data-entity.model.ts`).

Two of them are misspelled **in the product**, and the misspellings are load-bearing: the property `extentions` (the model carries a `TODO rename and fix typo` comment beside it) and the type `DataEntityFromSettings` — `From`, not `Form`, even though its source file is named `data-entity-form-settings.model.ts`. Emit them exactly as the product spells them; a correctly spelled `extensions` key is simply ignored.

Property	Type	Notes
`specificationVersion`	`number`	Required; default `1`.
`name`	`LocalizedValue`	Required.
`description`	`NullableLocalizedValue`	Optional.
`normalizedName`	`string`	Required, unique-indexed, pattern-validated.
`version`	`number`	Entity schema revision; *stripped* from packages.
`isImplicit`	`boolean`	Required.
`virtual`	`DataEntityVirtualOptions`	Optional — present on virtual entities and aggregators.
`implicitOptions`	`DataEntityIsImplicitOptions`	Optional. `allowEdit` decides whether an implicit entity is exportable at all.
`type`	`DataEntityTypeEnum`	Required, **numeric** (`Custom = 0`, `ADT = 1`, `Service = 2`, …).
`typeOptions`	`TypeOptions | VirtualProxyEntityOptions | AggregatorVirtualEntityOptions`	Optional; open-typed.
`icon`	`Icon`	Required.
`archiveInfo`	`ActionInfo`	Optional.
`state`	`DataEntityStateEnum`	Numeric: `Draft = 0`, `Active = 1`, `Archived = 2`. Archived entities are never packaged, so expect `0` or `1`.
`stateHistory`	`StateHistory[]`	*Stripped* from packages.
`creationInfo`, `updateInfo`	`ActionInfo`	*Stripped* from packages.
`internalEntities`	`InternalEntity[]`	Default `[]`. Nested entity definitions, each with its own `fields`.
`internalCollections`	`InternalCollection[]`	Default `[]`.
`fields`	`DataEntityField[]`	Default `[]` — in practice the bulk of every real payload. Each field carries a type discriminator and a type-specific options object; the catalogue is documented in Field Types [Build Apps › Choose a field type] and not reproduced here.
`validation`	`string | null`	Optional entity-level validation expression.
`tags`	`string[]`	Default `[]`.
`workflows`	`DataEntityWorkflow[]`	Default `[]`.
`meta`	`ServiceMeta | Record<string, any>`	Optional, open-ended.

Property	Type	Notes
`module`	`UUID`	Required — the membership edge to the owning `DataModule`.
`displayOption`	`DataEntityDisplayOptionEnum`	Optional numeric enum; default `KeyValue`.
`formSettings`	`DataEntityFromSettings`	Optional. The misspelled type name.
`deletedFields`	`DataEntityDeletedFields[]`	Optional.
`breakingFields`	`DataEntityBreakingFields[]`	Declared, but listed in the model's `hiddenProperties` — never serialized, so never packaged.
`extentions`	`DataEntityExtensions`	Optional. The misspelled property name.
`entryNames`	`DataEntityEntryNames`	Optional.
`discussions`	`DataEntityDiscussions`	Optional.
`templates`	`DataEntityExportTemplate[]`	Optional — export/print templates.
`treeStructure`	`DataEntityTreeStructureConfig`	Optional.
`indexes`	`DataEntityIndex[]`	Default `[]`.
`dataSources`	`DataEntityFieldSources[]`	Optional.
`storeRules`	`StoreRule[]`	Optional. Shared stores (`type === 2`) are additionally snapshotted into the package's `store-references.json`.
`sharing`	`ApplicationSharing`	Optional on this model (required on `DataModule` and `DataView`).
`updateIdentifier`	`string`	Optimistic-concurrency token; *stripped* from packages.
`displayTemplate`	`EntityDisplayTemplate | null`	Optional.
`scriptAttributes`	`object`	Optional, nullable, open-ended.

`Since 1.10` — a `VirtualProxy` entity's `typeOptions.sources[*].sourceViewId` is a new hard dependency edge to its bound `kind: Proxy` `DataView`; on the read side (not the packaged payload) the response-only `proxyCapabilities` enrichment joins `forbiddenOnWrite` alongside `typeDetails` / `moduleDetails`. See Proxy view governance [Build Apps › Virtual entities] in Virtual Entities.

DataView

46 declared properties — the widest Tier-1 shape. The class is named `DataViews` (plural) in `services/data-service/src/models/data-view/data-views.model.ts`, which is worth knowing because searching the backend for `class DataView` finds nothing. (A related misspelling lives nearby: `data-view-exceptions.ts` declares `DataViewExcpetion` — not part of the payload, but you will meet it reading the view code paths.)

Grouped by role:

Identity and placement — `specificationVersion` (required, default `1`), `name` (required), `normalizedName` (validated, not unique-indexed), `description`, `icon` (optional here, unlike entity/module), `dataEntity` (required — the entity the view reads), `module` (optional), `displayOrder` (default `0`), `domain` (`NullableReferenceProperty`).

Source — `dataSource` (`DataEntrySource`, required), `sourceOptions`, `sourceEntitiesMapping` (default `[]`), `relations` (`ViewRelation[]`, default `[]` — relation composites inside are remapped at import), and `fields` (`string[]`, **required**

with at least one item — the projected field set).

Access — `isPublic` (**deprecated** in the model's own schema), `anonymous` (`AnonymousViewOption | null`), `authenticated` (`AuthenticatedViewOption | null`), `isPublished` (default `false`), `isPinned` (default `false`), `sharing` (`ApplicationSharing`, required), `qrCode` (default `{enabled: false}`).

Configuration — the three required blocks that carry most of a real payload: `viewConfiguration` (`DisplayViewConfiguration`), `editConfiguration` (`EditDataViewConfiguration`), `entriesConfiguration` (`EntriesDataViewConfiguration`); plus the optional `dashboardConfiguration`, `sideBarConfig`, `displayConfig`, `actionButtonsConfig`, `systemButtons` (default `[]`), `systemButtonsOrder` (**deprecated** in favour of ordering within `systemButtons`), `screenOrder`, `defaultViewTemplate`, `redirectionRules` (default `[]`), `help`, and the open-ended `uiAttributes` / `scriptAttributes`.

Behaviour toggles — `showArchivedEntries` (default `false`), `showInvalidEntries`, `allowObsoleteFilter` (default `false`), `allowInvalidRequirementsFilter` (default `false`).

Aggregates enabled on the view `Since 1.10` — `viewConfiguration.aggregates` is a `string[]` of `DataEntityAggregate` ids: the aggregates this view displays. Import filters the array to non-empty strings, and an id that cannot be remapped — because the aggregate was not included in the package, or was deleted before the export — is **dropped rather than treated as an error**. The view simply ends up showing one aggregate fewer, which is the same outcome as deleting the aggregate directly. The type this array points at is documented below.

Derived and server-managed — `executionPlan` is explicitly deleted at export and re-derived by the target: a generated package that includes one is at best ignored. `isImplicit` (default `false`) and `implicitType` (numeric, default `NONE`) mark platform-generated views — and because data-service spawns implicit views *asynchronously* after an entity is created, the import pipeline has env-tunable settle/poll timings (`PKG_IMPLICIT_SETTLE_MS`, `PKG_IMPLICIT_POLL_ATTEMPTS`, `PKG_IMPLICIT_POLL_DELAY_MS` in `services/packaging-service/src/services/package-import.service.ts`) to wait for them. `creationInfo`, `updateInfo`, and `updateIdentifier` are declared but *stripped* from packages.

Client-owned keys and `packageLabels` `Since 1.10`

An installed view is locked by its package, except for a short list of keys the client may change and an update keeps (`CLIENT_KEYS` in `services/packaging-service/src/libs/update-carry-over.ts`): `isPublished`, `displayOrder`, `authenticated`, `anonymous` and `permissions` (the access-right rows the view read joins in). The content hash leaves them out, so a client changing one is not an edit of the package — and **a package must not rely on shipping them**: an update lays the client's values back over the incoming resource.

`packageLabels` (`{name?, description?, icon?}`) records what the package shipped for a label the client renamed on an installed view or page. It is stored on the document and excluded from the hash, but it is never accepted from an ordinary caller — data-service drops it from every create, and from an update unless the caller is exempt from the installed-resource guard — so **a generated package should not emit it**. How a client reads and changes those labels is on Data views [Build Apps › Data Views & Faces].

DataQuery

`services/data-service/src/models/data-query/data-query.model.ts`:

Property	Type	Notes
`specificationVersion`	`number`	Required; default `1`.

Property	Type	Notes
`name`	`LocalizedValue`	Required.
`normalizedName`	`string`	Pattern-validated; not required, not unique-indexed.
`description`	`NullableLocalizedValue`	Optional.
`dataEntity`	`UUID`	Required — the entity queried.
`dataSource`	`DataEntrySource`	Optional.
`visibility`	`VisibilityEnum`	Required, numeric: `Organization = 0`, `Private = 1`, `Public = 2`.
`criteria`	`FilterExpression`	Required — the recursive expression tree. See below.
`relatedTo`	`DataQueryRelatedTo`	Optional, nullable. A query whose `relatedTo.user` carries a `dataEntry` is user-scoped and therefore **never exported**.
`renderedCriteria`	`string`	Marked **deprecated** in the model schema. A derived rendering of `criteria`; if you edit one, keep the other consistent — or omit it.
`isImplicit`	`boolean`	Default `false`.
`useInQuickFilter`	`boolean`	Default `false` — whether the query is offered as a quick filter.
`creationInfo`, `updateInfo`	`ActionInfo`	*Stripped* from packages.
`tag`	`DataQueryTag | null`	Optional.
`updateIdentifier`	`string`	*Stripped* from packages.
`parameters`	`DataQueryParameter[]`	Optional.
`isParameterized`	`boolean`	Optional.

The single most useful fact here for a generator: **`criteria` is the very same `FilterExpression` class the HTTP API accepts as the `criteria` member of `filterCriteria`** — `data-service`'s criteria-filter schema imports it from `models/data-query/filter-expression.model.ts` (`customJsonSchema/filter-criteria-json-schema.ts`). The operator vocabulary is documented with the v2/v3 filter dialect, and a criteria tree can be validated against the live criteria-parser endpoint before being written into a package — the only pre-import validation available for any fragment of a resource payload.

DataReaction

The reaction model lives in a different service from its four siblings: `services/reaction-service/src/models/data-reaction/data-reaction.model.ts`.

Property	Type	Notes
`specificationVersion`	`number`	Required; default `1`.
`normalizedName`	`string`	Optional, pattern-validated.
`definingEntity`	`ReferenceProperty`	Required — the owning entity, as `{dataEntity, dataEntry}`. Drives the hard membership edge. At import the parser strips it from the write body (the parent travels in the URL path), but keep it in the payload: validation and the dependency graph read it.
`eventType`	`EventTypeEnum`	Required, numeric.

Property	Type	Notes
`executorType`	`CollectionValueModel<EventExecutorTypeEnum>`	Optional.
`query`	`ReferenceProperty | null`	Optional — a soft edge to a `DataQuery`.
`criteria`	`FilterExpression`	Optional — the same expression model as `DataQuery.criteria`.
`eventConfig`	`DataReactionEventConfig`	Optional, open-typed. Outline depth — see below.
`actions`	`DataReactionAction[]`	Required. Outline depth — see below.
`isImplicit`	`boolean`	Default `false`. Implicit reactions are **never exported**.
`disabled`	`boolean`	Default `false`.
`name`	`LocalizedValue`	Required.
`spaceScope`	`DataReactionSpaceScope`	Required.
`loopThreshold`	`number`	Optional; default `0` (execute the trigger loop once).
`relatedEntities`	`DataReactionRelatedEntities[]`	Optional, nullable.
`sourceEntity`	`DataReactionSourceEntity | null`	Optional. For relation-scoped reactions, `sourceEntity.sourceId` is a **relation composite** — `entityId:fieldId[:relatedEntityId]` — which packaging parses and remaps as a unit (`libs/relation-composite.ts`); it is a resource identifier, not an entry UUID.
`type`	`DataReactionTriggerTypeEnum`	Required, numeric; default `0`.
`updateIdentifier`	`string`	*Stripped* from packages.
`validations`	`DataReactionValidation | null`	Optional.
`creationInfo`, `updateInfo`	`ActionInfo`	*Stripped* from packages.

`actions` and `eventConfig` are deliberately left at outline depth: a reaction is an event-condition-action tree whose action vocabulary is broad and extensible, and a shallow table would mislead more than it helps. Use Reactions & Automation [Build Apps › Reactions & automation] for what the actions *do*, and a real export for their serialized shape. One import-time behaviour is worth knowing when generating: the reaction parser keeps `id` in the create body on purpose — the platform honours a client-supplied reaction id because self-referencing tokens inside action configs are pre-rewritten against the planned target id (`services/packaging-service/src/resource-parsers/data-reaction.parser.ts`).

DataEntityAggregate `Since 1.10`

The newest packageable type — the resource behind the **Aggregator** list on an entity: a named `(field, aggregationFunction)` pair the platform computes server-side over that entity's entries. Its model is `services/data-service/src/models/data-entity-aggregate/data-entity-aggregate.model.ts`, and it travels as `resources/DataEntityAggregate/{normalizedName}.json`. For what an aggregate is and how a view enables one, see Entity Aggregates & Entry Grouping [Build Apps › Totals & grouping].

Property	Type	Notes
`specificationVersion`	`number`	Required; default `1`.
`dataEntity`	`UUID`	Required — the owning entity. **Structurally immutable** (see below).
`normalizedName`	`string`	Required, pattern-validated. The cross-environment identity key.
`name`	`NullableLocalizedValue`	Optional — the backend deliberately permits an unnamed aggregate.
`field`	`string`	Required — the `fullNormalizedName` of the aggregated field.
`aggregationFunction`	`AggregationFunctionEnum`	Required, numeric.
`sensitiveDataAccessPolicy`	`SensitiveDataAccessPolicyEnum`	Optional; absent means the default (privileged) policy.
`id`	`UUID`	Server-generated at both ends, and **forbidden on write** (`forbiddenOnWrite = ['id']` in `resource-parsers/data-entity-aggregate.parser.ts`).

Note what is missing relative to every other type on this page: this model declares **no `creationInfo` / `updateInfo`** at all, so there is nothing for the server-managed strip to remove.

Identity across environments is `normalizedName`, not `id`. The importer keys an aggregate `DataEntityAggregate::{entityId}::{normalizedName}` — the owning entity is part of the key, because a `normalizedName` is only unique *within* its entity.

It is an entity child, and that is tracked separately from being entity-scoped. Entity-scoping decides how validation matches an incoming aggregate against the target: by `normalizedName` within the target entity, not space-wide. Being an entity *child* is what drives the "patch the existing child rather than create a new one" decision. Both matter: if an aggregate that already exists under the target entity took the create path, the uniqueness guard on `normalizedName` inside that entity would reject it and the resource would fail the import.

`dataEntity` is structurally immutable on an override — a payload override cannot repoint an aggregate at a different entity. `normalizedName` deliberately is **not** immutable: renaming an aggregate through an override is a supported action.

Its dependency-graph edges — a hard one from the entity, a deliberately soft one from each view that enabled it — are documented in The dependency graph.

Model truth versus packaged truth

The tables above are the **declared model properties**, read from the source classes named in each section, with the export-side stripping rules (`SERVER_MANAGED_KEYS`, `executionPlan`, collection `items`) verified in `package-export.service.ts` and `parser-utils.ts`. They have not been diffed field-by-field against every possible emitted archive, and model defaults tell you what data-service will fill in, not what every historical export contains. When a payload you are transforming disagrees with a table here, trust the payload for reading and the model for writing — and report the difference. For the nine Tier-2 types, export a real specimen; do not extrapolate.

The dependency graph

Every package carries a `dependencies.json` — a serialized array of **dependency edges** recording, for each packaged resource, what it depends on and where in its JSON that dependency lives. The edges are not documentation: they are the machine input that drives the two hardest parts of an import. The **remapping** pass reads each edge's `path` to know exactly which field to rewrite from a source-environment id to a target-environment id, and the **topological sort** reads the edge relation to decide creation order, so a resource is always written after everything it needs. A package whose resources lack matching edges still imports — but in arbitrary order, and with its cross-references left pointing at ids that do not exist in the target.

This page documents the edge format and how the import engine consumes it. The packaging overview covers the surrounding workflow and the live graph API served by `dependency-service`; the package format covers the archive that `dependencies.json` sits inside.

Where the edges come from

The graph is owned by `dependency-service`. It extracts edges from resource snapshots using a declarative schema per resource type, interpreted by a schema engine, plus custom extractors for the shapes a declarative rule cannot express, and rebuilds them event-driven as resources change. The node vocabulary is the `ResourceType` enum — eighteen values, defined. Note that this is the *graph* vocabulary, not the exportable set: `Dashboard` is a graph node but is never written to a package's `resources/` folder (see the package format).

Internally the service persists edges as flat MongoDB documents (`fromType` / `fromId` , `toType` / `toId`), which also carry a `resourceVersion` snapshot hash used for idempotent rebuilds. That flat shape never reaches a package. What `dependencies.json` contains is the **nested wire DTO** the service serves (`docToEdge`), captured at export time as the outbound closure of the exported selection: the exporter BFS-walks outgoing edges from the selected roots, and **only hard edges expand the walk** — soft edges are recorded but not followed.

Anatomy of an edge

The wire contract is the `DependencyEdge` interface. Two edges from a real view, one of each kind:

```
[
  {
    "from": { "type": "DataView", "id": "5f0c33…", "normalizedName": "open_tickets" },
    "to":   { "type": "DataEntity", "id": "a91d78…", "normalizedName": "ticket" },
    "kind": "MEMBERSHIP",
    "hard": true,
    "path": "dataEntity",
    "resolvedVia": "DIRECT_ID",
    "fromActive": true,
    "fromArchived": false
  },
  {
    "from": { "type": "DataView", "id": "5f0c33…", "normalizedName": "open_tickets" },
    "to":   { "type": "DataCollection", "id": "c22e41…", "normalizedName": "priorities" },
    "kind": "REFERENCE",
    "hard": false,
    "path": "dataSource.collection",
    "resolvedVia": "DIRECT_ID"
  }
]
```


Field	Type	Meaning
`from`	`ResourceRef`	The resource that **holds** the reference — the dependent.
`to`	`ResourceRef`	The resource being depended on.
`kind`	enum	`MEMBERSHIP` or `REFERENCE` — see below.
`hard`	boolean	Whether the target must exist for the source to be valid. Also blocks deletion of the target in the live graph.
`path`	string	JSON path in the **source resource's snapshot** where the reference lives, e.g. `"module"` or `"columns[0].entityId"`.
`resolvedVia`	enum	The strategy that discovered the edge — see below.
`fromActive` / `fromArchived`	boolean	Lifecycle flags of the source resource at extraction time.
`metadata`	object	Kind-specific extra context; optional.

The `from` and `to` references

Both ends are `ResourceRef` objects: `type` (a `ResourceType` value) and `id` are the identity; `normalizedName` and the localized `name` are carried for display and for name-based matching at validation. Three optional flags qualify the node — `isImplicit` (system-managed, not exportable), `isExternal` (belongs to a different module than the one queried), and `isService` (a service-type entity). A fourth, `withChildren`, is only meaningful in export *requests*, where it asks for the ref's `MEMBERSHIP` subtree to be expanded.

`kind` — membership or reference

`DependencyKind` has exactly two values (enforced as the enum on the persisted document too):

- **`MEMBERSHIP`** — the source *belongs to* the target: an Entity to its Module (`path: "module"`), a View to its Entity (`path: "dataEntity"`, hard). Membership is what `withChildren` traverses when an export expands a Module or Entity into its subtree.

- `REFERENCE` — a cross-reference from the source's payload to an independent resource: a view's `dataSource.collection`, a reaction's target query, a widget's entity id.

`hard` — must exist, or may be nulled

A **hard** edge is a validity requirement: the source cannot function without the target. Hard edges are the ones the export closure follows, the ones the import's cascade-skip logic honors (skip or fail a resource and every hard dependent is `cascade_skipped` with it), and the ones the hard-only sorting pass keeps. A **soft** edge is a recorded preference: if the target has no mapping in the target space, the import **nulls the value at the edge's `path`** rather than leaving a dangling source id — with deliberate exceptions such as `dataSource.query`, which is left intact so the data service can fail with a meaningful `QUERY_NOT_FOUND` instead of a type error.

`Since dev` one family of soft edges leaving the package is no longer nulled silently: a **page widget's** reference to a view or other resource outside the package is declared at publish and offered to the installer as an optional mapping; only when it is left unmapped is it nulled (`isMappablePageWidgetReference`; see Page widgets pointing outside the package).

Hardness is declared per rule in the extraction schemas, and it gets tuned when reality disagrees: a query-type view's `dataSource.query` edge is today declared **hard** precisely because shipping it soft produced packages that imported a view without its source query.

`path` — where the reference lives

`path` is a JSON path into the *source* resource's packaged snapshot, in the `lodash.get` / `set` dialect — `"module"`, `"dataSource.collection"`, `"columns[0].entityId"`. It is the single piece of information that makes precise remapping possible: without it the importer would have to guess which of a payload's string fields are ids. An edge without a `path` still participates in ordering and cascade-skip, but cannot drive a rewrite.

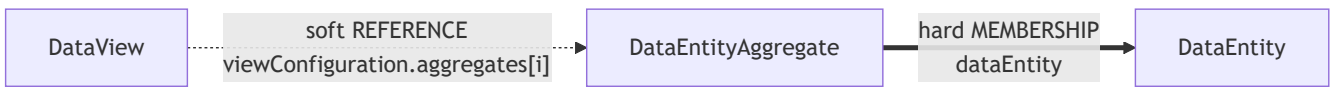
`resolvedVia` — how the edge was discovered

`DependencyResolution` records the extraction strategy. It is informational on import — remapping treats all strategies alike — but useful when auditing why an edge exists:

Value	Strategy
`DIRECT_ID`	The payload holds the target's UUID directly at `path`.
`NORMALIZED_NAME`	The target was resolved from a name at `path` rather than an id.
`REFERENCE_PROPERTY`	The value at `path` is a reference-property object (`{dataEntity, dataEntry}`); the edge targets the entity while the entry id is handled by the value-reference pipeline.
`PIPELINE_STAGE`	Declared in the enum, but no extractor currently emits it.

The aggregate node and its two edges `Since 1.10`

`DataEntityAggregate` — the resource behind an entity's **Aggregator** list — is the newest node type, and the two edges added with it are the clearest worked example of the hard/soft distinction above, because the two directions were chosen for opposite reasons.



The entity edge is hard. An aggregate belongs to exactly one entity and means nothing without it, so `DataEntityAggregate.dataEntity → DataEntity` is a `MEMBERSHIP` edge marked hard. The consequences are the ones hardness always carries: exporting an entity with its children pulls its aggregates along, and the entity cannot be deleted out from under them.

The view edge is deliberately soft. `DataView.viewConfiguration.aggregates[] → DataEntityAggregate` is a `REFERENCE` edge, and it is soft on purpose: deleting an aggregate must silently drop it from every view that had it enabled, and a hard edge would block that delete instead — the opposite of the required behaviour. The edge exists purely so that an import can remap the enabled ids onto the target space's own aggregates. When one of those ids has no mapping, the usual soft-edge treatment applies and the id is dropped from the array, so the view shows one aggregate fewer rather than failing.

The array is what makes this edge structurally new: its elements *are* the target ids, a shape neither of the existing rule primitives could express — a direct-id rule returns early on an array, and array-iteration walks objects. A new `DirectIdArray` primitive covers it, emitting one edge per element with the usual indexed path convention (`viewConfiguration.aggregates[0]`).

Operator consequence, and it is easy to miss: a graph built before this change will not show aggregates at all until it is rebuilt. The graph is maintained incrementally from resource lifecycle events, and adding a rule does not retroactively rewrite edges that were already computed. Until a rebuild runs, an entity's dependency view simply omits its aggregates and a view shows no dependency on the aggregates it displays — which looks like the feature is absent rather than stale.

Further edges added in 1.10 `Since 1.10`

The 1.10 line taught the graph about two new node types and several reaction actions. Each row changes what an export pulls in and what an import remaps; and, exactly as for the aggregate edges above, **a graph built before the change shows none of them until it is rebuilt**.

Edge (source path → target)	Kind	Hard?	What it means
`QuickFilterGroup.dataEntity → DataEntity`	Membership	Hard	A quick-filter group belongs to its entity and travels with it.
`TagGroup.dataEntity → DataEntity`	Membership	Hard	Likewise for a tag group.
`DataQuery.quickFilterGroup → QuickFilterGroup`	Reference	Soft	Deleting a group just ungroups its queries.
`DataQuery.tag.tagGroup → TagGroup`	Reference	Soft	Deleting a group leaves the tag ungrouped.
`CounterWidget.value.moduleId → DataModule`	Membership	Hard	A counter widget now belongs to its module.
Reaction *Execute Function*: `options.functionId → Function`	Reference	Hard	The function the action runs.
Reaction *Execute custom action*: `options.customActionId → DataReaction`	Reference	Hard	The reaction it runs; reaction-service refuses a create whose id does not resolve.
Same action: `options.entryToRun.relationId → DataRelation` , `options.entryToRun.targetEntityId → DataEntity`	Reference	Soft	Both optional.
Same action: `options.parameters[].value` → entries	Reference	Soft	Entry values, under the same rules as the create/update actions.

Edge (source path → target)	Kind	Hard?	What it means
Reaction *Delete archived entry*: `options.relation.relationId`, `options.querySource`	Reference	Soft	The same rules as *Archive entry*.
Reaction *Set permission* / *Remove permission*: `options.resource.entitlementId` → module, entity, view, page or report	Reference	Hard	Typed from the entitlement type; there is no edge for a `System` entitlement.
Same actions, **Fixed** principal: `options.principal.source.value` → the USER or GROUP entity	Reference	Soft	Remapped through the value-reference pipeline, as for *User activation toggle*.
Reaction *Send SMS* and *Send WhatsApp*, **Fixed** user or group recipient: `options.recipients[].source.value` → the USER or GROUP entity	Reference	Soft	One rule serves both actions. Before it covered *Send WhatsApp*, a WhatsApp reaction's fixed recipients left no edge and the reaction was missing from exported notifications. Remapped through the value-reference pipeline.

Two smaller corrections ride along. Field **default-value** and **currency** edges now record the full field path, so remapping rewrites exactly the field that holds the reference. And the old rule for an *Update data entity* reaction action is gone, because the create/update data-entity actions never ran and were removed from the product.

How edges drive remapping

At execution, the importer groups the package's edges by `from.id`, giving each resource its outgoing edge list, and builds an id map that pre-assigns a target id to every resource it will create — which is what allows references to be rewritten *before* the referenced resource physically exists. Then `RemappingService.remap` walks each resource's outgoing edges:

1. Deep-clone the payload (an override, if one was saved, has already replaced the packaged original at this point — overrides are applied *before* remapping).
2. For each edge with a `path`: read the value at `path`; look up `to.id` in the id map; if the current value **exactly equals** the source id, write the mapped id back at `path`. A value that does not match is logged and skipped — the exact-match guard is what prevents corrupting a field that merely resembles an id.
3. Reference-property objects get special handling: when the value at `path` is a `{dataEntity, …}` object, only its `dataEntity` member is remapped; the embedded entry UUID belongs to the value-reference pipeline.
4. A soft edge whose target has no mapping has its `path` **set to** `null` (minus the exceptions above); the resource's own `id` is remapped last.

Two scoped fallbacks then catch what the path-based pass missed: an exact-match rewrite over the ids this resource's outgoing edges name (covering edges whose recorded array index has drifted), and the unmapped-dependency replacements the user configured. Both replace whole-string exact matches only — never substrings.

The consequences for anyone producing or transforming a package: **an id you change in a payload without a matching edge stays unrewritten**, and **an edge whose `path` no longer points at the id it names is skipped**, silently, by design.

How edges drive import order

`index.json`'s `orderedResources` is written topologically sorted by the exporter — but the importer does not trust it. It re-sorts on every execution, using the package's edges, because older packages were exported unsorted (step 4e).

The sort itself is Kahn's algorithm, run in up to three passes:

1. **All edges, hard and soft.** The ideal order: even optional targets are written before their referrers, so a soft reference resolves rather than getting nulled.
2. **Hard-only fallback.** If pass 1 detects a cycle (two resources soft-referencing each other), soft edges are dropped and the sort re-runs — breaking a soft cycle is safe precisely because soft references are nullable at remap time.
3. **SCC collapse.** A residual *hard* cycle is collapsed into a strongly-connected component (Tarjan) so the cycle as a whole is still ordered before its dependents. Hard entity-to-entity cycles are then broken at execution by stubbing one member and patching the cross-cycle references in once every member exists.

The same edge relation, reversed, powers **cascade skip**: before any writes, the importer builds a reverse-dependency index from the edges and marks every hard dependent of a skipped or failed resource as `cascade_skipped`, transitively (`buildReverseDeps` / `applyCascadeSkip`).

Rules for a package producer

Everything above compresses into four rules for anyone generating or transforming `dependencies.json` — the practical paths for doing so are on Generating a package:

1. **Every internal cross-reference needs an edge**, with `from`, `to`, and the exact `path` of the referencing field. No edge, no rewrite, no ordering guarantee.
2. **Mark hardness truthfully.** A hard edge to something absent from both package and target does not fail loudly — it cascade-skips the dependent subtree. A soft edge to something absent gets nulled.
3. **Keep `path` synchronized with the payload.** If a transformation moves an array element, the recorded index in `path` must move with it; the exact-match guard will otherwise skip the rewrite without failing the import.
4. **Do not rely on `orderedResources` alone.** The importer re-derives order from the edges; an internally consistent edge set matters more than the order you wrote the index in.

Generating a package

Let us start with the honest answer to the question that usually brings a reader here: **a single standalone JSON file containing every definition a working system needs is not something OSOS produces or accepts.** A package is a multi-file ZIP whose parts — manifest, index, dependency edges, per-resource payloads — must agree with one another, and nothing in the platform emits or consumes a one-file bundle of everything (`services/packaging-service/src/libs/zip-utils.ts` defines the contract; the package format documents it). The nearest thing to a single JSON is data-service's separate module/entity transfer, which genuinely is one file but carries modules and entities only — see Module JSON transfer. The packaging engine this contract belongs to shipped in the **1.8** line (`Since 1.8`); on an older deployment these endpoints and the export UI are absent.

What exists instead are three practical paths to producing, transforming, or parameterising a package. This page documents them in the order you should consider them.

Choosing a path

Path	Supported?	Reach for it when
1. Export → transform → re-import	Yes — the mainline	You want a modified or templated variant of something a real environment already contains.
2. Payload overrides on an import session	Yes — the sanctioned programmatic hook	You want to inject environment-specific values into a package at import time, without forking the ZIP.
3. Hand-authoring from nothing	Possible, unsupported	You are generating resources no environment has ever contained — and you accept the costs below.

Paths 1 and 2 compose: the robust pipeline for repeated environment promotion is a reference package produced by export, parameterised per target through the override API.

Path 1 — export, transform, re-import

The only path that starts from a guaranteed-valid artifact. Export a reference package from a source environment, unzip it, transform the JSON, rebuild the archive, and import the result. You inherit a correct `manifest.json`, `index.json` and `dependencies.json`, and your only obligation is keeping them consistent with whatever you changed.

The round trip

The endpoints below are the packaging service's export and import routes; conventions — the `{success, result}` envelope, tokens, error shapes — are covered by Calling the HTTP API and are not repeated here.

Selecting resources from the admin UI, `Since 1.8`. Before calling the export endpoint below, an admin usually picks the modules/entities/pages to include from a checkbox tree in the Administration packaging screen. A bug fix landed this cycle: unchecking a container's only remaining child used to also uncheck the container itself, so "export this module or entity alone, with no children" was unreachable — the tree now tracks a container's own checked state separately from its derived (any-child-checked) state, so that combination works correctly. Ticking/unticking a container still cascades to its whole subtree, as before.

```
export BASE="https://acme.example.com"
export TOKEN="<your API token>"

# 1 — start an export job
curl -s -X POST "$BASE/api/packaging/v1/export" \
  -H "x-app-token: $TOKEN" -H "content-type: application/json" \
  -d '{"resources":[{"type":"DataModule","id":"<module-id>","withChildren":true}],
       "name":{"en":"crm-reference"}}'
# → {"success":true,"result":{"jobId":"export-<exportId>","exportId":"<exportId>"}}
#   (Since 1.10 the export is also kept as an export record: GET /v1/exports/<exportId>)

# 2 — poll until the job completes (note: the job's own result is nested
#     inside the envelope's result)
curl -s "$BASE/api/packaging/v1/export/export-<exportId>/status" -H "x-app-token: $TOKEN"
# → {"success":true,"result":{"status":"completed",
#      "result":{"fileId":"<fileId>","name":"crm-reference.zip", …}}}

# 3 — download the ZIP. The export stores it in storage-service as a
#     temporary file; the response is the raw file, not enveloped JSON.
curl -s "$BASE/api/storage/v2/files/download/<fileId>?temp=true" \
  -H "x-app-token: $TOKEN" -o package.zip

# 4 — unzip, transform the JSON under resources/, rezip

# 5 — upload the transformed archive, again as a temporary file
curl -s -X POST "$BASE/api/storage/v2/files?temp=true" \
  -H "x-app-token: $TOKEN" -F "file=@package-transformed.zip"
# → {"success":true,"result":[{"id":"<newFileId>", …}]}

# 6 — open an import session against it, then configure and execute
curl -s -X POST "$BASE/api/packaging/v1/import/validate" \
  -H "x-app-token: $TOKEN" -H "content-type: application/json" \
  -d '{"fileId":"<newFileId>","fileName":"package-transformed.zip"}'
# → {"success":true,"result":{"sessionId":"…","validationReport":{…},
#      "updateIdentifier":"…"}}
```

From the `sessionId` onward this is the standard session flow — configure mappings, execute, poll — documented on the import API page and in the packaging overview. Export is a background job (`POST /v1/export` enqueues it and returns a job id — the general pattern is covered in Long-running work); the `temp=true` flag on both storage calls matches how the platform itself stores package files (and every packaging download passes it, `services/packaging-service/src/services/package-export.service.ts` uploads with the temp flag).

`Since 1.10` the export `name` is a localized object and is **required** (`400 EXPORT_NAME_REQUIRED` without it); the ZIP is named from its English value, else its Arabic one. The caller needs Developer on every exported root, and resources that belong to an installed package cannot be exported at all (`403 EXPORT_INSTALLED_RESOURCE`) — see Export access.

Keeping the parts consistent

A transformation is safe exactly as long as the archive's members still agree. The rules, each traceable to import-side code:

- **Filenames come from `nameMap`, not from names.** `index.json`'s `nameMap` (`id → normalizedName`) is the authoritative key for `resources/{Type}/{name}.json`; export suffixes colliding slugs with the id, and the importer id-verifies name-based hits. Rename a file, or a resource's `normalizedName`, without updating `nameMap` and the resource is unreachable.

- **Path components are strict.** Every component must match `^[a-zA-Z0-9_.-]+$` — deriving a filename from a user-visible name with spaces or non-Latin characters is the most common way a rebuilt archive gets rejected at extraction.
- **Changed references need matching edges.** If your transformation adds or moves an id reference, `dependencies.json` must carry an edge whose `path` points at it, or the id will never be remapped — the dependency graph explains exactly how the edges drive rewriting and ordering.
- **Structural fields are load-bearing.** A resource's parent pointers (`module`, `dataEntity`, …) are what remapping and scope resolution key on; moving a resource between parents is a re-authoring exercise, not a field edit.
- **The manifest must still count.** `resourceCount` and `resourceTypes` are shape-checked at extraction (`assertManifestShape`), and the `schemaVersion` must be a supported one — `1.0` to `1.4` on 1.10, only `1.0` or `1.1` on an older target. Keep the version your export was stamped with unless you add or remove the part that version describes; see Schema versioning.

Path 2 — the payload override API

The sanctioned way to change package *content* programmatically — and for most readers the real answer to "can I generate this?". Within a live import session, between validate and execute, every packaged resource's payload can be read and replaced through one route:

```
GET    /api/packaging/v1/import/{sessionId}/resource/{sourceType}/{sourceId}/payload
PATCH  /api/packaging/v1/import/{sessionId}/resource/{sourceType}/{sourceId}/payload
DELETE /api/packaging/v1/import/{sessionId}/resource/{sourceType}/{sourceId}/payload
```

`sourceType` is the resource's `ResourceType` name and `sourceId` its id *in the package* — source-environment identifiers, as listed in the session's validation report. The ZIP itself is never touched: an override lives on the session (`ResourceOverride`) and is applied **in place of the packaged payload, before remapping**, at execution. That ordering matters — you write an override in source-environment ids, exactly as if it had been packaged, and the engine rewrites it like everything else. This is how a reference package is parameterised per environment — repointing an endpoint, adjusting a threshold, injecting tenant-specific configuration — while the artifact stays a single unforked file.

Read a resource's payload

`GET` returns the payload the execution will actually use — the override if one exists, otherwise the packaged original extracted from the ZIP. The response (`GetResourcePayloadResponse`):

Field	Meaning
`sourceType`, `sourceId`, `normalizedName`	The resource's identity in the package.
`payload`	What will be imported: the override when `isOverride` is `true`, else the packaged original.
`isOverride`	Whether an override is currently active.
`packagedPayload`	The packaged original, present **only** while an override is active — so a client can diff and preview a reset without a second call.
`editedBy`, `editedAt`	Provenance of the active override.

A resource present in the session but missing from the archive returns `404`; reads are refused with `409` while the import is running.

Save an override

`PATCH` takes `{payload, updateIdentifier}` (`SaveResourcePayloadRequest`) and persists the payload on the session after validation. Saving an override for a resource that previously **failed** (or was cascade-skipped) also resets its status to pending, so the session's *retry* picks it up — the override API doubles as the repair loop for a failed import. Every write is audit-logged with the session, resource, author and payload size.

The validator rejects:

Condition	Response
Payload not a plain JSON object, or carrying prototype-polluting keys	`400`
Serialized payload over 2 MB (`MAX_OVERRIDE_PAYLOAD_BYTES`)	`413`
A structural field changed from the packaged snapshot	`400`

What an override cannot change

Structural fields — the parent pointers that remapping and dependency resolution key on — are fingerprinted at validation (`structuralSnapshot`) and locked (`STRUCTURAL_IMMUTABLE`, same file). Changing them would silently desynchronise the payload from the package's dependency edges:

Resource type	Immutable fields
`DataEntity`	`module`, `dataModule`
`DataView`	`module`, `dataModule`, `dataEntity`, `definingEntity`
`DataQuery`, `DataReaction`, `DataCollection`, `StatisticQuery`	`dataEntity`, `definingEntity`
`DataRelation`	`dataModule`, `module`
`Announcement`	`entityId`

An override is a *content* hook, not a *structure* hook. To change where a resource lives, use the session's mapping configuration; to change what the package contains, use path 1.

Clear an override

`DELETE` (with the current `updateIdentifier` as a query parameter) removes the override so the next execution falls back to the packaged payload. Clearing an override that does not exist is a deliberate no-op `200`, so an idempotent "reset" in client code needs no existence check first.

Concurrency and session state

All three verbs share the session's guards: the caller's space must own the session, and writes are refused with `409` while the session is `importing` or `validating`. Mutations are optimistically locked — each write must echo the `updateIdentifier` the client last saw and each response returns a rotated one to adopt; a mismatch is a `409` whose `error.code` is `STALE_UPDATE_IDENTIFIER`, distinguishable from the status-gate `409`s. A generator driving this API is a loop: read the token, write, adopt the returned token, repeat.

Path 3 — hand-authoring from nothing

Building a package without exporting one is possible — the import trusts any archive that passes validation, wherever it came from — but it is **unsupported**, and the platform gives you no authoring tools for it. You are reimplementing the exporter by hand.

What you must produce

Every part of the package format, mutually consistent:

- A `manifest.json` whose shape passes `assertManifestShape` — `schemaVersion` (a supported `'1.0'` / `'1.1'`), `createdAt`, `resourceCount`, `resourceTypes` — plus the source-identity fields if you want stored mapping profiles to apply.
- An `index.json` whose `orderedResources` is a genuine topological order over the hard edges, and whose `nameMap` names every file actually present under `resources/`.
- A `dependencies.json` whose edges carry correct JSON `path` values for every internal reference — otherwise nothing is remapped and nothing is ordered (the dependency graph).
- One payload per resource under `resources/{Type}/{name}.json`, each matching its type's schema — Resource schemas documents Tier 1; for the rest, the reliable procedure is to export a real specimen and copy its shape.
- Nothing else: the archive's top level is allow-listed, and an unknown entry fails extraction (`UNKNOWN_ENTRY`). Stay under the safety limits — 200 MB decompressed, 10 000 entries, safe path components, no traversal.

Your only safety net

The strict validator at the extract boundary checks the *container*: required entries present, valid JSON, manifest shape, allow-listed top level. Its four typed failures — `MISSING_ENTRY`, `INVALID_JSON`, `SCHEMA_VIOLATION`, `UNKNOWN_ENTRY` — are the only errors that will point at your archive directly, and a producer iterating on a generator should expect to meet all four.

What it does **not** check is the mutual consistency the exporter normally guarantees. An `index.json` entry whose file is missing does not fail the import — the resource is *skipped* and its hard dependents are cascade-skipped with it. An edge whose `path` is wrong does not error — the reference is silently left unrewritten. A payload violating its resource schema surfaces as a per-resource failure from data-service at execution, in the target environment's terms rather than your generator's. Which is the argument, one last time, for paths 1 and 2: start from an artifact the platform built, and parameterise it through the API built for exactly that.

The import API

Packaging & distribution explains what the import engine does at each stage, and Export, import and install apps [Build Apps] walks the wizard as a builder sees it. This page documents the same surface as a **wire contract**, for a client driving an import programmatically: every session endpoint with its request and response bodies, the session state machine exactly as the service enforces it, and the concurrency rules that keep two writers from corrupting one session.

Everything here is served by `packaging-service` behind the gateway prefix `/api/packaging`; paths below are written from `/v1/…`. The shared conventions — routing, authentication, pagination, and the `{success, result}` response envelope — are owned by Calling the HTTP API and are not repeated here. Every response body on this page is the `result` member of that envelope, without exception. Two requirements apply to every call:

- an authenticated request — every packaging controller is `@authenticate`-guarded with the Auth2 strategy — made by a **space administrator** (`Since 1.10`: every import, execution, override and mapping-profile route requires a system-level Admin grant, or a service call; anything less is a `403`); and
- a target space. You do not send it: the gateway derives the space from the request's host and sets `x-space-id` itself, replacing any value a client sends (Tenancy and identity). Address the tenant host of the space you are importing into. A request that reaches `packaging-service` without the header — one that bypassed the gateway — is a `400`, and addressing a session that belongs to a different space is a `403`.

Sessions are not permanent. A TTL index deletes any session not updated for **seven days**; a client that parks a half-configured session longer than that will find it gone.

The state machine

A session's `status` is one of five values — `validating`, `validated`, `importing`, `completed`, `failed` (`IMPORT_SESSION_STATUSES`). There is no `uploaded` state: a session enters the lifecycle at `validating` while Phase-1 validation runs, and is normally first visible to a client as `validated`.

The transitions are not folklore. The service defines an explicit transition table, and code paths that write a status are expected to check it (`assertValidTransition`). The complete set:

Transition	From	To	Fired by
`VALIDATE`	`validating`	`validated`	Phase-1 validation completing (`POST /v1/import/validate`)
`START_IMPORT`	`validated`	`importing`	`POST /…/execute` claiming the session
`COMPLETE`	`importing`	`completed`	the import worker finishing all mappings — with or without per-resource errors
`FAIL`	`validating`, `validated`, `importing`	`failed`	an unrecoverable error during validation or execution
`RETRY`	`failed`, `completed`	`importing`	`POST /…/retry`, or `execute` called on a terminal session
`RECOVER_STUCK`	`importing`	`failed`	worker recovery, when a session has been stuck in `importing` past the recovery window (15 minutes, `IMPORT_STUCK_THRESHOLD_MS`)
`CANCEL`	`validated`, `importing`	`failed`	`POST /…/cancel`

Three consequences worth internalising before writing a client:

- `completed` **does not mean fully succeeded.** `COMPLETE` fires when every mapping has been processed, possibly with per-resource errors; the truth is in `importResult` and the per-mapping `status` values. A `completed` session remains retryable.
- `failed` **is also where cancellation lands.** There is no separate cancelled state; a cancelled session is `failed` with an `errorMessage` that begins `Cancelled:`.
- `failed` **is not always terminal-bad.** Validation that produces a resource report but cannot proceed still creates a `validated` session (the report carries the blockers); only a validation that cannot produce a report at all — an unreadable package, an incompatible schema version — lands the new session directly in `failed` (`createSession`). Both `failed` and `completed` accept a retry.

Creating a session

`POST /v1/import/validate`

The package ZIP is **not** posted to this endpoint. Upload it to the storage service first, then pass the resulting file id; the packaging service downloads the ZIP itself.

Body field	Required	Meaning
`fileId`	yes — its absence is a `400`	Storage-service id of the uploaded ZIP
`fileName`	no	Original file name, echoed in session listings
`importStrategy`	no	Initial bulk strategy: `MERGE_NEW_ONLY`, `OVERWRITE_ALL` or `MANUAL` (default)

The response is `{sessionId, validationReport, updateIdentifier}` (`ImportValidateRequest` / `ImportValidateResponse`). Keep the `updateIdentifier` — every later write must echo it (see the concurrency section). The shape and meaning of the validation report are covered on Packaging & Distribution; how the manifest's `schemaVersion` gates this call is covered in Schema versioning. One member of the report is documented on this page instead, because acting on it is a wire operation of its own: `mappedFields` `Since 1.10`.

Validation always creates a **new** session; there is no in-place re-validation of an existing one. Re-uploading a corrected package means a new session and a fresh configuration.

`Since 1.10` validation also reports, as blocking errors in the report:

- `TARGET_ARCHIVED` — a package resource would match an **archived** target resource; it is not treated as Existing, so an archived record is never overwritten;
- `TARGET_INSTALLED` — a resource the import would write is owned by a package **installation**; fork the installation first;
- the `INSTALL_*` codes, when the ZIP is a published package version and the session is therefore an install — see Install mode; `INSTALL_NAME_COLLISION` now lists **every** collision (`resourceType`, `name`, `displayName`, `scope` of `space` or `environment`, `targetId?`, `targetModuleId?`), not only the first;
- `MAPPED_FIELD_SOURCE_UNDECIDED` — a mapped field needs the client's decision before the run (see Mapped-field decisions).

A published-version file exported before package ids existed is refused outright with `422 PACKAGE_FORMAT_OUTDATED`.

Reading sessions

`GET /v1/import/sessions`

Lists the current space's sessions as summaries — `{sessions: [{id, status, sourceEnvironment?, packageDescription?, packageFileName?, resourceCount?, createdAt, updatedAt, createdBy?, updatedBy?, updateIdentifier?}]}` — newest activity first (`updatedAt` descending). Paging is by `skip` and `limit` query parameters; `limit` defaults to 50 and is clamped to 1–100, so a larger value is quietly reduced.

`GET /v1/import/{sessionId}`

Returns the full session document: manifest, validation report, resource mappings, value-reference mappings (with their packaged `entryData` intact — the payload the wizard's auto-create editor runs on), unmapped-dependency mappings, overrides and results. For a session in `validated` or `failed`, the service re-checks unresolved auto-create references on each read and stamps the fresh errors into the returned report — the response reflects the current target space, not a snapshot.

`GET /v1/import/{sessionId}/status`

The lightweight polling endpoint for an import in flight: `{status, importResult?, errorMessage?, resourceMappings?}` (`ImportStatusResponse`). `resourceMappings` is included only once the session has left the configuration phase — while `importing` and in the terminal states — so a poller can render per-resource progress without fetching the whole session.

Configuring a session

Configuration writes are accepted while the session is `validated` or `failed`; any other status is a `409`. The one exception is the bulk strategy, which requires `validated` exactly. All of them are optimistic-locked — see the concurrency section.

`PATCH /v1/import/{sessionId}/config`

The unified auto-save endpoint (`SaveImportConfigRequest`). Every section is optional; the client sends only what changed, and validation is deliberately relaxed — hard constraints are enforced at execute time, not here:

```
{
  "mappings": [
    {
      "sourceType": "DataEntity",
      "sourceId": "0b6d…",
      "normalizedName": "invoice",
      "action": "OVERWRITE",
      "targetId": "9f2a…"
    }
  ],
  "valueReferences": [
    {"entryId": "77c1…", "action": "REMAP", "remapTargetId": "a41b…"}
  ],
  "unmappedDependencies": [
    {"sourceId": "d305…", "targetId": "e77f…"}
  ],
  "mappedFieldDecisions": [
    {"entityId": "0b6d…", "fieldId": "f1…", "targetParentId": "9f2a…",
     "decision": {"kind": "dontCopy"}}
  ],
  "updateIdentifier": "58c0…"
}
```

Server-side behaviour a client should know about:

- `status`, `errorMessage` and `isHardDep` on a submitted mapping are **stripped** — they are server-controlled; each saved mapping is re-stamped `pending`.
- A mapping submitted without `targetId` for a resource that validation classified as existing gets the validation-resolved `targetId` back automatically, so a save cannot silently drop the source→target link.
- In `unmappedDependencies`, `targetId: null` **clears** a previously chosen target.
- `Since dev` a row of the report's `unmappedDependencies` may carry `hard: false`: only page widgets use it, and left unmapped those widgets are cleared (no `hard` field means hard). See Page widgets pointing outside the package.
- Value-reference choices are merged item-by-item; an `entryId` not already on the session is ignored, and remap targets are verified to exist — and to be live — before the save lands.
- A save that would clear a required workflow-step owner on a resource the import will write is rejected with `400` — see required workflow-step owners.
- `Since 1.10` `mappedFieldDecisions` is the **full** list of mapped-field decisions and replaces the stored one; a malformed item is `400 MAPPED_FIELD_DECISION_INVALID`. A save that changes mappings or unmapped-dependency targets recomputes the mapped-field statuses too, because either can move a field's target parent (see Mapped-field decisions).
- An empty body is a valid no-op: nothing is written and the token is not rotated, but the token is still checked, so a stale client learns it is stale.

The response is `{updatedAt, updateIdentifier, dependencyImpact?, valueReferenceErrors?, mappingSummary?, mappedFields?, canProceed?}` (`SaveImportConfigResponse`). `dependencyImpact` accompanies mapping changes (which resources lost hard dependencies, updated cascade-skip counts); `valueReferenceErrors` are advance warnings from the auto-create nested-reference guard — the save succeeded, but execution will gate on them; `mappingSummary` is the resolved/remaining principal count after a value-reference save; `mappedFields` and `canProceed` (`Since 1.10`) come back whenever the mapped-field statuses were recomputed.

`PATCH /v1/import/{sessionId}/strategy`

Body `{strategy, updateIdentifier}` (`ApplyStrategyRequest`). Recomputes every mapping's action from the chosen strategy — `MERGE_NEW_ONLY` creates new and skips existing, `OVERWRITE_ALL` creates new and overwrites existing, `MANUAL` leaves per-resource decisions to the client — while **preserving user-chosen module remaps** (a `DataModule` mapped by hand to a different target survives the sweep). Returns the full updated session. Only accepted in `validated`.

`PATCH /v1/import/{sessionId}/suggestions/accept-exact`

Body `{updateIdentifier}` (`AcceptExactSuggestionsRequest`). Converts every not-yet-actioned, exact-confidence principal suggestion into a remap to its suggested target, in one write, through the same merge-and-verify pipeline as `/config`. Fuzzy, ambiguous and already-resolved references are skipped, not failed. Returns `{acceptedCount, mappingSummary, updateIdentifier}`; with no qualifying suggestions it is a token-checked no-op that returns `acceptedCount: 0`.

The payload-override endpoints

`GET` / `PATCH` / `DELETE /v1/import/{sessionId}/resource/{sourceType}/{sourceId}/payload` read, replace and clear a single resource's JSON payload for the next execution. `sourceType` must be a literal `ResourceType` enum value — anything else is a `400` — and the pair must have a mapping on the session, else `404`. The `GET` returns the current override, or the packaged original — plus the packaged copy for diffing when an override exists. The save is a `PATCH` with `{payload, updateIdentifier}`; it validates the payload (a `400` for a bad payload or one that changes a structural field — immutability is checked against the per-resource fingerprints captured in the session's `structuralSnapshot` — and `413` for an oversized one), stores the override on the session as a `ResourceOverrideModel`, and resets a previously failed mapping to `pending` so a retry picks it up. Every override write is optimistic-locked and audit-logged. The `DELETE` takes `updateIdentifier` as a query parameter and is an idempotent revert to the packaged payload. Reads are blocked while `importing`; writes while `importing` or `validating`.

When to reach for these — parameterising a package per environment rather than fixing one bad value — is the subject of Generating a package.

Mapped-field decisions `Since 1.10`

A **mapped field** is a `DataEntityField` carrying `mapping.enabled` — a field that copies a value from a parent entity, naming that parent's field by UUID in `mapping.sourceFieldId`. Those UUIDs belong to the source environment, and the parent entity is usually *mapped* onto an entity that already exists in the target rather than written fresh, so the packaged id names nothing there. Import repairs this by matching the field's `fullNormalizedName` instead; the sidecar that carries that name across the boundary is `field-mapping-hints.json`.

When that match cannot be made, the **client decides** what the field copies from, and the session cannot run until every such field is decided. The decisions ride the ordinary configuration save.

The report: `mappedFields`

`validationReport.mappedFields?: MappedFieldStatus[]` (absent when the package has no mapped field the import writes) rides `POST /v1/import/validate` and `GET /v1/import/{sessionId}`, and `PATCH /…/config` answers with a fresh copy. There is one status per mapped field on an entity the import writes (mapped `CREATE` or `OVERWRITE`, plus fields that a skip-extension of an existing entity adds); fields whose parent is a core identity entity (users, groups, org units, working hours) are never listed, because those ids match everywhere.

```
type MappedFieldChoice =
  | {kind: 'copyFrom'; targetFieldId: string}
  | {kind: 'dontCopy'};

interface MappedFieldStatus {
  entityId: string; entityName?: string;   // the entity carrying the field (source-environment ids)
  fieldId: string; fieldName: string; fieldLabel?: ILocalizedValue;
  state: 'waiting' | 'matched' | 'unmatched' | 'unnamed' | 'decided' | 'unavailable';
  updateMode: number;                      // 1 Never, 2 ParentOnly, 3 TwoWay (the default when absent)
  sourceParentId: string;                  // the parent in the package's environment
  targetParentId?: string; targetParentName?: string; targetParentLabel?: ILocalizedValue;
  expectedSource?: {name: string; label?: ILocalizedValue; dataType?: number};
  matchedField?: MappedFieldCandidate;     // the field found on the target parent
  decision?: MappedFieldChoice;            // the standing decision, if any
  candidates: MappedFieldCandidate[];      // may be EMPTY: then only dontCopy is offered
}

interface MappedFieldCandidate {
  id: string;                              // target-parent field UUID
  fullNormalizedName: string;
  name?: Record<string, string | null>;
  dataType: number;
}
```

`state`	**Meaning**	**Client action**
`waiting`	The parent is created by this package; there is nothing to choose from yet.	None.
`matched`	The expected source field exists on the target parent (or the packaged parent is being written).	None.
`unmatched`	The package names the source field, but the target parent has no field with that name.	**Decide.**
`unnamed`	The package cannot name the source field (no sidecar hint, parent snapshot not in the package).	**Decide.**
`decided`	A valid decision stands for the current target parent.	None (it may still be changed).
`unavailable`	The target parent could not be read.	Retry later; a `MAPPED_FIELD_SOURCE_UNAVAILABLE` warning accompanies it.

Which fields qualify as candidates

`candidates` is not every field on the target parent. Three rules are mandatory — offering a candidate that fails any of them either silently produces wrong data or hard-fails the resource at data-service:

1. **Same data type.** The candidate's `dataType` must equal the mapped field's.
2. **Relation targets must be a subset.** For a reference field (`dataType` `7`), every entity the *candidate* can point at must be one the *local* field already accepts. A superset would let a reference be copied into a field that cannot hold it; requiring exact equality would wrongly reject a safe narrower candidate. A candidate with no determinable target set is rejected, and a local field with no determinable target set empties the whole list — the rule fails closed.
3. **`TwoWay` must agree on required.** When `updateMode` is `TwoWay` (`3`), the candidate's `validations.required` must equal the mapped field's, because data-service rejects the entire resource when a two-way source field disagrees.

Display-only fields (`options.displayOnly.enabled`) are excluded outright. A packaged mapping with **no** `updateMode` is treated as `TwoWay`, deliberately over-filtering rather than offering a candidate that would hard-fail later. `candidates` may legitimately be empty.

Saving decisions, and what blocks

Send the decisions with `PATCH /v1/import/{sessionId}/config` as `mappedFieldDecisions` — the **full list**, which replaces the stored one (so send every decision you want to keep, not just the one that changed):

```
{
  "mappedFieldDecisions": [
    {"entityId": "0b6d…", "fieldId": "f1…", "targetParentId": "9f2a…",
     "decision": {"kind": "copyFrom", "targetFieldId": "c7e3…"}},
    {"entityId": "0b6d…", "fieldId": "f2…", "targetParentId": "9f2a…",
     "decision": {"kind": "dontCopy"}}
  ],
  "updateIdentifier": "58c0…"
}
```

A malformed item is `400 MAPPED_FIELD_DECISION_INVALID`. The response adds `mappedFields` (the recomputed statuses) and `canProceed`. A decision is bound to the `targetParentId` it was made for and survives only while that is still the field's target parent and the choice is still offered (`dontCopy` always; `copyFrom` only the matched field or a listed candidate). A save that changes what the parent maps to — a resource mapping or an unmapped-dependency target — therefore drops decisions that no longer fit, and its response already carries the new statuses.

What blocks. Each field in `unmatched` or `unnamed` without a decision is a validation **error**, `MAPPED_FIELD_SOURCE_UNDECIDED` ("Choose what '*entity.field*' copies from before installing."), the first 20 in full and one summary item after that, so `canProceed` is `false` until all are decided. Execute and retry re-check and refuse with `409 MAPPED_FIELD_SOURCE_UNDECIDED` (naming up to ten fields) or, when a parent cannot be read, `503 MAPPED_FIELD_SOURCE_UNAVAILABLE`.

What a decision does at execute. `copyFrom` sets `mapping.sourceFieldId` to the chosen field. `dontCopy` sets `mapping.enabled = false` and, when `updateMode` is `Never` (`1`) or `ParentOnly` (`2`) and the field carries a `locked` block, also clears `locked.enabled` — otherwise the field would import locked with nothing able to populate it. A field with no decision still goes through name matching: a match rewrites `sourceFieldId`, and a miss leaves the mapping as declared with an import warning that it will not copy until it is fixed. Nothing is ever disabled without a decision.

A headless store install or update cannot decide for the user: an undecided mapped field comes back among its unresolved `choices` (`kind: 'mappedField'`), and the run is refused until it is decided on the import page.

Executing and finishing

`POST /v1/import/{sessionId}/execute`

No body. The gatekeeping sequence, in order:

1. If the session was **cancelled**, the call is refused with `409 IMPORT_SESSION_CANCELLED` (`Since 1.10`) — start a new import. Otherwise, if the session is already `failed` or `completed`, the call **delegates to retry** — a client may treat execute as "run it" without tracking which of the two it needs.
2. Every unmapped dependency required by a resource the import will actually write (action `CREATE` or `OVERWRITE`) must have a target — otherwise `400`. Dependencies needed only by skipped resources do not block. At the development-line head (`1.11.0-alpha.4`) this check does not yet distinguish an optional `hard:`

`false` row, although the import page lets **Execute** through with one unmapped — see the caution under Page widgets pointing outside the package.

3. `Since 1.10` every mapped field that needs a decision must have one — otherwise `409 MAPPED_FIELD_SOURCE_UNDECIDED`, or `503 MAPPED_FIELD_SOURCE_UNAVAILABLE` when a target parent cannot be read (see Mapped-field decisions).
4. Every value reference consumed by an actively written resource must have an action. Safe defaults are applied first — a reference that exists in the target defaults to `KEEP`, one with a packaged payload to `AUTO_CREATE` — and only what remains unresolvable is a `400`; `Since 1.10` its code is `VALUE_REFERENCE_ACTION_UNSET`. An `AUTO_CREATE` whose identity key (a user's email or username, for instance) already matches an entry in the target is refused for that reference with `VALUE_REFERENCE_IDENTITY_COLLISION` rather than creating a duplicate principal.
5. `Since 1.10` the tenant must have room for every entry the run will create: packaging asks data-service up front and refuses with `403 ENTRY_CAP_EXCEEDED` ("…past its entries limit. Archive or delete entries, or upgrade the plan, then install again."). Nothing has been written at that point. Any other failure of this check is logged and the run goes on, because each create is still checked on its own.
6. The session is claimed by an **atomic** `validated → importing` transition. Losing that race — a concurrent execute, or a session that moved — is a `409`, never a double execution.

The response is `{jobId, sessionId}` (`ImportExecuteResponse`): the work is queued, not done. Poll `/status` until the state machine reaches `completed` or `failed`.

`POST /v1/import/{sessionId}/retry`

No body. Accepted only in `completed` or `failed` (else `409`) — and, `Since 1.10`, never on a cancelled session (`409 IMPORT_SESSION_CANCELLED`) — claimed atomically like execute. `Since 1.10` it re-checks the mapped-field decisions and the entries limit exactly as execute does — the limit is counted against the rows the reset will re-open, so entries that only a cascade-skipped resource needed are included. Failed and cascade-skipped mappings are reset to `pending` — cascade-skipped ones get their original action back, recomputed from the session's strategy — and failed or skipped value references are re-opened. Already imported resources are not re-imported; the retry re-attempts only what did not land. Returns `{jobId, sessionId}`.

`POST /v1/import/{sessionId}/cancel`

No body. Accepted while `validated` or `importing`; any other status is a `400`. The session is atomically flipped to `failed` with a `Cancelled: …` error message, and the queued job is removed; a worker already running notices the marker between resources and aborts. Resources written before the abort **remain** — cancellation stops the import, it does not undo it. `Since 1.10` cancellation is also **final**: the session records `cancelledAt`, and execute or retry on it answers `409 IMPORT_SESSION_CANCELLED`. Returns `{outcome, state?}`.

The result of a finished run (`importResult` on `/status`) gained `importWarnings` in 1.10. It reports what the import changed rather than fail on:

- **Index replacement.** An imported entity's custom indexes **replace** the target's, and an index whose field path the target cannot index is dropped with a warning such as `Dropped index "<name>": field path <path> not found` instead of failing the entity.
- **Files that did not transfer.** A resource still carrying a source-tenant file id (the export could not download it, or the import could not upload it) used to fail with `FILE_EXPIRED`; the file is now removed from the resource — cleared, dropped from an array, or removed as a thumbnail — with one warning per document. A file id that resolves in the target storage (a same-tenant import) is kept.

- **Mapped fields left as declared.** A mapped field with no decision whose source the name match still cannot find at run time keeps its mapping as declared, with a warning that it will not copy until it is fixed.

`DELETE /v1/import/{sessionId}`

Deletes an abandoned session. Refused with a `409` while the session is `importing`; carries the `updateIdentifier` as a query parameter so a stale tab cannot delete a session someone else has moved.

The session document

`GET /v1/import/{sessionId}` returns the `ImportSession` (persisted in the MongoDB collection `ImportSession`). Its key fields:

Field	Purpose
`id`	UUID v4 (generated at validate).
`spaceId`	Target space.
`status`	`ImportSessionStatus` — see the state machine.
`packageFileId` / `packageFileName`	Storage-service reference to the ZIP (re-downloaded at execute).
`manifest`	Parsed package manifest.
`validationReport`	Phase 1 output.
`resourceMappings`	Per-resource decisions (see `ResourceMapping` below).
`valueReferenceMappings`	Entry-level reference decisions.
`unmappedDependencyMappings`	Targets for external dependencies not in the package.
`resourceOverrides`	User-edited payloads applied at execute.
`structuralSnapshot`	Per-resource immutable-field fingerprints.
`importResult`	Phase 2 aggregate result.
`importJobId`	BullMQ job id (for cancel).
`importStrategy`	Bulk strategy; default `MANUAL`.
`updateIdentifier`	Optimistic-concurrency token (UUID; rotated on each write — see below).

Enumerations

Enum	Values
`ImportSessionStatus`	`validating`, `validated`, `importing`, `completed`, `failed`
`ImportStrategy`	`MERGE_NEW_ONLY` (CREATE new, SKIP existing), `OVERWRITE_ALL` (CREATE new, OVERWRITE existing), `MANUAL` (default; per-resource as-is)
`MappingAction`	`CREATE`, `OVERWRITE`, `SKIP` — a `SKIP` on an existing resource still registers its id mapping, so other resources' references resolve to it (this is the web app's **Map**)
`ResourceImportStatus`	`pending`, `completed`, `failed`, `skipped`, `cascade_skipped`
`ValidationStatus`	`NEW`, `EXISTING`
`ValueReferenceAction`	`KEEP`, `REMAP`, `NULL`, `AUTO_CREATE`

Enum	Values
`EntryReferenceSourceStatus`	`FOUND`, `NOT_FOUND`, `ARCHIVED`
`ModuleConflictResolution`	`MOVE`, `KEEP`

`ResourceMapping`

The user-editable decision per resource: `sourceType`, `sourceId`, `normalizedName`, `displayName?`, `action`, `targetId?` (existing target, or filled after CREATE), `status?`, `errorMessage?`, `isHardDep?`, `moduleConflictResolution?` (`MOVE` / `KEEP` for a remapped-module conflict), `entityId?`, `moduleId?`.

`ImportResult`

Persisted after Phase 2 and returned by `GET /…/status`: `created`, `overwritten`, `skipped`, `cascadeSkipped`, `errors`, `resources[]` (`{sourceId, targetId, type, action}`), `importErrors[]` (`{sourceId, type, message}`), `valueReferenceResults[]`, `importWarnings` (`Since 1.10`, see above), plus the value-reference counters `entriesAutoCreated`, `entriesAutoCreateFailed`, `entriesKept`, `entriesRemapped` and `entriesNulled`.

Concurrency: the update identifier

Every mutating write on a session carries an optimistic-concurrency token. The session's `updateIdentifier` is seeded at creation and **rotated on every successful write**; each write must echo the token the client last saw, and the response returns the new one, which the client must adopt.

A mismatch is a `409` whose `error.code` is `STALE_UPDATE_IDENTIFIER` — deliberately distinguishable from the status-gate `409`s such as "cannot modify config for session in status: importing", which share the HTTP status but not the code. On `STALE_UPDATE_IDENTIFIER`, reload the session and let the user reconcile; on a status-gate `409`, re-read the status and follow the state machine.

No-op writes — an empty config save, an accept-exact with nothing to accept, a delete of an override that does not exist — still check the token but do **not** rotate it, so a client's lock state stays consistent across them.

Status gates at a glance

Endpoint	Accepted statuses	On violation
`PATCH /…/config`	`validated`, `failed`	`409`
`PATCH /…/suggestions/accept-exact`	`validated`, `failed`	`409`
`PATCH /…/strategy`	`validated`	`409`
`GET /…/resource/…/payload`	any except `importing`	`409`
`PATCH` / `DELETE /…/resource/…/payload`	any except `importing`, `validating`	`409`
`POST /…/execute`	`validated` (terminal states delegate to retry; never a cancelled session)	`409`
`POST /…/retry`	`completed`, `failed` (never a cancelled session)	`409`
`POST /…/cancel`	`validated`, `importing`	`400`
`DELETE /…/{sessionId}`	any except `importing`	`409`

Reads (`GET` sessions, session, status) carry no status gate — only the space-ownership check.

Schema versioning

Every package manifest carries a `schemaVersion` — the version of the **package format**, deliberately decoupled from the version of the platform that wrote it. This page documents which versions exist, which ones an importer accepts, the three distinct version checks that run at validation, and what a package producer should write. The format itself is documented in The package format. The packaging engine this contract belongs to shipped in the **1.8** line (`Since 1.8`); on an older deployment these endpoints and the export UI are absent.

The supported versions

The authority is a closed allow-list:

```
export const SUPPORTED_SCHEMA_VERSIONS = ['1.0', '1.1', '1.2', '1.3', '1.4'] as const;
```

`1.2` , `1.3` and `1.4` are all `Since 1.10` . A **pre-1.10** deployment (1.8 or earlier) still has the two-item allow-list `['1.0', '1.1']` and rejects a package declaring any of the three. That version gap is the subject of What 1.2 adds and What 1.3 and 1.4 add, and it is why a package declares a newer version only when it has to: an export produced on 1.10+ still needs to import cleanly into an environment that has not yet upgraded past 1.8.

The writer side is `pickSchemaVersion` , which returns the **lowest version that describes the package's actual contents** — `1.4` when it carries a README, else `1.3` when it carries a `package` block, else `1.2` when it carries the mapped-field sidecar, else `1.1` . `Constants.SCHEMA_VERSION = '1.4'` is the highest version the service can *produce*; a unit test asserts it also appears in the allow-list, so the service can never reject its own exports. A **plain export** carries neither a README nor a `package` block (`POST /v1/export` takes no README `Since 1.10`), so it is always `1.1` or `1.2` ; `1.3` and `1.4` come from published package versions.

Version	Status	What it is
`1.0`	accepted	The original layout
`1.1`	accepted — and still what export stamps on a package with no mapped fields, no `package` block and no README	Adds the source-identity manifest fields
`1.2` `Since 1.10`	accepted from 1.10 onward; **rejected on a pre-1.10 deployment**	Adds the `field-mapping-hints.json` sidecar; stamped only by a package that carries it
`1.3` `Since 1.10`	accepted from 1.10 onward; **rejected on a pre-1.10 deployment**	Adds the `package` manifest block, `hashes.json` and `package-references.json` ; stamped only by a published package version
`1.4` `Since 1.10`	accepted from 1.10 onward; **rejected on a pre-1.10 deployment**	Adds `README.md` and `readme/` ; stamped by any package that carries a README — in practice a published version with one
anything else	rejected	Blocking `INCOMPATIBLE_SCHEMA` at validation

Note that this is an allow-list, **not** a range check: a hand-authored manifest claiming an unlisted version is rejected exactly like one claiming `0.9` , even though it may look "newer". The strict package validator also requires `schemaVersion` to be a **string** — a bare number in the JSON fails the manifest shape check with a `SCHEMA_VIOLATION` before version comparison even happens.

What 1.2 adds `Since 1.10`

> **Since 1.10.** `1.2` shipped on the development line that became release 1.10. On a **pre-1.10** deployment the allow-list is still `['1.0', '1.1']`, and a package that declares `1.2` is refused there with a blocking `INCOMPATIBLE_SCHEMA`.

`1.2` adds the `field-mapping-hints.json` **sidecar** — one new optional top-level ZIP entry, and nothing else. It lets import resolve a mapped field's `sourceFieldId` by name when the parent entity's own snapshot is not in the package. No resource payload gained a property and no manifest field was added or removed; the entry itself is specified in the package format.

The version is **declared conditionally** — only by a package that actually carries the sidecar (`buildPackageManifest` via `pickSchemaVersion`). The rule is *declare the lowest version that describes this package's actual contents*: `1.2` when the sidecar it just built is non-empty (and there is no `package` block or README), `1.1` otherwise.

That conditionality is the whole design, because `schemaVersion` gates in **both** directions:

Package	Target	Outcome
`1.0` or `1.1`	a `1.2`-capable environment	**Accepted unchanged.** There is no sidecar to read, so a mapped field's source name falls back to the parent entity's in-package snapshot — exactly the pre-`1.2` behaviour.
`1.2` — it carries the sidecar	a pre-`1.2` environment, i.e. anything on 1.8 or earlier	**Refused outright.** Blocking `INCOMPATIBLE_SCHEMA`, `canProceed: false`, before any target-space lookup runs.
any package with **no** mapped fields, exported by a `1.2`-capable environment	anything	Stamped `1.1`, so it still imports into a pre-`1.2` environment exactly as before.
`1.2`	a `1.2`-capable environment	Full round trip: the sidecar travels, and a mapped field resolves by name even when its parent entity is not in the package.

Declaring `1.2` unconditionally would strand *every* export — including the majority that contain no mapped fields at all — on any environment not yet upgraded. That is row three, and it is the reason the choice is made per package.

For a package that does carry the sidecar, row two's rejection is correct and unavoidable rather than a gap: without the declaration, `validatePackageStrict` on the older service would fail the same package anyway, with an opaque `UNKNOWN_ENTRY` on the unrecognised top-level entry. `INCOMPATIBLE_SCHEMA` at least says why.

The operator rule that follows: upgrade the target environment before the source. A source already on `1.2` can keep shipping to an older target for as long as its packages happen to carry no mapped fields — but nothing on the target side can force that, and the moment one mapped field enters a package the export becomes unimportable there. Upgrading targets first is the only ordering that is never blocked.

What 1.3 and 1.4 add `Since 1.10`

Both versions belong to published package versions and the in-app store, and both follow the same conditional rule as `1.2`.

- `1.3` adds the optional `package` manifest block plus two top-level entries, `hashes.json` and `package-references.json`. Only a **published package version** declares it; a plain export never carries a `package` block. On import, the block is what turns the session into an install.
- `1.4` adds `README.md` and the `readme/` image folder, with `manifest.readme` pointing at them. **Any** package that carries a README declares `1.4`. Only a published version is written with one: a plain `POST /v1/export`

takes no README (it has no `readme` field), though an importer still reads a README it finds in a hand-built file.

The entries are specified in the package format. As with `1.2`, a pre-1.10 target refuses these packages outright with a blocking `INCOMPATIBLE_SCHEMA`; without the declaration it would fail them anyway on the unknown top-level entries.

The same validation step applies one more gate to versioned files: a `package` block whose `id` is not a valid `pkg_` id — a file published before package ids existed — is refused with `422 PACKAGE_FORMAT_OUTDATED`, and the file must be exported again from its environment. This is not a `schemaVersion` check: such a file may well declare a supported version.

What 1.1 adds

Two optional manifest fields: `sourceEnvironmentId` and `sourceSpaceId`. Together with the target space they form the key of a persisted **mapping profile** — the memory of how a previous import from the same source resolved users, groups and org-units. The stored-profile lookup returns early when either field is absent (`resolveStoredMappingProfiles`).

The consequence is graceful, not fatal: a `1.0` package imports fine, but no stored principal mapping is ever pre-filled for it — every principal falls back to natural-key matching, and remembered decisions from earlier imports do not apply.

Three version checks, three severities

Validation runs three independent version checks, and they are not equally serious.

The package format: `schemaVersion`

Blocking. An unsupported value produces the `INCOMPATIBLE_SCHEMA` error and validation returns early — no resource classification, no target-space lookups. The session is still created so the failure is inspectable, but it lands directly in `failed` with an empty resource report and `canProceed: false`.

The source release: `sourceVersion`

A warning, in one direction only. The manifest's informational `sourceVersion` records the platform release that exported the package. When it is *newer* than the importing environment, validation attaches the non-blocking `NEWER_SOURCE_VERSION` warning — the package may reference capabilities the older environment lacks. The comparison is `major.minor.patch` only; prerelease tags are ignored, and an unparseable version simply produces no warning. A package *older* than the environment raises nothing at all — that is the supported direction.

The per-resource contract: `specificationVersion`

A warning per resource. Each packaged resource payload may carry its own `specificationVersion`; a missing value is treated as `1`. Every resource parser declares the spec versions it supports, and today every parser supports exactly `[1]` (the shared default, overridden by none). A resource claiming another version gets an `UNSUPPORTED_SPEC_VERSION` warning naming the resource — validation proceeds, and the resource may still fail at execution.

The compatibility matrix

Package says	Importer's verdict	Effect
`schemaVersion: "1.4"` `Since 1.10`	accepted from 1.10 onward; **blocked** on a pre-1.10 deployment	The README is validated and kept with the import record
`schemaVersion: "1.3"` `Since 1.10`	accepted from 1.10 onward; **blocked** on a pre-1.10 deployment	The `package` block makes the import an install
`schemaVersion: "1.2"` `Since 1.10`	accepted from 1.10 onward; **blocked** on a pre-1.10 deployment	The sidecar is read, so a mapped field resolves by name without its parent's snapshot
`schemaVersion: "1.1"`	accepted	Full behaviour, including stored mapping-profile reuse
`schemaVersion: "1.0"`	accepted	Imports; mapping profiles never match — principals resolve by natural key only
`schemaVersion:` anything else	**blocked**	`INCOMPATIBLE_SCHEMA`; session created in `failed`, `canProceed: false`
`sourceVersion` newer than the environment	warning	`NEWER_SOURCE_VERSION`; import proceeds at your own risk
`sourceVersion` older, equal, or absent	silent	The supported direction
a resource's `specificationVersion` ≠ 1	warning	`UNSUPPORTED_SPEC_VERSION` on that resource
a `package` block with a pre-package-id `id` `Since 1.10`	**blocked**	`422 PACKAGE_FORMAT_OUTDATED`; export the file again

Forward migration

There is no migration step, and no upgrade transform anywhere: nothing rewrites a package from one `schemaVersion` to another, on import or on export. A `1.0` package is read **as-is**, and everything `1.1` to `1.4` added is optional. Absence degrades a feature — profile reuse at `1.1`, name-based mapped-field resolution at `1.2`, install mode at `1.3`, the README at `1.4` — rather than failing the import, which is the design intent recorded at the allow-list itself: every listed version is accepted "so older packages keep importing unchanged". Format versioning is thus one-directional: environments move forward, packages stay importable.

For a producer the guidance is short:

- **Stamp the lowest version that describes what you actually wrote.** Omit `field-mapping-hints.json` and stamp `"1.1"`; include the sidecar and stamp `"1.2"`; include a README and stamp `"1.4"`. Stamping a newer version than your contents need gains nothing and locks the package out of every environment that has not been upgraded.
- **Do not hand-write a `package` block.** A `1.3` package is a published version: its hashes, root hash and cross-package references are computed by the publishing environment, and an install validates them. Publish from the Packages area instead.
- Include `sourceEnvironmentId` and `sourceSpaceId` if imports from your pipeline should benefit from remembered principal mappings.
- Never invent a higher version to signal "newer content". The release that exported the resources belongs in `sourceVersion`; `schemaVersion` describes only the container layout, and an unknown value blocks the import outright.

Module JSON transfer

> ⚠ **Legacy path**
>
> This is an older, narrower mechanism kept for compatibility. To move an application between environments, use packaging — or, in the UI, Export, import and install apps [Build Apps]. See When to use packaging instead.

Alongside the ZIP packaging system, `data-service` carries a second, older transfer mechanism: exporting **modules and entities as a single JSON file** and replaying that file into another space. It is genuinely one file — which is why people find it attractive — but it moves *definitions of modules and entities* and very little else. This page documents its format, its endpoints, and — plainly — what it will not carry, so you can decide when the packaging system is the right tool instead.

The two systems do not share code, formats or endpoints. Packaging lives in `packaging-service` under `/api/packaging`; this path lives in `data-service` under the core `/api` prefix. The similarity of their URL shapes (`/v1/import/…` in both) is coincidence, not compatibility.

One file, two kinds

The file is an `ImportExportModel`:

Field	Type	Meaning
`specificationVersion`	number	File-format version. The current exporter writes `1`; the import path does not branch on it
`kind`	`"DataEntity"` | `"DataModule"`	What the file transfers (`ImportExportKindEnum`)
`dataModules`	array, optional	Module definitions — present for module-kind files
`dataEntities`	array	Entity definitions

Each entity in `dataEntities` is a `FullDataEntity`: the complete entity definition — fields, workflows, internal entities, form settings, sharing — plus embedded arrays the exporter fills in: `triggers` (the entity's reactions), `queries`, `views`, `lists` (collections) and `relations`. Do not let those arrays mislead you about what the import will do with them; see *What it will not carry* below.

Portability comes from **normalized names**: the exporter rewrites entity-reference field sources from ids to entity normalized names, resolves auto-increment counters into inline options, and (for module exports) records each entity's module by normalized name (`exportEntities`).

Exporting

Both export endpoints respond with a **file download** — an `application/json` attachment named after the first exported resource. They write the response themselves, so the body is the bare JSON document, not the `{success, result}` envelope that wraps ordinary API responses (see Calling the HTTP API). The import endpoints below, by contrast, are ordinary enveloped operations.

`POST /v1/export/entity`

Body: `{"dataEntities": [{"id": "…"}, …]}`. Exports the named entities as a `kind: "DataEntity"` file. Edit permission is checked per entity; a denial is a `403` with code `CANNOT_EXPORT_ENTITY`.

`POST /v1/export/module`

Body: `{"dataModules": [{"id": "…", "dataEntities": [{"id": "…"}, …]}, …]}`. Exports the named modules together with the listed entities of each, as a `kind: "DataModule"` file that carries both `dataModules` and `dataEntities`. Each module must exist in the calling space, and the per-entity permission check applies here too.

Importing

Import is deliberately two-phase, and the file travels by reference: upload the JSON to the storage service first, then pass its file id in the path. An unparseable file is rejected with `IMPORT_FILE_IS_INVALID`; a `kind` outside the enum with `FILE_KIND_NOT_SUPPORTED`.

`POST /v1/import/{fileId}` — the dry run

Parses the file and reports what *would* happen, writing nothing. The body is `{"moduleId": "…"}` — optional in the schema, but for an entity-kind file it is required in practice: such a file carries no modules of its own, so without a target module the import cannot resolve where the entities land and fails with a module-not-found error (`prepareImportData`). For a module-kind file the body's `moduleId` is ignored; modules come from the file.

The response is an `ImportResultDto`: per-item summaries for entities and modules — `{resourceTempId, normalizedName, name, alreadyExists, allowOverride}` — a `dependencyMap` of intra-file entity dependencies, and an `errors[]` array of `{resourceTempId, path, errorCode, errorMessage}`. Duplicated normalized names inside the file are flagged (`ENTITY_DUPLICATED_INSIDE_IMPORT_FILE`, `MODULE_DUPLICATED_INSIDE_IMPORT_FILE`), as are unresolvable entity references.

`POST /v1/import/{fileId}/confirm` — the apply

Same path shape, same body, plus one gate: confirming needs a license that includes administration — inspecting a package changes nothing, applying one is a configuration change (`403 LICENSE_ADMIN_FEATURE_REQUIRED` otherwise). The confirm re-runs the dry-run analysis and **refuses entirely** (`IMPORT_FILE_IS_INVALID`) if it reports any error, or any colliding resource that does not allow override — there is no partial apply and no per-resource opt-out.

When it proceeds, it creates or updates the modules, then creates or overwrites the entities in dependency order (`sortEntitiesByDependencies`; a reference cycle aborts with `CIRCULAR_DEPENDENCY_DETECTED`), and — for an **entity-kind** file only — recreates the reactions embedded as `triggers`. Entity creation and overwrite run asynchronously through the platform's command bus, so the HTTP response confirms acceptance of the whole set rather than completed materialisation of every entity.

`Since 1.10` two more checks run before anything is written, and each refuses the **whole** confirm:

- **Installed resources** — overwriting a module or entity that came from an installed package, or creating an entity inside an installed module, is refused with `403 PACKAGE_RESOURCE_READ_ONLY`. Fork the installation first; see installed resources.
- **The entity cap** — when the tenant's plan sets an entity cap, an import that would create more entities than the cap leaves room for is refused with `403 ENTITY_CAP_EXCEEDED`.

Collision rules

Existing resources are matched by `normalizedName` within the target space, and the `allowOverride` flag in the dry-run report is the whole contract (`isEntityExistsResult` / `isModuleExistsResult`):

Case	Overwritable?
Entity exists, has **no entries**, name not reserved	yes — the definition is overwritten in place
Entity exists and **has entries**	no — the import is blocked
Entity name is a reserved word (`switch`, `new`, `class`, …)	no
Module exists	yes — name, description, icon and sharing are updated
Module is the core module	no

The entries rule is the one that surprises people: this mechanism can refresh the *definition* of an entity only while that entity is still empty in the target. Once real data exists, the JSON transfer will not touch it — schema evolution of a live entity is a different problem, and this tool declines it.

What it will not carry

The file format looks richer than the import is. State of the code today:

- **Views, queries, collections, relations** — embedded in the file by export, but never replayed: the import materialises only modules, entities and reactions, and the entity-creation path carries the entity definition alone. The embedded arrays are dead weight on arrival.
- **Reactions — only half-carried.** An entity-kind confirm recreates the embedded triggers; a **module-kind confirm drops them** — its code path never dispatches the prepared reactions. If reactions must travel with a module, this path will not do it.
- **Entries** — definitions only, never data. Nothing entry-shaped exists in the format.
- **Everything outside the module/entity pair** — dashboards, reports, functions, system buttons, statistic queries, counter widgets, announcements, pages/generic views: no representation in the format at all.
- **Identity and permissions** — no principal mapping, no value references, no permission grants beyond the `sharing` field embedded in the module and entity definitions.
- **Process** — no session, no configure step, no per-resource actions, no payload overrides, no retry. Confirm is one shot: it applies everything or refuses everything.

When to use packaging instead

Use this JSON path for what it is: a quick, single-file copy of module and entity *skeletons* between spaces — a starting schema, a template of empty entities, a definition refresh while the target is still empty.

Reach for the packaging system the moment any of the following is true: views, queries, reactions-with-modules, reports or any other resource type must travel; references to users, groups or entries need remapping rather than luck; existing target resources need per-resource create/overwrite/skip decisions; you need a validation report you can act on, payload overrides, retry, or an audit trail. That system's single-file question — "can I get one JSON with everything?" — is answered honestly in The package format: a full solution package is a multi-file ZIP, and this JSON path is not a substitute for it.

Packages, Marketplace & the In-App Store

This page is for **integrators and engineers**: the access model, routes, error codes and server rules behind packages, published versions, the review inbox, the marketplace connection and the in-app store.

Doing this in the UI?

Creating packages and versions, filling in a store page, sending for review, reviewing, and installing, updating, forking or uninstalling apps are covered for builders in Export, import and install apps [Build Apps].

Availability

Since 1.10. A few later 1.10 additions name the patch they arrive in (problem reports from 1.10.1; the publishing-revocation notice and the README nudge from 1.10.2). The development line adds a few more, marked `Since dev` where they are described: stale saves are refused, page widgets that show a view outside the package, and web images pasted into the README.

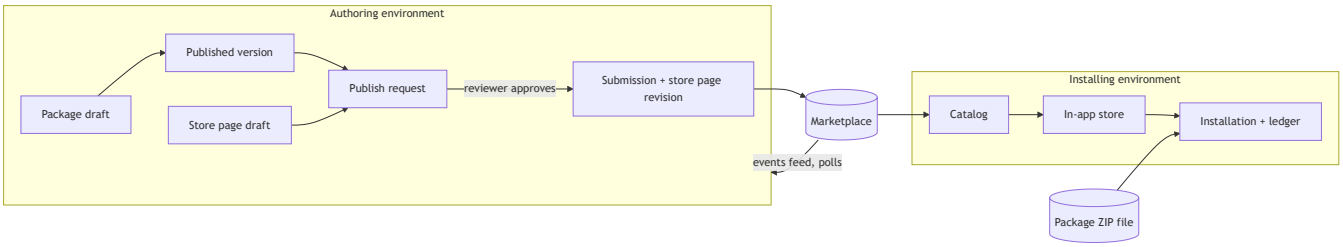
The file workflow moves resources between environments as an anonymous ZIP: export it, hand it over, import it. The 1.10 line builds a second layer on top of that workflow, for app builders who want to ship a **template** that other environments can install and keep track of:

- a **package** — a named, permanent publishing identity that owns one or more modules;
- **published versions** of it — immutable, semver-numbered builds of the package ZIP;
- a **store page** — the app's listing in the marketplace, drafted next to the package;
- an internal **review inbox**, so nothing reaches the marketplace without a second person approving it;
- a **marketplace connection** that registers the environment with the OSOS marketplace, and a separate **publisher** identity for environments that publish;
- an **in-app store** that browses the marketplace catalog and installs from it;
- an **installation ledger** on the receiving side, which locks what an installation created until someone deliberately forks it.

An install is still an import under the hood — the same session, the same remapping engine — run in an *install mode* that adds version, dependency and ownership checks. This page is the design and integration side of that layer: the access model, the routes, and the rules the server enforces. The package file format additions it relies on (the `package` manifest block, `hashes.json`, `package-references.json` and the README) are specified in The package format; the export, import and mapping mechanics underneath are in Packaging & Distribution.

Off until an operator turns it on

The marketplace connection is **disabled by default**. The packaging service only talks to a marketplace when its marketplace addresses are configured; the deployment chart ships with `marketplace.enabled: false`. Until an operator enables it, every marketplace route answers `503 MARKETPLACE_DISABLED` and the web app hides every marketplace surface (the Store, the Reviews page and the store-page editor). Packages, versions and installing from a package *file* work without it.



Vocabulary

Term	What it is
Package	The publishing identity of a template: a permanent id `pkg_<ULID>`, a localized display name and description, the modules it owns, and a semver range per package it depends on. A package is draft data in the authoring environment, not a file.
Package version	One immutable published build: a ZIP in storage with its `sha256` and canonical `rootHash`, a snapshot of its modules and dependencies, a status (`published` or `deprecated`) and, once sent, its marketplace submission.
Store page	The package's listing draft: store name, topics, documentation URL, maintenance note, icon and screenshots. Its description is the package's own description.
Publish request	An internal review item that carries either a version or a store page towards the marketplace. A version and its store page can travel together as one **group**.
Marketplace connection	The environment's registration with the marketplace. There is one per environment, kept in the ROOT space. It exists with or without a publisher.
Publisher	The store identity that published apps appear under: a name and a store address. It is set up separately from connecting, and only an environment that publishes needs one.
Installation	The receiving side's record of one package version installed into a space, with a ledger of every resource it created.
Import record	A history row for any import — a plain import, a package file, or a store install.
Store	The in-app browse-and-install experience over the marketplace catalog. Not to be confused with a data *store* (Data Stores [Build Apps › Share an entity's records across spaces]).

Everything that talks to the marketplace — the connection, publish requests, the store page, the catalog — is available **only in the ROOT space**. Packages, installations and import records are per space.

Access model

The three marketplace groups

Marketplace rights are membership in three system user groups, which data-service seeds into the users-group entity of the ROOT space (migration `1.10.0-a37-marketplace-permission-groups`):

Permission	Group id	What it lets a member do
`publish`	`MARKETPLACE_PUBLISHERS`	Set up publishing, prepare and withdraw publish requests, unpublish and republish an app, read the publisher terms.
`review`	`MARKETPLACE_REVIEWERS`	Read the review inbox, approve a request or return it with a note, accept the publisher terms.
`install`	`MARKETPLACE_INSTALLERS`	Browse the catalog, open an app page, connect the environment, install from the store.
`update`	`MARKETPLACE_INSTALLERS`	The same group as `install`; it gates updating an installed app in place.

An earlier `MARKETPLACE_UPDATERS` group is merged into the installers group by migration `1.10.0-a40-merge-marketplace-updaters`, which deletes the old group only once it is empty.

Publishing and reviewing need the groups, even for administrators. A system administrator (system-level `AdminAccess`) holds only `install` and `update` without belonging to a group; `publish` and `review` come from group membership alone. An internal service call holds all four. Membership is cached per tenant and user for **60 seconds**; a system administrator who is refused re-reads their groups once without the cache, so an administrator who has just added themselves to a group does not have to wait. A missing permission answers `403 MARKETPLACE_PERMISSION_REQUIRED`.

A few operations are reserved to system administrators regardless of group: **renaming the publisher**, **changing the store address**, and finishing a store install that still has unresolved choices.

Publishing is a plan entitlement

Any environment may install; only an environment whose plan includes the **`marketplacePublish`** entitlement may publish. Clients read it as `publishInPlan` on `GET /v1/marketplace/me`. Outside the plan, the server refuses with `403 MARKETPLACE_PUBLISH_NOT_IN_PLAN` on exactly these operations:

- preparing a publish request (a version, a store page, or both as a group);
- approving a publish request or a group;
- setting up publishing, renaming the publisher, and changing the store address;
- unpublishing or republishing an app.

Connecting, browsing, installing and updating, checking a store address, the terms routes, withdrawing or returning a request, and every read stay open. An environment without the entitlement can therefore connect *install-only*. The web app shows publishers and reviewers a notice when publishing is not in the plan; a backend too old to report `publishInPlan` is treated as allowing it.

Where each route is authorised

Routes	Who may call them
`/v1/packages/**`	Authenticated, with **Developer or higher on every module of the package** (`403 PACKAGE_DEVELOPER_REQUIRED`). Sending routes also need the ROOT space and a marketplace permission.
`/v1/packages/{id}/listing` (read, save, images)	ROOT space. **Any developer of the package writes the store-page draft** — no marketplace group is needed; a `review` holder may also read it. Sending, unpublishing and republishing need `publish`.
`/v1/publish-requests/**`, `/v1/publish-request-groups/**`	ROOT space, plus `review` (or `publish` for withdraw, `publish` or `review` for refresh).
`/v1/marketplace/connection` (read, connect)	ROOT space, plus `install` or `publish` — so every administrator qualifies through `install`.
`/v1/marketplace/me`	ROOT space; answers the caller's own permissions.
`/v1/marketplace/publisher` (set up), `/namespaces/{name}` (check an address)	ROOT space, plus a system administrator or `publish`.
`/v1/marketplace/publisher` (rename), `/namespace` (change the address)	ROOT space, system administrator only.

Routes	Who may call them
`/v1/marketplace/slugs/{slug}`	ROOT space, `publish`, and a publisher on the connection.
`/v1/marketplace/terms`	Reading: `publish` or `review`. Accepting: `review`.
`/v1/marketplace/catalog/**`, `/publishers`	ROOT space, plus `install`. `/topics` takes `publish` or `install`.
`/v1/installations/**`	Space admin (a system-level Admin grant in the space); uninstall and its preview need an **environment administrator** — a system administrator calling in the ROOT space. The update routes need an environment administrator **and** `update`. `POST /v1/installations/origins` is open to any user of the space, so a designer can learn a resource is locked before trying to save it.
`/v1/imports/**`	Space admin; `POST /v1/imports/origins` is open to any user of the space.

These rules sit beside the 1.10 rules of the file workflow: plain export is authorised by Developer on the exported roots, and every import route is space-admin only.

Package identity

A package id is `pkg_` followed by a 26-character Crockford-base32 ULID (`^pkg_[0-9A-HJKMNP-TV-Z]{26}$`). It is permanent and travels in every published version's manifest, which is how another environment recognises "the same package, a newer version".

- **A module belongs to at most one package.** A second package claiming it is refused (`409 MODULE_IN_OTHER_PACKAGE`, enforced by a unique index so concurrent saves cannot both win).
- **The store slug** is the package's name in the marketplace: lowercase letters, digits and hyphens, starting with a letter or digit, at most 50 characters. It is chosen when the first version is sent for review and **locked once the marketplace approves it**; after that a change answers `409 SLUG_LOCKED`. A slug used by another package answers `409 SLUG_TAKEN`, and one held by an open request `409 STORE_SLUG_IN_USE_BY_OPEN_REQUEST`. `GET /v1/marketplace/slugs/{slug}?packageId=` checks availability ahead of time (`available`, plus `TAKEN`, `INVALID` or `LOCKED`).
- **Older files.** A published-version file exported before package ids existed is refused at validation with `422 PACKAGE_FORMAT_OUTDATED` — export it again from its environment. Data-service migration `1.10.0-a39-package-identity` gives every pre-existing package a `pkg_` id.

Authoring packages and versions

Package drafts

Route	Purpose
`GET /v1/packages`	The space's packages the caller may author, most recently updated first. Each summary carries its marketplace state: the store slug and whether it is locked, whether the store name is locked, whether the app is unpublished, whether **another publisher owns it**, and its last known store status.
`POST /v1/packages`	Create a draft: `displayName` (localized, required), `description`, `moduleIds`, `pageIds` (global pages), `dependencyRanges` (package id → semver range). Neither list is required when saving; an empty package is refused when a version is created (`PUBLISH_EMPTY`).
`GET /v1/packages/{id}`	The draft with its modules and pages resolved (`missing: true` for a deleted one) and its package dependencies named.

Route	Purpose
`PUT /v1/packages/{id}`	Replace the draft. `Since dev` it takes the `updateIdentifier` the client last read and refuses a stale one — see Stale saves are refused.
`DELETE /v1/packages/{id}`	Only while the package has no version (`409 PACKAGE_HAS_VERSIONS` otherwise).

Validation codes on a save: `DISPLAY_NAME_REQUIRED`, `DEPENDENCY_RANGE_INVALID`, `MODULE_NOT_FOUND` and `PAGE_NOT_FOUND` (`404`), and `DESCRIPTION_TOO_LONG`. **The package description is also the store page's description**, so each language is limited to 120 characters when it is written (`400 DESCRIPTION_TOO_LONG`). A longer description stored before the limit existed is kept until that language is edited.

Stale saves are refused `Since dev`

The draft `PUT` replaces the whole draft and the README `PUT` the whole Markdown, so two people (or two tabs) editing one package used to overwrite each other without noticing. The development line adds an optimistic-concurrency token to both:

- **The token.** Each package carries an `updateIdentifier`, issued when the package is created and **rotated by every draft save and every README Markdown save — and only by those**. Background writes (store polling, publishing, the store-page draft) and adding or removing a README image leave it alone. A package created before the token existed has none until its first save.
- **Where it travels.** It is returned on the package list, the package detail and the `PUT` answer, and on the draft README (`GET /v1/packages/{id}/readme`). It is accepted in the body of `PUT /v1/packages/{id}` and `PUT /v1/packages/{id}/readme` (`{markdown, updateIdentifier?}`).
- **The contract.** A token that is **sent** and no longer matches is refused with `409 STALE_UPDATE_IDENTIFIER` ("…was changed by another save; reload it and redo your edit"). The check is made when the package is read and again inside the write, so a save that lands between the two cannot slip through. A token that is **omitted** is not checked, so clients that never send one keep working. Each `PUT` answers with the **new** token; the README `PUT` answers with the Markdown and token it wrote itself, never another save's.
- **In the web app.** The package page sends the token it last read and adopts a new one only when the rotation came from its own save, so an older tab's next save is refused rather than silently overwriting. The message asks the user to reload and redo the change.

Publishing a version

Publishing has a dry run. `GET /v1/packages/{id}/publish-preview?version=&minPlatformVersion=` lists every problem without publishing anything, along with the resource counts, the package dependencies it detected (with a default caret range), the external dependencies, the suggested next patch/minor/major version (`1.0.0` for a first publish), a `suggestedBump` computed from a content diff against the latest version, and `canPublish`.

`POST /v1/packages/{id}/versions` with `{version, releaseNotes, minPlatformVersion?}` answers `202 {jobId}` and runs as an export job: poll `GET /v1/export/{jobId}/status`, whose result then carries a `publish` block (`packageId`, `versionId`, `version`, `sha256`, `rootHash`).

The version rules:

- the version is **strict semver** and must be **higher than any version ever recorded** for the package, deprecated and pre-release versions included;

- **release notes are required**;
- `minPlatformVersion` defaults to the environment's own platform version and may only be **raised** — support can be narrowed, never widened;
- "latest" means the highest *published, non-pre-release* version.

A publish is refused with the first problem's code while any of these holds:

Problem	Meaning
`PUBLISH_EMPTY`	The package holds neither a module nor a page.
`PUBLISH_MODULE_MISSING`, `PUBLISH_PAGE_MISSING`	One of its modules or pages was deleted.
`PUBLISH_PAGE_NOT_GLOBAL`	A packaged page now belongs to a module or entity; a package carries only global pages.
`PUBLISH_MODULE_IN_OTHER_PACKAGE`	A module is claimed by another package.
`PUBLISH_MODULE_INSTALLED`, `PUBLISH_MODULE_FORKED`, `PUBLISH_PAGE_INSTALLED`, `PUBLISH_PAGE_FORKED`	A module or page came from an installation (forked or not); an installed template cannot be republished as your own.
`PUBLISH_SYSTEM_FUNCTION_REFERENCE`	A packaged resource calls a function that belongs to no module.
`PUBLISH_PACKAGE_DEPENDENCY_UNSUPPORTED`, `PUBLISH_DEPENDENCY_CYCLE`, `PUBLISH_DEPENDENCY_FORKED`, `PUBLISH_DEPENDENCY_RANGE_INVALID`, `PUBLISH_DEPENDENCY_REFERENCE_UNSUPPORTED`	A dependency on another package that cannot be expressed: unsupported, circular, forked, an invalid range, or a reference kind that cannot cross packages.
`PUBLISH_VERSION_INVALID`, `PUBLISH_VERSION_NOT_INCREASING`	The version is not strict semver, or not higher than every recorded version.
`PUBLISH_MIN_PLATFORM_INVALID`	The minimum platform version is malformed or lower than allowed.
`PUBLISH_RELEASE_NOTES_REQUIRED`	No release notes.
`PUBLISH_RESERVED_FIELD_PREFIX`	A field name uses the reserved `x_` prefix.

Other version routes: `GET /v1/packages/{id}/versions` (highest first) and `/versions/{version}`; `/versions/{version}/download` (the ZIP as an attachment); `POST /versions/{version}/deprecate` with a localized `message`, and `/undeprecate`; and `GET /v1/packages/{id}/compare?from=&to=`, where `to` may be `current` (the live modules) — it reports added, removed and changed resources and a `suggestedBump`.

External dependencies. Whatever the package references but does not carry is declared in the version's manifest as an external dependency, which the installer maps to something of their own. `Since dev` this also covers a view (or other resource) that a **page widget** shows from outside the package: it is declared without becoming a package dependency and is optional to map — see Page widgets pointing outside the package.

Comparison and integrity both rely on **canonical content hashes**: each resource is hashed with server-managed and environment-specific keys left out and ids replaced by package-relative keys, so the same content hashes the same way in any environment.

Deleting a version, the package's history and the export records belong to the file workflow and are described in Packaging & Distribution. One rule matters here: **deleting a version or an export removes its record only — the ZIP stays in storage**, because tenant backups list stored files without copying them.

Contents summary

Every exported manifest — plain export or published version — carries a `contents` summary: counts per resource type, a `modules[]` list with each module's entities and their field counts (implicit fields excluded), an `other` bucket for resources outside any exported module, and `truncated: true` when a cap is hit (**100 modules, 500 entities**). Names keep only `en` and `ar`. The installation stores the summary, and the store shows it as *What's inside*.

README

A package has a draft README — Markdown plus images — edited through `GET|PUT /v1/packages/{id}/readme` and `POST|GET|DELETE /v1/packages/{id}/readme/images[/{name}]`. Publishing freezes it into the version's ZIP. The limits:

Limit	Value
Markdown	256 KiB
Each image	2 MiB; PNG, JPEG, WebP or GIF
Images	at most 50, 20 MiB in total
Image names	lowercase letters, digits, `.`, `_` and `-`, at most 100 characters

A published version's README is read at `GET /v1/packages/{id}/versions/{version}/readme`; an import keeps the README it arrived with at `GET /v1/imports/{id}/readme`.

A version keeps the README it was created with. A README cannot be added to a version afterwards — that takes a new version — and the store page uses the README as the app's long description. The package page says so before the version exists (see In the admin area). The README `PUT` takes part in stale-save refusal `Since dev`.

The store page

The store page (the marketplace listing) is a separate draft on the package, edited through `GET|PUT /v1/packages/{id}/listing` and `POST /v1/packages/{id}/listing/images` (`kind` = `icon` or `screenshot`), and frozen into its own publish request with `POST /v1/packages/{id}/listing/publish-requests`. Its fields are a localized **store name**, **topics**, a **documentation URL**, a **maintenance** flag with a localized note, an **icon** and **screenshots** with captions.

- **Topics replace the old problem, features, contact and category fields.** Those four are stripped from any stored draft; a draft saved before topics existed gets its old category as its single topic. A topic is trimmed, lower-cased and has spaces or `_` turned into `-`; it must then be 1 to 35 letters, digits or hyphens starting with a letter or digit. Invalid entries and repeats are dropped when the page is sent, and at most **10** topics are allowed. A save that omits `topics` keeps the stored ones.
- **The description is the package's.** The store page has no description of its own: it always shows the package description, and one sent with a save is ignored. Sending still checks the 120-character limit and reports a problem under `description`. A store page that has been approved once is not sent without a description, because the revision would clear the store's.
- **Who writes it.** Any developer of the package, without a marketplace group. Sending it needs `publish`.

The limits mirror the marketplace's: store name 60 characters, description 120, documentation URL 2048 (http or https), maintenance note 2000, screenshot caption 140, at most 8 screenshots. Images are PNG, JPEG or WebP up to 2 MiB; the icon is square, **256 to 1024** px; screenshots are 16:9 (within 2 percent), **1280×720 to 3840×2160**. Saving checks shape and limits only, so a half-written page can be kept; sending checks the required fields (`422 LISTING_INVALID` with the issues listed). Once a store-page revision is approved the **name is locked** (`NAME_LOCKED`).

`POST .../listing/unpublish` and `.../republish` take an app out of the store and back. Both need `publish` and the plan entitlement; they are **not** gated by the publisher terms — only approval is.

Publish requests and the review inbox

Nothing goes to the marketplace directly. A publisher **prepares** a request with `POST /v1/packages/{id}/versions/{version}/submit` (with a `storeSlug` until the slug is locked), and a **reviewer in the same environment** approves it; only then does the platform upload anything.

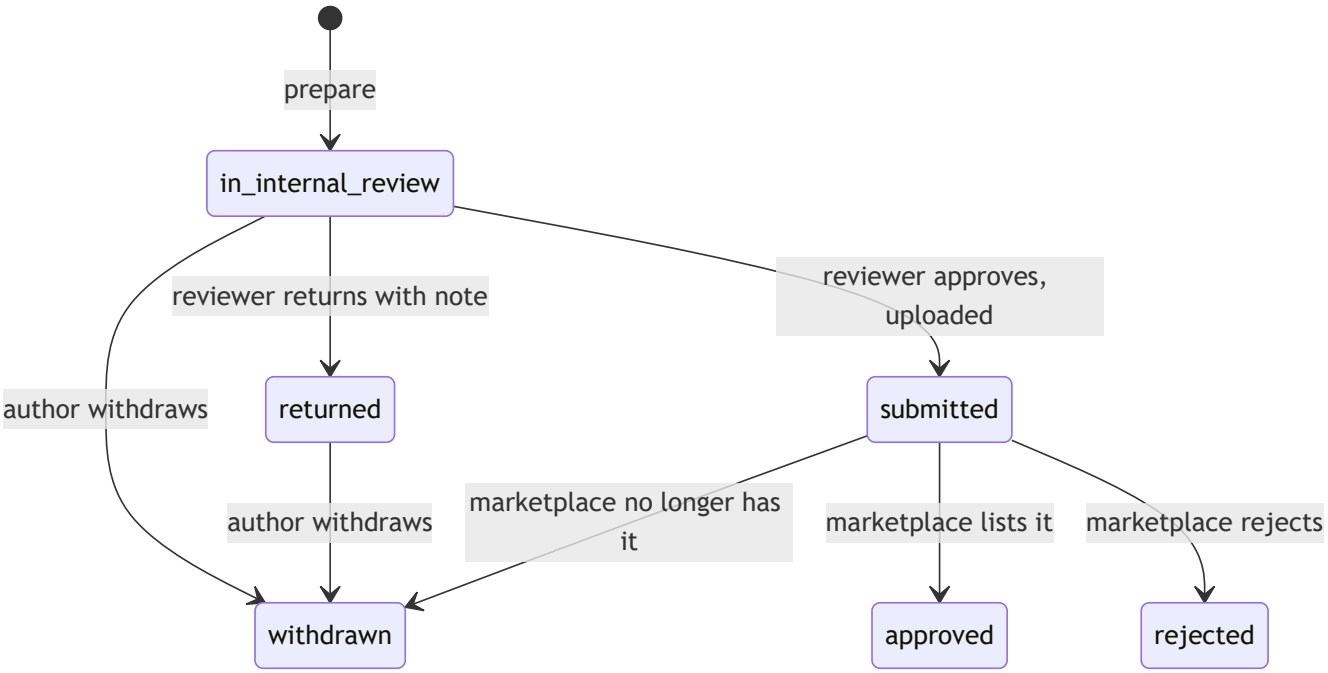
Preparing checks, in this order:

1. the `publish` permission;
2. the plan entitlement (`marketplacePublish`);
3. **a publisher on the live connection** — `409 MARKETPLACE_PUBLISHER_REQUIRED` ("This environment has no publisher in the marketplace yet. Set up publishing first."). A connection alone is not enough; see Connecting and setting up publishing.

A request is also refused while another marketplace publisher owns the app, for a deprecated version, and for a version that already has a marketplace submission. A version the marketplace **rejected** cannot be sent again: publish a new version.

The publisher terms are asked at approval, not when sending. Approving checks `review`, the plan, and then the terms: when the current terms are not accepted the answer is `409 MARKETPLACE_TERMS_REQUIRED`, and the approve dialog asks the reviewer to accept them (`POST /v1/marketplace/terms`, which needs `review`).

States



A request has a `kind` — `version` or `listing` — and at most one request per version and per store page can be open (`in_internal_review` or `submitted`) at a time (`409 LISTING_REQUEST_OPEN`; a transition from the wrong state is `409 PUBLISH_REQUEST_STATE`).

Route	Needs	Effect
`GET /v1/publish-requests?state=&skip=&limit=`	`review`	Every request in the environment, newest first, with its resource count.
`GET /v1/publish-requests/{id}`	`review`	The request with its package, version summary, README, frozen store page and, for a grouped request, the whole group.
`POST /v1/publish-requests/{id}/approve`	`review`, plan, terms	Uploads to the marketplace and records the submission. A claim left stale for more than 15 minutes can be re-taken.
`POST /v1/publish-requests/{id}/return`	`review`	Returns the request with a note of 1 to 1000 characters.
`POST /v1/publish-requests/{id}/withdraw`	`publish`	The author or a system administrator, from `in_internal_review` or `returned` — or a `submitted` request the marketplace no longer has.
`POST /v1/publish-requests/{id}/refresh`	`publish` or `review`	Re-reads the marketplace's state.

`GET /v1/packages/{id}/publish-requests` lists one package's requests (`publish` or `review`), and a reviewer reads a request's README images and store-page images through `GET /v1/publish-requests/{id}/readme/images/{name}` and `.../listing/images/{imageId}`. The reviewer who approved is recorded, including whether they were the author.

One review for a version and its store page

A version and the store page can travel as one **group**, reviewed and decided in one step:

Route	Effect
`POST /v1/packages/{id}/publish-requests/first-publish`	`{versionId, storeSlug?}`. **The first publish sends both** — the version and the store page of a package that has never been live. Refused with `409 ALREADY_LIVE` once the slug or the store name is locked or any request of the package has been approved.
`POST /v1/packages/{id}/publish-requests/together`	The same pair, whether or not the app has been live. Refused only while *that* version or the store page is already waiting for a decision.
`POST /v1/publish-request-groups/{groupId}/approve`	Approves the version, then its store page. A store page that fails stays in review and can be approved alone.
`POST /v1/publish-request-groups/{groupId}/return`	Returns the requests still in review with one note.
`POST /v1/publish-request-groups/{groupId}/withdraw`	Withdraws the requests not yet decided; every request must belong to the caller unless the caller is a system administrator.

Both requests pass the same checks before either is stored, so a failed check stores nothing. Returning or withdrawing a group moves none of its requests if one of them is being approved on its own at that moment. The sender gets one notification for the whole group.

Withdrawing a request the marketplace no longer has

When a `submitted` request is read again and the marketplace answers *not found*:

- while the events feed is in use, the feed — not the 404 — decides: the request is **marked** as gone but stays `submitted`, and a person may **withdraw** it. Withdrawing closes it as `withdrawn` with the reason

`MARKETPLACE_SUBMISSION_GONE` (a version) or `MARKETPLACE_REVISION_GONE` (a store page). A store-page request is not marked while the app is delisted;

- without the feed, the request is closed directly with the same reason.

A third reason, `MARKETPLACE_APPLICATION_MOVED`, closes the submitted requests of an app that moved to another publisher.

Notifications

The review flow is the first producer of **in-app notifications with a link** — the notification entity gained a `link` field (migration `1.10.0-a43-add-notification-link`) that may only point inside the app. Notifications are written in English and Arabic, in the ROOT space. Internal review steps are in-app only; **marketplace decisions, store changes and publisher changes also go by email** (an English-and-Arabic message whose link opens the package page). Every notification also tells open package pages to reload.

Event	Who is notified	Email	Link
Sent for review	Members of `MARKETPLACE_REVIEWERS`, except the author. With no reviewer, the system administrators are told a reviewer is needed.	no	`/admin/packages/reviews/{requestId}` (reviewer needed: the user-groups page)
Approved by a reviewer and sent	The author, unless they approved it themselves	no	`/admin/packages/build/{packageId}`
Returned with a note	The author, unless they returned it themselves; one message for a group	no	same
Approved or rejected by the marketplace	The author and every member of the publisher and reviewer groups; a rejection carries the reason	yes	the package page, or its store page when the decision concerns the store page
Store status changed (not live yet, live, delisted with its reason, back in the store, unpublished)	Same audience	yes	same
Store address changed or added, apps moved in or out	Publishers and reviewers	yes	`/admin/packages/build`
The marketplace ended the connection	Publishers and reviewers	yes	`/admin/packages/build`
A customer reported a failed install or update of one of your apps `Since 1.10` (from 1.10.1)	Publishers and reviewers	yes	`/admin/packages/build/{packageId}`

Links written before the Build page was renamed (`/admin/packages/mine/...`) are redirected by the web app to `/admin/packages/build/...`.

Connecting to the marketplace

Configuration

The packaging service reads three marketplace addresses: the API origin, the origin that package file transfers must use, and an optional web origin. Each must be a bare `https` origin (plain `http://localhost` is accepted outside production). In production an unset or malformed value means the marketplace is **off** — `503 MARKETPLACE_DISABLED` on every marketplace route. The web app needs both its marketplace API and web addresses to be valid before it shows any marketplace surface. How an operator enables the connection, and which marketplace it points at, is deployment configuration; at the chart level `marketplace.enabled` defaults to `false`.

The **name an environment registers and publishes under** comes from the tenant: in a multi-tenant cell, the tenant's display name in the tenant registry; in a single-mode deployment, the `TENANT_DISPLAY_NAME` setting, falling back to the host of `APP_BASE_URL`. A deployment with no usable host answers `422 BASE_URL_INVALID`.

Registering the environment

Connecting needs **no marketplace account** and no hand-off between sites. `POST /v1/marketplace/connection` registers the *environment*, under the tenant: it sends the environment id, the tenant's display name and company, the environment's packaging and web addresses, the platform version and the package schema versions it supports. The marketplace answers with a connection key, which is stored encrypted, and with the signing keys used to verify downloads. Connecting is idempotent (a live connection is returned unchanged), and nothing connects automatically.

- **Connecting creates no publisher.** A new registration is **install-only**: the connection is live with no publisher and no store address, and the store works for installing. Only an environment that was already a publisher at the marketplace gets its publisher back on connect. Publishing is a separate step — setting up publishing.
- `GET /v1/marketplace/connection` returns the stored connection — `status` `active`, `unreachable` or `revoked`, the publisher, store address, scopes, `registrationConflict`, `connectedAt`, `namespaceLocked` (the store address can no longer change once anything has been approved) and `publishingRevoked` (`null`, or the marketplace's reason and date while it has revoked publishing) — and, while there is no live connection, a preview of what connecting would share. It also returns a **publisher suggestion** while the environment has no publisher: the last publisher name after a revoke, otherwise the environment's name, and the first label of the host as the store address when that is a valid address.
- **When the preview fails** (for example, the tenant host does not form a plain https address), the response carries `registrationError: {code}` instead of hiding the connection, and the card still offers **Connect**, which reports the full error.
- **Already registered.** If the marketplace reports the environment as already registered, the connection is marked `registrationConflict` and is not retried automatically.
- **Rate limits.** Marketplace rate limits surface as `429 RATE_LIMITED` with a `Retry-After` header.

Connecting and setting up publishing

Connecting and setting up publishing are **two separate steps**:

1. **Connect** — `POST /v1/marketplace/connection`. Needs `install` or `publish`, so anyone who may install can connect, and every administrator can. Installing from the store connects the environment first when it is not connected yet (see Installing from the store connects first).
2. **Set up publishing** — `POST /v1/marketplace/publisher` with `{publisherName, namespace?}`, on the live connection. Needs a system administrator or a `publish` holder, and the plan entitlement. The environment chooses its **publisher name** (1 to 60 characters, no control or formatting characters) and, optionally, its **store address**; the marketplace may assign a different address than the one suggested. An invalid name answers `422 PUBLISHER_NAME_INVALID` (also when the marketplace refuses the name, with its message), an invalid address `422 NAMESPACE_INVALID`, and a connection without publishing rights `403 PUBLISH_NOT_ALLOWED`. A connection that already has a publisher is returned as it is.

After set-up, **only system administrators** change what was chosen:

Route	Effect
`PUT /v1/marketplace/publisher`	`{name}`. Renames the publisher; the store address is untouched.

Route	Effect
`PUT /v1/marketplace/namespace`	`{name}`. Changes the store address while the marketplace still allows it (`namespaceLocked` false). Addresses are 1 to 50 lowercase letters, digits or hyphens, starting with a letter or digit.
`GET /v1/marketplace/namespaces/{name}`	Checks an address: `available`, plus `TAKEN`, `INVALID` or `LOCKED` (an administrator or `publish`).
`GET /v1/marketplace/terms`, `POST /v1/marketplace/terms`	The publisher terms and their acceptance (`{version}`, which must be the marketplace's current one). Read with `publish` or `review`; accept with `review`.

Revocation, heartbeat and reporting

- **Revocation.** When the marketplace revokes the connection, the environment ends it locally and the call answers `409 MARKETPLACE_NOT_CONNECTED`, so the store can offer Connect again. Publishers and reviewers are notified. Connecting again registers anew under the **last publisher name** and starts the events feed afresh; the apps published under the old publisher stay with it, so the marketplace then reports another owner and those packages are refused for sending.
- **Heartbeat.** A daily job sends the platform version and the supported schema versions.
- **Reporting.** Installs, uninstalls and forks are reported to the marketplace in the background.

Publishing revoked by the marketplace `Since 1.10`

Marketplace staff can stop one publisher from publishing **without ending the connection**. This arrives on the 1.10 line after 1.10.1 (it ships from **1.10.2**). Installing and updating apps keeps working; only publishing stops.

- **Where it comes from.** The marketplace's description of the connection may say that publishing is revoked, with a reason and a date. Each refresh of the connection stores that as `publishingRevoked` `{reason, revokedAt}` (an absent reason or date is stored as an empty string), or `null` when publishing is not revoked. A marketplace older than the feature never sends it.
- **Where it surfaces.** `GET /v1/marketplace/connection` always returns `connection.publishingRevoked`. `GET /v1/marketplace/me` returns `publishingRevoked` from the last stored refresh (`null` for a caller with none of `install`, `publish` or `review`); a `me` call on a stale connection starts a background refresh (at most one a minute), so the package page learns of a revocation without anyone opening the connection card.
- **Cleared** when the connection is revoked and when a new registration starts, so a reconnect never shows an old revocation.
- **Not enforced by the platform.** The packaging service does not refuse a send, an approval or a rename because of it: the marketplace refuses every submission while publishing is revoked, and that refusal surfaces as the usual `502 MARKETPLACE_REFUSED`. No event or notification announces a revocation; an open package page sees it when it reloads (it reloads on every package change).
- **In the web app.** The package page shows a warning with the revocation date, the marketplace's reason as written, and "New versions and store page changes can't be sent until the marketplace restores publishing. Installing and updating apps still works." *Send for review* is disabled. The connection card shows a *Publishing revoked* tag (the reason as its tooltip) and the same notice, and stops offering rename and store-address editing.

Events and store status

The marketplace never calls the environment, so the environment reads from it. Four per-tenant background jobs are scheduled when a connect succeeds, and again at start-up for every tenant with a live connection:

Job	Period	What it does
Heartbeat	24 h	Reports the platform and schema versions.
Events read	60 s	Reads the marketplace's events feed and applies it. Runs only once the connection has a publisher.
Decision poll	15 min	The safety net behind the feed: re-reads submitted requests and store statuses.
Update check	6 h	Refreshes, for every installation in the ROOT space, whether a newer version is listed.

The events feed

The feed is a cursor over the marketplace's events for this publisher. The cursor and the feed's health live on the connection; one run per tenant holds a lease at a time, and the cursor is saved after each step, so a failed step is retried by the next run (and skipped after ten consecutive failures). A first read, a return after the feed was unavailable, or a *resync* page starts at the head of the feed and runs a full catch-up pass instead of replaying the gap. While the feed is in use and fresh (read within the last 5 minutes), the decision poll only checks due store statuses and runs a full pass every 6 hours; without a fresh feed it polls up to 50 submitted requests each time, as before.

Event	Effect
A submission or store-page revision was decided	The matching `submitted` request becomes `approved` or `rejected`, and the audience is notified (once for a group).
The app was **delisted**	Records the store status with the marketplace's **reason** and notifies "Removed from the store". The reason is kept on the package while it stays delisted.
The app was **relisted**	Records the new status and notifies "Back in the store", with anything still missing.
The store address changed or was added	Refreshes the connection and tells the publishers.
Apps **moved in** to this publisher	Clears the "owned by another publisher" mark of those packages and tells the publishers.
Apps **moved out** to another publisher	Marks those packages as owned by another publisher, closes their submitted requests (`MARKETPLACE_APPLICATION_MOVED`) and tells the publishers.
A customer environment sent a **problem report** `Since 1.10` (from 1.10.1)	Notifies the publishers and reviewers in-app and by email (see What the publisher receives). A report whose details the marketplace has already discarded is skipped; when nothing could be delivered the step fails, so the next run retries it without duplicating.

Store status on the package

Each package keeps the last store status the marketplace reported, returned as `store` on every package summary (empty until one has been read): `live`, `maintenance`, `unpublished`, `delisted` or `not_live`. While `not_live`, it lists the steps still missing — the version, the store page, or both — each either missing or in review; a delisting carries its reason. The status is written by the decision poll's store checks, by decisions, by the events above, and right after an unpublish or republish.

Only real changes are announced, and not all of them: going live is announced only when the environment saw the earlier non-live status; a delisting or unpublish seen for the first time is silent, and an unpublish this environment did itself is not announced. A step newly missing announces "Not in the store yet" with what to send.

App owned by another publisher

The environment asks the marketplace who owns each package id. When **another publisher** owns it — for example after this environment reconnected under a new publisher, or after the app was moved — the package is marked, and **every send is refused** with `409 PACKAGE_OWNED`: preparing a version, a group or a store page, and again just before the

upload at approval. When the marketplace cannot answer, the stored mark decides. The mark is cleared when the app moves back or the marketplace answers that this publisher (or nobody) owns it.

Installing

There are two ways in: from the **store**, or by importing a **published-version file** (a ZIP whose manifest carries a `package` block) through the normal import wizard. Both end in the same install mode and the same installation record.

The store routes

All of them need the ROOT space and the `install` permission (`/topics` also accepts `publish`).

The store is browsable without a connection. When the environment is not connected — never connected, or revoked — the catalog, the publishers and the app pages are read anonymously from the marketplace's public catalog, sending the platform version and supported schema versions with each read. Only installing needs a live connection.

Route	Behaviour
`GET /v1/marketplace/catalog?q=&cursor=&limit=&publisher=&topic=&sort=&lang=`	A page of store items (24 by default, at most 50). `publisher` filters by store address and `topic` by one topic; `sort` is `name` (the default), `newest`, `updated` or `installs`, and `lang` (`en` or `ar`) is the language a name sort uses. Each item says whether it is **installed** here (installation, version, status, source, `updateAvailable`), whether it is **blocked** (the platform is too old, or the package is incompatible) and, when the marketplace counts it, its **install count** — how many live environments have it installed now. An invalid parameter answers `400 STORE_REQUEST_INVALID`, a stale cursor `400 STORE_CURSOR_INVALID`.
`GET /v1/marketplace/publishers`	The store's publishers — store address, name, verified flag and app count. Empty while the marketplace is off or unreachable.
`GET /v1/marketplace/topics?q=&limit=`	Topics used by published store pages, most used first, as suggestions (`q` is a prefix).
`GET /v1/marketplace/catalog/{publisher}/{slug}`	The app page: store-page fields and topics, README, the latest version (minimum platform, contents, resource types, dependencies with the locally installed version, external dependencies) and the version list. `404 STORE_APP_NOT_FOUND` when the app is not in the store.
`POST /v1/marketplace/catalog/{publisher}/{slug}/install-check`	Body `{version}`. Would this version install now? See Store install check.
`POST /v1/marketplace/catalog/{publisher}/{slug}/install`	Body `{version}`. `202 {installationId}` when the install started; `200 {sessionId}` when a person has to make choices on the import page. Needs a live connection (`409 MARKETPLACE_NOT_CONNECTED`).
`GET /v1/marketplace/packages/{packageId}/versions/{version}/readme/images/{name}`	A README image of a store version, proxied.

Install from the store is idempotent per user, package and version for 24 hours: a repeat returns the existing installation, and a concurrent attempt answers `409 IDEMPOTENCY_KEY_IN_PROGRESS`.

Store install check

`install-check` answers "would this version install now?" without installing anything: it fetches and verifies the package exactly as an install does, stages it, applies the safe defaults, and always discards the session. The answer says:

Field	Meaning
`installable`, `refusal`	Whether it would install, or the first hard refusal. An app **already installed** here is refused with `INSTALL_ALREADY_INSTALLED` — the store offers the update instead.
`collisions`	The items **blocking** the install: each colliding name with its resource type and whether it collides in this space or elsewhere in the environment (a data-entity name must be unique across the environment).
`choices`	The unresolved choices an installer would have to make.
`publishedHere`	This environment built and published the package, so its resources are already here.
`forkedHere`	This space **forked** an earlier install of the app; its items are now the space's own, keep their names, and block a fresh install until renamed or deleted.

A user without administrator rights cannot finish an install that has unresolved choices, so the app page shows the choices instead of enabling **Install**.

Installing from the store connects first

On the app page, the install check runs when the page opens. On **Install**, when the environment is not connected yet, the page asks the user to confirm in a *Connect to the marketplace* dialog that shows what connecting shares; it then connects and runs the install. Connecting this way creates no publisher.

What happens when you press Install

The platform downloads the listed version (at most **200 MiB**), verifies its signature, its `sha256` and its manifest hash against the signed catalog entry (`422 MARKETPLACE_SIGNATURE_INVALID`, `MARKETPLACE_HASH_MISMATCH`, `MARKETPLACE_PACKAGE_TOO_LARGE`), and stages it as an install-mode import session with safe defaults — exact-match suggestions are accepted for value references.

If choices remain — external dependencies, principals or package dependencies the defaults cannot settle — a **system administrator** is handed the session to finish on the import page; any other installer is refused with `409 UNRESOLVED_CHOICES`, which lists the choices. Hard refusals (wrong package, platform too old, name collisions, a dependency missing or out of range) are raised before anything is written. The store publisher is recorded on the installation and on the import record.

Install mode

An import becomes an install when the validated package has a `manifest.package` block. Install mode adds these checks to validation:

Code	Raised when
`INSTALL_MANIFEST_INVALID`	The `package` block is malformed.
`INSTALL_PLATFORM_TOO_OLD`	The package's `minPlatformVersion` is higher than this platform.
`INSTALL_DEPENDENCY_MISSING`, `INSTALL_DEPENDENCY_OUT_OF_RANGE`	A package it depends on is not installed in this space, or is installed at a version outside the declared range (pre-releases count).
`INSTALL_REFERENCE_UNRESOLVED`	A cross-package reference does not resolve here.
`INSTALL_ALREADY_INSTALLED`	The package is already installed in this space — update it in place instead.

Code	Raised when
`INSTALL_NAME_COLLISION`	A resource name already exists here, or lives in another space of the environment.

`INSTALL_VERSION_DEPRECATED` and undeclared external dependencies are warnings, not errors. A **failed** installation of the *same version* is taken over by a new session, which is what makes a re-run safe.

While installing, every mapping creates new resources; `TARGET_ARCHIVED` and `TARGET_INSTALLED` are refused up front, as is a value reference left without an action. Each resource written is recorded in the installation's ledger, and a **lock row** is kept for it. A ledger write that keeps failing stops the install, and a failed install keeps its lock rows, so what it created stays protected.

Installation statuses are `installing`, `active`, `updating`, `failed`, `forked`, `uninstalling` and `uninstalled`.

Route (per space, space admin)	Purpose
`GET /v1/installations?status=&skip=&limit=`	List (50 by default, at most 100); `status` takes a comma-separated list, an unknown value is `400 INVALID_STATUS_FILTER`.
`GET /v1/installations/{id}`	The installation with its resources and their **integrity** (`ok`, `changed` or `missing` against the shipped hash), `platformAdjusted` (the target server normalised the resource on write), dependencies, dependents and the contents summary.
`GET /v1/installations/{id}/status`	A cheap poll: `{id, status, lastError?}`.
`GET` / `POST /v1/installations/{id}/problem-report`	`Since 1.10` (from 1.10.1). Preview and send a problem report about a failed store install or a stopped store update to the app's publisher — see Reporting a failed install or update.
`POST /v1/installations/origins`	`{targetIds}` (1 to 500): which installation owns each target, and whether it is `readOnly`.

Installed resources are read-only. data-service and reaction-service refuse an update or delete of a resource that an installation locks with `403 PACKAGE_RESOURCE_READ_ONLY` ("fork it to edit") — for every caller, system administrators included; only packaging-service's own writes are exempt. The lock lasts until the installation is forked. A plain import that would overwrite such a resource is blocked with `TARGET_INSTALLED`. A plain export refuses installed resources as well (`403 EXPORT_INSTALLED_RESOURCE`), and an installed module cannot be published as part of your own package.

Updating an installed app in place

An installed marketplace app is **updated in place** to a newer listed version — no uninstall and reinstall. The update routes need an environment administrator in the ROOT space plus the `update` permission (the installers group):

Route	Effect
`POST /v1/installations/{id}/update-sessions`	`{version}`. Stages the newer version as an update session: `201 {sessionId}`, or `200 {sessionId, choices}` when choices remain. Refused with `409 UPDATE_NOT_NEWER`, `409 INSTALLATION_FORKED` or `409 UPDATE_IN_PROGRESS`.
`GET /v1/installations/{id}/update-sessions/{sessionId}`	The update preview, including the catalog's release notes between the installed and the target version.
`POST /v1/installations/{id}/update`	`{sessionId}`. `202`: runs the update (idempotent per user, installation and session).

Route	Effect
`POST /v1/installations/{id}/update/retry`	`202`: retries a failed update.
`POST /v1/installations/updates/check`	Refreshes "update available" now (at most once a minute; the 6-hour job runs regardless).

`active` and `failed` installations can take an update — a failed one is finished by a fixed newer version. The store's `installed.updateAvailable` comes from the same check. How an update is planned, and how the client's own changes are carried over, belongs to the update mechanics in Packaging & Distribution. In the web app the update preview opens at `/admin/packages/installed/installations/{id}/update`.

Fork and uninstall

- **Fork** — `POST /v1/installations/{id}/fork` with `{confirm: true}` — detaches an *active* installation in place. Its resources become editable and keep their provenance, but they no longer follow the package, no longer receive updates, and can never be exported or published. A fork is refused while another installation depends on this one (`409 FORK_BLOCKED_BY_DEPENDENTS`) and unless the installation is active (`409 INSTALLATION_STATUS_CONFLICT`).
- **Uninstall preview** — `GET /v1/installations/{id}/uninstall-preview` (environment administrator) — counts what would be deleted per type, the entries per store, the packages that depend on this one, and the caller's own resources that would break.
- **Uninstall** — `POST /v1/installations/{id}/uninstall` with `{confirm: true}` (environment administrator) — is allowed from `active`, `failed`, `forked` or `uninstalling`. A background job deletes the installation's resources and their entries in dependency order and ends at `uninstalled`. A module that still holds entities **your users created** is kept, and an uninstall is refused while items outside the package still reference its modules. If the job fails, the status stays `uninstalling` with `lastError.code = UNINSTALL_FAILED`, and calling uninstall again resumes it.

The web app treats uninstall like deleting a module: the dialog lists the links from **outside** the package that must be removed first — dependency-service answers this with `GET /v1/external-dependents/{type}/{id}` — and keeps **Uninstall** disabled until every one is unlinked and the confirmation is ticked.

Reporting a failed install or update to its publisher `Since 1.10`

An administrator whose store install failed, or whose store update stopped, can send the app's publisher a **problem report** from the installation page. It ships from **1.10.1**.

Route	Effect
`GET /v1/installations/{id}/problem-report`	The preview: exactly what would be sent, so the administrator can review it first.
`POST /v1/installations/{id}/problem-report`	Sends it and answers `{at, by, errorAt}`, which is also stored on the installation as `lastProblemReport`.

Both need an **environment administrator** in the ROOT space and a **live marketplace connection** (`409 MARKETPLACE_NOT_CONNECTED` otherwise). They are **not** gated by `publish` or by the publishing plan entitlement: reporting a problem is part of installing.

What can be reported. Only a store installation (`source: marketplace`) whose last error is not a cancellation (a code ending in `_CANCELLED`) and that is either `failed` (an install problem) or `updating` with a pending update (a stopped update). Anything else answers `409 PROBLEM_REPORT_NOT_AVAILABLE`.

The preview carries the package id and version (for an update, the target version and the installed one), the error (code, message and time), the **failed resources** of the run — type, source id, name and message, read from the run's import session — with a count of those left out, `detailsExpired: true` once that session no longer exists (only the overall error can then be reported), the recipient (the app's publisher as the marketplace knows it now), the caller's own name and email as a contact preview (`null` without an email), and a fresh `reportKey`.

Limits (text is cut on a character boundary): error message 1000 characters, **50** resources, resource message 500, resource name 300, note **2000**.

Sending. The body is `{note, shareContact, errorAt, recipientPublisherId, reportKey}`; all five are required (`note` may be `null`). The server never trusts the client's copy: it re-reads the installation and rebuilds the report.

1. An `errorAt` that is no longer the installation's last error answers `409 REPORT_STALE` — the error changed since the preview, so preview again.
2. `shareContact: true` for a caller without an email answers `400 CONTACT_UNAVAILABLE`.
3. The report — with the note (a blank note is sent as `null`) and the contact only when shared — goes to the marketplace with `reportKey` as its idempotency key.
4. On success `lastProblemReport` is stored without touching the installation's `updatedAt`. Nothing prevents a second report; the web app treats a report as current while its `errorAt` still equals the installation's last error.

Marketplace refusals pass through as `409` with the marketplace's message: `NOT_INSTALLED` (the marketplace has no record of this environment downloading the app), `RECIPIENT_CHANGED` (the app has a new publisher — preview again), `PUBLISHER_UNREACHABLE` and `PUBLISHER_CANNOT_RECEIVE` (the publisher cannot receive reports yet). A marketplace rate limit becomes `429 RATE_LIMITED` with `details.retryAfterSeconds` (the marketplace's value, else 60).

Installation summaries now also carry `publisher` (`namespace`, `name?`) and `lastProblemReport`.

What the publisher receives

The report reaches the publishing environment through its events feed. That environment tells every member of `MARKETPLACE_PUBLISHERS` and `MARKETPLACE_REVIEWERS`, in-app and by email, with a link to the package page. The title is "Problem report: *app version*"; the body gives the failed install or update with its code and message, the failed resources (whole lines only, then "…and N more"), a line saying the details had expired when they had, the reporter's note, "Reply to *name* <*email*>" or "The reporter did not share contact details.", and the installation id with the reporter's platform version. The text is capped at 4000 characters and every reported value is sanitised. Unlike other marketplace notifications, it does not make open package pages reload.

In the web app. On the installation page an environment administrator sees **Report to *publisher*** when the installation is a store install with a known publisher, its last error is not a cancellation, and it has failed or its update stopped. The dialog loads the preview, lists the error and the failed resources, offers an optional note and a "Let *publisher* reply to me at *email*" checkbox (only when there is a contact), and sends. A rate limit says when to try again; a changed recipient, a stale error or a missing contact reloads the preview. After a send the page shows "Reported to *publisher* on *date* by *person*" while the report is current.

Import records

`GET /v1/imports?skip=&limit=&source=&q=&state=` is the space's import history (50 by default, at most 100). `source` is `marketplace`, `package-file` or `import`; `q` searches (at most 100 characters); `state` filters on the server, before

paging and in the total:

- `installed` — plain imports, and installs whose installation is `installing`, `active`, `updating`, `failed` or `forked` (a running or failed install has resources in the space, so it counts as installed);
- `failed` — failed installations only;
- `uninstalled` — installs that are `uninstalled` or `uninstalling`.

Unknown values answer `400 INVALID_STATE_FILTER` or `400 INVALID_SOURCE_FILTER`. Each record carries the installation status, the store publisher for a store install, who imported it and per-type counts. `GET /v1/imports/{id}/resources` lists what an import wrote, and `POST /v1/imports/origins` says where each target resource came from — the most recent import wins — with the exporting environment, space and user taken from the package's `origin` block.

In the admin area

Packaging lives under `/admin/packages`; the earlier `/admin/packaging` URLs redirect there and keep their query string, so bookmarks and older links still land correctly. The authored-packages page is **Build** at `/admin/packages/build`; its earlier address `/admin/packages/mine` (and anything below it, including older notification links) redirects there.

Route	What it is	Who sees it
`/admin/packages/store` (+ `/{publisher}/{slug}`)	**Store**: the catalog and an app page with Install, Open, Update or Uninstall	Marketplace enabled; inside the page, ROOT space and `install`
`/admin/packages/installed` (+ `/import`, `/installations/{id}`, `/installations/{id}/update`, `/{importId}`)	**Installed**: import history, the import wizard, an installation's page, the update preview	System `AdminAccess`
`/admin/packages/build` (+ `/exports`, `/export`, `/{packageId}`, `/{packageId}/store-page`)	**Build**: your authored packages and your exports (two tabs), the export page, a package's page (Contents and Versions) and its store-page editor	A Developer on a module, or a `publish` holder; the export page needs a module Developer; the store-page editor needs the marketplace enabled
`/admin/packages/reviews` (+ `/{requestId}`)	**Reviews**: the review inbox and a request	`review`; marketplace enabled

In the menu, the **Packages** group holds four items in this order: **Store** (when the marketplace is configured and the user has `install`), **Installed**, **Build** (labelled *Build*, not "My packages") and **Reviews** (when the marketplace is configured and the user has `review`). Build and Reviews follow the marketplace groups rather than an access right, so a publisher or reviewer without a system grant still sees them. When the store cannot be shown, the page says why, in this order: ROOT space only, loading, marketplace unreachable, or no install permission (with a link to the user-groups page). There is **no "not connected" state**: browsing needs no connection.

A few behaviours worth knowing when you build on this:

- **Browse the store.** Search, **Publisher** and **Topic** quick filters and a sort — **Name**, **Newest**, **Recently updated**, **Most installed** — are kept in the address (`q`, `publisher`, `topic`, `sort`), so a link carries them. Cards show how many environments installed the app.
- **The app page** follows the marketplace's own layout (overview with README, at a glance, what's inside, screenshots, details, changelog). The install check runs when it opens; its refusal, collisions and the forked-here and published-here messages are shown, with *Check again*.
- **One publishing workflow on the package page.** Contents → version → store page → *Send for review*, which sends the version and the store page together. Unpublish and Republish sit in the header. The page waits for

publishing to be set up (a missing publisher blocks the store steps and points to the set-up), shows the store status and a delisting reason, shows the "owned by another publisher" notice (with no send offered), reloads when the package changes elsewhere, and offers **Withdraw** on a request the marketplace no longer has. The store-page editor saves or cancels from its own bar before sending, and explains why the store name can differ from the package name.

- **Set up publishing** is a card on the Build page. An administrator or a `publish` holder enters the publisher name and an optional store address (checked as it is typed); an administrator can later rename the publisher or change the address in place. A user who may not set it up is told to ask a publisher or administrator. After a revoke the card says the existing apps stay with the old publisher, and offers Connect.
- **Terms at approval.** The approve dialog asks the reviewer to accept the publisher terms when they are not accepted yet.
- **Developers can export without a system grant.** A Developer on a module reaches Build and the export page, and the export tree offers only the roots that user can develop — the same rule the server applies.
- **Locked resources say so.** A module or entity that came from an installation shows **Customise** and **View installation** as header actions. Customise offers the ways forward that keep the installation intact — add your own field, hide what you do not use, build your own view, ask the publisher — and, as a last resort, fork.
- **Installed** is filtered from the toolbar — *Status* and *Source* selects beside the search and sort (see Store and Installed polish) — and shows who installed each package (*Installed by*) and, for store installs, its publisher (*From*). An installed store app with a newer listed version offers **Update** on its installation page and on its store app page; both open the update preview. A failed or stopped store install offers **Report to *publisher*** on its installation page `Since 1.10` (from 1.10.1; see Reporting a failed install or update).
- **Create package** is offered in the Store header and empty state to module developers (it opens Build with the create form open), and **How packages work** opens a short guide in the Help panel.
- **README before the version** `Since 1.10` (from 1.10.2). Because a version keeps the README it was created with, the package page says so before the version exists. To a user who can edit, the README section shows a notice that the next version keeps a copy of this README when there is one, and a warning to add the README before creating a version when there is none; the *Create version* dialog says "This version will have no README" with an **Add README** button that closes the dialog and opens the README editor. **The missing-README warning appears only once the README is known to be missing** — a read that returned empty Markdown or *not found*, or a save — so a README that merely failed to load triggers no warning.
- **Pasted web images are copied into the package** `Since dev`. Pasting a web (`http(s):`) or `data:` image into the README editor makes the browser download it and store it as an ordinary README image, so it travels with the package and counts towards the README image limits. The download sends no credentials and no referrer, gives up after 15 seconds, and decides the type from the bytes (PNG, JPEG, GIF or WebP), never from the server's declared type. If the copy fails — no cross-origin access, offline, an error status, an unrecognised type or a refused upload — the image becomes a link to its source and the editor says it could not be copied ("Download it and use Add image instead"). Web images already in a README open in the editor as the links readers see, and saving keeps them as links.
- **Publishing revoked** `Since 1.10` (from 1.10.2): the package page and the connection card show the marketplace's revocation and reason and stop offering to send — see Publishing revoked by the marketplace.

Store and Installed polish `Since 1.10`

These web-app refinements are part of the 1.10 release:

- **Installed is filtered from the toolbar, not tabs**: a *Status* select (Installed, Failed, Uninstalled) and a *Source* select beside the search and sort. Both start empty, so every status and source shows; there is no *All* option, and

the sort offers the column headers' fields.

- **One icon tile for store apps** — a bordered tile the image fills, showing the first letter of the name when there is no icon or it fails to load — in the store list, the app page header (icon and name together) and the Build page's store preview. The store's card fills the page however short the list is.
- **Explanations in the set-up and connect steps**: the *Set up publishing* card says what a publisher profile is and what each field is for, and previews an app card "by" the typed name and the publisher page address; the install-connect dialog says why installing connects the environment and that connecting publishes nothing.
- **Smaller fixes**: a compact quick-filter search box aligned with the toolbar; a single sort keeps its field after *Reset*; a reload that finds a store app gone shows the store's not-found state.

Error codes

The marketplace-related codes a client can meet:

Status	Codes
`400`	`MARKETPLACE_ORIGIN_INVALID`, `STORE_REQUEST_INVALID`, `STORE_CURSOR_INVALID`, `UPDATE_REQUEST_INVALID`, `DESCRIPTION_TOO_LONG`, `CONTACT_UNAVAILABLE` (problem report, from 1.10.1)
`403`	`MARKETPLACE_PERMISSION_REQUIRED`, `MARKETPLACE_PUBLISH_NOT_IN_PLAN`, `PUBLISH_NOT_ALLOWED`, `PACKAGE_DEVELOPER_REQUIRED`, `EXPORT_INSTALLED_RESOURCE`, `PACKAGE_RESOURCE_READ_ONLY` (data-service, reaction-service)
`404`	`STORE_APP_NOT_FOUND`, `STORE_VERSION_NOT_FOUND`
`409`	`MARKETPLACE_NOT_CONNECTED`, `MARKETPLACE_CONNECTION_CHANGED`, `MARKETPLACE_TERMS_REQUIRED`, `MARKETPLACE_PUBLISHER_REQUIRED`, `PACKAGE_OWNED`, `ALREADY_LIVE`, `UNRESOLVED_CHOICES`, `IDEMPOTENCY_KEY_IN_PROGRESS`, `PUBLISH_REQUEST_STATE`, `STORE_SLUG_IN_USE_BY_OPEN_REQUEST`, `LISTING_REQUEST_OPEN`, `FORK_BLOCKED_BY_DEPENDENTS`, `INSTALLATION_STATUS_CONFLICT`, `UPDATE_NOT_NEWER`, `UPDATE_BREAKS_DEPENDENTS`, `UPDATE_IN_PROGRESS`, `UPDATE_SESSION_EXPIRED`, `INSTALLATION_FORKED`; the problem-report codes `PROBLEM_REPORT_NOT_AVAILABLE`, `REPORT_STALE` and, from the marketplace, `NOT_INSTALLED`, `RECIPIENT_CHANGED`, `PUBLISHER_UNREACHABLE`, `PUBLISHER_CANNOT_RECEIVE` (from 1.10.1); `STALE_UPDATE_IDENTIFIER` on a stale package draft or README save (`Since dev`)
`422`	`MARKETPLACE_SIGNATURE_INVALID`, `MARKETPLACE_HASH_MISMATCH`, `MARKETPLACE_PACKAGE_TOO_LARGE`, `MARKETPLACE_ORIGIN_NOT_ALLOWED`, `IDEMPOTENCY_KEY_REUSED`, `PACKAGE_FORMAT_OUTDATED`, `PUBLISHER_NAME_INVALID`, `NAMESPACE_INVALID`, `BASE_URL_INVALID`
`429`	`RATE_LIMITED` (with `Retry-After`; on a problem report, with `details.retryAfterSeconds`)
`502`	`MARKETPLACE_UNREACHABLE`, `MARKETPLACE_REFUSED` (with `details.upstreamCode`)
`503`	`MARKETPLACE_DISABLED`, `MARKETPLACE_STORAGE_UNAVAILABLE`, `MARKETPLACE_KMS_UNAVAILABLE`

Some refusals come from the marketplace itself and are passed through with its message — among them `SLUG_TAKEN`, `SLUG_LOCKED`, `NAME_LOCKED`, `NAMESPACE_TAKEN`, `NAMESPACE_LOCKED`, `LISTING_INVALID`, `MEDIA_INVALID`, `REVISION_PENDING`, `NOT_LIVE`, `README_INVALID`, `MANIFEST_INVALID` and `ENVIRONMENT_REGISTERED`.

These routes are documented here by hand: packaging-service and dependency-service do not publish an OpenAPI description, so they do not appear in the generated backend reference.

Functions: run your own code

This page is for developers who need logic the no-code tools cannot express: calling an external service, a calculation, a custom validation. You write a small TypeScript function, upload it, and the platform builds it, deploys it and calls it when an automation runs, when a record is saved, or when a request comes in.

When to use a function

You need…	Use
To react to an event with built-in actions (create a record, send an email, call a URL)	A reaction [Build Apps › Reactions & automation] — no code
A computed field, a dynamic value, a template token	The built-in expression engine — no deployment
Your own code, external network calls, or heavier computation	**A function**

A reaction can call a function when it needs custom logic, so the two combine well.

How it works

- A function is a record in **Administration › Applications › Functions**. Its source is a ZIP file you upload.
- When you save it, the platform builds the source and deploys it as a serverless service inside the cluster. Its **Status** moves from **Provisioning** to **Provisioned** (or **Build failed**), and its URL is filled in. Only then can it be called.
- The platform calls it over HTTP using the CloudEvents format and reads the JSON you return.
- Uploading new source rebuilds it; archiving the record removes the deployment.

A function has one of four **Type**s, which decides who calls it and with what:

Type	Called	It receives	It returns
Reaction Custom Action	By a reaction's **Execute Function** action	`{ context, parameters }`	Any JSON object; later actions can use it
Standalone Validator	When a record is saved, edited or archived	`{ dataEntry, existingEntry }`	`{ success, errors?: [{ errorCode }] }`
Value Validator	When a field value is saved	`{ value }`	`{ success, errors?: [{ errorCode }] }`
Interceptor	Before a matching incoming request is handled	The request's method, path, query, headers and body	A rewritten request, or a response of its own

Write a function

A function is a Knative Functions TypeScript project. Its folder looks like this:

```
my-function/
├── func.yaml        # required
├── params.yaml      # optional: the parameters a reaction can map
├── package.json
├── tsconfig.json    # compiles src/ to build/
└── src/
    └── index.ts     # exports handle()
```

`func.yaml`

```
specVersion: 0.36.0
name: order-total          # replaced by the platform's deployed name (Since 1.10)
runtime: typescript
build:
  builder: pack
  buildEnvs:
    - name: BP_NODE_RUN_SCRIPTS
      value: build
```

The platform deploys your function under a name it derives from the record, so the `name` here does not decide the deployed name `Since 1.10`. It also makes every function private to the cluster; you never get a public URL for it.

`params.yaml`

Declare the inputs a builder maps in the **Execute Function** action:

```
version: "1"
parameters:
  - name: amount
  - name: currency
```

The platform reads the names when it deploys the function and offers them in the action. They describe inputs; they do not validate them.

`package.json`

```
{
  "name": "order-total",
  "version": "1.0.0",
  "scripts": {
    "build": "npx -p typescript tsc",
    "start": "faas-js-runtime ./build/index.js"
  },
  "dependencies": {
    "cloudevents": "^9.0.0",
    "faas-js-runtime": "^2.4.0"
  },
  "devDependencies": {
    "typescript": "^5.5.2"
  }
}
```

The handler: a reaction custom action

This function receives an amount and a currency mapped by the reaction, applies VAT, and returns a result that later actions in the same reaction can use:

src/index.ts

```
import { Context } from 'faas-js-runtime';

interface Input {
  context?: unknown;                      // the event that triggered the reaction
  parameters?: { amount?: string | number; currency?: string };
}

const handle = async (context: Context) => {
  // The platform sends a CloudEvent; depending on its mode the payload is the body
  // itself or the body's `data` property.
  const body = (context.body ?? {}) as { data?: Input } & Input;
  const input: Input = body.data ?? body;

  const amount = Number(input.parameters?.amount);
  if (!Number.isFinite(amount)) {
    return { success: false, message: 'amount must be a number' };
  }

  const vat = Math.round(amount * 0.05 * 100) / 100;
  return {
    success: true,
    message: 'VAT calculated',
    result: { net: amount, vat, gross: amount + vat, currency: input.parameters?.currency ?? 'AED' },
  };
};

export { handle };
```

- **Return a JSON object.** The platform parses your answer as JSON. An answer sent as plain text is treated as an error.
- **Keep the shape stable.** `Since 1.10` later actions and chained reactions can bind `success`, `message` and any key under `result` (for example `result.gross`) — see Execute Function output [Build Apps › Reactions & automation]. If you rename a key, those bindings break.
- **Report failures in the body.** The action is recorded as run whenever your function answers; return `success: false` so later actions can tell.

The handler: a validator

A standalone validator returns error codes; the platform shows the message you entered for each code in the function's **Errors** list:

src/index.ts

```
import { Context } from 'faas-js-runtime';

const handle = async (context: Context) => {
  const body = (context.body ?? {}) as any;
  const { dataEntry } = body.data ?? body;

  const end = new Date(dataEntry?.value?.end_date);
  const start = new Date(dataEntry?.value?.start_date);
  if (end < start) {
    return { success: false, errors: [{ errorCode: 'END_BEFORE_START' }] };
  }
  return { success: true };
};

export { handle };
```

A value validator works the same way and receives `{ value }`.

Build and try it locally

```
npm install
npm run build
npm start          # serves the function on http://localhost:8080

curl -s -X POST http://localhost:8080 \
  -H 'Content-Type: application/json' \
  -d '{"parameters": {"amount": 200, "currency": "AED"}}'
```

The runtime also answers health checks on `/health/readiness` and `/health/liveness`.

Package it

Create a ZIP with `func.yaml` at its root, next to `package.json` and `src/`. Leave out `node_modules` and `build`; the platform installs and builds when it deploys.

```
cd my-function && zip -r ../order-total.zip . -x 'node_modules/*' 'build/*'
```

Only ZIP uploads are deployed. Any other source is ignored without an error.

Add the function

1. Open **Administration › Applications › Functions**, then select **Add function**.
2. Enter a **Name**, choose the **Type** and the **Function Scope** (**System**, **Module** or **Entity**).
3. Leave **Provisioning Type** on **Internal** (the default, with the ZIP option) and upload the ZIP in **Function Zip File**.
4. Save. The **Status** shows **Provisioning** while the platform builds.
5. When the status is **Provisioned** and a **Function URL** appears, the function is ready. If it shows **Build failed**, read **Provisioning Error**, fix the source, and upload a new ZIP.

For a validator, add each error code you return, with its message, under **Errors**.

You need the system **Admin** level to manage functions.

Call a function from a reaction

1. Make sure the function is **Provisioned**.
2. A builder opens the entity's **Reactions** tab, adds a reaction, and chooses the event.
3. They add the **Execute Function** action, choose the function, and map a value to each parameter from `params.yaml`.

Reactions and their actions are covered in Reactions & automation [Build Apps].

You can also run a provisioned function from your own code:

```
curl -s -X POST "https://acme.example.com/api/functions/v1/<function-id>/execute" \
  -H "Authorization: Bearer <access-token>" \
  -H "Content-Type: application/json" \
  -d '{"parameters": {"amount": 200, "currency": "AED"}}'
```

The call needs a credential. It answers your function's JSON, or `{ "success": false, "error": … }` when the function failed or did not answer with JSON. A function that is missing, not active or not provisioned is refused with `404` or `400`. To list the functions available in a scope, call `GET /api/functions/v1/{scope}` (`SYSTEM`, `MODULE` or `ENTITY`, with `?scopeId=` for the last two); only provisioned, active functions are listed.

Validators

A validator function runs only where a builder attaches it to an entity:

- A **Standalone Validator** checks the record's whole value. When it runs — before the record is saved, edited or archived — is chosen when it is attached. It only applies to records of the entity in its scope.
- A **Value Validator** checks a single field value when the record is saved.
- The first validator that returns errors stops the save. Each `errorCode` is looked up in the function's **Errors** list to show its message; for a value validator the error is attached to the field.

Request interceptors

An **Interceptor** function runs before an incoming request that matches its path and HTTP method is handled. It can return a rewritten request — body, headers, query — or answer the request itself, so it never reaches the platform. Only the before-handling stage exists, and an interceptor is not an access control: do not rely on one to refuse requests.

How your function is deployed `Since 1.10`

- **Each tenant's functions are isolated.** In a multi-tenant deployment every tenant has its own space in the cluster, so two tenants' functions with the same name never collide.
- **The deployed name comes from the record**, not from `func.yaml`: a short form of the function's normalized name plus part of its id, for example `order-total-e93838fe1c2d`. It stays the same when you rename the function.
- **Archiving deletes the deployment.**
- **The build and its callbacks are secured by the operator.** How the platform authenticates to the function builder, and how the builder's status callbacks are verified, are deployment settings — see Functions in a multi-tenant deployment and, for the mechanics, Functions — internals.

Call the platform from your function

`Since 1.10` every function receives its tenant's address in its environment:

Variable	Value
`OSOS_API_BASE_URL`	The tenant's API, for example `https://acme.example.com/api`
`APP_ID`	The tenant's id

Always build URLs from `OSOS_API_BASE_URL`; never hard-code a host. A tenant with no public host gets its function without this variable.

On the 1.10 line a function has **no credential of its own**, and the platform keeps **no secrets** for it: nothing in its environment lets it call the API, and the function record has no secrets list. Both arrive on the development line, described below.

Its own API token `Since dev`

On the development line every function also receives `OSOS_API_TOKEN`, the API token of **its own application user**. The function calls the API as itself:

```
const res = await fetch(`${process.env.OSOS_API_BASE_URL}/v2/entities/orders/entries/count`, {
  headers: { 'x-app-token': process.env.OSOS_API_TOKEN ?? '' },
});
const { result } = await res.json();
```

- The function's user **starts with no access**. A tenant administrator grants it exactly what it needs, like any application user — see Use an API token and, for the administrator, Create an API token — Step 2 [Administration › Create an API token for an integration].
- Re-uploading the ZIP does not change the token.
- When an administrator regenerates the token, the new one goes straight to the function, which restarts with it; the old one keeps working for the usual grace period. It is never emailed.
- Archiving the function archives its user, so the token stops working (after the gateway's short cache time).

Secrets `Since dev`

On the development line, a function record has a **Secrets** list: one row per secret, with a key and a value. Each value reaches the running function as an environment variable named by its key:

```
const apiKey = process.env.PAYMENT_API_KEY;
```

- **Values are stored encrypted** and shown masked. Nobody can read them back, including administrators.
- **Change a value** by entering a new one; saving the masked value unchanged keeps it. **Delete** a secret by removing its row.
- **A change takes effect on a new revision.** The platform updates the secret and restarts the function without rebuilding it; a running instance keeps the old value until then.
- **Packages carry names, not values.** Exporting a function keeps its secret keys and drops the values. After installing a package, fill the values in on the target; an installed function lets you change existing secret values, and nothing else.

A save that breaks a rule is refused with `422`:

Code	When
`FUNCTION_SECRET_KEY_INVALID`	A key is not a valid environment-variable name (letters, digits and `_`, not starting with a digit).
`FUNCTION_SECRET_KEY_DUPLICATE`	Two rows use the same key.
`FUNCTION_SECRET_KEY_RESERVED`	The key is set by the platform: `APP_ID`, `OSOS_API_BASE_URL`, `PORT`, `K_SERVICE`, `K_CONFIGURATION`, `K_REVISION`.
`FUNCTION_SECRET_SYSTEM_MANAGED`	You added, changed or removed the `OSOS_API_TOKEN` row.
`FUNCTION_IDENTITY_SYSTEM_MANAGED`	You set the function's identity.

Size limits are checked when the function is deployed, not when you save: each `KEY=value` at most 131071 bytes, all secrets together at most 1 MiB, and no NUL character. If they are exceeded, or the secrets cannot be delivered, **Provisioning Error** shows one of:

Code	Meaning
`IDENTITY_UNAVAILABLE`	The function's user could not be created; nothing was deployed.
`SECRETS_SYNC_FAILED`	The secrets could not be delivered before the deploy; nothing was deployed.
`SECRETS_UPDATE_FAILED`	A secrets change could not be delivered to the running function; the platform retries.

None of these messages contains a secret value.

Functions vs the expression engine

For logic that does not need a deployment — computed fields, dynamic values, conditional targets, template tokens — use the built-in, sandboxed expression engine. A field can take its value from an expression or a pointer, recalculated when the form opens, on save, when the record changes, or when chosen fields change. Two calls resolve expressions from code:

- `POST /api/v1/entities/{entityId}/expressions/resolve` — an entity-level expression;
- `POST /api/v1/entities/{entityId}/{entryId}/tokens/resolve` — tokens for one record.

Good to know

- **TypeScript and ZIP only.** Other languages and source types (such as a GitHub repository) are not deployed.
- **Only new source redeploys.** Editing other fields of the function record does not rebuild it. `Since dev` a secrets change restarts it without a rebuild.
- **Deployment is asynchronous.** Until the status is **Provisioned**, the function cannot be listed or run.
- **Functions are private to the cluster.** They are reached only through the platform.
- **Isolation comes from the serverless runtime**, not from the platform: a function runs as its own service.

Troubleshooting

The status never reaches Provisioned. The platform's function builder may not be set up in this deployment; in that case saving a function does nothing. Ask your operator.

The status is Build failed. Read **Provisioning Error**. Common causes: `func.yaml` missing or invalid at the ZIP root, a runtime other than `typescript`, or a build error in your code. Run `npm run build` locally first.

The reaction runs but later actions get nothing. Check that your handler returns JSON (not text), and that the keys the builder bound under `result` exist in your answer.

Go deeper

For builders: Reactions & automation [Build Apps] · For operators: Functions in a multi-tenant deployment · For platform engineers: Functions — internals

Files: storage rules

This page is for developers who upload, attach and serve files through the API. It explains the rules behind those calls: how long a file lives, what happens when a record that uses it is saved, which files are accepted, how storage limits apply, and how to show a file in a browser without a token.

The calls themselves — upload, attach to a record, download, and their errors — are in Working with files. Start there.

How files are held {#how-file—image—document-fields-use-storage}

A file, image or document field never holds the bytes. It holds the **descriptor** the upload returned:

```
{ "id": "5f0e9b2a-…", "name": "invoice.pdf", "mimeType": "application/pdf", "size": 20481 }
```

The `id` is the file's handle everywhere: in record values, in downloads and in signed links. Upload first, then put the descriptor in the record.

Temporary and permanent files

	Temporary (`?temp=true`)	**Permanent (the default)**
Lifetime	Deleted automatically after **one day**, unless a save keeps it	Kept until the platform cleans it up
Use it for	A file you are about to attach to a record	A file you manage yourself
Download	Same call, same id	Same call, same id

Upload attachments as temporary files `Since 1.10`. When you then save a record that names the file, the platform **keeps** it: it becomes permanent under the same id, and the platform records that the record uses it. A temporary file that no save keeps is gone after a day.

What a save does with files `Since 1.10`

When a record — or an entity, view, page, report or reaction — is saved with file descriptors in it, the platform checks every file that is **new** to that record:

- A temporary file is kept and becomes permanent.
- A permanent file is recorded as used by that record.
- A file that no longer exists fails the save (below).

Files the record already held are not checked again, so re-saving an old record never fails because of them.

Errors on save

Status / code	**Meaning**	**What to do**
`400 FILE_EXPIRED`	A file named in the save does not exist — usually a temporary upload older than a day. `details` lists each file with its field (`{fileId, fieldName}`) or position (`{fileId, path}`) and a reason.	Upload the file again and save again.
`503 STORAGE_SERVICE_UNAVAILABLE`	File storage could not be reached, and the save adds a new file. A save that adds no new file goes ahead.	Retry with backoff.

Which files are accepted

Uploads through the API are checked in three ways:

1. **File type, by extension.** Files with no extension are refused. By default these extensions are accepted (your operator can change the list):

   ```
   .jpg .jpeg .png .gif .bmp
   .txt .csv .pdf .md .rtf .html
   .doc .docx .xls .xlsx .ppt .pptx .odt .ods .odp
   .mp3 .mp4 .avi .mov .wmv .flv .mkv .webm .wav .m4a .3gp
   .psd .ai .indd .xd .fig
   .css
   .ttf .otf .woff .woff2
   .zip .rar
   ```

2. **Content matches the type** `Since 1.8`. For binary types — images, PDF, Office files, media, Photoshop, fonts, archives — the platform reads the start of the file and refuses it if the content does not match the extension. Text formats (`.txt` , `.csv` , `.md` , `.html` , `.rtf` , `.css`) and some design formats (`.ai` , `.indd` , `.xd` , `.fig`) are checked by extension only.

3. **No JavaScript in PDFs** `Since 1.8`. A PDF that contains a JavaScript action is refused.

Refusal	Meaning
`400 INVALID_FILE_TYPE`	The extension is not accepted, or there is none.
`400 INVALID_FILE_CONTENT`	The content does not match the extension.
`400 PDF_JAVASCRIPT_NOT_ALLOWED`	The PDF contains JavaScript.

There is no per-file size limit of the platform's own; a limit may come from the network in front of your deployment. Downloads are always sent as attachments, never shown as active content.

Storage limits `Since 1.10`

If the tenant's plan limits storage, an upload is refused with `403 STORAGE_CAP_EXCEEDED` when the tenant is already at the limit, or when the upload would cross it — also part-way through a large upload, in which case the partial file is removed. `details` carries `capBytes` and `usedBytes`. Temporary files count too. Retrying does not help; the limit has to be raised for the tenant. How to read the tenant's limits is in Plan limits.

Show a file in a browser without a token `Since 1.6`

A browser can load a file with a plain `<img src>` or link if you give it a short-lived **signed link**:

1. Ask for links, with your normal credential, for up to 100 file ids at once:

   ```
   curl -s -X POST "https://acme.example.com/api/storage/v2/files/links" \
     -H "Authorization: Bearer <access-token>" \
     -H "Content-Type: application/json" \
     -d '{"ids": ["5f0e9b2a-…", "a71c…"]}'
   ```

   The answer is `[{ id, url, expiresAt }]`. Ids that do not exist are left out silently. Invalid ids (not 1–64 letters, digits, `_` or `-` , or more than 100) are refused with `400 INVALID_FILE_IDS` .

2. Use each `url` directly. It needs no credential and works until `expiresAt` — between one and two hours with the default settings.

What the link answers:

- `200` with the file. Images, video, audio and PDF are sent to display inline; SVG and everything else as a download. Responses are cacheable until the link expires, and a repeat request with the same `ETag` answers `304`. There is no byte-range support, so for seeking in video use the normal download.
- `403 LINK_INVALID` (tampered) or `403 LINK_EXPIRED`; `404` if the file is gone.
- `503 FILE_LINKS_NOT_CONFIGURED` from both calls when your operator has not set links up. Fall back to the normal download.

These refusals use file storage's own error shape: read the code from `result.details.error`, not from `error.code` (see Errors from the storage service).

Links for the same file are identical within a signing window, so browsers cache them across page loads. Request new links shortly before `expiresAt`.

> ⚠ **A signed link is a bearer capability**
>
> A signed link carries its own authority. Anyone who holds a link can fetch the file until it expires. Do not log links, put them in shareable URLs or hand them to third parties.

Missing files answer `404` `Since 1.6`

A download or signed link for a file whose record exists but whose stored bytes are gone answers `404`, like an unknown id.

Automatic clean-up `Since 1.10`

The platform can delete files that nothing uses any more, but carefully:

- Only files that were **kept by a save** (uploaded as temporary, then saved) are ever considered. Files uploaded straight to permanent storage are never cleaned up automatically.
- A file is considered only after every record that used it let it go, and a grace period (7 days by default) has passed.
- Every record that ever used the file must confirm it no longer does. Any doubt keeps the file.
- A record keeps every file in its history, so files are not deleted while the record exists.

Your operator decides whether clean-up actually deletes or only reports; the platform ships it in report-only mode.

Downloading a tenant export `Since 1.10`

A tenant data export (see the backup API) is downloaded through a **single-use link** that the platform issues, valid for 5 to 10 minutes, on the tenant's own host. It needs no credential; the link is the credential. A used link answers `410 LINK_USED`, an expired or altered one `403 LINK_EXPIRED` or `403 LINK_INVALID`, an unknown export `404 EXPORT_NOT_FOUND`. The download is a gzip archive.

Deleting a package version or an export record keeps its stored file.

Go deeper

For operators: Configuration reference — storage-service · File sweeper settings · For platform engineers: File storage — internals

Criteria and field-path dialects

This page is for **integrators** who write criteria or field paths directly — through the API, a package, or a hand-edited payload. Builders who create conditions in the designer never type these paths; the designer writes them. Criteria conditions appear on five distinct API surfaces in OSOS: **view criteria**, **saved queries**, **statistic-query aggregation paths**, **reaction trigger criteria**, and **reaction action `Dynamic` value paths**. Each surface resolves the condition's field-path string in its own way, and the dialects are close enough to look interchangeable while being different enough to bite. This page is the single reference for which dialect each surface requires, why, and what a wrong choice looks like at run time.

> ⚠ **Wrong paths fail silently — almost everywhere**
>
> No criteria or aggregation surface validates a field path against the entity. A wrong-dialect path is never rejected:
>
> - **View and saved-query criteria** return HTTP `200` with an **empty entries array**.
> - **Statistic queries** return a **single bucket** with `category: null` (and `Sum`-style functions yield `0`).
> - **Reaction trigger criteria** simply never match — the trigger is skipped with nothing but a service log line.
>
> The one exception is a reaction action's `Dynamic` value path, which fails loudly — the action errors and the failure is recorded in the trigger execution history (see Silent or loud).

Dialects by surface

Entry field data is stored under the `value` property of every entry document, so most surfaces want the field's full normalized name prefixed with `value.` — but not all of them, and not for entry-metadata paths.

Surface	Required dialect	Example path	A wrong choice yields
View criteria (`filter.criteria` + quick filters)	`value.<fullNormalizedName>`; entry metadata un-prefixed (`state`, `creationInfo.doneAt`)	`value.amount`	HTTP `200`, empty entries array
Saved queries (`DataQuery.criteria`, incl. as a view's data source)	Same as view criteria — merged into the same filter at query time	`value.status.id`	HTTP `200`, empty entries array
Statistic-query aggregation (`aggregation.field`, `aggregation.groupByField`)	`value.<fullNormalizedName>`; metadata exceptions un-prefixed (`creationInfo.doneAt` / `doneBy`, `updateInfo.doneAt` / `doneBy`); relation hops `value.<field>.:relation.value.<field>`	`value.department`	One bucket with `category: null`; `Sum` / `Average` over the missing path yield `0` / `null`
Reaction trigger criteria (`DataReaction.criteria`)	`value.<field>` (legacy) or canonical `scope.entries.<entityNormalizedName>.value.<field>` — **never bare**	`value.priority`	Criteria never match; the trigger is silently skipped
Reaction action `Dynamic` value paths	Canonical `scope.entries.<entityNormalizedName>.value.<field>`; `value.<field>` and bare known field names are auto-expanded	`scope.entries.task.value.assignee`	Action **fails with a recorded error** in the trigger execution history

Rule of thumb: always write the `value.` prefix. It is required on the first three surfaces and accepted (auto-expanded) on both reaction surfaces. The one reaction surface where bare names genuinely are the dialect is not a criteria path at

all: the `DataEntryUpdated` event's `eventConfig.fields` watch-list, which matches bare `fullNormalizedName` values.

Why each surface reads paths the way it does

View and saved-query criteria

When a view's entries are fetched, the data service merges the request's `filter.criteria`, the view's query-type data-source criteria, and any active quick-filter query into **one** filter expression, then hands it to the query engine with **no field scope**. The engine uses the condition's `field` string **verbatim** as the match key, and the resulting `$match` runs directly against stored entry documents — whose field values all live under the document's `value` property. That is the whole dialect: `value.<fullNormalizedName>` reaches your data, while entry metadata such as `state` or `creationInfo.doneAt` sits at the top level of the document and is therefore written without the prefix.

Nothing checks the path at save time either: the save-time parse of a query's criteria validates **operators only** — an unknown operator or an unsupported field expression is rejected, but a syntactically fine path aimed at the wrong level sails through and simply matches nothing later.

Statistic-query aggregation paths

`GET /v2/entities/{entity}/statistic-queries/{id}/execute` runs the stored aggregation. Internally, the service strips the `value.` prefix from `aggregation.field` / `aggregation.groupByField` **only to look the field up on the entity** — the caller's raw string is kept as the lookup path and interpolated verbatim into the `$unwind` / `$group` / `$addFields` pipeline stages. Because entry documents nest data under `value`, the paths must be `value.`-prefixed to hit anything; the backend's own contract for these paths is `value.<fieldName>` (or `creationInfo.<doneAt>` for metadata). Relation hops use the `:relation` marker: `value.<field>.:relation.value.<field>`. Metadata is again the exception, handled without the prefix: `creationInfo.doneBy` / `updateInfo.doneBy` and `creationInfo.doneAt` / `updateInfo.doneAt`.

A worked example, with the numeric aggregation-function enum the model requires (`Count = 0`, `Sum = 1`, `Min = 2`, `Max = 3`, `Average = 4`):

```
{
  "aggregation": {
    "aggregationFunction": 1,
    "field": "value.amount",
    "groupByField": ["value.department"]
  }
}
```

Write `"field": "amount"` instead and the pipeline groups on `$amount` — a key absent from every document — so every entry collapses into one `category: null` bucket and the sum is `0`, with no error anywhere.

One adjacent footgun on the same surface: `source.store` must be the entity's **normalized name**. The store is looked up by `normalizedName` only, and at execution time the string is used verbatim as the MongoDB collection name. Passing a valid entity **id** or a display name fails at save time with the misleading `Entity <store> not found` (`ENTITY_NOT_FOUND`) — the entity exists, it just is not found under `normalizedName`. (The store's schema pattern also rejects UUID hyphens and spaces outright.)

Reaction trigger criteria

Reaction criteria are **not** evaluated against the live entry document. The trigger context is flattened into a synthetic entry whose `value` object holds keys like `scope_entries_<entityNormalizedName>_value_<field>`; that entry is

inserted into a temporary collection, and the criteria paths are rewritten to match it — `scope.entries.<entity>.value.<path>` and legacy `value.<field>` both map onto the flattened keys.

A **bare** path is untouched by that rewrite, and the engine's field-scope logic then strips the first path segment — which for a single-segment bare path leaves the **empty string**. The condition becomes a match on `""`, which matches nothing, and the trigger is skipped. So the reaction-criteria dialect is effectively the same `value.` anchor as views: write `value.<field>`, or the full `scope.entries.<entityNormalizedName>.value.<field>` form.

If you have seen bare field names "work" in reactions, that was the `DataEntryUpdated` watch-list (`eventConfig.fields`) — a list of bare `fullNormalizedName` values that selects *which field changes fire the event*, not a criteria path.

Reaction action `Dynamic` paths

A `Dynamic` value on a reaction action is not resolved through MongoDB at all: it is a property lookup over the **in-memory trigger context**. The resolver accepts three dialects — canonical `scope.entries.<entityNormalizedName>.value.<field>`, legacy `value.<field>` (auto-expanded to the canonical form), and a bare `<fullNormalizedName>` when it matches a known field. The `value.` prefix therefore always works here too, but this is the one surface where it is not required.

Category and switch fields match the option id

A category field's stored value carries the option object, and criteria compare against its `id`:

- **In / Not in** with option objects: the engine's explicit category case rewrites the match key to `value.<field>.id` and maps each argument to its `id`, compiling to `{"value.<field>.id": {"inq": ["<optionId>", …]}}` (`nin` for Not in).
- **Equal / Not equal** with an object argument expand per key: `{"id": "<uuid>"}` becomes `{"value.<field>.id": {"eq": "<uuid>"}}`.
- **Statistic group-bys** append `.id` to the group-by path for category **and switch** fields, then resolve each result bucket's id back to its display label — category options from the field's configured option list, switch values from the entity's internal collections.

The option `id` is a plain required string in the schema; it is a UUID by front-end convention, not by schema enforcement. Match against the **option id**, never the label. For example, an *In* condition:

```
{
  "field": "value.priority",
  "args": [{ "id": "0b9f3c2e-…" }, { "id": "6d41a8f0-…" }]
}
```

compiles to `{"value.priority.id": {"inq": ["0b9f3c2e-…", "6d41a8f0-…"]}}`.

`In` / `NotIn` with a single reference or category value `Since 1.10`

`In` and `NotIn` match a **reference** field on `<field>.dataEntry` and a **category** field on `<field>.id`, reducing each argument to that id. Before 1.10 the reduction only ran when the resolved value was already a **list**. A single object — a filled Parameter from a single-select input (`status: {id: 'doing'}`), or a reaction criterion such as `assignee In scope.me()` — stayed whole inside the `$in` and matched nothing, with no error.

Both query engines (the data service's and the reaction service's) now compile the pair the same way: the value is **wrapped in a list first**, then an element carrying `dataEntity` + `dataEntry` switches the key to `.dataEntry`, and an element carrying `id` switches it to `.id`. The rewrite is chosen by inspecting the **values**, not the field's data type — so a

hand-written criterion with a bare id string is still compared verbatim. In reaction criteria a `null` element in a reference list no longer throws.

The `Expression` value source and its result shape `Since 1.10`

A criteria argument of type **Expression** (`FilterExpressionValueTypeEnum.FieldExpression`, `2`) holds a JavaScript expression **string**; a non-string value is refused at save time. Both engines evaluate it as a global expression — **with no entry in scope** — and put the **raw result** into the condition. There is **no casting by field type**, so it matches only when it returns the shape the field stores.

The query builder offers the Expression source for these field types, each with the result it must produce (shown as a hint under the box):

Field type	The expression must return
Text	a `string`
Number	a `number` — `10`, not `'10'`
Date / Date-time / Time	a `string` in the stored format (e.g. `yyyy-MM-dd`), not a `Date` object
Boolean	a `boolean`
Reference	`{dataEntity, dataEntry}` — e.g. `scope.me()`, not a bare id

`In` / `NotIn` accept the same type or a list of it. Category, localized text, JSON / geo, file, icon and block fields do **not** offer the source. A wrong shape is only a **warning** in the editor — the editor type-checks the expression against the accepted types, but **Apply is never blocked** — so a mismatching expression can still be saved and then silently matches nothing, exactly like a wrong field path. The end-user entries **Filter** panel uses the same auto-complete expression editor as the admin builder; the editor's code is loaded only when an Expression row is rendered.

Date pointers in reaction criteria

Reaction criteria resolve **both** `@today()` and `@now()` — the expression scope also contains `@StartOfDay`, `@StartOfWeek`, `@StartOfMonth`, and `@StartOfYear`. What differs between the two headline tokens is the **return shape**, and that difference is the real footgun:

Token	Resolves in reaction criteria?	Returns	Use with
`@now('±N<unit>')`	Yes	Full **ISO datetime**	DateTime-typed comparisons only
`@today('±N<unit>')`	Yes	**Date-only** string (start of local day, with a hard-coded `+2h` offset before formatting)	Date-typed fields

Because `@now()` yields a full datetime, an equality or range comparison against a **date-only** stored value never matches — which presents exactly as "`@now()` doesn't work in reaction criteria", even though the token resolves fine. Use `@today('±Nd')` for date-only fields and reserve `@now()` for DateTime comparisons.

Manipulation grammar, shared by both tokens:

- The amount **must** be signed: `'+1d'`, `'-3w'`. An **unsigned** amount such as `'1d'` is silently ignored and the base date is returned.
- Units: `d` days, `w` weeks, `M` months, `y` years, `h` hours, `m` minutes, `s` seconds.

A pointer that fails to evaluate is **not** silent: expression execution throws an `ExpressionExecutionException`.

Silent or loud: where a wrong path surfaces

Surface	Save-time check	Wrong path at run time	Visible where
View / saved-query criteria	Operator validation only — never a bad-path error	`$match` on a nonexistent key → HTTP `200`, empty entries	Nowhere
Statistic-query aggregation	Field-lookup failures are caught and logged	One `category: null` bucket; `Sum` of a missing field → `0`	Nowhere — the response shape looks valid
Reaction trigger criteria	—	Criteria never match; the trigger is skipped	A service log line only; no trigger-history record
Reaction action `Dynamic` path	—	Throws `Path "<path>" does not exist in the context.`; resolution failures are wrapped as `INVALID_ENTRY_FIELD`	Recorded in the **trigger execution history** and the service error log — nothing surfaces to the originating request

The trigger execution history is the **only** place a wrong `Dynamic` path is visible; from the caller's point of view even the loud failure looks quiet.

Everything on this page is long-standing behaviour, unchanged from the 1.2 line through the current development line, except the two sections marked `Since 1.10` (`In` / `NotIn` with a single value and the `Expression` value source), which carry their own badges.

Calling the HTTP API

This page is for developers who call the OSOS API from code. It covers the rules every call shares — where to send it, how to authenticate, how to read the answer and its errors, how to page and filter, and where to find the machine-readable specification. Read it once before you write a client; the Backend quickstart gets you to a first working call in minutes.

Your first two calls

Read a page of records, then create one. Replace the host, the token and the entity name (the entity's normalized name, for example `employees`):

```
export BASE="https://acme.example.com"
export TOKEN="<your-api-token>"

# Read: the first 20 active employees in HR, newest first
curl -s -G "$BASE/api/v2/entities/employees/entries" \
  -H "x-app-token: $TOKEN" \
  --data-urlencode 'filterCriteria={"limit":20,"skip":0,"order":["value.hireDate DESC"],"criteria":{"operator":"Equa
l","field":"value.department","args":[{"type":0,"value":"HR"}]}}'

# Write: create one record
curl -s -X POST "$BASE/api/v1/entities/employees/entries" \
  -H "x-app-token: $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"value": {"name": "Lina Haddad", "department": "HR"}}'
```

Both answer with the standard envelope: `{ "success": true, "result": … }`. The rest of this page explains each part.

Where to send requests

Send every call to the **tenant's own host**, with paths starting `/api`. The gateway on that host authenticates you, works out the tenant from the host name, checks access, and forwards the call to the right part of the platform. You never call an internal service directly and you never send a tenant id.

The path prefix selects the area:

Path prefix	Area
`/api`	Core data platform: entities, entries, views, queries, permissions
`/api/comments`	Comments
`/api/notification`	Notifications and push tokens
`/api/timeline`	Timeline posts
`/api/workflow`	Workflow execution and ownership
`/api/reaction`	Automation rules (reactions)
`/api/functions`	Server-side functions
`/api/integrations`	Third-party integrations
`/api/dependency`	Dependency graph
`/api/packaging`	Packages, export and import, publish requests, the in-app store, installations

Path prefix	Area
`/api/logs`	Email, SMS and WhatsApp logs (read-only), and the WhatsApp status webhook `Since 1.10`
`/api/storage`	Files — see Working with files
`/api/rtc`	Audio and video call tokens

An operation documented for an area other than the core data platform — in the Endpoint reference or in that area's own OpenAPI document — is relative to its prefix. For example, the WhatsApp template list `GET /v1/whatsapp/templates` of the automation area is called as `/api/reaction/v1/whatsapp/templates`.

View-scoped calls. Many reads and writes also exist in a form that goes through a view: `/api/v1/v/{viewId}/…`. The view's own access settings are checked first, so a `403` on such a path usually means the view does not allow that operation — not that your credential is wrong.

Bodies are JSON in both directions (`Content-Type: application/json`, `Accept: application/json`), except file uploads and downloads.

Versions

The version is in the path, right after the prefix, and it belongs to each operation, not to a whole area. One resource often has `v1`, `v2` and `v3` side by side:

```
GET  /api/v1/entities/{entity}/entries          (deprecated)
GET  /api/v2/entities/{entity}/entries
GET  /api/v3/entities/{entity}/entries
POST /api/v1/entities/{entity}/entries
GET  /api/v1/entities/{entity}/entries/{id}
```

- Old versions are marked deprecated in the OpenAPI document, not removed.
- A newer version usually adds capability; do not assume `v1` and `v2` of "the same" operation return the same shape.
- Use the highest version a resource offers unless you need a legacy shape.

Authenticate

Send one credential with every call:

Credential	Header	Get it from
API token	`x-app-token: <token>`	Use an API token
OAuth access token	`Authorization: Bearer <token>`	Authentication & OAuth

Which to choose: Choosing a credential.

Read the response

Successful answers are wrapped in one envelope:

```
{ "success": true, "result": { "count": 42 } }
```

`result` is `null` when an operation returns nothing. A few answers are not wrapped:

Request	Body on success
`GET` on a `/api/v3/…` path `Since 1.10`	**The bare payload**, for example `{ "entries": [ … ], "tags": [ … ] }`
`POST`, `PATCH`, `PUT`, `DELETE` on `/api/v3/…`	Wrapped
Every `v1`, `v2` and unversioned JSON call	Wrapped
File downloads and other streams, health checks	The raw bytes or stream
OpenAPI documents (`…/openapi.json`, `…/openapi.yaml`)	The bare document

Errors are always wrapped, `/v3` reads included.

> ### Unwrap by shape, not by version
>
> On 1.8 deployments up to 1.8.21, `/v3` reads are still wrapped. A client that talks to more than one release line should unwrap `result` **only when the body carries** `"success": true`. The platform's own portal does exactly that.

Handle errors

Every error has the same shape:

```
{
  "success": false,
  "result": null,
  "error": {
    "code": "ENTITY_NOT_EXIST",            // stable, machine-readable
    "message": { "en": "…", "ar": "…" },   // localized text for people
    "serviceName": "…",                     // which part of the platform answered
    "traceId": "…"                          // quote this when you report a problem
  }
}
```

- **Branch on `error.code`, never on the message text.** Use the HTTP status for coarse handling.
- **A `500` is not always temporary.** Many deterministic refusals still answer `500`. Check the code against Errors and retries before you retry.
- Some errors carry a `details` object. A validation failure (`VALIDATION_FAILED`) names the failing path and the rule — for an invalid choice, the allowed values.
- A duplicate value in a unique field answers `400 UNIQUE_VALUE`, with the field path and value in `details`.
- A credential the platform cannot resolve to a user answers `401`.
- Files are the exception to this shape; see Working with files.

Tenancy and capacity refusals `Since 1.10`

Status	`error.code`	When	What to do
`404`	`SPACE_ERROR`	The host you called matches no space.	Use a host registered for the space.
`404`	`TENANT_NOT_RESOLVED`	Multi-tenant deployments: the host matches no active tenant.	Use the tenant's own host.

Status	error.code	When	What to do
429	RATE_LIMITED + Retry-After	Multi-tenant deployments: the tenant used up its per-second or per-minute request budget. details names the window and limit; RateLimit-* headers are usually present.	Wait Retry-After seconds.
503	RATE_LIMIT_UNAVAILABLE + Retry-After: 1	The rate limiter could not count the request and refused it.	Retry after 1 second.
503	TENANT_IN_MAINTENANCE + Retry-After	Multi-tenant deployments: the tenant is being restored from a backup or moved. Every call is refused except the status reads of the backup API.	Wait Retry-After seconds (1–3600, default 30), then retry.
400	SPACE_CONTEXT_MISSING, SPACE_NOT_RESOLVED	Only when a service is called directly, not through the gateway.	Call through the gateway.

Treat any 503 with Retry-After the same way: back off for the time it names, then retry. A TENANT_IN_MAINTENANCE body has traceId NO-TRACE-ID and an empty Arabic message. What happens to the tenant's events meanwhile is in Data model & events.

Refusals worth handling

These refusals of your input or of your access answer a 4xx with a stable code. Rows marked Since 1.8 were also delivered in 1.8 updates; the others are 1.10.

Status	error.code	Where	When
422	WORKFLOW_ACTION_EXECUTION_NOT_ALLOWED Since 1.8	Executing a workflow action	The action is hidden on the entry. (Deliberately not 403.)
422	CURRENT_STEP_ALREADY_OWNED Since 1.8	Taking ownership of a workflow step	The current step already has an owner.
422	ENTITY_ARCHIVE	DELETE /api/v1/entities/{id}, updating or moving an archived entity	The entity is already archived — often a double click or a retry.
400	INVALID_POINTER	Any filter or criteria with a pointer value, such as …/entries/count?whereCriteria=	The pointer is not a known pointer function, or not of the form @name, @name() or @name('<amount>'). @Today, @Me, @none and any other form are refused.
400	DATA_VIEW_INVALID_ALLOWED_OPERATIONS	Saving a view	Entry editing is on but no supported edit operation is set.
400	MISSING_REQUIRED_PARAMETER	POST /api/v1/entities/{entity}/entries, …/entries/draft, POST /api/v1/views/{id}/entries, …/entries/draft	The request has no body at all.
400	MISSING_REQUIRED_PARAMETER Since dev	PATCH /api/v1/entities/{entity}/views/{id}	A view save with no body at all. A partial body is accepted.
403	ACCESS_OBJECT_NO_AUDIENCE	GET /api/v2/access-object/{id}	The view, page or report allows neither visitors nor signed-in users, so nobody can open it.

Status	error.code	Where	When
404	QUERY_NOT_FOUND	Saving a query, view, summary or proxy view that names a query	The named query does not exist. (Reading a stored view whose query has since gone still answers 500.)
404	—	GET /api/v1/spaces/{id}	The space does not exist.
400	FILE_EXPIRED	Saves that attach uploaded temporary files: entries, entities, views, pages, reports, reactions	A temporary file expired before the save. details lists {fileId, fieldName?, path?, reason} per file. Upload again and save again.
503	STORAGE_SERVICE_UNAVAILABLE	The same saves, and file reads	File storage could not be reached and the save adds a new file. Retryable.
as relayed Since 1.8	the other part's code	Any operation that calls another part of the platform	A 4xx from that part keeps its own status and body.

Plan-limit refusals are in Plan limits. The full catalogue, with retry advice per code, is in Errors and retries.

Avoid overwriting someone else's change `Since 1.10`

Every entry carries an **updateIdentifier** that changes on every write. Reads return it with the entry. Send it back with your change, and the write is refused with **409 UPDATE_CONFLICT** if someone else changed the entry since you read it.

The check is opt-in per request. Leave the token out and the write behaves as it always did. There is no header form (no If-Match, no ETag):

Operation	Where the token goes
PATCH /api/v1/entities/{entity}/entries/{id}	Body property updateIdentifier
PATCH /api/v2/entities/{entity}/entries/{id}	Query parameter updateIdentifier (the body is a JSON Patch array)
PUT /api/v1/views/{id}/entries/{entryId}	Body property updateIdentifier

An entry that has no token yet is never refused on this account; it gets one on that write. The read-keep-resend recipe, with the conflict body, is in Writing data.

> ### Known limitations
>
> The guard covers the three operations above and no others. The platform's own web client does not send the token on inline edits, and a few internal write paths do not change it, so concurrent edits from those paths are not detected.

Page and filter

There are two filter dialects, one per generation of operations. The same filter model is used by saved queries, statistics, view filters and reaction conditions.

v1: the classic filter

v1 list and count operations take a filter object as a query parameter (?filter={…}, JSON or URL-encoded); count operations take where:

Key	Purpose
`where`	Match condition
`fields`	Include or exclude properties
`order`	Sort: `"<field> ASC"` or `"<field> DESC"`, or an array of them
`limit`	Page size (at least 1)
`skip` / `offset`	Rows to skip (aliases)
`include`	Load related records

`where` operators: `eq`, `neq`, `gt`, `gte`, `lt`, `lte`, `inq` (in), `nin` (not in), `between`, `like`, `nlike`, `regexp`, and `and` / `or` grouping.

```
GET /api/v1/entities/employees/entries?filter={"where":{"value.active":true},"limit":50,"skip":100,"order":["value.name ASC"]}
```

v2 and v3: the criteria filter {#v2—v3—structured-criteria-filter}

`v2` and `v3` operations take a `filterCriteria` object (count operations take `whereCriteria`) and a free-text `search` parameter. `filterCriteria` keeps `limit`, `skip`, `offset`, `order`, `fields`, `where` and `include`, and adds a `criteria` tree. Each node of the tree has:

Field	Notes
`field`	The field's normalized name. Absent on `And` / `Or` nodes.
`operator`	Required. One of: `And`, `Or`, `Equal`, `NotEqual`, `LessThan`, `LessThanOrEqual`, `GreaterThan`, `GreaterThanOrEqual`, `In`, `NotIn`, `Contains`, `DoesNotContain`, `StartsWith`, `DoesNotStartWith`, `Like`, `IsNotLike`, `IsEmpty`, `IsNotEmpty`, `ElemMatch`, `Type`, `Under`
`fieldOperator`	Optional aggregate applied to the field first: `Size`, `Sum`, `Avg`, `Min`, `Max`, `First`, `Last`, `Any`
`args`	Required. Child nodes (for `And` / `Or`) or values: `{ "type": <value type>, "value": … }`

```
// GET /api/v2/entities/employees/entries?filterCriteria=…&search=cairo
{
  "limit": 50,
  "skip": 0,
  "order": ["value.hireDate DESC"],
  "criteria": {
    "operator": "And",
    "args": [
      { "field": "value.department", "operator": "Equal",
        "args": [ { "type": 0, "value": "HR" } ] },
      { "field": "value.salary", "operator": "GreaterThanOrEqual",
        "args": [ { "type": 1, "value": 5000 } ] }
    ]
  }
}
```

Check a complex filter before you save it with `POST /api/v1/queries/parser`, which reports whether the expression is well-formed. Counts, sums, minimums, maximums and averages are expressed as statistic queries. `Since 1.8` the tree-list operations (`…/entries/tree`) also honour a `where` in the filter. The field-path rules are in Criteria & field-path dialects, and choosing a dialect is covered in Reading data.

Paging rules

- Page with `limit` plus `skip` / `offset`. There are no cursors.
- **The server sets no maximum page size.** Choose a sensible `limit` yourself.
- Counting is a separate call: `GET …/entries/count` (v1, `?where=`) or `GET /api/v2/entities/{entity}/entries/count` (`?whereCriteria=` and `search`), answering `{ "count": <n> }`.

Other list parameters:

Parameter	Applies to	Meaning
`search`	v2, v3	Free-text search across searchable fields
`appliedQuickFilters`	list, count, kanban	Quick-filter ids to apply
`appliedTagsFilters`	v3, tree, kanban	Tag-filter ids; the answer carries a matching `tags` map
`include_virtual`	list, count	Include virtual (computed) entries
`fields`	`GET …/entries/{id}`	Comma-separated projection, such as `?fields=value.name,value.email`; still subject to the view's access rules
`excludeResolved`	`GET …/entries/{id}`	Skip resolving relations and categories, for a lighter read

Useful operations

- **Bulk changes** `Since 1.4` — `PATCH /api/v1/entities/{entity}/bulk-update` and `…/bulk-upgrade` (also `v2`) take an explicit list of ids or a "select all matching" selector and return a result per row. `Since 1.6` there is a queued form and a second route, `PATCH /api/v2/entities/{entity}/bulk-update`. See Writing data.
- **Human verification** `Since 1.6` — creating an entry may require a passed verification challenge (for anonymous visitors).
- **Masked fields** `Since 1.8` — entry reads list the fields you may not read in `sensitiveMaskedFields`, so you can show them as locked.
- **Aggregated records** `Since 1.8` — for an entity that aggregates other entities, each field's locked state comes from the source record.

Reactions in a module `Since 1.8` — `GET /api/reaction/v1/reactions/module/{moduleId}` lists the reactions defined on any entity in a module. It accepts an optional `filter` (`where`, `fields`, `order`, `limit`, `skip`, `include`) — for example to narrow by `eventType`. Your `where` is combined with the module scope, which you cannot widen. A module with no entities answers an empty list.

Small utility calls

Call	Returns
`GET /api/v1/version`	The backend version
`GET /api/v1/icons`	The available icons
`GET /api/v1/configs`	Organization-level configuration
`GET /api/v1/requests/{entity}/{id}/{mode}`	Whether you may perform `{mode}` on that entry

The sidebar of views (`GET /api/v2/side-bar`) is covered in Pages and widgets [Build Apps › Pages & Widgets].

Long-running work

Some operations answer at once with a job handle; you then poll a status call until the job finishes. For live progress instead of polling, follow the job over the real-time channel.

Operation	Start	Poll
Generic background job	(returns a `messageId`)	`GET /api/v1/job/{messageId}`
Package export	`POST /api/packaging/v1/export` → `{ jobId }`	`GET /api/packaging/v1/export/{jobId}/status`
Package import	`POST /api/packaging/v1/import/{sessionId}/execute`	`GET /api/packaging/v1/import/{sessionId}/status`
Full dependency rebuild	`POST /api/dependency/v1/rebuild` → `202 { jobId }`	`GET /api/dependency/v1/rebuild/status/{jobId}`
Tenant backup `Since 1.10`	`POST /api/v1/admin/backups` → `202`	`GET /api/v1/admin/backups/{backupId}`
Tenant restore `Since 1.10`	`POST /api/v1/admin/restores` → `202`	`GET /api/v1/admin/restores/{restoreId}`
Tenant data export `Since 1.10`	`POST /api/v1/admin/exports` → `202`	`GET /api/v1/admin/exports/{exportId}`

- Only one full dependency rebuild runs at a time; a second answers `409`.
- Polling another space's export or import job answers `403`.
- Some results are files, not JSON: data exports (XLSX, PDF), theme CSS, translation files and fonts. Handle them as downloads.

Status values, retention and cancellation per job type are in Long-running work.

Plan limits `Since 1.10`

A tenant's plan can limit entries, entities, spaces, storage and automation runs per month. A limit that is not set is not enforced.

What each limit refuses — all `403`, with the tenant's numbers in `details`:

`error.code`	Refuses	`details`
`ENTRY_CAP_EXCEEDED`	Creating an entry (single, batch, draft, through a proxy) once the weighted total reaches the hard limit without overage. Reading, editing and exporting are never refused.	`{cap, hardLimit, current}`
`ENTITY_CAP_EXCEEDED`	Creating an entity beyond the limit. Implicit and core entities do not count.	`{cap, current}`
`SPACE_CAP_EXCEEDED`	Adding a space beyond the limit, restoring an archived one included. Archived spaces do not count; deactivated ones do. The default space is never refused.	`{limit, current}`
`STORAGE_CAP_EXCEEDED`	An upload once stored bytes reach `maxStorageGb`, or when the upload would cross it. File storage answers in its own error shape.	`{capBytes, usedBytes}`

A package install that would pass the entries limit is refused before it starts.

The monthly automation-run limit is not an HTTP error. A reaction run over the limit fails no request: it is recorded with status `refused` (code `REACTION_RUN_CAP_EXCEEDED`, with `limit`, `used`, `period`), and none of its actions run. Find such runs with `outcome=refused` in the run history (Reading data), `status=refused` in the reaction logs, or `refusedCount` in the reaction list. A single-tenant deployment does not meter automation runs.

Tenant backups, restores and exports `Since 1.10`

In a multi-tenant deployment, the tenant's own system administrators can back up, restore and export its data through the API (reference: BackupAdmin). There is no screen for this yet.

- **Who:** the tenant's system administrators only.
- **Where:** multi-tenant deployments only. Elsewhere every call answers `501 BACKUP_NOT_SUPPORTED_IN_SINGLE_MODE`.
- **How:** each start answers `202` with a handle; poll the matching `GET`.

Call	Answers	Notes
`GET /api/v1/admin/backups`	`200 {items, quotas}`	Newest first. `quotas`: `[{quota, used, limit, resetsAt}]` for `backupsPerDay`, `restoresPerDay`, `dumpDownloadsPerDay` (UTC day)
`GET /api/v1/admin/backups/{backupId}`	`200`	`{backupId, reason, state, createdAt, restorePoint?, sizeBytes?, expiresAt?}`
`POST /api/v1/admin/backups`	`202`	Start a backup now. No body.
`POST /api/v1/admin/restores/preview`	`200`	Body `{backupId}` or `{pointInTime}` (ISO-8601). Returns what the restore would do, warnings, and a single-use `confirmationToken` with `confirmationExpiresAt`.
`POST /api/v1/admin/restores`	`202`	Body `{backupId}` or `{pointInTime}`, plus the `confirmationToken`. The tenant is under maintenance from here.
`GET /api/v1/admin/restores/{restoreId}`	`200`	`{restoreId, backupId, pointInTime?, state, step?, safetyBackupId?, startedAt, finishedAt?, error?, hint?, …}` — poll this
`POST /api/v1/admin/exports`	`202`	Start an export archive of the tenant's data. **User accounts are not included.**
`GET /api/v1/admin/exports`	`200 {items}`	The tenant's exports
`GET /api/v1/admin/exports/{exportId}`	`200`	`{exportId, state, createdAt, finishedAt?, expiresAt?, sizeBytes?, error?, linksLeft, manifest}`
`POST /api/v1/admin/exports/{exportId}/link`	`200 {url, expiresAt}`	A single-use download link on the tenant's own host, valid up to 10 minutes; at most 5 links per export

During a restore, the tenant answers `503 TENANT_IN_MAINTENANCE` to everything except `GET` on `/v1/admin/restores…`, `/v1/admin/backups…` and `/v1/admin/exports…`, so you can keep polling.

Status	`error.code`
`400`	`VALIDATION_FAILED` (no `backupId` or `pointInTime`, or no `confirmationToken`), `CONFIRMATION_INVALID`, `RESTORE_TARGET_INVALID`
`404`	`BACKUP_NOT_FOUND`, `RESTORE_NOT_FOUND`, `EXPORT_NOT_FOUND` — another tenant's id reads as not found
`409`	`TENANT_IN_MAINTENANCE`, `TENANT_NOT_BACKUPABLE`, `TENANT_NOT_RESTORABLE`, `TENANT_NOT_EXPORTABLE`, `BACKUP_NOT_RESTORABLE`, `POINT_IN_TIME_NOT_RESTORABLE`, `RESTORE_IN_PROGRESS`, `EXPORT_IN_PROGRESS`, `EXPORT_NOT_READY`, `EXPORT_LINKS_EXHAUSTED`
`429`	`BACKUP_QUOTA_EXCEEDED`, `RESTORE_QUOTA_EXCEEDED`, `EXPORT_QUOTA_EXCEEDED`, with `Retry-After` until the quota resets
`501`	`BACKUP_NOT_SUPPORTED_IN_SINGLE_MODE`

Status	error.code
502	BACKUP_ENGINE_FAILED, EXPORT_LINK_FAILED
503	BACKUP_ENGINE_UNAVAILABLE, EXPORT_ENGINE_UNAVAILABLE, BACKUP_SERVICE_UNAVAILABLE, with Retry-After: 60

The operator side — retention and daily quotas — is in Tenant backup and restore.

More API families

System settings

GET /api/v1/system-settings/public needs **no credential**. It returns the system-settings values marked public, as {block: {field: value}}, for the current space. Saving a system-settings entry can be refused with 422: SYSTEM_SETTINGS_SPACE_MISMATCH, SYSTEM_SETTINGS_TENANT_DEFAULT_ARCHIVE (the tenant-wide entry cannot be archived) or SYSTEM_SETTINGS_DUPLICATE_PHONE_NUMBER (another active entry already uses that WhatsApp number). Reference: SystemSettings. Since 1.10

WhatsApp Since 1.10

Call	Who may call	Notes
GET /api/reaction/v1/whatsapp/templates	System View or above	The approved templates of the space's WhatsApp business account, for the **Send WhatsApp** action: [{name, language, category, bodyParameterCount}]. Authentication templates are left out. 409 whatsapp_not_configured when the account is not set up; 502 whatsapp_provider_error when the provider refuses (details {metaCode, metaStatus}). Reference: WhatsAppTemplates.
GET /api/logs/v1/whatsapp/, GET /api/logs/v1/whatsapp/count	As the email and SMS logs	The WhatsApp message log, with a filter. Not in any OpenAPI document.
GET / POST /api/logs/v1/whatsapp/webhook/{spaceId}	Public — Meta calls it	Meta's subscription handshake (GET) and delivery-status callback (POST).

The webhook is public because Meta cannot sign in. It is protected by the space's verify token and a signature:

- spaceId must be ROOT or an existing space, otherwise 404.
- GET: hub.mode must be subscribe and hub.verify_token must equal the space's verify token. Success answers 200 with the bare hub.challenge as plain text; anything else is 403.
- POST: the raw body must match the X-Hub-Signature-256 header, signed with the space's app secret. No secret, or a missing or wrong signature, is 401; malformed JSON under a valid signature is 400; otherwise 200. Only delivery statuses (sent, delivered, read, failed) are recorded; incoming messages are not.

WhatsApp settings are fields of the System Settings entity, edited through the entry editor or the API. How the **Send WhatsApp** action behaves is in Reactions & automation [Build Apps].

Packages and the store

The packaging area (/api/packaging) has no OpenAPI document, so it is not in the Endpoint reference. Its routes and who may call each are in Marketplace and store. Since 1.10 it also covers package history, deleting an unpublished version, first and grouped publish requests (approve, return, withdraw), publisher set-up and renaming, the store's

publisher and topic lists, an install check, installation updates, and export records (`GET /api/packaging/v1/exports` , `GET` / `DELETE …/exports/{id}` — deleting an export record keeps its file; see Export records).

Renaming installed views and pages `Since 1.10`

A view or page installed by a package is read-only through `PATCH /api/v1/entities/{entity}/views/{id}` and `PATCH /api/v1/generic-views/{id}` , apart from publishing it — and, `Since 1.10` , renaming it: `name` , `description` and `icon` are accepted and kept across package updates. A rename must carry the current `updateIdentifier` , or it answers `409 UPDATE_CONFLICT` . `description: null` clears the description; `icon: null` clears the icon only with the query parameter `clearIcon=true` . Any other change is refused.

Routes for the platform's own services are not callable through the gateway and are not listed here; you may still see them in traces.

Machine-readable specifications

Each area of the platform publishes its own OpenAPI 3 document; there is no merged document. Where your operator has the documents switched on, get them from the area's prefix:

Area	Document	Explorer
Core data platform	`/api/openapi.json` (or `.yaml`)	`/api/explorer`
Any other area	`/api/<area>/openapi.json` , for example `/api/reaction/openapi.json`	`/api/<area>/explorer`

What to know before you generate a client from them:

- **The documents describe the envelope.** `Since 1.8` each document declares the envelope schemas and states the rule; each operation's success schema describes the inner `result` , so generated types fit once your client unwraps `result` . A document exported before 1.8 describes bare bodies — regenerate it.
- **Some areas have no document**: packaging, logs, dependency and integrations. Their routes are on this page and in their own guides.
- **Internal operations are left out** of every document.

How to generate and keep a client current is in Generating a client.

A document for one view `Since 1.10`

A view can publish a small OpenAPI 3.0 document of its own. Turn on **Swagger document** in the view's display options, then:

```
GET /api/v1/views/{viewId}/openapi.json
```

Answer	Meaning
`200`	The document (`application/json`)
`403` `VIEW_ACCESS_DENIED`	The view exists but you cannot open it
`404` `DATA_VIEW_NOT_EXIST`	The view does not exist, or its entity is archived

> ⚠ **The answer is not wrapped, but the operations it describes are**
>
> Parse this response as a bare OpenAPI document — there is no `result` . The five operations the document describes **do** answer inside the envelope, like the rest of the API, and the document says so.

It describes exactly five operations, each through the view:

Documented operation	What it does
`GET /v1/v/{viewId}/v1/entities/{id}`	The entity's schema
`GET /v1/v/{viewId}/v3/entities/{entity}/entries`	List entries
`GET /v1/v/{viewId}/v1/entities/{entity}/entries/{id}`	Read one entry
`POST /v1/v/{viewId}/v1/entities/{entity}/entries`	Create an entry
`PATCH /v1/v/{viewId}/v3/entities/{entity}/entries/{id}`	Update an entry

- The view id and entity id are already filled into each path, so the rows are copy-pasteable; only the entry `{id}` stays a placeholder.
- Reading the entity schema always needs a credential, even on a view open to visitors. The four entry operations work anonymously when the view allows it.
- The entry `value` schema follows the view's own field selection and types. It is not filtered by sensitivity: a sensitive field the view shows appears by name and type (never with a value).

With the switch on, any signed-in user who can open the view can fetch its document; a user with system Developer, Edit or Admin can fetch it regardless of the switch. Do not build on anonymous access to the document itself. As a source for a client, this document is narrower and more stable than the full one — see Generating a client.

Quick reference

- **Host and paths:** the tenant's own host; `/api` for the core data platform, `/api/<area>` for the others; the version in the path; JSON in and out.
- **Credentials:** `x-app-token: <token>` or `Authorization: Bearer <token>`. Never set identity headers yourself.
- **Success:** `{ "success": true, "result": … }`, except `GET /api/v3/…` (bare payload `Since 1.10`), downloads and OpenAPI documents. Unwrap when `success: true` is present.
- **Errors:** `{ "success": false, "error": { "code", "message": {en, ar}, "traceId" } }`. Branch on `error.code`; a `500` may be permanent; `503` with `Retry-After` means wait.
- **Paging:** `limit` + `skip` / `offset`; no cursors; no server maximum.
- **Filtering:** `filter` / `where` (v1) or `filterCriteria` / `whereCriteria` + `search` (v2, v3).
- **Concurrency:** send back the `updateIdentifier` you read; `409 UPDATE_CONFLICT` on a mismatch.
- **Long-running work:** start, then poll a status call, or follow it live.

> **Go deeper**
>
> For operators: Gateway operations · The security boundary · For platform engineers: HTTP API — internals

Authentication & OAuth

This page is for developers who need a credential the platform trusts: to sign people in to their own app, or to let a server or script call the API. You will find your tenant's OpenID provider, get a client registered, run the right OAuth flow, call the API with the resulting token, and keep the session alive.

A token proves **who** is calling. What the caller may do is decided separately, by that user's permissions in the tenant.

Pick a method

Your caller	Method
A person in your web or mobile app	Authorization code with PKCE
Your own server acting as itself	An API token owned by a dedicated application user. Not client credentials: that token has no user, so it can reach almost nothing
A script, service or automation tool with no person present	An API token — the simplest option
A trusted first-party script that holds a user's password	The password grant
A visitor who is not signed in, on a public page	An anonymous token

Choosing a credential explains the trade-offs in more depth.

Find your tenant's OpenID provider

Every tenant has its **own** OpenID provider, with its own issuer, signing keys and clients. You reach it through the tenant's host, so you never pass a tenant id.

1. Ask the tenant for its host, for example `https://acme.example.com`.

2. Read the tenant's public configuration. It names the provider (`authority`) and the web client the tenant's own portal uses:

   ```
   curl -s https://acme.example.com/api/identity/v1/tenant-config
   ```

   ```
   {
     "schemaVersion": 1,
     "appId": "<tenant-id>",
     "authority": "https://<authority-host>",
     "clientId": "<web-client-id>",
     "scope": "<scopes-the-web-client-requests>"
   }
   ```

3. Load the discovery document from the authority and take every endpoint from it — do not hard-code paths:

   ```
   curl -s https://<authority-host>/.well-known/openid-configuration
   ```

Most OAuth 2.0 / OpenID Connect client libraries need only the authority URL and do step 3 themselves.

> ⚠ **Validate tokens against the tenant you call**
>
> `Since 1.10`, in a multi-tenant deployment, the token issuer and the signing keys belong to the tenant. Validate a token's `iss` against the authority of the tenant you called, never against a cell-wide one. After a key rotation the previous key stays in the tenant's key set for 24 hours, so refresh your key cache at least daily. A tenant whose identity is not set up yet answers `503`.

Get a client registered

There is no self-service client registration. Ask the platform operator to register an OAuth client for your application, and tell them:

- whether it is a **public** client (a browser or mobile app, no secret, PKCE) or a **confidential** one (a server that can keep a secret);
- the grant types you need — authorization code with refresh tokens, client credentials, or the password grant;
- your **redirect URIs** and post-logout redirect URIs;
- the scopes you need, usually `openid email profile offline_access`;
- if you need it, a shorter or longer access-token lifetime for this client.

Redirect URIs must match exactly, with one exception: a single wildcard may stand for a **whole subdomain** in the host name, such as `https://*.apps.example.com/auth-callback`. A wildcard in the path, more than one wildcard, a partial label such as `*suffix.example.com`, and a wildcard over a public suffix such as `*.co.uk` are refused.

You receive a `client_id`, and for a confidential client a `client_secret`. Keep the secret on the server only. (Operators register clients as described in Identity server — OIDC clients.)

Sign users in (authorization code with PKCE)

This is the flow for any app where a person signs in. The hosted sign-in page handles passwords, passkeys, second factors and external identity providers for you.

1. **Send the user to the authorization endpoint** with a PKCE challenge:

```
https://<authority-host>/auth
  ?client_id=<client-id>
  &response_type=code
  &redirect_uri=https://app.example.com/auth-callback
  &scope=openid%20email%20profile%20offline_access
  &state=<random-state>
  &code_challenge=<S256-challenge>
  &code_challenge_method=S256
```

2. **Receive the code** on your redirect URI, and check that `state` matches. The code is valid for 10 minutes.

3. **Exchange it for tokens** at the token endpoint:

```
curl -s -X POST https://<authority-host>/token \
  -d grant_type=authorization_code \
  -d code=<code> \
  -d redirect_uri=https://app.example.com/auth-callback \
  -d client_id=<client-id> \
  -d code_verifier=<pkce-verifier>
```

A confidential client also authenticates with its `client_secret`.

4. **Call the API** with the access token — see Call the API.

Public clients must use PKCE. The implicit flow is still accepted for older clients; do not use it for new work.

Start sign-in with a specific provider `Since 1.10`

If the tenant offers external sign-in (for example a national identity provider or Microsoft Entra ID), add `idp_hint=<provider name>` to the authorization request to skip the sign-in page and go straight to that provider. Use the provider's unique name, not its display label; ask the tenant administrator for it. An unknown or inactive name is ignored and the normal sign-in page is shown.

When a domain offers exactly one external provider and nothing else, the sign-in page is skipped for every client by default. Do not rely on the sign-in page being shown (for example, to display its **Cancel** link) on such a domain.

When the user cancels `Since 1.4`

If the user selects **Cancel** on the sign-in page, or leaves it idle for 10 minutes, your redirect URI receives `error=access_denied` with the `state` you sent. Treat it as "the user declined", not as a failure — for example, return to your home page.

Refresh tokens

You get a refresh token when you request the `offline_access` scope, or when your client is a public web client. Refresh before the access token expires:

```
curl -s -X POST https://<authority-host>/token \
  -d grant_type=refresh_token \
  -d refresh_token=<refresh-token> \
  -d client_id=<client-id>
```

- A refresh token lives 14 days, and can be renewed for at most one year in total.
- Refresh tokens **rotate**: a response may carry a new refresh token. Always store the one you received last; an old one may stop working.
- A public web client's rotated refresh token keeps the remaining lifetime of the one it replaced, so a browser session cannot be extended forever.

Designing a client that keeps sessions alive across expiry and revocation is covered in Token lifecycle.

Server-to-server: client credentials

A confidential client can get a token that acts as the client itself, with no user:

```
curl -s -X POST https://<authority-host>/token \
  -u "<client-id>:<client-secret>" \
  -d grant_type=client_credentials
```

Send the client credentials the way your client is registered for — HTTP Basic as above, or `client_id` and `client_secret` as form fields. The token lives 10 minutes unless your client was registered with another lifetime. It carries no session. Request a new one when it expires.

What the token can reach: almost nothing. It identifies the client, not a user, so the gateway passes no user id on, and the platform resolves the caller to a profile with no permissions. The request is authenticated, and then nearly every read comes back refused or empty. On the real-time socket it connects, but subscribing to notifications is refused, because that needs a user. For your own server acting as itself, use an API token owned by a dedicated application user instead. Choosing a credential explains why.

Trusted scripts: the password grant

A trusted first-party client may exchange a user's username and password directly:

```
curl -s -X POST https://<authority-host>/token \
  -d grant_type=password \
  -d username=<username> \
  -d password=<password> \
  -d scope="openid email profile offline_access" \
  -d client_id=<client-id> \
  -d client_secret=<client-secret>
```

The response carries `access_token`, `expires_in`, `token_type: Bearer`, `scope` and, when allowed, `refresh_token`.

Answer	Meaning
`400 invalid_grant`, "invalid credentials provided"	Wrong username or password. Repeated failures lock the account for an increasing time.
`400 invalid_grant`, "account is locked"	The account is locked. Wait, or ask an administrator.
`400 invalid_scope`	You asked for a scope the client is not allowed.

Use this grant only where the script can safely hold a password. It bypasses second factors and external sign-in, so prefer the authorization code flow anywhere a person is present.

Call the API

Send the access token on every request to the tenant's host:

```
curl -s https://acme.example.com/api/v2/entities/<entity>/entries/count \
  -H "Authorization: Bearer <access-token>"
```

- A `401` with `AUTHENTICATION_ERROR` means the token is missing, expired or no longer active: refresh it, or sign the user in again.
- For live updates over a socket, use the **same** access token; see Real-time.
- An `x-app-token` header, if present, is used instead of the bearer token.

Token lifetimes

Token	Lifetime
Access token	1 hour, unless your client was registered with another lifetime
Client-credentials token	10 minutes, unless your client was registered with another lifetime
Authorization code	10 minutes

Token	Lifetime
ID token	1 hour
Refresh token	14 days (rotated public-client tokens keep the remaining lifetime)
Grant and sign-in session	14 days
Anonymous token	10 minutes

Scopes and claims

Scope	Claims released
`openid`	`sub`
`email`	`email`, `email_verified`
`profile`	`preferred_username`
`offline_access`	A refresh token

A user's custom claims, when the tenant defines them, are released as well. `sub` is the user's id in the tenant.

Anonymous visitors `Since 1.10`

A browser page that shows public content to visitors who are not signed in can get an **anonymous token** without any credential:

```
curl -s -X POST https://acme.example.com/api/identity/v1/anonymous/token
```

```
{ "access_token": "<token>", "token_type": "Bearer", "expires_in": 600 }
```

The token carries only the `anonymous` scope, so it reaches only what the tenant opens to visitors. Never put a client secret in browser code: earlier integrations that fetched an anonymous token with a system client's secret must switch to this endpoint, because that system client may not exist.

The tenant configuration shown in Find your tenant's OpenID provider is also public, so a browser app can start without any deployment-specific settings.

Validate a token in your own service

If you build a service that receives the platform's access tokens, do not decode them yourself. Ask the provider: call the **introspection endpoint** from the discovery document with your client's credentials:

```
curl -s -X POST https://<authority-host>/token/introspection \
  -u "<client-id>:<client-secret>" \
  -d token=<access-token>
```

(Use form fields instead of HTTP Basic if your client is registered that way.) Accept the token only when the answer has `"active": true`, and use its `sub` as the user id. Check `exp` and `client_id` as your service requires.

Sign out and revoke

- **Sign out:** send the user to the provider's end-session endpoint (from discovery) with your registered `post_logout_redirect_uri`. If the user signed in through an external provider that supports it, they are signed out there too.

- **Revoke a token:** call the revocation endpoint (from discovery) with the token and your client credentials.
- **Back-channel logout** `Since 1.6`: if your client is registered with a back-channel logout URI, the provider notifies it when the session ends.

Act on behalf of another user

A person can act for a colleague during an authorized period — this is **delegation**, set up by an administrator in Delegation [Administration]. To use an active delegation on a call, add its id in the `X-Delegation-Id` header. The action is recorded as done for the colleague and by the person, so history stays correct. Outside the delegation's period the request is refused with `403 DELEGATION_EXPIRED`. Delegation is for people covering for each other, not for machine integrations.

API tokens

For a script or service that cannot run an OAuth flow, an **API token** sent in the `x-app-token` header is simpler. It acts as an application user that a tenant administrator creates — see Create an API token for an integration [Administration] — and you use it as described in Use an API token.

Good to know

- **Second factors and external providers are handled by the hosted sign-in page.** Depending on the tenant, users may be asked for an authenticator-app code, a code by SMS, email or `Since 1.10` WhatsApp, or a passkey. Use the authorization code flow and let the page present what the tenant requires, rather than building your own sign-in.
- **A tenant under maintenance** `Since 1.10` answers `503 TENANT_IN_MAINTENANCE` with `Retry-After` on sign-in, token and refresh calls — for example while it is restored from a backup. Wait and retry; do not sign the user out. See Calling the HTTP API.
- **Tokens and the tenant.** An access token from the authorization code or refresh flow is bound to the tenant's sign-in host, so it cannot be replayed against another tenant.

Go deeper

For administrators: Identity & identity providers [Administration › Sign-in methods & single sign-on] · For operators: Identity server · Gateway operations — authentication · For platform engineers: Identity, OAuth and API tokens — internals

Real-time: live events and calls

This page is for developers who want changes pushed to them instead of polling: keep an open record current, show notifications as they arrive, or follow a job you started over HTTP. It also covers the tokens for audio and video calls.

The live channel is **push-only and best-effort**. You subscribe and receive events; every write, and every authoritative read, still goes through the HTTP API.

Connect

The channel speaks **Socket.IO v4** — a plain WebSocket client cannot connect. Use a Socket.IO v4 client library and connect to the **tenant's own host**, on the default Socket.IO path:

```
import { io } from 'socket.io-client';

const socket = io('https://acme.example.com', {
  transports: ['websocket', 'polling'],
  // A function, so every reconnect sends a fresh token
  auth: (cb) => cb({ token: getAccessToken() }),
});

socket.on('connect_error', (err) => console.error('socket refused:', err.message));
```

- **The credential is an OAuth access token** — the same one you use for HTTP calls — or, on public pages, an anonymous token. An **API token cannot open a socket**: a service that calls the API with `x-app-token` needs a user's OAuth token for the socket. For a service, that means the password grant for a dedicated service account, which must be an ordinary user: an application user has no password and cannot sign in. A client-credentials token is not enough. It opens the socket, but it has no user, so `subscribeToNotifications` is refused.
- **A handshake without a token is refused, and so is** an invalid or inactive token (`Invalid token`).
- **The token is checked once, at the handshake.** Every reconnect is a new handshake — supply a fresh token each time, as in the example.
- **Connect to the tenant's own host** `Since 1.10`. On a multi-tenant deployment the socket belongs to the tenant of the host you connect to. A host that matches no tenant is refused with `Tenant not resolved`, and a token from another tenant is not accepted.

A headless worker or backend process follows the same rules; the extra care it needs (second credential, reconciliation) is in Subscribing to events.

Subscribe

A new socket receives nothing until you subscribe. Each subscription has an acknowledgement — **wait for it** before assuming you are subscribed, because some failed joins report no error.

Emit	Payload	Acknowledgement	You then receive
`subscribeToNotifications`	none — your identity comes from the token	`subscribedToNotifications {userId}`	Your notifications and job completions
`entries/subscribe-to-entry` `Since 1.4`	`{entityId, entryId}`	`subscribedToEntry {entityId, entryId}`	Changes to that record

Emit	Payload	Acknowledgement	You then receive
`entries/unsubscribe-to-entry` `Since 1.4`	`{entityId, entryId}`	`unsubscribedFromEntry {entityId, entryId}`	—
`subscribeToJobExecution` `Since 1.6`	`{messageId}`	`subscribedToJobExecution {messageId}`	The job's completion
`unsubscribeFromJobExecution` `Since 1.6`	`{messageId}`	`unsubscribedFromJobExecution {messageId}`	—

A missing `messageId`, `entityId` or `entryId` produces an `error` event (`{message}`) instead of a subscription. `subscribeToNotifications` on a socket with no resolved user emits `error` and disconnects.

Events you receive {#server—client-events}

Event	Arrives when you subscribed to	Meaning
`notification`	Notifications	A notification was created for you; the payload is the notification record.
`entries/value-updated`	The record	Field values changed. `Since 1.10` the payload may carry the record's `updateIdentifier`.
`entries/workflow-updated`	The record	The workflow moved: a step reached or left, an action executed, an owner changed, an ad-hoc step added. Carries `eventName`.
`entries/comments-updated`	The record	A comment was added or changed. Carries `eventName`.
`entries/archived` `Since 1.6`	The record	The record was archived.
`entries/restored` `Since 1.6`	The record	The record was restored.
`entries/virtual-entry-created` `Since 1.8`	The source record	A virtual record was created from it.
`entries/virtual-entry-updated` `Since 1.8`	The source record	A virtual record created from it was updated.
`osos/infrastructure/job-executed`	The job, **and** notifications	A background job finished. The payload carries `messages` and `result`.
`icons-updated`	— (every socket of the tenant)	The icon set changed.
`request-entry-modified`	— (every socket of the tenant)	A request was created, updated or archived: `{requestId, originalEntryReference, operation}`.
`packages/changed` `Since 1.10`	— (every socket of the tenant)	A package's review or store state moved: `{ $name, $version: 1, packageId }`. Re-read the package over HTTP.
`error`	—	A subscribe or unsubscribe call was invalid.

Recipes

Keep an open record current

```
socket.emit('entries/subscribe-to-entry', { entityId, entryId });
socket.on('entries/value-updated',    (e) => refreshFields(e));
socket.on('entries/workflow-updated', (e) => refreshWorkflow(e));
socket.on('entries/comments-updated', (e) => refreshComments(e));

// when the screen closes:
socket.emit('entries/unsubscribe-to-entry', { entityId, entryId });
```

Warn when someone else saved first `Since 1.10`

`entries/value-updated` may carry the record's current `updateIdentifier`. Compare it with the one you read:

- **No token yet, or the same token** — nothing changed for you.
- **A different token** — someone else saved after your last read. Tell the user, offer a reload, and stop an open editor from saving over the newer version.

Two cautions:

- This is a courtesy, not the protection. A client that missed the event still gets `409 UPDATE_CONFLICT` from the write itself.
- Your own save causes several follow-up writes (indexing, automations), each with a new token. Ignore the comparison while your own save is in flight, or you will warn the person who just saved.

Follow a job you started over HTTP

```
// An HTTP call answered with a messageId
socket.emit('subscribeToJobExecution', { messageId });

socket.on('osos/infrastructure/job-executed', (evt) => {
  if (evt.messageId !== messageId) return;   // may also arrive via notifications
  confirmOverHttp(messageId);                // the status call is authoritative
});
```

Subscribe right after starting the job: an event that fires before your subscription is not replayed. Confirm the final state with the job's status call — see Long-running work.

Delivery rules

- **Best-effort, no replay.** Record events are sent only while someone is subscribed, and nothing is kept for a disconnected client. After every connect or reconnect, subscribe again and re-read what you show over HTTP.
- **Notifications are not filtered by space.** You receive notifications from every space you belong to. `Since 1.10` the platform's web client hides those of other spaces itself; a custom client should filter on the notification's `value.space`.
- **Delegated actions notify two people.** When a notification comes from an action done on someone's behalf, both the recipient and the person who acted receive it.
- **Unsubscribing quickly can suppress virtual-record events.** They are sent about a second after the change, and only if you are still subscribed.

Audio and video calls

The platform issues tokens for audio and video channels run by Agora. It does not carry the media itself. Get a token through the API with your normal credential, then join the channel with the Agora client SDK:

```
curl -s "https://acme.example.com/api/rtc/<channel>/publisher?expiry=3600" \
  -H "Authorization: Bearer <access-token>"
```

```
{ "success": true, "result": { "rtcToken": "<token>", "userId": "<your-user-id>" } }
```

Part	Values
`<channel>`	The channel name. Required.
role	`publisher` (sends audio/video) or `audience` (receives only). Anything else is `400`.
`expiry`	Seconds the token is valid. Optional, default `3600`.

The token is issued for **your** user id; you cannot request one for another user. The Agora app id the client needs comes from your operator's configuration.

> **Go deeper**
>
> Headless clients: Subscribing to events · For operators: Configuration reference — web-socket-service · For platform engineers: Real-time — internals

Webhooks & integrations

This page is for developers who connect OSOS to other systems through events: calling your system when something happens in a tenant, connecting Zapier, n8n or similar tools, and letting those tools trigger automations in the platform.

Example: when a purchase request is approved, notify the finance system. A builder adds a reaction on the Purchase Requests entity that runs on the approval and uses the **Call API** action to post the amount and the requester to your endpoint. Your job as the integrator is the receiving endpoint and the contract.

Two related surfaces have their own pages: reading platform data from Grafana (Grafana dashboards `Since 1.6`), and running your own code inside a process (Functions).

Choose an approach

You want to…	Use
Call your system when a record changes or a workflow moves	The Call API action
Run code you wrote, with the platform's data	The **Execute Function** action — see Functions
Feed Zapier Zaps from platform events	Notify Zapier
Let a Zap or another tool make something happen in the platform	Zapier actions, or the HTTP API with a token
Keep a screen live	Real-time
Check or rewrite incoming requests before they are handled	Request interceptors

Call your system when something happens

Outbound calls are **reaction actions**. There is no separate webhook-subscription screen: a builder adds a reaction to an entity, picks the event that triggers it, and adds one of these actions:

Action	What it does
Call API	Sends an HTTP request — `GET`, `POST`, `PATCH`, `PUT` or `DELETE` — with the URL, headers and body the builder defines, filled with values from the record. A `2xx` answer counts as success, and the response is kept in the run's history.
Notify Zapier	Posts the record to the Zapier hooks that subscribed (see below).

Setting up the reaction is a builder task — see Reactions & automation [Build Apps]. What you agree with the builder is the contract: method, URL, headers, body fields and what your endpoint answers.

Design your receiving endpoint

- **Authenticate the call yourself.** The platform does not sign outbound requests. Ask the builder to send a secret you issue — for example in an `Authorization` or `X-Api-Key` header — and check it on every request.
- **Keep the secret out of history.** Each run's request and response are stored in the reaction's history. Values sent under well-known credential headers (`Authorization`, `Cookie`, `X-Api-Key`), values read from sensitive fields, and any value the builder marks **redact in history** are stored as `****` `Since 1.6` — see Redacting secrets in reaction history [Build Apps › Reactions & automation].

- **Answer quickly with a `2xx`.** `Since 1.8` the whole exchange must finish within 30 seconds by default, and anything other than a `2xx` is recorded as a failure.
- **Expose a public address.** Calls to private and reserved networks (internal IP ranges, cloud metadata addresses) are refused, unless the platform operator has allowed that destination. At most 5 redirects are followed, and request and response bodies are capped at 10 MiB. `Since 1.10` a few header names are reserved for the platform and refused when the reaction is saved (`400 RESERVED_HEADER_NAME`); see Call API: what the platform allows [Build Apps › Reactions & automation].
- **Make it idempotent.** Treat each call as "this record changed" and re-read the record over the HTTP API if you need the authoritative state.

To see exactly what a call sent and received, a system administrator opens the run in **Reaction Logs** `Since 1.10` — see the Reaction Logs API [Build Apps › Reactions & automation].

The Zapier connector

The connector lets Zaps subscribe to platform events, receive the records when those events fire, and push actions back into the platform. It is long-standing — it predates the 1.2 line.

From the platform to Zapier {#outbound-flow-platform—zapier}

When a reaction runs **Notify Zapier**, the platform posts this payload to every subscribed hook URL:

```
{
  "dataEntity": { "id", "specificationVersion", "normalizedName", "name" },
  "dataEntry":  { "id", "value", "specificationVersion", "dataEntity",
                  "updateInfo", "creationInfo" },
  "trigger":    { "eventType", "actionOptions", "criteria" }
}
```

Manage trigger subscriptions

A Zapier app subscribes with REST hooks under `/api/integrations`:

Call	Purpose	Body
`POST /api/integrations/v1/zapier/trigger/subscribe`	Subscribe a hook	`{ hookUrl, zapName, triggerType, entity, entry? }`
`GET /api/integrations/v1/zapier/trigger/subscriptions`	List subscriptions	—
`DELETE /api/integrations/v1/zapier/trigger/unsubscribe/{subscription_id}`	Unsubscribe	—
`GET /api/integrations/v1/zapier/trigger/perform-list`	A sample payload for the Zap editor's test step	—

From Zapier to the platform {#inbound-flow-zapier—platform}

A Zap action makes something happen in the platform by posting an event. This call needs an OAuth access token (the tenant's Zapier OAuth client uses the authorization code flow — see Authentication & OAuth):

```
curl -s -X POST https://acme.example.com/api/integrations/v1/zapier/action \
  -H "Authorization: Bearer <access-token>" \
  -H "Content-Type: application/json" \
  -d '{"profile": {"entityId": "<entity-id>", "zapId": "<zap-id>",
       "event": {"eventType": 2, "output": {"fileUrl": "https://…"}}}}'
```

The platform runs every active reaction on that entity whose trigger is a **Zapier event** with the same Zap id. The `zapId` is the link between a Zap and its reactions. `output` is free-form data from Zapier that the reactions can use. `eventType` is one of:

Value	Zapier event
`0`	Google spreadsheet created
`1`	Google spreadsheet row added
`2`	PDF file created
`3`	Google Calendar event created
`4`	Gmail email received
`5`	Google Calendar event cancelled

The call answers success even if the event could not be passed on; check the reaction history to confirm the reactions ran.

Known limitations

- **The sample data is fixed.** `perform-list` returns one synthetic record, not live data.
- **There is no inbound "catch hook".** `PATCH /api/integrations/v1/zapier/webhook/catch` accepts a request and does nothing. Use the action call above instead.

Request interceptors

A deployed function can run before an incoming request is handled, for requests matching a path and HTTP method. It can:

- **rewrite the request** — change its body, headers or query before it continues; or
- **answer it directly** — return its own status, body and headers, so the request never reaches the platform.

Interceptors run in priority order. Only the before-handling stage exists; there is no interceptor for responses.

View redirections in your own client

A builder can make a view send different people to different places — for example, visitors to a public page and staff to an internal view. When your own client opens a view for an end user (`GET /api/v2/views/end-user/{id}` , and the equivalent for pages), the answer includes `redirection` : `null` , or the destination `{ "type": <type>, "redirectTo": "<id or URL>" }` . **Your client must act on it**; the platform does not redirect by itself.

`type`	Destination
`0`	A fixed URL
`1`	A data view
`2`	A page

`type`	**Destination**
`3`	A report

Rules are checked in priority order (lowest number first), and the first one that applies to the user wins: everyone, signed-in users, visitors, or named users and groups. A rule set to **Restricted** applies only to the users and groups it names. Redirection rules are long-standing.

Other integration surfaces

- **Short links.** The platform shortens links in the emails and SMS it sends, as `https://<host>/sh/<code>`. It is not an API for clients.
- **QR codes.** Views can show QR codes, generated by a platform function.
- **Directory import.** Users and groups can be imported from Active Directory (LDAP) and Microsoft Entra ID. These are batch jobs run by administrators — see Directory sync [Administration › Users from your company directory].
- **Platform events.** The richest surface is the platform's own event and command vocabulary; see Data model & events.

> ⓘ **Go deeper**
>
> For builders: Reactions & automation [Build Apps] · For platform engineers: Integrations — internals

Grafana Dashboards

> **Availability**
>
> **Since 1.6.**

OSOS ships a **Grafana data source plugin** — `Apptomator` — that lets you build Grafana dashboards on top of OSOS data. Its defining property is that an administrator builds a dashboard **once**, and every viewer who opens it sees only the rows their own OSOS access allows.

That is not a filter you configure. The plugin holds no credentials of its own: every request travels through Grafana's data source proxy with the signed-in viewer's OAuth token attached, so the gateway and permissions [Administration › Permissions & access] evaluate the request exactly as they would for that person in the OSOS web portal. Two people opening the same panel get different rows, and neither of them can widen what they see.

This page is for the person **building panels**. Installing the plugin, wiring Grafana's OAuth against the OSOS identity service, and publishing the plugin archive are covered in Grafana Datasource Plugin in Setup & Maintenance.

What you can and cannot build

The plugin executes **saved OSOS queries** — it is not an ad-hoc query builder. Everything a panel can read has already been modelled in OSOS:

You have	A panel can
A data view [Build Apps › Data Views & Faces]	Read its entries as rows
A saved data query [Build Apps › Queries & filters]	Apply it as a quick filter on those rows
A saved **statistic query**	Return the aggregate it computes, chart-shaped

Two consequences follow, and they shape how you design for Grafana:

- **A view is required for every panel.** Entries are only ever read through a view, because a view's execution plan is what the gateway authorizes. There is no entity-level shortcut — if the data should be on a dashboard, it needs a view.
- **Anything you want aggregated should be a statistic query.** Aggregation in OSOS is far cheaper than pulling rows into Grafana and grouping them there, and the result arrives ready to chart.

Setting up the data source

Done once per Grafana instance, under **Connections → Data sources → Add new data source → Apptomator**.

Setting	Value
URL	The OSOS gateway base URL — the customer's **real domain**, e.g. `https://osos.example.com`
Forward OAuth Identity	**ON.** Not optional
Save & test	Confirms the URL, the token forwarding and the gateway in one call

Two of these deserve emphasis:

- **The URL must be a host that resolves to the customer's domain.** The gateway resolves both the domain and the space from the `Host` header, and a request whose host does not resolve is rejected with `MISSING_DOMAIN_ID`. An internal cluster address will not work.
- **Forward OAuth Identity is what makes the model work.** With it off, the proxy forwards no viewer token and the per-viewer permission model silently stops applying — panels do not fail, they just stop being personal.

Access mode is pinned to `proxy` by the plugin and is not offered as a choice. Direct (browser) access bypasses the proxy entirely, so Grafana would never attach the viewer's token.

Building a panel

The query editor is a cascade that mirrors the OSOS access model — module, then view, then query. `Since 1.8` the cascade starts at **Module**: the previous Entity picker is gone, and a panel saved before this change self-heals its module the first time you open its editor.

Field	Meaning
Type	`Rows (Data Query)` for row-level data, `Statistic Query` for an aggregate
Module	The module holding the view to read from — the cascade's starting point
View	**Required.** The view whose execution plan the gateway authorizes, and which determines the entity [Build Apps › Create and design an entity] — the entity is never picked directly; it travels with the view and is what gets saved on the query
Saved query	Optional for rows (a quick filter on top of the view); **required** for statistic panels
(parameters)	One input per parameter the selected saved query declares. Values are free text, so dashboard variables such as `$department` work
Time field	Rows only. A `Date` / `DateTime` field to expose as the frame's time column — and to filter the panel by the dashboard time range
Limit	Rows only. Defaults to 1000; the server clamps it to its own ceiling regardless
Fields	Rows only. Comma-separated column names to keep; empty means all
Metadata	Rows only, off by default. Shows the `_meta*` bookkeeping columns

`Since 1.8`, this also changed who can build a panel: discovery now goes through the same end-user endpoint an OSOS dashboard itself uses, so an ordinary OSOS user's own view access is enough to author a working panel. Before, the underlying discovery calls required an OSOS admin-scoped token, so only an admin's credentials could build one. Grafana's own authoring permission — who may open the query editor at all — is unchanged and remains the separate, pre-existing gate; see `analytics.oauth.authoringGroup` in Grafana Datasource Plugin.

Text inputs commit when you leave the field or press Enter, not on every keystroke.

Parameters and dashboard variables `Since 1.8`

A saved query's parameters appear as inputs, and each accepts a Grafana dashboard variable. `$department` in a parameter box makes the panel follow the dashboard's variable picker.

One rule matters when a parameter is bound to a **Number** or **Boolean** field: **a blank value is sent as "unset", never as `0` or `false`**. That is deliberate — a blank Number silently becoming `0` would filter the data to nothing and look like an empty result rather than a mistake. Leave a parameter blank to mean "don't filter on this".

> ⚠ **Caution**
>
> On a view with **filter first** enabled, each parameter value does *two* jobs — it is used as a query parameter *and* injected as an equality match on the field of the same name. That is the view's own behaviour, not the plugin's, but it means the same value narrows the rows twice.

Time fields and the dashboard time picker `Since 1.8`

Without a Time field, the dashboard's time picker has no effect on the panel. Setting one does three things at once: it becomes the frame's time column, the rows are sorted by it, and the panel is narrowed to the picker's range server-side.

The dropdown offers:

- the entity's own `Date` / `DateTime` fields;
- `Date` / `DateTime` fields **inside blocks**, shown as `Basic info / Birthdate`;
- the two system audit dates, **Created at** and **Last updated**.

`_metaData.createdAt` and `_metaData.updatedAt` are deliberately **not** offered. They carry the same instants, but they are computed when an entry is read rather than stored, so no filter could ever match them — offering them would give you a time column the picker silently ignores. **Created at** and **Last updated** come from stored fields and do work with the picker.

Range bounds are formatted to match how the field stores its values — a `Date` field holds `"2026-05-19"`, a `DateTime` field a full ISO instant — and date-only bounds use the **dashboard's timezone**, so the days match what the picker shows.

> ⓘ **Note**
>
> A panel saved before time-range filtering existed carries no field type, and silently ignores the picker. Open its editor once and the type is filled in automatically.

How OSOS values are displayed

Most OSOS field values are objects, so each is flattened into a display cell:

Value	Shown as
Reference [Build Apps › Link entities]	its resolved main title, e.g. `Super Admin`
Category	its resolved label, e.g. `Approved`
File	`[file] avatar.jpg` — a placeholder; attachments are not rendered
Localized value	the viewer's Grafana language, falling back to English
Multi-value field	its parts joined
Block (`address: {city, country}`)	**one column per sub-field**: `address.city` , `address.country`

Blocks [Build Apps › Choose a field type] expand into columns exactly as they do in the OSOS table view. The block name is kept as a prefix because Grafana column names must be unique, so two blocks that both have a `city` stay distinct. A

block that cannot become columns — one nested inside a repeated field — falls back to `city: Damascus, country: Syria` in a single cell.

Numeric OSOS fields stay numeric, so they remain summable. Every object-valued field becomes text, and text can only be counted or grouped — not summed.

The `_meta*` columns — `_metaData`, `_metaOrder`, `_metaCurrentSteps`, `_metaWorkflowHistory`, `_metaAdhocSteps` — sit on every entry and explode into many columns, so they are hidden unless you turn **Metadata** on. Naming one in **Fields** shows it regardless.

Audit columns `Since 1.8`

> ⚠ **Breaking change — `systemTimeField` is removed**
>
> The `systemTimeField` panel option no longer exists — **removed, not deprecated**. A dashboard that used it to surface a single system date as a column stops working with this plugin version, and no query-editor setting recreates the old one-column behaviour, because the four columns below are unconditional. **Re-author any such panel to read `creationInfo.doneAt` / `updateInfo.doneAt` directly.**

Every row frame now carries four audit-stamp columns, appended **after** the entry's own fields so they never bump a Time field you chose out of the frame's leading position:

Column	Shows	Type
`creationInfo.doneAt`	when the entry was created	time, always
`creationInfo.doneBy`	who created it — resolved display name, or the raw actor id if it can't be resolved	text
`updateInfo.doneAt`	when the entry was last updated	time, always
`updateInfo.doneBy`	who last updated it — resolved display name, or the raw actor id if it can't be resolved	text

They are unconditional — no setting turns them on — so a breakdown of entries by creation day, or a count grouped by author, needs only a **Group by** transformation, with no Time field chosen first. Name one or more of them in **Fields** to keep only those columns on a panel that doesn't want all four. Both `…doneAt` columns are typed as time whether or not either is the panel's chosen Time field, so bucketing one by day is a *Format time* transformation away.

A reference that no longer resolves — a deleted `doneBy` actor, for instance — now falls back to its raw id, the same as any other unresolved reference, rather than a fixed rendering bug that used to show the literal string `"{}"` in the cell.

Charting

Panels are not limited to tables — pick any visualization. What a chart needs is a numeric column and something to plot it against, so **which query type you choose matters more than which visualization you pick**.

Aggregates — use a Statistic Query. The shortest path to a bar, pie or stat panel, because the aggregation happens in OSOS. The frame comes back chart-shaped already: a `category` column (the group-by), one column per sub-grouping, and a numeric `value`. Point a **Bar chart** or **Pie chart** at it and it renders with no transformations at all.

Counts and sums over rows — use a Rows query plus a transformation. Rows come back un-aggregated, so add **Transform** → **Group by**: group by the column you want on the x-axis, and set another field to *Calculate* → *Count* (or Sum/Mean). Then switch the visualization to Bar chart.

Trends over time — use a Rows query with a Time field. The column becomes a real Grafana time column, rows are sorted by it and narrowed to the picker's range, so a **Time series** panel draws in order and follows the picker. Pair it with a numeric column, or with **Group by** on a time interval to count entries per day.

Two things worth knowing:

- **Multi-series charts from a sub-grouped statistic query** arrive in long format (`category`, `department`, `value`). Grafana's bar chart wants one column per series, so add **Transform** → **Partition by values** on the sub-group column, or **Rows to fields**.
- **Combining two queries in one chart** is done with **Transform** → **Join by field** — the plugin does not join server-side. Add both queries as separate targets on the panel.

Embedding a dashboard in OSOS

A Grafana dashboard can be shown inside OSOS using a page [Build Apps › Pages & Widgets] of type **Embedded**, which renders a plain iframe — nothing special is needed on the OSOS side.

Goal	URL
Whole dashboard, no Grafana chrome	`<grafana>/d/<uid>/<slug>?kiosk`
A single panel	`<grafana>/d-solo/<uid>/<slug>?panelId=<n>`

Whether the embed actually renders depends on two server-side conditions — Grafana permitting framing at all, and the two sites being same-site so Grafana's session cookie is sent. Both are deployment settings; see Grafana Datasource Plugin.

Limits to design around

These are properties of the model, not bugs — plan for them rather than around them.

- **Saved queries only.** No ad-hoc query builder. If a dashboard needs a new cut of the data, model it as a view or a saved query first.
- **No alerting and no scheduled reports.** Grafana alert rules and reports run **without a viewer session**, so there is no identity to forward and no way to honour per-viewer permissions. Anything alert-shaped belongs in OSOS's own notifications [Administration › Notifications & email templates] and SLA/time-monitoring [Administration › Working hours & holidays] machinery.
- **Row cap** `Since 1.8.` The server clamps every request to its own ceiling (10 000 by default), so a panel can never stream a whole collection. Aggregate instead.
- `Fields` **trims columns after the rows arrive**, it does not reduce what is transferred. The **view** decides which fields come back, so trim there if volume matters.
- **Statistic panels take no runtime parameters.** The statistic execute route accepts only the query's own filter field. If a statistic must vary by input, make the variation part of the query design.
- **Attachments are not rendered** — files show as `[file] <name>`.
- **The sharing paths that render without a session — snapshots, public dashboards — must stay disabled.** They would serve one user's data to whoever opens the link. See the setup page.

Related

- Data Views & Faces [Build Apps] — the unit every panel executes against

- Queries [Build Apps › Queries & filters] — saved data and statistic queries
- Dashboards, Summaries & Counter Widgets [Build Apps › Summaries, dashboards & counters] — OSOS's own in-product dashboards
- Reports [Build Apps] — scheduled, document-shaped output
- Webhooks & Integrations — the other outbound integration surfaces
- Grafana Datasource Plugin — installing and operating it

Data Model & Events

To integrate effectively you need a mental model of how OSOS stores data and how its services talk to one another. This guide describes the developer-facing data model and the **message-bus vocabulary** that integrators observe and react to — enough to design against, and detailed enough to serve as a reference when you write triggers/reactions, custom functions, or external integrations (webhooks/Zapier) that respond to activity inside a tenant.

The vocabulary here is the platform's shared contract — the single definition every backend service imports, where events, commands, socket messages and the domain models are all declared. Match on the wire identifiers shown below, never on class names.

The core model

Almost everything in OSOS is a **data entry** with a dynamic `value` payload, described by a **data entity** (its schema). A few consistent ideas run through the whole model:

- **Entities and entries.** A **data entity** defines a schema — its fields, validation, workflows, and triggers. A **data entry** is a single record of that entity, holding its field values plus state and history.
- **References.** Records link to one another through a universal reference pointer of the form `{ dataEntity, dataEntry }`. Wherever one record points at another, this is the shape you will see.
- **Localized text.** Human-facing names and text are **localized strings** carrying both Arabic and English (`{ ar, en }`). Send and expect this shape for names and labels.

The main building blocks

Concept	Role
Data entity	Schema definition: fields, validation, state (Draft / Active / Archived), workflows, triggers
Data entry	A record of an entity, with its `value` payload, state, and history
Data module	Groups entities (drives the sidebar)
Data view	A per-entity presentation (table, kanban, calendar, timeline, and more)
Data query / statistic query	Saved filters and aggregations
Workflow	A process definition (steps, ownership, SLA) attached to an entity
Reaction	An automation rule (event + condition + actions)
Function	User-authored serverless code
Access right	A permission, expressing a level (None / View / Edit / Admin) over a target
Delegation	An "act as" grant with a time window

Identity objects — users, groups, organization units, roles, and assignments — are themselves modeled as data entities, so the same concepts apply when you read or write them.

Field types

A field's type comes from a fixed set: Text, Number, Date, Time, DateTime, Boolean, Blob, **Entity** (a reference to another record), Icon, **Block** (a sub-table), localized JSON, GeoJSON, and JSON. A field's value can be entered manually, defaulted, sourced dynamically, computed from an **expression**, or resolved through a **pointer** — with recalculation

timed to form open, save, or field change. The computed options connect to the expression engine described in the Functions & Integrations guide.

The reference shape

Cross-record references everywhere in OSOS use a two-part pointer, not a bare id:

```
export interface ReferenceProperty {
  dataEntity: Id;   // which entity/collection the target lives in
  dataEntry: UUID;  // which record within it
}
```

A `ReferenceProperty` identifies both *what kind of thing* is referenced and *which instance*. When a field value in an entry is a reference, its value is this shape.

Data entity (the schema)

A `DataEntity` is a record-type definition — the closest analogue to a table/collection schema. Key fields an integrator reads:

Field	Type	Notes
`id`	`UUID | string`	Entity id.
`normalizedName`	`string`	Stable machine name; use this to identify an entity in code rather than the localized `name`.
`name` / `description`	`LocalizedString`	Human labels, per-language.
`type`	`DataEntityTypeEnum`	Entity category (see enum below). `Custom = 0` is a normal builder-defined entity.
`version`	`number`	Schema version, bumped on breaking field changes. Entries record the `entityVersion` they were saved against.
`module`	`UUID`	The module the entity belongs to.
`fields`	`DataEntityField[]`	The field definitions.
`workflows`	`DataEntityWorkflow[]`	Workflow definitions attached to the entity.
`state`	`DataEntityStateEnum`	`Active = 0`, `Archived = 1`.
`isImplicit`	`boolean`	System-owned entities (see enum values `Core`, `System`, `Space`, …).
`virtual`	`VirtualEntityOptions?`	Present for computed/aggregator entities (no physical rows).

A `DataEntityField` declares one field: `normalizedName`, `dataType` (`DataTypeEnum`), `renderOptions`, `validations`, and the optional `virtualField`, `mapping`, `sources`, `sensitiveAccess`, and `valueValidators`. When you react to a field-level change you match on the field's `normalizedName` / `fullNormalizedName`.

Data entry (the record)

A `DataEntry<T>` is a single record of an entity. This is the object most events hand you:

Field	Type	Notes
`id`	`UUID`	Entry id.
`dataEntity`	`UUID`	The entity this entry belongs to.

Field	Type	Notes
`value`	`T extends DataEntryValue`	The field values, keyed by field `normalizedName`. Reference-typed fields hold `ReferenceProperty` values.
`state`	`DataEntryStateEnum`	`Draft = 0`, `Active = 1`, `Archived = 2`.
`stateHistory`	`DataEntryStateHistoryRecord[]`	Audit of state transitions.
`lifeCycleStatus`	`LifeCycleStatusEnum?`	Workflow lifecycle: `Failed = -1`, `Active = 0`, `Completed = 1`; `undefined` = not in a workflow.
`entryType`	`EntryTypeEnum?`	`Normal = 0`, `Virtual = 1`.
`virtual`	`boolean?`	`true` for calculated entries that don't physically exist (aggregator/virtual entity output).
`basedOn`	`ReferenceProperty[]?`	For virtual entries, the source records the calculation derived from.
`entityVersion`	`number`	The entity schema version this entry was saved against.
`store`	`ReferenceProperty`	The store/partition the entry lives in.
`isImplicit`	`boolean`	System-owned entry that cannot be modified or archived.
`isInvalid` / `invalidReason`	`boolean` / `DataEntryInvalidReasonEnum | null`	Set when a schema change invalidated the entry: `Obsolete = 0`, `Invalid = 1`.
`creationInfo` / `updateInfo`	`ActionInfo`	Who/when created/updated.

The `value` object also carries workflow meta under reserved keys: `_metaCurrentSteps`, `_metaWorkflowHistory`, `_metaAdhocSteps`, `_metaOrder`. Treat any `_meta*` key as system-managed.

> Note on statuses: `lifeCycleStatus` (`undefined` / `Active` / `Completed` / `Failed`) is the workflow-lifecycle tracking value that rides on the entry. It is distinct from the human-facing UI statuses shown for request-style entities (for example Draft / Open / Completed / OnHold with a result of Succeeded / Failed / Rejected / Timedout / Cancelled / Obsolete). When you automate against a request, read `lifeCycleStatus` for the tracking state and the entity's own status field for the UI state.

Reactions, triggers, and event configs

A **reaction** (aka trigger) is the builder-facing object that says "when *this* event happens matching *this* criteria, run *these* actions." It is modeled by `DataReaction`:

Field	Type	Notes
`eventType`	`EventTypeEnum`	Which event this reaction listens to.
`eventConfig`	`DataReactionEventConfig`	Event-type-specific matching config (see below).
`executorType`	`{ id: EventExecutorTypeEnum }?`	Whose identity actions run as: `SYSTEM`, `MANAGER`, `EXECUTOR`.
`criteria`	`FilterExpression?`	Value-based filter on the affected entry.
`query`	`ReferenceProperty?`	A saved query that overrides `criteria` when set.
`actions`	`DataReactionAction[]`	The ordered actions to run.
`spaceScope`	`DataReactionSpaceScope`	`{ rootSpace, currentSpace, otherSpaces }` — which spaces the reaction fires in.
`type`	`DataReactionTriggerTypeEnum?`	`NormalTrigger = 0`, `SLATrigger = 1`.

Field	Type	Notes
`sourceEntity`	`DataReactionSourceEntity?`	`Self`, `Relation`, or `AbsoluteEntity` — the entity whose events fire this.
`loopThreshold`	`number?`	Guard against runaway reaction chains.
`disabled`	`boolean?`	Reaction paused.
`relatedEntities`	`DataReactionRelatedEntities[]?`	Related entities whose changes also count.

Event configuration

`DataReactionEventConfig` is a discriminated union keyed by event type. The ones an integrator most often configures:

- `DataEntryCreatedEventConfig` — fires on create.
- `DataEntryUpdatedEventConfig` — `fields[]` (match specific `fullNormalizedName`s; empty = any field) and `relatedFields[]` for changes reached through references.
- `DataEntryArchivedEventConfig` — fires on archive.
- `CustomEventConfig` — a custom action (the builder's **Custom Action** trigger): a required `customActionType` (`UI`, `External` or `Local` — `CustomActionTypeEnum`; it replaced the retired `uiEvent` flag `Since 1.8`), a `contextual` flag (default `true`, no runtime behaviour yet), an optional `askUser` dialog config (inline entity or an external entity and view), and start/success `messages`.
- `WorkflowStepReachedEventConfig` / `WorkflowStepExitedEventConfig` / `WorkflowActionExecutedEventConfig` — scoped by `steps[]` (`workflowId` + `stepId`, where `stepId` may be the literal `'ADHOC_STEP'`) and, for exit/action configs, matched `actions[]`.
- `SLATimeExceededEventConfig` — `occurrencesConfig` (`startingAfter` and `repeatAfter` in seconds, `repetitions` a count) plus step scoping.
- `TimerEventConfig` — `timerOptions` with a `oneTime` timestamp or a `repeat` cron `expression` (+ timezone / start / end).
- `ReactionSucceededEventConfig` / `ReactionFailedEventConfig` — chain off another reaction by `entityId` + `triggerId`.
- `FileGeneratedEventConfig` — matched by `template`.
- `EventAnnouncedEventConfig` — `entityId` + `announcementId` of an announced event.

Every config extends `BaseEventConfig`, which carries `relatedInfo` (`relatedEntity` + `relatedFieldPath`, null for the reaction's own entity) and `triggerContextId`. **Since 1.8**, `triggerContextId` lives on the event config (it owns the context entity), having moved up from individual action options.

Actions

`DataReactionAction` has an `actionType` (`AutomatedActionTypeEnum`) and a typed `options` union (`DataReactionActionOptions`). Actions are what a reaction executes; the `actionType` enum is the catalog of things the platform can do automatically — create/update/archive entities and entries, move workflow steps, send email/SMS/notifications/push, call an external API, run a custom function, announce an event, notify Zapier, and more. See the enum table below.

The message bus

Internally OSOS runs on an internal message bus (`@node-ts/bus`) using a **command/event** pattern:

- **Commands** are imperatives — "do this." Examples include creating, updating, patching, archiving, restoring, or bulk-updating entries; executing a workflow action or changing a workflow step; scheduling a job; or notifying an external integration.
- **Events** are facts — "this happened." Examples include an entry being created, updated, deleted, archived, or restored; a workflow step being reached or exited; an access right changing; an API token being generated; a background job finishing; or a scheduled job firing.

You do not need the bus to use OSOS, but the vocabulary is worth knowing because it shapes the surfaces you *do* touch: **reactions** subscribe to domain events and issue commands as their actions (the no-code automation surface), and **real-time updates** are OSOS re-publishing selected domain events to your client over Socket.IO.

Message taxonomy

Three message families are exported from `@osos/shared/messages`:

Family	Base class	Direction / semantics	Where declared
Events	`Event` (`@node-ts/bus-messages`)	"Something happened" — published after a fact, fan-out to any number of subscribers. This is the primary vocabulary an integrator reacts to.	`messages/events`
Commands	`Command`	"Do this" — an instruction addressed to the service that owns the action, handled by exactly one handler.	`messages/commands`
Socket messages	`BaseSocketMessage`	Envelopes relayed to connected UI clients over the realtime socket (e.g. job/command completion, UI custom events).	`messages/socket-messages`

Every event and command carries a stable string identity and a schema version:

```
export class DataEntryCreatedEvent extends Event {
  static readonly NAME = 'osos/data-entry/data-entry-created';
  readonly $name = DataEntryCreatedEvent.NAME;
  readonly $version = 1;
  constructor(
    readonly dataEntry: DataEntry,
    readonly userId: string,
    readonly normalizedEntity: { normalizedName: string; name: LocalizedString; type: DataEntityTypeEnum; moduleId:
string },
    readonly valueDiff?: UpdateAdditionalData[],
    readonly isSystemModification?: boolean,
  ) { super(); }
}
```

- `$name` is the wire name (namespaced `osos/<domain>/<event>`). Match on this string, never on the class name.
- `$version` is the payload schema version. At the time of writing **every** event and command is `$version = 1`.
- The constructor parameters are the payload. All fields are `readonly`.

> Note: the `$name` string and the class filename are not always identical, and a few names contain historical typos frozen for wire compatibility (`osos/scheduler/scheduled-job-triggerd`, `osos/reaction/event-announced`). Always copy the exact `NAME` constant from the source.

Message attributes (envelope metadata)

Payload aside, each dispatched message carries an attribute envelope. Two shapes are defined in `@osos/shared`.

`BaseMessageAttributes`:

Field	Type	Meaning
`messageId`	`string`	Unique id of this message instance.
`instanceId`	`string?`	Correlates a dispatched job/command with its completion message.

`BaseStickyAttributes` — *sticky* attributes propagate along a causation chain (they ride from a triggering message onto every message it causes):

Field	Type	Meaning
`userId`	`string?`	The acting user, carried across the whole reaction chain.
`spaceId`	`string?`	The space (tenant sub-scope) the activity happened in.

A handler receives both the payload and the attributes:

```
async handle(
  event: AccessRightChangedEvent,
  messageAttributes: MessageAttributes<BaseMessageAttributes>,
): Promise<void> { /* ... */ }
```

> Many events *also* carry a `userId` in their own payload (e.g. `DataEntryCreatedEvent.userId`). The code marks this as gradually being superseded by `stickyAttributes.userId`. For now, prefer the payload `userId` where present and fall back to the sticky attribute.

Core enumerations

These integer/string enums appear throughout event payloads and models. Numeric values are the wire values — do not assume ordering, copy them exactly.

`EventTypeEnum` — reaction trigger types

Name	Value
`DataEntryCreated`	0
`DataEntryUpdated`	1
`DataEntryArchived`	2
`WorkflowActionExecuted`	3
`WorkflowStepReached`	4
`WorkflowStepExited`	5
`Timer`	6
`CustomEvent`	7
`ReactionSucceeded`	8
`FileGenerated`	9
`ReactionFailed`	10
`EventAnnounced`	11
`DataEntryRestored`	12
`ZapierEvent`	20
`SlaTimeExceeded`	100

`AutomatedActionTypeEnum` — reaction action types

Name	Value	Name	Value
`CreateDataEntity`	0	`MoveWorkflowToStep`	20
`UpdateDataEntity`	1	`ExecuteWorkflowAction`	21
`ArchiveDataEntity`	2	`ChangeCurrentStepOwner`	22
`CreateDataEntry`	10	`UserActivationToggle`	30
`UpdateDataEntry`	11	`SetPermission` `Since 1.10`	31
`ArchiveDataEntry`	12	`RemovePermission` `Since 1.10`	32
`CreateBlockItem`	13	`CallApi`	90
`UpdateBlockItem`	14	`SendNotification`	100
`DeleteBlockItem`	15	`SendEmail`	101
`DeleteArchivedEntry`	16	`SendSms`	102
		`GenerateFileFromTemplate`	103
		`ExecuteCustomFunction`	104
		`AnnounceEvent`	105
		`ExecuteCustomAction` `Since 1.8`	106
		`SendWhatsApp` `Since 1.10`	107
		`NotifyZapier`	200

`CreateDataEntity` (0) and `UpdateDataEntity` (1) stay declared, but `Since 1.10` a reaction carrying either is refused on save and a stored one is skipped at run time — see Action types [Build Apps › Reactions & automation].

`SendWhatsApp` (107) `Since 1.10` carries `SendWhatsAppActionOptions`: `recipients` (resolved like the Send SMS recipients), `templateName` (an approved Meta template), `language` (the template language code, e.g. `en` or `ar`) and `parameters` (`TokenSource[]`; index *i* fills `{{i+1}}`, may be empty). How the action behaves when it runs is on Reactions & Automation [Build Apps › Reactions & automation]; the list of approved templates comes from the WhatsApp template API.

Other frequently-seen enums

Enum	Values
`DataEntryStateEnum`	`Draft = 0`, `Active = 1`, `Archived = 2`
`DataEntityStateEnum`	`Active = 0`, `Archived = 1`
`EntryTypeEnum`	`Normal = 0`, `Virtual = 1`
`LifeCycleStatusEnum`	`Failed = -1`, `Active = 0`, `Completed = 1` (plus `undefined` = not in a workflow)
`DataEntryInvalidReasonEnum`	`Obsolete = 0`, `Invalid = 1`
`EventExecutorTypeEnum`	`SYSTEM`, `MANAGER`, `EXECUTOR`
`DataReactionTriggerTypeEnum`	`NormalTrigger = 0`, `SLATrigger = 1`
`TriggerSourceEntityTypeEnum`	`Self = 0`, `Relation = 1`, `AbsoluteEntity = 2`
`FunctionTypeEnum`	`REACTION_CUSTOM_ACTION`, `INTERCEPTOR`, `STANDALONE_VALIDATOR`, `VALUE_VALIDATOR`
`FunctionStatusEnum`	`BUILD_FAILED`, `PROVISIONING`, `PROVISIONED`, `DELETED`

`DataEntityTypeEnum` is large; the values integrators most often encounter are `Custom = 0`, `Service = 2`, `Request = 3`, `Notification = 7`, `Task = 8`, `User = 9`, `Group = 10`, and the system band (`Core = 30`, `System = 40`, `VirtualProxy = 50`, `VirtualAggregator = 52`, `Space = 60`).

Event catalog

The full published event vocabulary, grouped by domain. The payload column lists the constructor parameters (the wire payload). `?` marks optional fields.

Data entities (schema lifecycle)

Event (`$name`)	Payload
`osos/data-entity/data-entity-created`	`dataEntity: DataEntity`, `userId`
`osos/data-entity/data-entity-updated`	`dataEntity`, `userId`, `requiresReIndex?`, `workflowStatus?`, `entityChanges?[]`, `isSystemModification?`, `rawDiff?`
`osos/data-entity/data-entity-archived`	`dataEntity`, `userId`
`osos/data-entity/data-entity-version-changed`	`dataEntity`, `oldVersion`, `newVersion`, `userId`
`osos/data-entity/data-entity-deleted` `Since 1.10`	`dataEntity` (read before the purge), `owned: DeletedEntityOwnedIds`, `moduleNormalizedName`, `userId` — see below
`osos/data-entity-field-meta/data-entity-updated`	`dataEntity`, `changedMeta` (`mainTitle` / `mainPicture`), `userId`

`DataEntityUpdatedEvent.entityChanges[]` reports structural field changes via the `EntityChanges` enum: `FIELD_DELETED`, `FIELD_ADDED`, `FIELD_REQUIRED_VALIDATION_CHANGED`, `FIELD_RENDER_OPTIONS_CHANGED`, `FIELD_INTERNAL_COLLECTION_CHANGED`, `VIRTUAL_ENTITY_OPTIONS_CHANGED`, `TREE_STRUCTURE_DELETED`. `rawDiff` is the complete diff from the `just-diff` library.

`DataEntityDeletedEvent` — permanent entity delete `Since 1.10`

Published at the end of an administrator's **permanent entity purge** (`DELETE /v1/entities/{id}/permanent`), just before the entity document itself is deleted. `owned` lists the ids of what was removed with the entity, by kind: views, queries, collections, statistic queries, relations, aggregates, quick-filter groups, tag groups, reports, system buttons, counter widgets, dashboards, pages and functions. Reactions and announcements are not listed — their services find them by the entity id.

- **Consumers must be idempotent.** A retried purge can publish the event again.
- **No archive event is published** by the purge, so the entity's archive reactions never fire for it.
- The audit service records it as `DATA_ENTITY_DELETED`.

Data entries (record lifecycle)

Event (`$name`)	Payload
`osos/data-entry/data-entry-created`	`dataEntry: DataEntry`, `userId`, `normalizedEntity`, `valueDiff?`, `isSystemModification?`
`osos/data-entry/data-entry-updated`	`dataEntry`, `userId`, `normalizedEntity`, `metaDataChanges`, `valueChanges?[]`, `valueDiff?`, `isSystemModification?`, `refreshedEntities?`
`osos/data-entry/data-entry-deleted`	`dataEntry`

Event ($name)	Payload
osos/data-entry/data-entry-archived	dataEntry , userId
osos/data-entry/data-entry-restored	dataEntry , userId
osos/data-entry/data-entry-invalidated	dataEntry , userId , normalizedEntity
osos/data-entry/data-entry-permanently-deleted	dataEntry , userId , stateBefore , entryNameBefore
osos/data-entry/data-entries-archived	`dataEntries: Pick<DataEntry,'id'
osos/data-entry/data-entries-permanently-deleted	items: PermanentlyDeletedEntrySnapshot[] , userId
osos/data-entry/draft-data-entry-upserted	dataEntry , userId , rawDiff
osos/data-entry/draft-data-entry-deleted	dataEntityId , draftDataEntryId , userId

DataEntryUpdatedEvent.valueChanges[] (IValueChangesEntryUpdated) is the field-level change set: each item has fullNormalizedName , optional fieldId , an optional mapConfig (mapped-field source/reference/ updateMode), and isTreeStructureField? . This is what a DataEntryUpdated reaction's fields[] config is matched against. refreshedEntities is an internal de-duplication list used by the aggregator refresh handler to avoid re-refreshing an entity already refreshed inline. **Since 1.6**, DataEntryInvalidatedEvent announces entries invalidated by a schema change. DataEntryRestoredEvent and the permanently-deleted family are **Since 1.8**.

A write whose entry is deleted before its re-read publishes nothing Since dev . After a create, update, patch or upgrade, data-service re-reads the entry to put the resolved version on DataEntryCreatedEvent / DataEntryUpdatedEvent . When a concurrent delete won that race, the event used to go out with dataEntry undefined, and every consumer failed on it and was retried. Now the write is reported as a non-retryable failure and **no** created/updated event is published; the deletion has its own DataEntryDeletedEvent . The payload contract is unchanged (dataEntry is always present on an event that is published), but a write can now end without any event, so do not wait for an updated event on an entry that may have been deleted. audit-service also skips, with a warning, an updated event that arrives without dataEntry (from an older publisher or a replayed message).

Collections, relations, views, queries, and modules

Event ($name)	Payload
osos/data-collections/data-collection-created / -updated / -archived	dataCollection: DataCollections , userId (-updated adds rawDiff?)
osos/data-relation/data-relation-created / -updated / -deleted	dataRelation: IDataRelation , userId (-updated adds rawDiff?)
osos/data-view/data-view-created / -updated / -deleted	dataView: IDataViews , userId , plus permissions? , changedStatus? (published/pinned), isSystemModification? , rawDiff? on -updated
osos/data-query/data-query-created / -updated / -deleted	dataQuery: DataQuery (-updated adds criteriaChanged: boolean , rawDiff?)
osos/data-module/data-module-created / -updated / -deleted	dataModule: DataModule , userId (-updated adds changes?: Record<string, {oldValue,newValue}>)

Event (`$name`)	Payload
`osos/generic-view/generic-view-created` / `-updated` / `-deleted`	`genericView: IGenericView`, `userId`, `permissions?`, `changedStatus?`, `defaultPageForSpaces?`, `rawDiff?`
`osos/statistic-query/statistic-query-created` / `-updated` / `-deleted`	`statisticQuery: IStatisticQuery`
`osos/report/report-deleted`	`reportId`, `userId`

> Reports are a **web-only** surface; they are not available in the mobile app, so any report deep-link surfaced to a mobile client is treated as unsupported.

Reactions, triggers, and custom events

Event (`$name`)	Payload
`osos/reaction/reaction-created` / `-deleted`	`dataReaction: DataReaction`, `userId`
`osos/reaction/reaction-updated`	`dataReaction`, `previousDataReaction`, `userId`, `rawDiff?`
`osos/reaction/announcement-created` / `-deleted`	`announcement: Announcement`, `userId`
`osos/reaction/announcement-updated`	`announcement`, `previousAnnouncement`, `userId`
`osos/reaction/event-announced`	`dataEntityId`, `announcementId`, `dataEntry`, `userId`, `announcementEntry?`
`osos/triggers/custom-event-triggered`	`dataEntityId`, `dataEntries: ReferenceProperty[]`, `userId`, `triggerId`, `timestamp`, `askUserValue?`, `askUserDataEntry?`
`osos/triggers/trigger-executed`	`dataEntityId`, `triggerId`, `triggerHistoryId`
`osos/triggers/email-generated`	`subject`, `body`, `to`, `cc`, `bcc`, `link?`, `attachments?`, `relatedTriggerExecutionId?`, `eventType?`, `dataEntry?`, `from?`
`osos/triggers/sms-generated`	`body`, `recipients`, `link?`, `relatedTriggerExecutionId?`
`osos/triggers/whatsapp-generated` `Since 1.10`	`recipients: string[]`, `template: {name, language, category: 'authentication' | 'other'}`, `parameters: string[]` (for `authentication`: `[code]`), `relatedTriggerExecutionId?`

Note the class name `EmailGeneratedEvent` maps to `$name` `osos/triggers/email-generated`, and `SMSGeneratedEvent` to `osos/triggers/sms-generated`.

WhatsApp messages `Since 1.10`. `WhatsAppGeneratedEvent` (`osos/triggers/whatsapp-generated`) asks notification-service to send one approved WhatsApp template. Two services publish it: reaction-service, for the Send WhatsApp action, and users-service, for WhatsApp one-time codes during multi-factor sign-in. notification-service answers with one `osos/events/whatsapp-send-status` per recipient, which logging-service records. That publish is best effort — a failed publish is logged and never retried, because a redelivery would message every recipient again. Meta's later delivery callbacks reach logging-service through its public webhook (WhatsApp API); they update the same log row by provider message id and are **not** republished on the bus.

Reaction-history storage `Since 1.10`. The `TriggerHistory` row that `trigger-executed` points at (one per reaction run) is now indexed for the Reaction Logs filters, its `result` is optional with default `[]` (a run that recorded no action), and the entry-history row an entry-created / entry-updated message writes is unique per source message, so a redelivered message writes it once. Details: Reaction history indexes and clean-up [Build Apps › Reactions & automation].

Workflow

Event (`$name`)	Payload
`osos/workflow/step-reached`	`dataEntry`, `workflowId`, `stepId`, `userId`, `isAdhoc?`
`osos/workflow/step-exited`	`dataEntry`, `workflowId`, `stepId`, `userId`, `action?`, `isAdhoc?`
`osos/workflow/action-executed`	`dataEntry`, `workflowId`, `stepId`, `action`, `userId`, `isAdhoc?`
`osos/workflow/ended`	`dataEntry`, `workflowId`, `userId`
`osos/workflow/workflow-step-owner-changed`	`dataEntry`, `userId`, `changes: ReassignOwnerChanges`, `doneAt`, `additionalData?`
`osos/workflow/adhoc-step-created`	`dataEntry`, `adhocStep: WorkflowStep`, `userId`
`osos/sla/time-exceeded`	`dataEntry`, `workflowId`, `stepId`, `userId`, `isAdhoc`, `receivedDate`, `dueDate`, `workingHours`, `holidays?`

Integrations, functions, files, jobs, and messaging

Event (`$name`)	Payload
`osos/events/zapier-event-triggered`	`entityId`, `zapId`, `eventType`, `eventOutput`
`osos/functions/function-provisioning-status-update`	`functionModel: FunctionModel`, `status: FunctionStatusEnum`, `details: { url }`
`osos/files/file-generated`	`file: IFile`, `fileName`, `template`, `exportType`, `dataEntityId`, `dataEntryIds`
`osos/app-tokens/app-token-generated`	`token`, `tokenName`, `userId`, `email`
`osos/app-tokens/function-app-token-rotated` `Since dev`	`functionId`, `userId` (the function's own user), `token` (the new raw token: never log or store it). Published instead of the e-mailed token events when a function's own user has its token regenerated; functions-service stores it as the function's `OSOS_API_TOKEN` secret, and audit-service records `FUNCTION_APP_TOKEN_ROTATED` without the token. See the per-function identity.
`osos/events/job-executed`	`messageId`, `instanceId`, `userId`, `messages{startMessage,successMessage}`, `result{success,error?,result?}`
`osos/events/email-sent`	*(no payload)*
`osos/events/email-send-status`	delivery report: `from`, `to`, `cc`, `bcc`, `subject`, `body`, `status:'success'`
`osos/events/sms-send-status`	`to`, `body`, `status`, `strategy`, `relatedTriggerExecutionId`, `errorMessage?`, `smsProviderResponse?`, …
`osos/events/push-notification-sent`	`notificationId: UUID`
`osos/events/whatsapp-send-status` `Since 1.10`	`payload`: `{to, status, providerMessageId?, messageId?, spaceId, templateName, templateLanguage, category, parameters?, errorCode?, errorMessage?, relatedTriggerExecutionId?, timestamp}`; `status` is `accepted`, `sent`, `delivered`, `read`, `failed` or `mocked` (a send skipped in test mode). `parameters` is left out for `authentication` templates, because it holds the code.
`osos/icons/icons-updated`	`timestamp: number`
`osos/packaging/package-exported`	`packageName`, `resourceCount`, `resourceTypes[]`, `description?`

Event (`$name`)	Payload
`osos/packaging/package-imported`	`sessionId` , `created` , `overwritten` , `skipped` , `failed` , `packageFileName?`
`osos/packaging/package-changed` `Since 1.10`	`packageId` — published by packaging-service when a package's review or store state moves. A UI signal only; see the socket message.
`osos/dependency/rebuild-dependency-requested`	`reason` , `resourceTypes?`

The `FileGeneratedEvent.exportType` identifies the produced document format; data-export files are generated as **XLSX** or **PDF**.

Access, spaces, and users

Event (`$name`)	Payload
`osos/access-rights/access-rights-changed`	`entitlementType: EntitlementEnum` , `entitlementId: string
`osos/spaces/space-created-event`	`space: SpaceDataEntry` , `userId` , `valueDiff?` , `isSystemModification?`
`osos/spaces/space-updated-event`	`space` , `userId` , `rawDiff?` , `isSystemModification?`
`osos/spaces/space-activated-event` / `space-deactivated-event`	`space` , `userId`
`osos/spaces/end-user-activated` / `end-user-deactivated`	`space` , `userId` , `doneAt`
`osos/user/user-signed-up`	`userId` , `value` , `defaultGroup` , `defaultSpace` , …
`osos/user/user-signin-attempted`	`userId` , `timestamp` , `outcome:'success'`
`osos/user/user-account-activated` / `-deactivated` / `-locked`	`userId`
`osos/user/user-external-id-linked`	`userId` , `externalId` , `syncPatches?`
`osos/user/user-password-changed` / `-reset` / `-reset-by-system` / `-reset-completed`	`userId` (+ email/token/etc. per variant)

Command vocabulary

Commands are one-to-one instructions handled by the owning service. An integrator issues these (directly or via reaction actions) rather than merely observing them. Full name list from `messages/commands` :

Data entries — `create-data-entry` , `update-data-entry` , `patch-data-entry` , `upsert-data-entry` , `upsert-draft-data-entry` , `delete-data-entry` , `archive-data-entry` / `archive-data-entries` , `restore-data-entry` / `restore-data-entries` , `permanently-delete-data-entry` / `permanently-delete-data-entries` , `upgrade-data-entry` , and the bulk variants (`bulk-patch-data-entries` , `bulk-update-data-entries` , `bulk-update-entries` , `bulk-upgrade-data-entries`).

Data entities — `create-data-entity` , `update-data-entity` , `create-virtual-data-entity` , `update-virtual-data-entity` , `archive-data-entity` , `recalculate-references` , `migrate-history-for-entity-entries` .

Workflow — `change-step` , `change-step-owner` , `execute-action` , `insert-adhoc-step` .

Authorization projection `Since 1.10` — `osos/authz/reconcile-authz-tenant` (`ReconcileAuthzTenantCommand`): `mode` (`repair` writes what differs, `report` only counts it) and an optional `trigger` (`nightly` or `on-demand`). It is sent in the Root Space as the system user and reconciles the sending tenant's OpenFGA projection; the on-demand sender is an internal, service-only route. It is internal plumbing, listed so you recognise it in bus traces.

Reactions / integrations / messaging / scheduling — `osos/reaction/create-reaction`, `osos/integrations/notify-zapier`, `osos/notifications/send-push-notification`, `osos/sms/send-sms`, `osos/data/generate-file-from-template`, `osos/shared/export-template-in-background`, `osos/scheduler/schedule-job`, `osos/scheduler/schedule-interval-job`, `osos/scheduler/remove-scheduled-job`.

Example command shapes:

osos/data-entry/create-data-entry

```
new CreateDataEntryCommand(
  store: string,
  dataEntry: Pick<DataEntry, 'id' | 'value' | 'specificationVersion' | 'dataEntity'>,
  userId: UUID,
  isAdminModification: boolean,
  skipValidation?: boolean,
  viewId?: UUID,
  systemButtonId?: UUID,
  isSystemModification?: boolean,
);

// osos/notifications/send-push-notification
new SendPushNotificationCommand(
  tokens: string[], title: string, content: string, data: any, notificationId?: UUID,
);
```

Note that commands intentionally carry `Pick<...>` slices of `DataEntry` / `DataEntity`, not the full model — only the fields the handler needs travel over the wire.

Socket (real-time) messages

Socket messages wrap results relayed back to connected UI clients (`messages/socket-messages`). They implement `BaseSocketMessage` (`{ $name, $version }`) and, unlike bus events, embed the full correlation envelope in the message body:

Message (`$name`)	Purpose
`osos/infrastructure/command-executed`	A dispatched command finished.
`osos/infrastructure/job-executed`	A background job finished (mirrors `JobExecutedEvent`; carries `messageId`, `instanceId`, `userId`, `messages`, `result`).
`osos/infrastructure/job-progress`	Progress update for a running job.
`osos/infrastructure/job-scheduled` / `job-started`	Job lifecycle notifications.
`osos/infrastructure/register-service`	Service registration handshake.
`osos/triggers/custom-event-triggered`	UI custom event relayed to clients.
`packages/changed` Since `1.10`	Server → client, broadcast to every socket of the event's tenant when a package's review or store state moves. Payload `{ $name, $version: 1, packageId }`. An open package page reloads.

`packages/changed` differs from the other rows in two ways. Its message class (`PackageChangedMessage`) lives in `web-socket-service`, not in `@osos/shared`. And it is derived from a bus event rather than relayed from a command result: packaging-service publishes `PackageChangedEvent` (`osos/packaging/package-changed`) and web-socket-service

broadcasts it. It carries no store or review detail — treat it as "re-read the package", never as state to build on. Delivery details: Real-Time.

`CustomEventTriggeredMessage` wraps the `CustomEventTriggeredEvent` plus `messageId`, `instanceId`, `traceId`, `userId`, `spaceId`, and optional `delegateeId` / `delegationId` — useful when a UI custom event runs under delegation.

Delivery while a tenant is under maintenance `Since 1.10`

In a multi-tenant deployment a tenant can be put under **maintenance**: `restoring` while its data is restored from a backup, or `moving` while an operator holds it for a move between cells. Bus consumers that run the shared maintenance gate — the LoopBack services' message bus, `indexing-service`, `notification-service`, `users-service` and `web-socket-service` — neither process that tenant's messages meanwhile nor return them to the queue (which would dead-letter the tenant's traffic after five attempts). They **park** them: the original is acknowledged and a copy goes to `<queue>-parked`, a queue with a fixed TTL that dead-letters back to the main queue, so the message is checked again on its next delivery.

What a subscriber can observe:

- **Delay, not loss, for most work.** Once the maintenance clears, parked messages replay. This applies to a `moving` hold and to a restore that did not rewind the data.
- **Deliberate loss after a restore that rewound the data.** A message first parked under a `restoring` window is **dropped** once the restore has rewound the tenant's data (its `lastRestoredAt` is at or after the window's start): it describes data as it was after the restore point, and replaying it would, for example, index an entry the restore removed or run a reaction twice. Drops are counted (`outcome="dropped"`) and logged with the restore id. An integration that relies on an event for an outside effect — a webhook, a Zapier call — must treat events emitted inside the restore window as possibly never delivered.
- **A time cap.** A message waits at most 2 hours by default, or the maintenance's own cap (a move carries the cell's `moving` cap); beyond it the message goes to the real dead-letter queue with the reason `restore-timeout`.
- **Cell-level messages** that name no tenant are never parked, and single-tenant installations have no maintenance state.

The HTTP side of the same state is the `503 TENANT_IN_MAINTENANCE` refusal with `Retry-After` — see Calling the HTTP API.

How integrators consume events

There are several consumption surfaces, all keyed off the same event vocabulary:

1. **Reactions/triggers (no code)** — a builder configures a `DataReaction` bound to an `EventTypeEnum` with an `eventConfig` matcher, `criteria` / `query` filter, and a list of `DataReactionAction`s. The platform matches published events against these and runs the actions. This is the primary, config-driven path.

2. **Custom functions** — a reaction action of type `ExecuteCustomFunction` (`AutomatedActionTypeEnum = 104`) runs a `FunctionModel` of `FunctionTypeEnum.ReactionCustomAction`. Functions can also be `Interceptor`, `StandaloneValidator`, or `ValueValidator` types, hooking the save/edit/archive lifecycle (`StandaloneFunctionValidatorTriggerTypeEnum`: `BEFORE_SAVE`, `BEFORE_EDIT`, `BEFORE_ARCHIVE`; `ValueFunctionValidatorTriggerTypeEnum`: `ON_SAVE`). A function's build/deploy status is reported via `FunctionProvisioningStatusUpdateEvent`.

3. **In-service handlers (engine code)** — backend services subscribe by implementing `@node-ts/bus-core`'s `Handler<TEvent>`: set `messageType = SomeEvent` and implement `handle(event, messageAttributes)`. Example:

```
export class AccessRightChangedHandler implements Handler<AccessRightChangedEvent> {
  handlerName = AccessRightChangedHandler.name;
  messageType = AccessRightChangedEvent;
  async handle(event: AccessRightChangedEvent, attrs: MessageAttributes<BaseMessageAttributes>) { /* ... */ }
}
```

4. **External integrations** — the `ZapierEventTriggered` event (`osos/events/zapier-event-triggered`) and the `NotifyZapierCommand` bridge tenant activity to external Zapier flows; `AppTokenGeneratedEvent` backs API-token issuance for programmatic access.

For most integrations, prefer **events** for reactive workflows and the **HTTP API** for request/response work. If you only need to know that something changed, the real-time channel is lighter than any deeper integration.

A note on the data-service observers

The files under `backend/services/data-service/src/observers` are **not** bus-event handlers — they are LoopBack `LifeCycleObserver`s (startup/shutdown booters), grouped by `ObserverGroupsEnum.Preparation`:

Observer	Role
`ApplicationObserver`	Stops the bus cleanly on app shutdown.
`DomainRedisBooter`	Warms the domains cache in Redis at startup (`DomainService.refreshDomainsInRedis`).
`SpaceRedisBooter`	Warms the spaces cache in Redis (`SpaceService.refreshSpaceInRedis`).
`IconManifestRedisBooter`	Regenerates the icon manifest cache.
`GeojsonValidatorObserver`	Registers a custom GeoJSON `Position` validator (lon −180..180, lat −90..90).
`LanguagesBooter`	Loads active language entries into the app (falls back to English/Arabic defaults if none exist).

Actual event *handling* for the data-service lives under `data-service/src/handlers/events/**`, and events are published from controllers/services. If you are looking for "who reacts to event X," search the `handlers/events` trees, not `observers`.

Versioning and compatibility

- Match events by their `$name` string constant and check `$version` (currently all `1`). If a `$version` ever bumps, treat the older payload as a distinct contract.
- Field additions to event payloads are made as optional (`?`) trailing constructor parameters — safe to ignore if you don't use them (e.g. `rawDiff`, `isSystemModification`, `refreshedEntities`).
- Use `normalizedName` (entity, field, reaction) as your stable identifier in code; localized `name` values change and are per-language.
- `rawDiff` on `*-updated` events is the raw `just-diff` operation list — prefer the higher-level `valueChanges` / `entityChanges` / `changes` summaries where provided; use `rawDiff` only when you need the exact operation set.

Feature availability

- `DataEntryInvalidatedEvent` — **Since 1.6**.

- `rawDiff` on update events — **Since 1.6**.
- `triggerContextId` on the event config (moved up from action options) — **Since 1.8**.
- `refreshedEntities` on `DataEntryUpdatedEvent` — **Since 1.8**.
- `DataEntryRestoredEvent`, `EventAnnouncedEvent`, `DataEntriesPermanentlyDeletedEvent` — **Since 1.8**.
- `DataEntityDeletedEvent` (permanent entity delete) — **Since 1.10**.
- `SetPermission` (31) / `RemovePermission` (32) action types — **Since 1.10**.
- `WhatsAppGeneratedEvent`, `WhatsAppSendStatusEvent`, `PackageChangedEvent` (and the `packages/changed` socket message), `ReconcileAuthzTenantCommand`, the `SendWhatsApp` action type (107) and the maintenance parking and drop behaviour — **Since 1.10**.
- `FunctionAppTokenRotatedEvent`, and no created/updated event for an entry deleted before its re-read — **Since dev**.
- `ZapierEventTriggered`, `FunctionProvisioningStatusUpdateEvent`, `DraftDataEntryUpsertedEvent` — **Long-standing** (available before 1.2); no availability badge.

Run the platform from source (local development)

This page is for **platform contributors**, and for **integrators** who need a complete local sandbox rather than a hosted tenant. It collects the machine-level techniques for running OSOS **from source** on your own workstation: generating the per-developer configuration, giving the instance a real hostname, launching the whole backend fleet at once, recovering the first login, and diagnosing a bring-up that will not come up.

It is deliberately separate from the rest of this section, and self-contained. The other pages describe building **on** a running platform — through its API, events and functions; everything here describes standing the platform **up** on a laptop. The operator-level reference for the same Docker Compose setup — what runs where, seeds and migrations — is Running locally. A few steps still edit files you must **not commit**; those are marked **local-dev only**.

> ⚠ **Local-dev only**
>
> Where this page tells you to edit a tracked file, the edit exists to make a split, single-machine setup behave like a real deployment. It is not a fix, it is not upstream, and it must never reach a shared branch. Keep it in your working tree, or in a local-only commit you rebase away.
>
> Most of what used to need that treatment no longer does: the local endpoints, the loopback host mapping and the loopback sign-in callbacks are all **configuration** now, and they live in files git ignores.

Generate the local configuration files `Since 1.10`

Two files a workstation run needs are deliberately not in the repository, and a script creates them for you. `pnpm install` runs it through `postinstall` (skipped when `CI` is set), and you can re-run it at any time:

```
pnpm run setup:local
```

It **never overwrites**, so re-running it is safe and will not undo your edits. It creates:

File it creates	Copied from	What it is for
`config/app-config.local.json`	`config/app-config.json`	The endpoints the portal calls — your copy, free to edit
`services/data-service/.env.migration`	`services/data-service/.env.migration.example`	The super-admin identity the migration seeds

Why `app-config.json` is still tracked, and how to override it `Since 1.10`

The obvious move — untrack `config/app-config.json` and let every developer keep their own — is wrong here, and the reason is worth knowing because the failure is silent. Compose **creates a directory** at a missing bind-mount source. A clone that skipped the copy step would get a *directory* where the portal expects its config file, and the portal would be served a broken configuration with no error anywhere. Setting `bind.create_host_path: false` does not prevent it.

So the tracked file stays, and the mount is indirected instead:

```
- ${OSOS_APP_CONFIG:-./config/app-config.json}:/app/app-config.json
```

Unset — which is every existing setup, and all of CI — that resolves to exactly the path it always used. To point the portal at your own endpoints, set the variable in `backend/.env` and edit the local copy rather than the tracked one:

```
OSOS_APP_CONFIG=./config/app-config.local.json
```

Everywhere below that tells you to set `API_BASE_URL`, `AUTHORITY` and friends, that is the file to edit once this variable is set.

> **A 1.10 portal reads less of this file** `Since 1.10.` The current portal copies `app-config.json` through an allowlist and ignores `API_BASE_URL`, `SOCKET_BASE_URL`, `APP_URL`, `AUTHORITY` and `CLIENT_ID`: it derives the API and socket bases from the page's own origin and takes its OIDC settings from the host's `tenant-config` — see Portal sign-in comes from the host's tenant-config. The settings below matter only for a pre-1.10 portal build.

Use a local domain instead of `localhost` or an IP

The single highest-value change you can make to a local environment is to stop browsing `localhost` or a raw IP and give the instance a **real hostname** — `osos.lan`, say — with its subdomains pointed at your machine. It removes every loopback workaround further down this page, and it is the only way to exercise the parts of the platform that key off the request host.

Why the hostname is load-bearing

OSOS is multi-tenant **by hostname**. On every request the gateway resolves the `Host` header into a Space and a Domain, and rejects anything it cannot match. Space hosts are derived from the Space's domain record:

- `domain_identifier === 'root'` → the Space is served on the bare URL.
- Any other identifier → the Space is served on `<domain_identifier>.<url>`.

So a second Space with identifier `hr` on domain `osos.lan` is reachable only at `hr.osos.lan`. With `localhost` you cannot meaningfully exercise that: the default host alias rewrites the literal string `localhost:4200` to the gateway's port, because the portal (`:4200`) and gateway (`:8000`) are split across two ports locally — something no real deployment does. A real domain has no alias entry, so it resolves to itself and the generic host-building path runs, exactly as it does in production.

The payoff:

- **Spaces and Domains become testable.** Create a Space, give it an identifier, browse its subdomain, and watch `space-resolver` and `domain-resolver` do the real thing.
- **Per-host sign-in behavior becomes testable** — `supportLocalSignIn`, per-Domain branding, and identity-provider selection are all resolved from the host.
- **No loopback patching.** The `127.0.0.1` workarounds in the next section stop being necessary, because the browser, the portal, and the gateway all agree on one host.
- **The topology matches production**: one hostname, path-routed to the portal, the gateway, and identity.

Point a wildcard domain at your machine

Use a name in a reserved suffix — `.lan`, `.internal`, or `.test` — never a real TLD you do not own, and never `.local` (mDNS). You need `osos.lan` **and** `*.osos.lan` resolving to the workstation's LAN address.

Approach	Wildcards	Good for
`dnsmasq` on the workstation or a home server	Yes — `address=/osos.lan/192.168.1.50` covers the domain and every subdomain	The standard choice on Linux/macOS; also what most router firmware runs underneath

Approach	Wildcards	Good for
Pi-hole (or any LAN DNS appliance)	Yes — one wildcard record	A LAN-wide setup where phones and other machines must reach the instance too
Router DNS entry	Depends on firmware	Zero extra software when the router supports wildcard host overrides
Acrylic DNS Proxy (Windows)	Yes — `192.168.1.50 osos.lan` `*.osos.lan` in `AcrylicHosts.txt`	A Windows workstation with no LAN DNS available
`hosts` **file**	**No**	Only viable if you enumerate every Space subdomain by hand; fine for one or two Spaces, useless for testing Space creation

The `hosts`-file limitation is the reason to prefer a real resolver: the whole point of the exercise is that a newly created Space resolves without you touching anything.

Point the domain at the machine's **LAN IP**, not `127.0.0.1`, if you want to open the instance from a phone or a second machine — which is also how you test the mobile app against a local backend.

Put one front door in front of the portal and the gateway

A production instance serves everything from one host and path-routes behind it. Locally the portal and the gateway are separate published ports, so put a small reverse proxy on port 80 and mirror the production path map.

The production ingress routes, in order of specificity:

Path	Backend
`/socket`	public websocket service
`/api`	API gateway
`/sh/`	shortener
`/identity`, `/users-assets`, `/.well-known`, `/jwks`, `/me`, `/session`, `/ext-auth`, `/token`, `/interaction`, `/auth`	identity
`/auth-callback`	portal
`/`	portal

A Caddy container reproduces that locally. Add it to `backend/docker-compose.override.yml` (the same local-only override file that carries the image remaps):

```
# backend/docker-compose.override.yml — local-dev only
services:
  front-door:
    image: caddy:2-alpine
    ports:
      - "80:80"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile:ro
    extra_hosts:
      - "host.docker.internal:host-gateway"
```

```
# backend/Caddyfile — local-dev only
http://osos.lan, http://*.osos.lan {
  handle /api/* {
    reverse_proxy osos-api-gateway:8000
  }
  handle /socket/* {
    reverse_proxy host.docker.internal:3017
  }
  handle /sh/* {
    reverse_proxy host.docker.internal:3010
  }
  @identity path /identity/* /users-assets/* /.well-known/* /jwks /me /session /ext-auth/* /token/* /interaction/* /
auth/*
  handle @identity {
    reverse_proxy host.docker.internal:3001
  }
  handle {
    reverse_proxy osos-web:5000
  }
}
```

The gateway and portal are addressed by their Compose service names because Caddy shares their network; identity and the websocket service run natively on the host, so they are reached through `host.docker.internal`. `/auth-callback` is not matched by the `/auth/*` matcher and correctly falls through to the portal, matching the production ingress.

Configure the platform for the domain

Four places carry the host, and all four must agree.

1. **Portal runtime config** — the file mounted into the portal container. Edit `backend/config/app-config.local.json` and point `OSOS_APP_CONFIG` at it (see above) rather than editing the tracked `backend/config/app-config.json`:

   ```
   API_BASE_URL     = http://osos.lan/api
   SOCKET_BASE_URL  = http://osos.lan/
   APP_URL          = http://osos.lan
   AUTHORITY        = http://osos.lan
   ```

2. **Identity** — `backend/services/users-service/config.env`. `AUTHORITY_URL` is the OIDC issuer and must match `AUTHORITY` above exactly; `RP_ID` is the WebAuthn relying-party id and must be the registrable domain:

   ```
   AUTHORITY_URL="http://osos.lan"
   CLIENT_ORIGIN="http://osos.lan"
   RP_ID="osos.lan"
   BEHIND_PROXY=true
   ```

3. **OIDC client** — add the domain's callbacks to the `localhost-angular` client in `backend/services/users-service/src/clients.json` (and the built copy under `dist/`, or rebuild the service): `http://osos.lan/auth-callback` and `http://osos.lan/silent-renew.html` in `redirect_uris`, `http://osos.lan` in `post_logout_redirect_uris`.

4. **The seed** — run the initial-setup migration with the domain, so the seeded Domain record and Root Space are bound to it from the start:

   ```
   DEFAULT_DOMAIN="osos.lan" node ./dist/migrate
   ```

If the instance was already seeded against `localhost:4200`, either reset it and re-seed with the new `DEFAULT_DOMAIN`, or edit the Domain record's `host` and the Root Space's domain URL, then restart `data-service` so its boot observers rewrite the routing caches.

Testing Spaces and Domains on the domain

With wildcard DNS in place, create a second Space with a `domain_identifier` (lowercase letters, digits, and underscores only), and it becomes reachable at `<identifier>.osos.lan` with no further DNS or proxy work. Confirm the host actually landed in both routing caches before blaming the browser — a Space without a matching Domain produces `SPACE_ERROR` or `DOMAIN_ERROR` even though the UI shows the Space as created. The full model, both records' fields, and the resolver behavior are described in Spaces & Domains [Administration › Spaces & domains].

> **Note**
>
> One behavior changes when you leave `localhost:4200`: the Space provisioner short-circuit is keyed to that literal string, so on a real domain the platform will emit space-provisioner CloudEvents on Space create/activate/deactivate. With no `SPACE_PROVISIONER_URL` configured the call fails and is logged without failing the operation, which is harmless locally — but it is why the log gains provisioner errors the moment you switch domains.

Loopback fallback: the `127.0.0.1` workarounds

If you are not going to set up DNS, the instance has to be reached over the loopback address, and on Windows that takes several coordinated settings. This is the fallback, not the recommendation — the domain setup above exists precisely to avoid it.

Why `localhost` fails on Windows

Windows resolves `localhost` to the IPv6 loopback `::1` first, and Docker Desktop does not forward the IPv6 loopback. Published container ports — the portal on `:4200`, the gateway on `:8000`, the RabbitMQ UI on `:15672` — are therefore dead on `localhost` and alive on `127.0.0.1`. Services running natively (identity on `:3001`, `data-service` on `:3000`) bind both, so they work either way, which is what makes the failure so confusing: half the stack answers on `localhost` and half does not.

The browser makes it worse. Page navigation falls back from `::1` to IPv4, but `fetch` and `XHR` do not — so the portal can load on `localhost:4200` and every API call it makes still fails. The portal must be pointed at `127.0.0.1` explicitly.

An alternative is to make Windows prefer IPv4 globally, from an elevated prompt:

```
netsh interface ipv6 set prefixpolicy ::ffff:0:0/96 60 4
```

That keeps `localhost` usable everywhere, at the cost of a machine-wide network policy change — and the space and domain host mappings still have to cover whichever host the browser ends up sending.

The loopback OIDC callbacks are already registered `Since 1.10`

Nothing to do here any more. The development-only `localhost-angular` client in `backend/services/users-service/src/clients.json` now carries both host spellings out of the box:

```
redirect_uris:              http://localhost:4200/auth-callback
                            http://localhost:4200/silent-renew.html
                            http://127.0.0.1:4200/auth-callback
                            http://127.0.0.1:4200/silent-renew.html
post_logout_redirect_uris:  http://localhost:4200
                            http://127.0.0.1:4200
```

Previously only the `localhost` spellings were listed, so opening the portal on `http://127.0.0.1:4200` — the address Docker Desktop on Windows forces — ended sign-in in an invalid-redirect error, and the file had to be hand-edited locally. It grants nothing the `localhost` entries did not; it is the same client, reachable by both names.

Any **other** host still needs its own entries added by hand — that is the local-dev edit in step 3 of Configure the platform for the domain.

Keep the OIDC `AUTHORITY` on `localhost:3001` — identity is a native service that binds both stacks, and the issuer must keep matching what it announces.

Map the loopback host with `DEV_HOST_ALIASES` `Since 1.10`

This used to be a source edit. It is configuration now — one environment variable, no patched file, nothing to keep out of a commit.

A Space and a Domain each record **one** host: the address a browser is expected to use. Locally that is not the address the gateway answers on, so the routing caches built from those records never match the `Host` the browser actually sends, and every API call returns `SPACE_ERROR` and then `DOMAIN_ERROR`. Both caches are now built through one shared aliasing layer, configured by `DEV_HOST_ALIASES` in `backend/.env`:

Shape	A JSON object mapping a recorded host to one host, or to a list of hosts
Example	`DEV_HOST_ALIASES={"localhost:4200":["localhost:8000","127.0.0.1:8000"]}`
Default when unset	`{"localhost:4200": ["localhost:8000"]}`
A host with no entry	Resolves to itself
Malformed value	Logged, not thrown — it is read while refreshing the cache, so a typo must not take the refresh down. The default is used instead.
Production	Sets nothing, so nothing changes for a real deployment.

> ⚠ **The default does not cover `127.0.0.1`**
>
> The unset default aliases `localhost:4200` to `localhost:8000` **only**. Docker Desktop on Windows publishes container ports on `127.0.0.1`, so a developer browsing that address hits a host neither cache recognises — and the symptom is **a blank page with no error at all**. The gateway's plugins answer `SPACE_ERROR` / `DOMAIN_ERROR`, but nothing surfaces that in the browser. If you browse `127.0.0.1`, you must set `DEV_HOST_ALIASES` explicitly and list it, exactly as in the example above.

One semantic matters when writing the value: the alias list **replaces** the recorded host, it does not add to it. That mirrors the hardcoded rewrite it generalises, where `localhost:4200` was replaced by `localhost:8000` rather than served alongside it. To keep *both* `localhost:8000` and `127.0.0.1:8000` usable, list both.

The Domain side is the part that changed most: it previously took its host verbatim, with no rewrite at all, so covering the Space alone was never enough. Both sides now run through the same function, which is why one setting fixes both.

After changing the value, **restart** `data-service` — the caches are rewritten by boot observers, so a running service keeps serving the old host lists.

Add the loopback host to the Default Domain

`DEV_HOST_ALIASES` fixes what the gateway resolves. Identity has its own check: it shows the username and password form only when a **Domain** lists the host the browser returns to. The seed gives **Default Domain** one host, `DEFAULT_DOMAIN` (default `localhost:4200`), so on `127.0.0.1:4200` the sign-in page is **blank**. Seed with `DEFAULT_DOMAIN="127.0.0.1:4200"`, or on an instance already seeded add `127.0.0.1:4200` as another **Host** on Default Domain (**Administration › Applications › Data Entities › Domain**, see Spaces & Domains [Administration › Spaces & domains]). To get a session for that, sign in on `localhost:4200` with the browser forced onto IPv4 — for example Chromium started with `--host-resolver-rules="MAP localhost 127.0.0.1"`. The saved change reaches identity and the gateway's domain cache through the update event, with no restart; a direct edit of the Mongo `domain` collection skips that event and leaves identity's copy stale.

Portal sign-in comes from the host's `tenant-config` `Since 1.10`

The portal no longer reads its OIDC settings from `app-config.json`. Before Angular starts it requests `GET /api/identity/v1/tenant-config` on the host it was opened on and takes `authority`, `clientId` and `scope` from the response (the contract is in Authentication & OAuth). Consequences for a workstation run:

- **`OIDC_WEB_CLIENT_ID` must be set on the identity service.** `users-service/config.env` ships `OIDC_WEB_CLIENT_ID="localhost-angular"`; without it `tenant-config` answers `503` and the portal shows its "sign-in service not available / still being set up" page instead of starting.

- `nx serve` proxies `/api` and `/socket.io` to the gateway named by `OSOS_API_TARGET` (default `http://localhost:8000`, or a remote environment) with `changeOrigin`, so the gateway resolves the tenant from the *target's* host. The development build signs in through the `localhost-angular` client, because a real environment's web client does not list `http://localhost:4200`. The production bundle has no such override, and CI fails a production build that contains it.

- **The `osos-web` image** has no such proxy: its nginx serves the portal only, so on the split-port stack (portal `:4200`, gateway `:8000`) `/api/identity/v1/tenant-config` falls through to the portal's page and the portal stops on `BAD_CONFIG`. The image (`Since 1.10`) reads any `*.conf` mounted into `/etc/nginx/conf.d/extra/` inside its server block, so mount a **local-dev only** include that does what `nx serve` does. Behind a front door you do not need it: the front door already routes `/api` and `/socket`.

```
# same-origin-api.conf — local-dev only. Mounted into osos-web at /etc/nginx/conf.d/extra/
location ^~ /api/ {
    proxy_pass         http://osos-api-gateway:8000;
    proxy_http_version 1.1;
    proxy_set_header   Host 127.0.0.1:8000;      # a host the gateway already resolves
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Proto $scheme;
    client_max_body_size 512m;
    proxy_read_timeout 300s;
}
location ^~ /socket.io/ {
    proxy_pass         http://host.docker.internal:3017;   # the public websocket service
    proxy_http_version 1.1;
    proxy_set_header   Upgrade $http_upgrade;
    proxy_set_header   Connection "upgrade";
    proxy_set_header   Host $host;
    proxy_read_timeout 3600s;
}
```

Mount it from `backend/docker-compose.override.yml` (keep the file outside the repository), then `docker compose up -d osos-web`:

```
# backend/docker-compose.override.yml — local-dev only
services:
  osos-web:
    volumes:
      # keep the file outside the repository
      - /path/to/osos-local/portal-nginx/same-origin-api.conf:/etc/nginx/conf.d/extra/same-origin-api.conf:ro
```

The include sends `Host: 127.0.0.1:8000`, so keep that host in `DEV_HOST_ALIASES`.

- A broken bootstrap renders a static bilingual error page with a `data-reason` code and the gateway correlation id. `BOOTSTRAP_ENDPOINT_MISSING` usually means the backend image or the `/api/identity` gateway route is older than the frontend.

Other local-run defaults that changed `Since 1.10`

- **`space-resolver` answers `404 SPACE_ERROR`** for a host that matches no Space (it was `400`); `domain-resolver` still answers `400 DOMAIN_ERROR`. The portal classifies either as an unresolved host.
- **`APPLY_RATE_LIMITER`** — the `/identity` rate limiter (10 non-`GET` requests per 15 minutes per client address) is **on unless the value is `false`** (`Since 1.4`: the flag used to be inverted). The local `users-service/config.env` sets `"false"` so repeated sign-in tests are not throttled; leave it unset for a production-like run.
- **`OIDC_COOKIE_KEYS`** is required unless `NODE_ENV` is `development` or `test` (the local `config.env` sets `NODE_ENV=development`, so a key is generated with a warning). Signing keys are no longer read from a committed `jwks.json`; they are generated into the tenant's key store on first boot, or read from PEM files in `OIDC_SIGNING_KEYS_DIR`.
- **`SPACE_SCOPE_ENFORCEMENT`** defaults to **`enforce`** in code when unset — the usual case on a workstation — so a script that calls a service directly without an `x-space-id` header gets `400 SPACE_CONTEXT_MISSING` instead of silently using the Root Space. Through the gateway the header is always supplied from the host.
- **`ENABLE_SPEC=false`** now really stops `/api/openapi.json` and `/api/explorer` (`Since 1.2`, an update on every line); it used to leave the JSON document served.

- `SIGNIN_AUTO_REDIRECT_SINGLE_PROVIDER` is a commented-out example in `config.env`; unset means on, so the login page is skipped when one external provider is the only way in.

Launch the whole backend fleet

The backend services are not in Compose; they run natively. Two root scripts start them, and they are not interchangeable:

Script	What it starts
`pnpm run start:core`	The merged `core` process, plus only the services `core` does **not** host (`oidc`, `storage`, `indexing`, `scheduler`, `notification`, `web-socket`, `rtc`). This is the one to run day to day.
`pnpm start`	Every workspace service as its own process — **except** the merged `core` bootstrap.

```
cd backend
pnpm run start:core
```

Both scripts build first (Turbo's `start` task depends on `build`), and both default `TENANCY_MODE=single`, `APP_ID=local` and `ALLOW_INSECURE_FIELD_ENCRYPTION_KEY=true` for every service they launch; a value already set in your shell wins. A third script, `pnpm run start:main`, used to start a hand-picked minimal subset; it was removed in 1.10 `Removed 1.10` because `start:core` already brings up everything a local run needs.

`core` is a bootstrap that boots several service applications (`audit`, `notification-api`, `reaction`, `dependency-service`, `data-service`, `comments`, `timeline`, `workflow-service`, `logging-service`, `integrations`, `packaging-service`, `functions`) inside **one** process, so starting it alongside the same services individually would run each of them twice. `pnpm start` therefore leaves it out, and `start:core` is the script that runs it — leaving out, in turn, every application `core` already hosts.

Two constraints then shape a hand-rolled launcher: every service defaults its Prometheus metrics server to the same port, so each needs a distinct `METRICS_PORT` — non-fatal for HTTP services, fatal for the port-less bus workers — and a service started on its own does not get the defaults the root scripts supply, so it refuses to boot without `APP_ID` (single-tenant mode), and the services that encrypt fields also need a master key or, locally, `ALLOW_INSECURE_FIELD_ENCRYPTION_KEY=true`.

The altcha and shortener services are **not** in this list: they are not workspace packages. They live in their own repositories and run locally as Compose images, published on the ports the other services expect (altcha on `3023`, the shortener on `3010`):

```
cd backend
docker compose up -d altcha-service shortener-service
```

A launcher that assigns ports, sets those defaults, and keeps one log per service:

```
#!/usr/bin/env bash
# ~/osos-start-all.sh — local-dev only
cd "<path-to>/backend"
export BASH_ENV="$HOME/.pnpm-bashenv"     # Windows-only; see the note below
mkdir -p "$HOME/osos-logs"
mp=9101
# the defaults the root start scripts would have supplied
export TENANCY_MODE="${TENANCY_MODE:-single}" APP_ID="${APP_ID:-local}" \
       ALLOW_INSECURE_FIELD_ENCRYPTION_KEY="${ALLOW_INSECURE_FIELD_ENCRYPTION_KEY:-true}"
for s in oidc data-service logging-service workflow-service reaction storage \
         notification-api indexing audit comments timeline scheduler packaging-service \
         dependency-service web-socket notification; do
  env METRICS_PORT=$mp pnpm --filter "$s" run start > "$HOME/osos-logs/$s.log" 2>&1 &
  echo "launched $s (METRICS_PORT=$mp)"
  mp=$((mp+1))
done
echo ALL_LAUNCHED
wait
```

Run it in the background and leave it running — it owns the service processes:

```
bash ~/osos-start-all.sh &
```

The list is the **workspace** name, which often differs from the directory name (identity's package is `oidc`, `web-socket-service` is `web-socket`). `BASH_ENV` is only needed on native Windows, where it supplies the `nullglob` setting the package scripts assume.

Services must be listening **and** their bus queues bound before the initial-setup migration runs, because provisioning is published over the bus and a topic exchange drops messages nobody is consuming. Confirm the two that matter have a consumer:

```
docker exec backend-rabbitmq-1 rabbitmqctl list_queues name consumers \
  | grep -E "data-service-queue|users-service-queue"     # both should show 1
```

Recover the first login

Who the super-admin *is* comes from `services/data-service/.env.migration` — the file `pnpm run setup:local` creates — through `SUPER_ADMIN_EMAIL` and `SUPER_ADMIN_USERNAME`. Set both before you migrate, and use an inbox you can read for `SUPER_ADMIN_EMAIL`: the defaults in the code and in the example files are not addresses you will receive mail at. With a mail catcher (below) any address works: every message lands in the same inbox.

The super-admin gets **no password**. When the identity service creates the account it gives it an unusable random password, issues a password-reset token, and emails the set-password link to `SUPER_ADMIN_EMAIL`. Since 1.10 only a SHA-256 hash of the token is stored (`resettokens.tokenHash`), so the link cannot be rebuilt from the database — reading the collection gives you nothing you can open. Catch the email instead.

Run a local mail catcher and point the email sender at it. Mailpit takes SMTP on `1025` and shows every message in a web inbox on `8025`. Add it to `backend/docker-compose.override.yml`, and give the `email-service` container its SMTP settings (the same `SMTP_*` variables a cluster deployment sets):

```
# backend/docker-compose.override.yml — local-dev only
services:
  mailpit:
    image: axllent/mailpit
    ports:
      - "8025:8025"
    environment:
      MP_SMTP_AUTH_ACCEPT_ANY: "1"
      MP_SMTP_AUTH_ALLOW_INSECURE: "1"
  email-service:
    environment:
      SMTP_HOST: mailpit
      SMTP_PORT: "1025"
      SMTP_SECURE: "false"
      SMTP_USERNAME: local
      SMTP_PASSWORD: local
```

Recreate the two containers (`docker compose up -d mailpit email-service`), then run the migration. Open `http://localhost:8025`, find the set-password email, and follow its link. The reset router is mounted under `/identity`, so the link reads `/identity/reset-password/<token>` on the host you configured — `http://osos.lan/identity/reset-password/<token>` on a domain setup. Set a password, then sign in at the portal.

If the link has expired (24 hours by default) or the email never arrived, open `/identity/reset-password` on the same host and request a new link for the super-admin email you seeded. Each request replaces the previous link, so only the newest one works.

Diagnose a bring-up that will not come up

The failure modes move as you fix each layer — connection refused, then `SPACE_ERROR`, then `DOMAIN_ERROR`, then a real response. That progression means you are making progress; each one is a different layer answering.

```
# What is actually listening? Authoritative on Windows, where curl is easy to misread.
powershell -c "Get-NetTCPConnection -State Listen | ? {$_.LocalPort -ge 3000 -and $_.LocalPort -le 3025} | select Lo
calPort"

# Curl a published port without the proxy/loopback quirks getting in the way:
curl -s --noproxy '*' -o /dev/null -w "%{http_code}\n" http://127.0.0.1:8000/api

# Can the gateway container reach the native services on the host?
docker exec backend-osos-api-gateway-1 sh -c \
  'curl -s -o /dev/null -w "%{http_code}\n" http://host.docker.internal:3000/'

# Which plugin is rejecting the request? Replay the preflight and read the body:
curl -s -i --noproxy '*' -X OPTIONS http://127.0.0.1:8000/api/language \
  -H "Origin: http://127.0.0.1:4200" \
  -H "Access-Control-Request-Method: GET" \
  -H "Access-Control-Request-Headers: authorization"

# A service's boot log, if you used the launcher above:
tail -n 40 ~/osos-logs/data-service.log
```

The preflight replay is the fastest way to separate a genuine CORS problem from a tenancy one. The gateway allows all origins, so a rejected preflight is almost always the space or domain resolver refusing the host — the error body names which.

Notes on versioning

Most of this page is **Long-standing** — the split-port local topology, the host-derived tenancy, and the seed behavior all predate the 1.2 milestone.

The mechanics that changed recently are **developer-environment facts rather than product capabilities** — they change nothing a tenant, an administrator or an app builder can see, and they are not part of any release the product *announces*. They do, however, live in the backend repository itself, so they ship with the code: `Since 1.10` and later, a checkout of `develop` or the `1.10` release branch has them; a checkout of `1.8` or an earlier release branch will not. That is exactly what the version filter is for: they carry `` `Since 1.10` `` so filtering this page to an earlier milestone hides the steps that would not work there.

- `DEV_HOST_ALIASES` and the shared aliasing layer behind it, on both the Space and the Domain side. Before it, the loopback host mapping was the hand-applied source patch this page used to document.
- The `127.0.0.1` callbacks on the `localhost-angular` OIDC client.
- `scripts/setup-local-config.js` (`postinstall` / `pnpm run setup:local`) and the `OSOS_APP_CONFIG` mount indirection.
- The AD and Entra ID directory importers leaving this repository for their own, which also retired the `azure-ad-importer` exclusion `pnpm start` and `pnpm run start:core` briefly carried.
- The portal bootstrap from `tenant-config` and the other local-run defaults listed under Other local-run defaults that changed — these *are* product changes, shipped on the 1.10 line.

OSOS as a Backend

Everything the OSOS UI does is an HTTP call. The web application is one client of a gateway API on the order of 600 routes (about 450 of them are typed operations described by the services' OpenAPI documents; the rest — packaging, files, sign-in and a few others — are documented by hand) — it has no private side channel, and nothing it can do is closed to other callers. That makes the platform usable **headless**: as the backend behind your own scripts, services and products, with its UI out of the picture entirely.

This cluster of guides is about building those other clients. This page sets the model, names the three shapes an integration takes, states what the server keeps enforcing no matter which client calls, and maps which existing guide owns which part of the story — so the rest of the cluster can stay task-focused and never restate the fundamentals.

The headless model

A headless client needs three things: a base URL, a credential, and the conventions in Calling the HTTP API. There is no required SDK.

All traffic goes to the **gateway host**, and the host itself names the tenant: a gateway plugin resolves the request's `Host` header to a space id and forwards it as `x-space-id`. Your client never sends a tenant parameter — calling `https://acme.example.com/api/…` *is* selecting the tenant.

Credentials are validated at the edge, once. The gateway's authentication plugins turn a bearer token or an API token into a set of identity headers that downstream services trust — which is why those services must never be exposed directly, a boundary The security boundary covers in full.

On the way back, every JSON response arrives in the same `{success, result}` envelope, with no version condition — except that a `GET` on `/v3` is raw again `Since 1.10` (details) — and every error in a single localized shape with a stable machine code. The contract — routing, URL versioning, envelope, error body, pagination, obtaining the OpenAPI documents — is owned by Calling the HTTP API and holds for everything in this cluster.

Three integration shapes

Almost every integration is one of three shapes, and the shape decides your credential before you write any code:

Shape	Typical example	Credential	Requests run as
First-party script or automation	A data loader, a nightly sync, a migration	API token (`x-app-token`)	The token's application user
Headless service	Your own product using OSOS as its backend	An API token per environment, or a machine-to-machine OAuth client	A dedicated service identity
Third-party acting for a user	A BI tool, a portal surfacing platform data to its own users	Each user's own OAuth bearer token	Each end user, individually

A **first-party script** is the fastest path to a working call: get a token, send it as a header, done — the Quickstart takes exactly this route. The token carries no scopes of its own; it is bound to a dedicated *application user*, and its permissions are precisely that user's roles and space membership. Least privilege therefore means constraining the user, not the token — Choosing a credential works through the consequences.

A **headless service** uses the same mechanics with more discipline: its own application user (or OAuth client) per environment, so staging can be revoked without touching production, and rotation can happen without downtime —

covered in Token lifecycle and Environments and promotion.

A **third-party acting for a user** should hold *no* credential of its own. Each user authenticates through the platform's OIDC provider, and the integration forwards that user's token, so every request executes with exactly that user's permissions. The flows live in Authentication & OAuth — and the platform ships a worked example.

A shipping example: the Grafana data source

The Grafana data source is the third-party shape running in production, and two of its design choices are worth copying rather than re-deriving:

- **It holds no credential.** Grafana's server-side proxy attaches the signed-in viewer's own OAuth token to every request (`oauthPassThru`). Two viewers of the same dashboard can see different data from the same panel, because each request runs as the person looking at it.
- **It reads through the gateway's view route.** Every call goes to `/api/v1/v/{viewId}/…`, where the Kong `osos-access` plugin validates the remainder of the path against that view's execution plan, checks the caller may access the view, and injects the `x-view-id` header itself. The integration can reach only what the view deliberately exposes — a server-side allowlist, not a client-side promise.

If you are building anything that shows platform data to someone else's eyes, start from this pattern: pass the viewer's token through, and scope reads to a view.

What the server enforces, whatever the client

The platform does not trust its own UI, so it does not need to trust yours. Four guarantees hold for every caller:

- **The permission ladder.** Roles, space membership and per-view access are evaluated server-side on every request. One behaviour to know up front: a plain (non-view-scoped) entries list on an entity the identity may not read returns a *successful, empty* result rather than an error, while a view-scoped path answers `403` when the view forbids the operation.
- **Business rules.** Writes from your client enter the same pipeline as writes from the UI: workflow transitions, automation reactions, and the verification challenge that entry creation can demand all still apply.
- **Validation.** Field and model validation runs server-side and fails with typed, stable error codes — your client's local checks improve UX, they do not create safety.
- **Tenancy scoping.** The space is resolved from the host at the gateway and every operation executes inside it. A client cannot address another tenant by adding a header; what identity can and cannot express is the subject of Tenancy and identity.

Who owns what

This cluster is deliberately additive. The platform's existing guides own the ground they already cover, and pages here link to them rather than restating them — where wording ever differs, the owning page wins:

Concern	Owning guide
Routing, URL scheme and versioning, the response envelope, the error body, pagination and filtering conventions, obtaining the OpenAPI specification	Calling the HTTP API
The OIDC provider, grant types, the gateway's authentication plugins, identity-header propagation	Authentication & OAuth

Concern	Owning guide
Creating an application user and its API token	Adding an application user [Administration › Create an API token for an integration]
Regenerating and revoking API tokens	API Tokens · Token lifecycle
Sockets from the browser, RTC	Real-Time (Sockets & RTC)
Outbound webhooks and inbound integrations	Webhooks & Integrations
Reading platform data from Grafana	Grafana Dashboards

What changed for integrators `Since 1.10`

These changes on the 1.10 line affect a headless client directly. Each is covered where it belongs; this is only the pointer:

- **Entries now carry an optimistic-concurrency token.** Send back the `updateIdentifier` you read and a write that would overwrite someone else's change fails with `409` instead of silently winning. It is opt-in per request, and the coverage has known gaps — Writing data has the recipe, Calling the HTTP API has the contract.
- **A view can publish its own OpenAPI document**, five operations wide, which is a narrower and more stable thing to generate a client from than a whole service specification — Generating a client.
- `GET /v1/collections/{id}` **serves your own and shared collections**, so opening a list you just created no longer answers `404` — Reading data.
- **The** `entries/value-updated` **socket event carries the entry's token**, so a connected client can notice a conflict before it tries to save — Real-Time.
- **Refusals that used to answer** `500` **now answer a 4xx with a stable code** — re-archiving an archived entity, a hidden workflow action, an unknown filter pointer, an entry create with no body, an expired uploaded file, and more; a 4xx relayed from another service keeps its status (the hidden-action `422` and the relayed status are also `Since 1.8`) — Calling the HTTP API, with retry guidance in Errors and retries.
- **Plan limits can refuse creates** with `403` `ENTRY_CAP_EXCEEDED`, `ENTITY_CAP_EXCEEDED`, `SPACE_CAP_EXCEEDED` or `STORAGE_CAP_EXCEEDED` — never reads, edits or exports — Plan limits.
- **A tenant under maintenance answers** `503 TENANT_IN_MAINTENANCE` **with** `Retry-After` in a multi-tenant deployment — while its data is restored or it is moved; back off and retry — Tenancy- and space-related refusals.
- **A tenant's system administrator can back up, restore and export the tenant's data** through an API (multi-tenant deployments only; no web screen in 1.10) — Tenant backups, restores and exports and Long-running work.
- **The gateway strips a client's** `x-service` **and clears its** `x-view-id` — What the gateway strips.
- **New REST families** — the public system-settings read, the WhatsApp template list and webhook, and the packaging routes, which no generated specification contains — Other REST families added in 1.10.

The Endpoint reference has been regenerated from the current OpenAPI snapshots and now lists this cycle's new operations alongside the rest. Read the operations there rather than from a list in prose — it is generated, so it does not drift.

The pages in this cluster

Start here. The Quickstart gets a token and walks through your first authenticated read, the envelope, one real error, one paged read and one create — in under fifteen minutes.

Credentials and identity. Choosing a credential is the decision matrix across the three shapes; Token lifecycle covers refresh, expiry, rotation without downtime and revocation lag; Tenancy and identity explains space scoping and delegation; The security boundary draws the line between what a client may hold and what must stay server-side.

Working with data. Reading data covers the filter dialects, projection and safe paging in depth; Writing data covers creates, updates, bulk operations and their per-row results; Files covers upload, download and the attachment lifecycle; Subscribing connects a non-browser client to the public socket; Long-running work covers job handles, polling, terminal states — and the responses that are downloads, not JSON.

Shipping a client. Errors and retries is the error-code catalogue with a written retry policy; Generating a client turns the per-service OpenAPI documents into typed code; Environments and promotion takes the result from development to production.

Backend Quickstart

In the next fifteen minutes you will make your first authenticated call against a OSOS tenant and leave with the four habits every client needs: sending a credential, unwrapping the response envelope, branching on a real error, and paging a listing safely. Everything runs with plain `curl`; every response body shown is the enveloped shape the wire actually carries.

Six steps: get a token → make one authenticated read → parse the envelope → handle one error → paginate → create an entry.

What you need

- **A tenant you can reach**, for example `https://acme.example.com`. The host itself names the tenant — the gateway resolves the space from the request's `Host` header — so no tenant parameter appears anywhere below.
- **An application user and its API token**, which a tenant administrator creates for you (step 1).
- **An entity with a few entries** the token may read (and, for step 6, create in). The examples use `employees`; substitute the id or the normalized name of any entity in your space — the path accepts either.
- `curl`, and optionally `jq`.

Set two variables and the rest of the page is copy-paste:

```
export BASE="https://acme.example.com"
export TOKEN="<your API token>"
```

Step 1 — Get a token

An API token belongs to an **application user** — a user of type *Application*. A tenant administrator creates one in **Administration › Organization › Users** (see Adding an application user [Administration › Create an API token for an integration]), and the platform **emails the new token to the administrator who created it**. It is stored only in hashed form: it is never shown on screen and cannot be read back, so if it is lost, a system administrator regenerates it (**Regenerate Token**). Rotation and revocation are covered in Token lifecycle.

What you just created is not a free-floating secret. The token is bound to a dedicated **application user**, and every request made with it runs *as that user* — its reach is exactly that user's roles and space membership, because the token itself carries no scopes. Before continuing, make sure the application user may list your chosen entity's entries, which this quickstart reads straight from the entity. That needs **Edit** on the entity or its module: **View** shows the entity's design, not its records, and the list then comes back empty. Edit also covers creating entries, for step 6. An administrator grants that like any other access: see Give a group access to a module or data entity [Administration › Permissions & access]. For a real integration that only reads, prefer a view shared with the application user and read through it (`/api/v1/v/{viewId}/…`, see Calling the HTTP API): that needs access to the view only, not Edit on the entity. If an app token's fixed identity is the wrong shape for your integration — say, you act on behalf of interactive users — see Choosing a credential.

Step 2 — Make an authenticated read

List two entries. The token travels in the `x-app-token` header; the filter travels as a JSON object in the `filterCriteria` query parameter, which is why `-G --data-urlencode` is used:

```
curl -sG "$BASE/api/v2/entities/employees/entries" \
  -H "x-app-token: $TOKEN" \
  --data-urlencode 'filterCriteria={"limit":2}'
```

This is the v2 entries listing. A `200` comes back with the entries under `result`:

```
{
  "success": true,
  "result": [
    {
      "id": "665f1c0d2e9a4b3a9c1d0e2f",
      "dataEntity": "664ade91b2f7a2418c0c11aa",
      "specificationVersion": 1,
      "entityVersion": 4,
      "state": 1,
      "value": {
        "full_name": "Layla Hassan",
        "department": "HR"
      },
      "creationInfo": {
        "doneAt": "2026-05-02T09:14:33.201Z",
        "doneBy": { "dataEntity": "…", "dataEntry": "…" }
      }
      // … further entry metadata trimmed
    },
    { /* second entry */ }
  ]
}
```

Each element is a data entry: your business fields sit under `value`, keyed by each field's normalized name, and the platform's metadata (entity reference, versions, state, audit info) sits around it.

One behaviour to know now, because it looks like success: if `result` is `[]`, either the entity has no entries **or the token's application user is not allowed to list them** — for a plain (non-view-scoped) read the server returns an empty list rather than an error when list permission is missing, and listing needs **Edit** — View is not enough. An empty first read is a prompt to check the application user's access, not proof the data is absent.

Step 3 — Parse the envelope

Every JSON response from the API is wrapped in the same envelope — `{"success": …, "result": …}` — with no version condition, with one exception `Since 1.10`: a `GET` on a `/v3/` route is raw again (details). This quickstart reads through `v2`, so the payload is always under `result`, and `result` is `null` when an operation has nothing to return. The only responses that arrive unwrapped are the ones that are not JSON API results at all: file downloads and streams, which write to the response themselves, and the OpenAPI document. The full contract lives in Calling the HTTP API.

So a client unwraps once, at the transport layer, and everything above it works with plain payloads:

```
curl -sG "$BASE/api/v2/entities/employees/entries" \
  -H "x-app-token: $TOKEN" \
  --data-urlencode 'filterCriteria={"limit":2}' \
| jq -r '.result[].value.full_name'
```

```
Layla Hassan
Omar Haddad
```

If you generate a typed client from the OpenAPI documents, the same rule applies — the per-operation schemas describe the `result` payload, and Generating a client shows where the unwrapping belongs.

Step 4 — Handle one error

Ask for an entity that does not exist (`-i` prints the status line):

```
curl -siG "$BASE/api/v2/entities/no-such-entity/entries" \
  -H "x-app-token: $TOKEN"
```

The status is `404`, and the body is the platform's single error shape:

```
{
  "success": false,
  "result": {
    // remnants of the server-side exception — do not build on this field
    "errorCode": "ENTITY_NOT_EXIST",
    "errorMessage": "The data entity 'no-such-entity' does not exist",
    "service": "data-service",
    "retryable": false,
    "requestCode": 404
  },
  "error": {
    "code": "ENTITY_NOT_EXIST",
    "message": {
      "en": "The entity does not exist",
      "ar": "الكيان غير موجود"
    },
    "traceId": "8c2f5e0d-…"
  }
}
```

Four rules, in the order they matter:

1. **Branch on `error.code`.** It is a stable machine string, raised here by a typed not-found exception. `error.message` is localized display text and may change; never match on it.
2. **Trust the HTTP status for coarse handling.** The platform's typed exceptions carry their status with them — validation `400`, unauthorized `401`, forbidden `403`, not-found `404`, conflict `409`. The full code catalogue, with retryability per code, is in Errors and retries.
3. **Log `error.traceId`.** It is the request's correlation id; quoting it lets a problem be traced end to end through the platform's logs.
4. **Ignore `result` on failure.** It is *not* `null` — the response factory currently leaves remnants of the internal exception object there. Treat `"success": false` plus `error` as the failure signal and read nothing else.

Step 5 — Paginate

Count first — count is its own operation, separate from list:

```
curl -sG "$BASE/api/v2/entities/employees/entries/count" \
  -H "x-app-token: $TOKEN"
```

```
{ "success": true, "result": { "count": 137 } }
```

Then page with `limit` and `skip` inside the same `filterCriteria` object, always with an explicit `order`:

```
total=$(curl -sG "$BASE/api/v2/entities/employees/entries/count" \
  -H "x-app-token: $TOKEN" | jq '.result.count')

for ((skip = 0; skip < total; skip += 50)); do
  curl -sG "$BASE/api/v2/entities/employees/entries" \
    -H "x-app-token: $TOKEN" \
    --data-urlencode "filterCriteria={\"limit\":50,\"skip\":$skip,\"order\":[\"creationInfo.doneAt ASC\"]}" \
  | jq '.result[].value'
done
```

Three facts to design around, all set by the filter schema:

- Paging is **offset-based** — `limit` plus `skip` (or its alias `offset`); there are no cursors. A set that changes while you page can shift rows between pages, so always sort on a stable key, as the creation timestamp does above.
- `order` takes `"<field> ASC|DESC"`, as a string or an array, exactly as in the examples above.
- The count you fetched can drift while you page. Treat it as a progress estimate, not a contract.

Step 6 — Create an entry

Writes take a JSON body. Send the new entry's business fields under `value`, keyed by each field's normalized name:

```
curl -s -X POST "$BASE/api/v1/entities/employees/entries" \
  -H "x-app-token: $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"value": {"full_name": "Sara Nasser", "department": "Finance"}}'
```

The created entry comes back under `result`, with its new `id` and the same metadata you saw in step 2:

```
{
  "success": true,
  "result": {
    "id": "6660a1b2c3d4e5f6a7b8c9d0",
    "dataEntity": "664ade91b2f7a2418c0c11aa",
    "state": 1,
    "value": { "full_name": "Sara Nasser", "department": "Finance" }
    // … further entry metadata trimmed
  }
}
```

If a field fails the entity's rules — a required field left out, say — the answer is a `400` whose `error.code` is `ENTRY_VALIDATION_ERROR` (or `ALL_ENTRY_VALIDATION_ERROR`, with one item per failing field) and whose details name the field; branch on it exactly as in step 4. Updates, JSON Patch, batch creates and the per-row results of bulk writes are in Writing data.

Where to go next

- Reading data — the `criteria` expression tree inside `filterCriteria`, projection with `fields`, validating a filter before saving it, quick filters and tag filters.
- Writing data — creates, updates, and bulk operations with per-row results.
- Errors and retries — the full `error.code` catalogue and a written retry/backoff policy.
- Choosing a credential and Token lifecycle — when an app token is the wrong shape, and how to rotate without downtime.

- Calling the HTTP API — the conventions this page leaned on: routing, versioning, the envelope, the two filter dialects.

Choosing a Credential

Before an external client can call the OSOS API it needs a credential the gateway will accept, and picking the wrong kind is the most common way an integration starts on the wrong foot. The gateway understands two credential shapes: an **OAuth 2.0 bearer token** in the `Authorization` header, validated by introspection, and a long-lived **application token** in the `x-app-token` header. Behind the first sit several grant types, each meant for a different kind of caller.

This page owns one decision: **which credential fits which caller**. It does not explain how to mint any of them — Authentication & OAuth covers the OAuth provider, its grants and endpoints in depth, and API Tokens covers creating and managing application tokens. Once you have decided, the quickstart takes you from a fresh token to a working call.

The decision at a glance

You are building…	Use	Because
An interactive app a person signs into (web, mobile)	OAuth **authorization code + PKCE**	The platform hosts login, consent, MFA and passkeys for you; the user's own identity flows through.
A trusted first-party script that holds a user's credentials	OAuth **password grant**	Direct exchange of username/password for tokens — acceptable only when the script can safely hold a raw password.
A headless service, integration or scheduled job	**Application token** (`x-app-token`)	No token exchange, no refresh loop; a stable machine identity with exactly one user's permissions.
A third-party product acting on behalf of your users	OAuth **authorization code** with consent	Each user authorizes your client individually; you never see credentials, and one user revoking does not affect another.

The rest of this page walks each row, then covers the limits of the application token — the credential most headless clients end up with.

An interactive user

Use the **authorization code flow with PKCE**. The shipped SPA clients (`osos-<env>`, `osos-<env>`) are the pattern to copy: `token_endpoint_auth_method: "none"` (a public client), `authorization_code` + `refresh_token` grants, and wildcard subdomain redirect URIs so one client registration serves every tenant subdomain. Clients are provisioned statically — there is no dynamic registration endpoint — so registering yours means adding it to the clients file (or its `additional-clients.json` override) and restarting the identity service. `Since 1.10` those files are a seed written into the tenant's client collection at boot, and in a multi-tenant deployment the standard clients are templated per tenant instead (see Per-tenant OpenID Providers). A browser page that only needs anonymous access uses `POST /api/identity/v1/anonymous/token` — never a client secret in the browser.

Two behaviours the provider gives interactive clients for free:

- A public web client receives a **refresh token even without requesting** `offline_access` (`issueRefreshToken`), so sessions survive access-token expiry.
- The hosted login UI presents whatever factors the tenant has enabled — password, OTP, passkeys, external identity providers — without your client implementing any of them.

A trusted first-party script

The provider registers a custom **password grant**: the script posts a username and password to `/token` and receives tokens directly. This is the right shape only when *first-party* and *trusted* both genuinely hold — the script sees the raw password, so it must be yours, and the account is subject to normal account policy (bad credentials count toward lockout).

Weigh it against an application token before choosing it. The password grant ties your automation to a human account: its permissions change when the person's do, its password rotates on the person's schedule, and a lockout stops your job. If the script is really a service in disguise, a dedicated application user with its own token is the more stable identity.

A headless service

Use an **application token**. It is minted for a dedicated **application user** — a `USER` entry whose type is `APPLICATION_USER` — and sent as `x-app-token` on every request. The gateway's `app-token-auth` plugin validates it against the identity service and injects the owning user's id as `x-user-id`; from there the request is indistinguishable from that user acting directly. There is no OAuth exchange, no refresh loop, and no session to keep alive — the operational lifecycle it does have (expiry, rotation, revocation) is covered in Token Lifecycle.

Why not `client_credentials`? The machine-to-machine grant exists and is enabled, but a client-credentials token identifies a *client*, not a *user*: its introspection result carries no subject, so the gateway forwards no `x-user-id`, and downstream services resolve the caller to a profile with **empty permissions**. The request authenticates — and then can read almost nothing, because the data plane authorizes users, not clients. The grant only becomes useful in combination with the gateway's `trustedClientId` impersonation configuration, which is a deployment-level trust decision for internal infrastructure, not an application pattern. For a headless caller that needs to see tenant data, the application token is the supported answer. The one surface it does not open is the real-time socket, which needs a user's OAuth token: for a service, the password grant for a dedicated ordinary user — an application user cannot sign in with a password (Subscribing to events).

A third party acting for a user

Run the **authorization code flow** and let each user consent. The shipped `zapier`, `wordpress` and `n8n-localhost` clients in `clients.json` are working examples of exactly this pattern: a fixed redirect URI on the third-party side, `authorization_code` + `refresh_token` grants, standard scopes.

Two tempting shortcuts, and why they are wrong here:

- **Not the password grant.** A third party must never handle users' passwords; the grant exists for first-party trust only.
- **Not a shared application token.** An application token has one fixed identity, so every user's actions would be attributed to the same application user, per-user permissions would collapse into one profile, and revoking one customer's access would mean revoking everyone's.

The platform's **delegation** mechanism (the `X-Delegation-Id` header) is not this either: it lets one *person* act for another inside the platform during an authorized window, and is not a machine-integration surface. It is covered in Tenancy and Identity.

What an application token cannot express

An application token carries **no scopes**. The stored token document holds identity and lifecycle metadata — the argon2 hash, a name, active/expiry flags — and nothing else; the gateway consumes only the owning user's id from a successful validation. There is no field anywhere in that path that could narrow a token to "read-only" or "this entity only".

Everything the token may do is therefore decided by the **owning application user**: its roles, its space memberships, its view permissions. This has three practical consequences:

- **Least privilege is modelled on the user, not the token.** Create a dedicated application user per integration and grant it exactly what that integration needs.
- **One integration, one token.** Since a token cannot be scoped down, isolation between integrations comes from separate owning users — which also makes revocation surgical.
- **Auditing follows the user.** Actions appear as the application user, so a shared token across several systems destroys attribution.

The same inheritance also bounds the token spatially: it can only operate in spaces its owning user belongs to, which is enforced per request at the gateway — see Tenancy and Identity.

Deciding quickly

If a person is present at a screen, use authorization code + PKCE. If no person is present and the caller is yours, use an application token owned by a purpose-built application user. If no person is present but the caller belongs to someone else, make their users consent through the authorization code flow. Reserve the password grant for the narrow case of a first-party script that genuinely must hold a password, and treat `client_credentials` as infrastructure plumbing rather than an integration credential.

Token Lifecycle

Every credential ages: access tokens expire in minutes, refresh tokens rotate, application tokens reach their expiry date or get rotated by an administrator, and any of them can be revoked. This page is the operational side of that story for a client author — what expires when, how to refresh and rotate without an outage, and the one property that surprises most integrators: **revocation is not instant**, because the gateway caches validation results.

Choosing a Credential decides which credential you hold; Authentication & OAuth documents the provider's full TTL table and grant mechanics; API Tokens documents creating and managing application tokens. This page assumes you have a credential and need to keep it working.

Two credentials, two clocks

The two credential shapes age on entirely different scales, and a client is designed around whichever clock it lives on:

	OAuth bearer token	Application token
Typical lifetime	Access token: 1 hour by default, per-client override (`ttl.AccessToken`)	A calendar expiry date, or unlimited
Kept alive by	The refresh cycle (refresh token: 14 days, rotating)	Nothing — it is simply valid until expiry, deactivation or rotation
Renewal actor	The client itself, programmatically	An administrator, deliberately
On compromise	Revoke at the provider	Deactivate or rotate the owning application user's token

An OAuth client therefore carries token-refresh machinery as a core component; an application-token client carries none, and instead needs an operational answer to "what happens on the day the token rotates".

Refreshing an OAuth session

Access tokens are short-lived by design; the refresh token is what persists. Grants that yield refresh tokens, rotation policy and TTLs are specified and documented in Authentication & OAuth; what matters to a running client is the discipline:

- **Always store the refresh token from the last response.** Rotation means a refresh response may carry a *new* refresh token that silently replaces the one you used — public web clients rotate on every use, and every client rotates once a token passes 70% of its lifetime (`rotateRefreshToken`). Reusing a rotated-away token fails.
- **Rotation is not immortality.** Total rotated lifetime is capped at one year, and a rotated public-client token inherits the *remaining* TTL of its predecessor rather than restarting the 14-day clock.
- **Refresh proactively, not reactively.** The token response's `expires_in` tells you when the access token dies; refreshing shortly before that beats racing a `401` in production. When you do receive a `401` mid-flight, one refresh-and-retry is the correct recovery; a second `401` means the session is genuinely gone and the client must re-authenticate.

Application-token expiry

`Since 1.8` — an application token's expiry date is **enforced at validation time**. The stored `expiresAt` is a calendar date (`yyyy-MM-dd`), and the token remains valid **through the end of that day (UTC)**; a token flagged `unlimitedExpiry` , or with no date set, never expires (`isTokenExpired`). An `expiresAt` value that does not name a real calendar date is treated as **expired** — the check fails closed and logs the bad value rather than granting an accidental immortal token.

Practical consequences:

- Expiry is a **day boundary**, not a timestamp. Plan rotations for the day before, not the hour before.
- A token that stopped working on schedule is not an incident; regenerating it (below) both issues a fresh secret and re-synchronizes the expiry fields from the application user's entry.
- **An expiry can actually be set `Since 1.10` .** Before 1.10 the application user form kept its *Expiry date* field hidden, so every token was created without an expiry and never expired. The field is now a date (`yyyy-MM-dd`), shown unless *Unlimited expiry* is on. An entry whose expiry was stored as a full timestamp is treated as expired until the date is re-entered — see API Tokens.

Rotation without downtime

`Since 1.8` — an administrator can **regenerate** an application's token from the admin area. Regeneration issues a new secret, emails it to the requesting administrator, and — this is the part that makes zero-downtime rollover possible — keeps the **previous** token valid for a grace window (`regenerateApiToken`). The window is deployment-configured via `APP_TOKEN_GRACE_PERIOD_MS` , defaulting to **24 hours**; setting it to `0` revokes the replaced token immediately.

The rollover procedure that follows from the mechanism:

1. Regenerate. The old token keeps authenticating; nothing breaks yet.
2. Deploy the new secret to the integration within the grace window.
3. Verify traffic on the new token before the window closes — the token record tracks `lastUsedAt` , so an integration still on the old secret is discoverable.

One sharp edge, directly from the code: the record remembers exactly **one** previous hash. Regenerating twice in quick succession overwrites the first grace entry — the token your integration is still running on — and revokes it on the spot. Rotate, migrate, verify, and only then rotate again. (`Since 1.10` two *concurrent* regenerations are serialised per application across pods, so they can no longer both rotate from the same token; the sequential case above still applies.)

Revocation

For OAuth credentials the provider exposes the standard **revocation endpoint** (`/token/revocation` , enabled features); revoking a refresh token ends the session's ability to renew, and an access token can be revoked individually.

An application token is revoked through its owning application user's entry, in either of two strengths:

- **Deactivate** the application user — the update handler flips the token's `isActive` off; reactivating restores it.
- **Archive** the entry — the token document is soft-deleted and stops validating permanently; records are retained for audit, never hard-deleted.

Both take effect at the *identity service* immediately. Whether the *gateway* still honours the token for a while longer is the next section.

The revocation-visibility lag

The gateway caches successful validations in Redis so that steady-state traffic costs no identity-service round-trips. The same cache means a credential revoked at the provider can keep authenticating at the edge until its cached entry dies. The two paths cache differently — both TTLs sit in the plugin schemas and are deployment-configurable; the shipped configuration (`api-gateway/config/kong.yml`) overrides neither, leaving the defaults:

Path	Cached result	TTL	Expiry honoured from cache?
OAuth bearer (`oauth2-introspection` plugin)	The introspection response	`cache_ttl`, default **300 s**, additionally capped to the token's remaining lifetime minus 10 s; tokens within 30 s of expiry are not cached at all	**Yes** — a cached entry past the token's `exp` is evicted (with a 5-second clock-skew buffer) and re-introspected
Application token (`app-token-auth` plugin)	The validation result	`cache_ttl`, default **300 s**	**No** — a cache hit is trusted as-is for the full TTL; nothing is re-checked until it lapses

Read the table as a client author and as an operator:

- **Expiry is prompt; revocation is eventually consistent.** An expired bearer stops at the edge within seconds, but a *revoked-yet-unexpired* bearer — and a deactivated or freshly-rotated-away application token — can continue to authenticate for up to the configured `cache_ttl` after revocation.
- **The lag bound is a deployment choice.** It equals the plugins' `cache_ttl` (300 s as shipped), and on the OAuth path is further capped by the token's own remaining lifetime. A deployment with stricter revocation requirements sets a smaller `cache_ttl` and pays for it in validation traffic.
- Authentication is not the only cached thing: what a user is *permitted* to do is cached separately and on longer clocks — see the cache table in Tenancy and Identity.

Designing a client around all of this

The lifecycle facts above compress into a small set of client rules. Refresh OAuth sessions ahead of expiry and treat a single `401` as a refresh trigger, not a failure. Persist whichever refresh token arrived last. For application tokens, schedule rotation ahead of the expiry day, complete rollovers within the grace window, and never assume a revoked credential died instantly anywhere — build de-provisioning flows on the identity source (deactivate the user, revoke the session) and let the edge caches drain in their own bounded time.

Tenancy and Identity

OSOS is multi-tenant, and the tenant model reaches into every API call: the same operation, with the same credential, returns different data depending on which **space** the request is scoped to. This page explains how that scope is established, how identity and delegation travel with a request, and — the part that matters most when designing an external client — what a credential *cannot* do across spaces, no matter how it is presented.

The identity headers themselves (`x-user-id`, `x-shared-key` and friends) are catalogued in Calling the HTTP API; why downstream services trust them at all is the subject of the security boundary. Here we follow the tenancy and delegation story end to end.

The space comes from the host, not the token

A client never sends a tenant identifier. The gateway's `space-resolver` plugin runs at the head of the access chain, ahead of every authentication plugin: it reads the request's `Host` header, looks it up in a Redis key named `:spaces`, and stamps the matching space's id onto the request as `x-space-id` (plus `x-space-maintenance` when the space is closed to end users). A host that matches no space is rejected outright with `404 SPACE_ERROR` (`Since 1.10`; it was `400`) — before authentication is even attempted.

That `:spaces` cache is owned by the data service, which rewrites it whenever spaces change (`refreshSpaceInRedis`): an array of `{id, hosts, name, is_end_user_deactivated}` entries covering every active, non-deactivated space, each space's host derived from its domain and subdomain identifier. A sibling plugin, `domain-resolver`, resolves the same `Host` against a `:domains` key into an `X-DOMAIN-ID` header the view-access check later requires.

Two consequences for a client author:

- **You address a tenant with a URL.** Pointing the same client at `hr.example.com` and `sales.example.com` yields two different data worlds with no other change.
- **Whatever `x-space-id` you send is irrelevant.** The gateway sets the header itself from the host lookup; `Since 1.10` it first strips any `x-space-id` / `x-space-maintenance` the client supplied. (For OAuth flows the tenant is additionally bound *into the signed token* as the `extra.domain` claim — see Authentication & OAuth.)

Membership is checked on every request

Resolving a space from the host says nothing about whether *you* belong there. That is the `user-space-checker` plugin: for every authenticated request it verifies that the resolved `x-space-id` is among the calling user's spaces, answering `403 SPACE_FORBIDDEN` otherwise — helpfully carrying an `available_spaces` list (id, name, icon, URL) so a client can offer the user their actual spaces instead of a dead end.

Membership is fetched from the data service (`GET /api/v1/spaces/user/{userId}`) and cached in Redis for a configurable TTL (default **300 s**), keyed per user. The staleness window is shorter than it looks: when a user's space membership changes, the data service actively deletes that user's gateway cache entry (`invalidateKongUserSpacesCache`), so the TTL is a backstop rather than the propagation delay.

A small allow-list of bootstrap endpoints — profile, own-spaces, language, themes and similar (`configured at the gateway`) — skips the check, which is what lets a client ask "which spaces do I have?" before it is *in* any space.

How the space reaches the data plane

Past the gateway, the space travels as the `x-space-id` header and nothing else. LoopBack services resolve it through a request-scoped provider (`SpaceIdProvider`) — which, before 1.10, fell back to the root space id when the header was absent (now it fails closed) — and controllers inject that resolved id into every command and query they issue; the data-plane controllers thread it through reads, writes, counts and exports alike.

Services with their own tenancy-sensitive resources enforce it again at their boundary. The packaging service is the clearest example: every operation requires the header outright (`400` without it), and polling or manipulating an import session that belongs to another space is refused with `403` by an explicit ownership assertion. A long-running job is tenant property, not a global handle.

Services fail closed without a space `Since 1.10`

The root-space fallback above is gone (`SpaceIdProvider`):

- **Resolution order**: the single, non-blank `x-space-id` header, then the ambient space scope (an in-process dispatch, or a background job re-entering the space its payload carries). If neither exists the call answers `400` `SPACE_CONTEXT_MISSING` instead of acting in the root space. A header sent twice counts as missing.
- The space is established as a **scope for the whole request**, so in-process calls inherit it. A header naming a different space than the one already in scope, or an unusable header inside a scope, answers `400` `SPACE_NOT_RESOLVED` . Infrastructure paths (`/health` , `/ready` , `/live` , `/ping` , `/metrics` , `/explorer` , `/openapi.json`) run in the root space.
- Outbound service-to-service calls carry `x-space-id` from the scope.
- **Definition reads are space-scoped**: entity and view definition lists return the `Public` definitions plus the `Private` / `Shared` ones that list the current space; another space's private definitions are no longer returned.

Through the gateway none of this changes what a client does — the host still supplies the space. It matters to a script that calls a service directly on the cluster network, which now gets the `400` instead of silently reading the root space.

In a multi-tenant deployment `Since 1.10`

When one deployment serves several tenants, the host also decides the **tenant**. A host that maps to no active tenant answers `404` (`APP_ERROR` "Unknown host" at the gateway, `TENANT_NOT_RESOLVED` from a service). Each tenant has its own OIDC issuer, JWKS and clients, so validate tokens against the issuer of the tenant you call — see Per-tenant OpenID Providers. A browser client bootstraps from the public `GET /api/identity/v1/tenant-config` and obtains anonymous tokens from `POST /api/identity/v1/anonymous/token` (details). The gateway may also enforce a per-tenant request budget (`429 RATE_LIMITED` with `Retry-After`) — see Calling the HTTP API. And a tenant can be put **under maintenance** — while its data is restored from a backup, or while the operator holds it for a move to another cell — during which every service answers its requests with `503 TENANT_IN_MAINTENANCE` and a `Retry-After` (`Since 1.10` ; same link). The tenant stays routable and your credential stays valid: wait and retry.

View-scoped requests and the access object

Requests under `/api/v1/v/{viewId}/…` pass one more gate: the `osos-access` plugin loads the view's **access object** and enforces, in order:

1. **Domain** — the view must belong to the request's resolved domain (`403` otherwise; a missing domain id is `400` `MISSING_DOMAIN_ID`).

2. **Space** — unless the view is marked public-space, the request's `x-space-id` must appear in the view's configured space list (`403`).
3. **Execution plan** — the concrete URL and method must match the view's allow-list of routes; a view with no plan accepts reads but rejects writes.
4. **Authentication mode** — the access object declares who may enter: anonymous-only (`0`), any authenticated user (`1`), everyone (`2`), or explicitly-permitted users only (`3`), checked against the caller's cached permission profile.

Only then is the path rewritten, the view id set as `x-view-id`, and the request forwarded to the owning service. `Since 1.10` the plugin first clears any `x-view-id` the client sent, so a view can only be named through the path (details). This is why a `403` on a view-scoped path so often means "this view does not allow that" rather than "your token is bad" — the credential may be perfectly valid and the view still out of bounds in this space.

Two caches make this gate cheap, and both have visibility consequences: the access object itself (default **600 s**) and the user's permission profile (**3600 s**, keyed per user — and per delegation when one is active). A permission granted or revoked can therefore take up to an hour to be seen on view-scoped calls; compare the *authentication* caches in Token Lifecycle, which are tighter.

Delegation: acting as someone else, visibly

Delegation lets one person act for another during an authorized window, and it is the only sanctioned way a request runs under a different identity than its credential. The client's part is a single header: `X-Delegation-Id` .

The gateway's `user-delegation` plugin does the rest. It requires an authenticated caller, loads the delegation record (cached, default **600 s**), and enforces the time window — outside it the request dies with `403 DELEGATION_EXPIRED` . Within the window it **rewrites the identity**: `x-user-id` becomes the *delegator* (the person being acted for), the original caller moves to `x-delegatee-id` , and `x-is-delegation: true` marks the request. Both identities therefore reach every downstream service, which is what keeps history and audit truthful while permissions resolve as the delegator.

Downstream, delegation is not a free pass. The authentication strategy surfaces the delegation on the user profile, the data service narrows request-related views to the delegation's configured scope by rewriting their filters, and the view gate demands *explicit* view permission for most views while a delegation is active. Delegation is a person-to-person instrument — for machine integrations, model the identity properly instead with a dedicated application user (Choosing a Credential).

What a token cannot do

The mechanics above add up to a set of hard limits that hold for *any* credential — OAuth bearer or application token — because they are enforced by the edge and the services, not by the token's contents:

- **It cannot choose a tenant.** The space and domain derive from the request host; client-supplied `x-space-id` is overwritten, and the identity headers cannot be forged through the gateway — the edge strips and re-issues `x-shared-key` , and only sets `x-user-id` from a validated credential.
- **It cannot enter a space its user does not belong to.** Membership is checked per request; the only cross-space visibility a client ever gets is the `available_spaces` hint on the `403` .
- **It cannot reach across tenants through side doors.** View access is space-checked, and long-running job handles (exports, import sessions) are ownership-checked in their services.

- **It cannot escalate through a view.** The access object's authentication mode and execution plan bound what any caller may do on view-scoped paths, whoever they are.
- **It cannot impersonate.** Acting as another user exists only as a live, time-boxed delegation — always leaving both identities in the audit trail — or as deployment-internal trust configuration that no external credential can invoke.

What a token *can* do across spaces is exactly one thing: be reused. The same credential works against every space host its owning user is a member of — the scope changes with the host, the identity does not. A client that serves several spaces should treat the host as part of its configuration, not attempt to multiplex tenants through headers.

Reading Data

This is the read-side guide for an external client. Calling the HTTP API owns the shared conventions — the URL scheme, the response envelope, the filter-model reference and the pagination and list parameters. This page is about using them well: choosing the right filter dialect, validating criteria before you commit to them, fetching less than the whole Entry, and paging safely against a server that will not protect you from your own page sizes. Examples read one entity, `employees`, with an application token; every successful JSON response arrives as `{"success": true, "result": …}` regardless of the operation's version — except a `GET` on `/v3`, which is raw again `Since 1.10` (details).

Choosing a filter dialect

Two filter dialects coexist, selected by the operation version, and the choice is already made for you: the v1 list operation is marked `@deprecated()` in the controller, surfaced as `deprecated: true` in the OpenAPI document. It still works — deprecation on this platform is additive, not a cutover — but a new client should not adopt it.

Operation	Path	Filter input
List *(deprecated)*	`GET /api/v1/entities/{entity}/entries`	`filter` (LoopBack)
List	`GET /api/v2/entities/{entity}/entries`	`filterCriteria` + `search`
List with tags	`GET /api/v3/entities/{entity}/entries`	`filterCriteria` + `search` + `appliedTagsFilters`
Count *(v1)*	`GET /api/v1/entities/{entity}/entries/count`	`where` (LoopBack)
Count	`GET /api/v2/entities/{entity}/entries/count`	`whereCriteria` + `search`
Single entry	`GET /api/v1/entities/{entity}/entries/{id}`	`fields`, `excludeResolved`

The rule: **write new read paths against v2 or v3 and pass `filterCriteria`**. Take v3 when you want tag information with the rows — its result is an object carrying `entries` plus a `tags` array — and v2 when you want just the array. Counting is always a separate operation from listing.

`filterCriteria` is a superset of the LoopBack filter: it keeps `limit`, `skip` / `offset`, `order`, `fields`, `include` and a legacy `where`, and adds the recursive `criteria` expression tree. The expression model — operators, field operators, typed values — is documented in the conventions guide; this page assumes it.

```
curl -G "https://{host}/api/v2/entities/employees/entries" \
  -H "x-app-token: $TOKEN" \
  --data-urlencode 'filterCriteria={
    "limit": 100, "skip": 0, "order": ["value.hire_date DESC"],
    "criteria": {
      "operator": "And",
      "args": [
        { "field": "value.department", "operator": "Equal",
          "args": [ { "type": 0, "value": "HR" } ] }
      ]
    }
  }'
```

```
{ "success": true, "result": [ /* entries */ ] }
```

Malformed criteria fail with `400` and a stable code from the query engine: an operator the engine does not know is `UNKNOWN_OPERATOR_ERROR`, and a `fieldOperator` outside the supported set is `UNSUPPORTED_FIELD_EXPRESSION`.

Since 1.8 — criteria that reference a sensitive field [Build Apps › Protect a field] the calling user cannot access are **rejected outright** with `400` and code `SENSITIVE_FIELD_QUERY_NOT_ALLOWED`, rather than silently dropped — a deliberate design decision recorded in the engine. Sub-paths cannot slip past the guard: filtering on `salary.en` or `manager.dataEntry` is treated as filtering on `salary` or `manager`. Expect this error to vary per credential — a filter that works for one user can 400 for another.

Validating criteria with the parser

Before persisting a complex expression — into a saved Query, a client-side filter builder, a configuration file — validate it against the same engine the live endpoints use:

```
curl -X POST "https://{host}/api/v1/queries/parser" \
  -H "content-type: application/json" -H "x-app-token: $TOKEN" \
  -d '{
    "operator": "And",
    "args": [
      { "field": "value.department", "operator": "Equal",
        "args": [ { "type": 0, "value": "HR" } ] }
    ]
  }'
```

The body is a bare `FilterExpression` — the `criteria` node alone, not a whole `filterCriteria` object. On success the response is the parsed condition rendered as a string inside the envelope; treat that string as opaque — the useful signal is `200` versus a `400` carrying the same `UNKNOWN_OPERATOR_ERROR` / `UNSUPPORTED_FIELD_EXPRESSION` codes as a live query.

Two caveats. The parser has no entity context: field names are not checked against any Entity, so a structurally valid expression can still match nothing at runtime. And it does not run the per-user sensitive-field check above, which only applies when a query actually executes — a criteria tree that parses cleanly can still be rejected at read time.

Pointer values are checked strictly `Since 1.10`. A pointer value in a filter — `@me`, `@today('-1d')` and the like — must name a known pointer function in the form `@name`, `@name()` or `@name('<amount>')`; anything else (`@Today`, `@Me`, `@none`) is refused with `400 INVALID_POINTER`. A saved filter that relied on a loose spelling now fails where it used to be tolerated — fix the spelling.

Projection

On a **list**, projection is the standard `fields` clause inside `filter` or `filterCriteria` — include or exclude properties per LoopBack semantics.

On a **single read**, `GET /api/v1/entities/{entity}/entries/{id}` takes two dedicated parameters:

- `?fields=value.name,value.email` — a comma-separated projection. The read still runs through the view's access pipeline, deliberately: a raw projected fetch would let `?fields=` bypass row-level scoping, so the scoping stages stay on while the expensive resolve stages (virtual fields, relation and category resolution) are switched off. The projected read is therefore fast, but relation and category values come back **unresolved** — raw references, not display-ready objects.
- `?excludeResolved=true` — returns the full value but skips the relation- and category-resolution stages only. Use it when you need the whole Entry cheaply and can resolve references yourself, or don't need them resolved at all.

Paging safely

The server enforces no maximum page size. The filter schema requires only `limit >= 1` — there is no upper bound, in either dialect — and a request with no `limit` at all returns every matching row. The platform will not protect you from a full-collection read, so the discipline is the client's:

- Send an explicit `limit` on **every** list request, including exploratory ones. Pick a fixed page size in the low hundreds and keep it.
- Page with `skip` (or `offset`) in `limit`-sized steps; a page shorter than `limit` is the last one.
- Sort on a stable, unique key — append the entry id to whatever business ordering you need (`"order": ["value.hire_date DESC", "id ASC"]`). Offset paging over a mutating set skips or repeats rows when the sort key is not stable; there are no cursors to save you.
- Treat the count operation as advisory: rows can be created or archived between the count and the pages that follow.

Quick filters and tag filters

Both features let a request reuse filters that were defined once, server-side, instead of re-encoding them in every client.

`appliedQuickFilters` is an array of **saved Query ids**. Each referenced Query's criteria are AND-merged with your own `filterCriteria`, and a parameterized Query has its parameters resolved from the request. An id that does not resolve to a Query with criteria is **silently skipped** — a typo narrows nothing and raises no error, so a suspiciously large result set is worth re-checking against the ids you sent.

`appliedTagsFilters` (v3 list, tree and kanban operations) takes **tag ids or tag-group ids** — a group id applies every enabled tag in the group. The v3 response's `tags` array then reports which returned entries carry which tag, as `{tagId, entriesId, groupId}` objects. An empty `tags` array means no applied tag matched the returned rows — not that the tags don't exist.

Reading a collection by id

`Since 1.10` — `GET /api/v1/collections/{id}` now serves four kinds of collection to an authenticated caller: the caller's **own** saved collections; **open** ones, shared with every authenticated user; ones shared with them **directly**; and ones shared with **a group they belong to**. Previously it recognised only a narrower set, so a client that created a collection and then opened it by id got a `404` for a list it had just made itself — a confusing failure, because nothing about the credential or the id was wrong.

Authorization did not change. The visibility rule is still applied as a `where` predicate at query time rather than as a check after the document is fetched, so a `fields` projection cannot be used to smuggle out a collection you are not entitled to; a Restricted caller holding a grant on the entity still cannot retrieve another user's saved collection by guessing its id. Anything outside those four cases still answers `404`, with the same body as before. No route, method, parameter or response shape changed — only which collections the predicate matches.

Proxy entity capabilities `Since 1.10`

`GET /v1/entities/{id}` — the single-entity read, not the list endpoints — now returns an additional `proxyCapabilities?: ProxyCapabilitiesDto` property on `VirtualProxy` entities: a response-only, per-request-derived capability ceiling (which fields, which operations, which surfaces the entity's bound Proxy view currently permits). It is never accepted on write. An absent property means "not a proxy entity"; a present one with `bound: false` means "a

proxy entity whose capability chain did not resolve" — read the field to know what the entity currently allows rather than assuming from its `type` alone. The `data-service` OpenAPI snapshot has since been regenerated and now carries the property on `DataEntityDto`. Full shape and derivation rules: Virtual Entities § Proxy view governance [Build Apps › Virtual entities].

Reading a reaction's run history `Since 1.10`

`GET /v1/reaction/{entity}/{id}/history` (reaction-service) returns a keyset-paginated page of one reaction's recorded runs, newest first — query params `outcome` (`failed` / `succeeded` / `refused` / `all`; `refused` is a run the tenant's monthly run limit stopped — see Plan limits), `from` / `to`, `before` (keyset cursor), `limit` (default 10, capped 100); response `{items, hasMore, nextBefore}`, no total count by design. `GET /v1/reaction/{entity}` also now returns per-reaction `executedCount` / `succeededCount` / `failedSinceLastChangeCount` / `statsSince`, and `refusedCount` (refused runs are not counted in `executedCount`). Both paths are relative to `reaction-service`'s base path, `/api/reaction`. The regenerated `reaction-service` snapshot now carries both — see the DataReaction reference. Full contract — including why the run history deliberately carries unredacted per-action input/output — is on Reactions & Automation § Execution and failure counts [Build Apps › Reactions & automation].

When an empty result is not what it seems

Outside a view context, a caller without the entries-list grant does not get a `403` from the list and count operations — they deliberately return an **empty list and a zero count**. For a headless client this is the classic silent failure: an application token whose owning application user lacks read access sees `[]` everywhere and no error.

The diagnostic is the single-entry read, which does *not* soften the failure: reading one known entry id without the grant throws `ForbiddenToReadEntryException`. So when a list you expect to be populated comes back empty, fetch one known entry — an explicit `403` tells you it's the credential's permissions, and a clean entry tells you it's your filter. Permission design for application users is covered in Choosing a credential; the meaning of `403` on view-scoped paths is covered in the conventions guide.

Writing Data

This is the write-side guide for an external client: the operations that create and change Entries, the result shapes bulk operations return per row, what validation failures actually look like on the wire, how conflicts and unique-value collisions surface, and the one write gate that has nothing to do with your credential. The shared conventions — envelope, error model, status-code resolution — live in Calling the HTTP API; the full error-code catalogue lives in Errors and retries. Every error body on this page is shown as it arrives: enveloped, `success: false`, with the machine-readable code in `error.code`.

The write surface

The core Entry write operations:

Operation	Path	Notes
Create	`POST /api/v1/entities/{entity}/entries`	Body is a `CreateDataEntryDto`: `value` (required), `specificationVersion` (default `1`), optional `saveAsInvalid`, `systemButtonId`, draft fields. Returns the created Entry.
Update, full value	`PATCH /api/v1/entities/{entity}/entries/{id}`	The body carries the Entry's full `value`. Returns the updated Entry.
Update, JSON Patch	`PATCH /api/v2/entities/{entity}/entries/{id}`	Body is an array of JSON Patch operations, applied server-side (`fast-json-patch`). Prefer it when you touch a few fields — it narrows both the payload and the blast radius.
Batch create	`POST /api/v1/entities/{entity}/entries/batch`	One main Entry plus related Entries per relation, created in a single call.
Archive	`DELETE /api/v1/entities/{entity}/entries/{id}`	Soft delete; reversible with `PATCH …/{id}/restore`.
Permanent delete	`DELETE /api/v2/entities/{entity}/entries/{id}/permanent`	Irreversible.

Batch create returns per-row results, not all-or-nothing: a `mainResult` of `{id, success, error}` and, per relation, an `entries` array of `{id, index, success, error}`. Through a view, every relation you write must be one of the view's configured relations, or the whole call fails with `403` before anything is created.

Bulk operations and per-row results

Bulk update and bulk upgrade select their targets the same way: either an explicit `dataEntriesIds` array or a `selectAllModel` — a "select all matching" descriptor with a `sourceMode` (all entries, a collection, or a relation), an optional `query` criteria tree, `search`, and `appliedQuickFilters`. Supplying neither is a `400`. Before a large run, `POST /api/v1/entities/{entity}/entries/versions/count` previews how the selection spreads across entity schema versions, as `{version, count}` pairs.

Bulk update applies a list of `updateDefinitions` — each `{field, type, value}` where `type` selects the value source: `0` fixed value, `1` expression, `2` the field's default, `3` null (`BulkUpdateFieldValueSourceEnum`). Two variants:

- `PATCH /api/v1/entities/{entity}/bulk-update` runs synchronously and returns the per-row accounting directly:

  ```
  {
    "success": true,
    "result": {
      "successfulUpdates": 8,
      "savedAsInvalid": 1,
      "failures": [
        { "entryId": "3f2c…", "errors": ["ENTRY_VALIDATION_ERROR"] }
      ]
    }
  }
  ```

  A partial success is the normal case, not an edge case: iterate `failures` and handle each row. The `errors` array carries error-code strings (or, for untyped failures, raw messages) — the same codes you would get from the single-row operation, minus the `details`. When you need the details for a failing row, replay that row alone through the single-Entry endpoint.

- `PATCH /api/v2/entities/{entity}/bulk-update` is asynchronous: it enqueues the work and returns a message id inside the envelope (`{"success": true, "result": "<messageId>"}`). Poll `GET /api/v1/job/{messageId}` for the outcome — the pattern is described in Long-running work.

Bulk upgrade rewrites Entries onto a newer entity schema version, with the same selector and the same per-row response shape (`successfulUpgrades` + `failures`). `v1` is deprecated; `v2` runs synchronously; `PATCH /api/v3/entities/{entity}/bulk-upgrade` is the asynchronous variant returning a message id. One accounting trap in the synchronous v1 path: a selected row whose stored `entityVersion` does not match the request's `entryVersion` is **skipped silently** — it appears in neither `successfulUpgrades` nor `failures`, so the numbers may not add up to the rows you selected.

`saveInvalidEntries` (and `saveAsInvalid` on single writes) lets a row that fails value validation be stored anyway, flagged invalid, instead of failing — such rows are counted in `savedAsInvalid`. On view-scoped calls this is only honored when the view's edit configuration allows saving invalid entries; otherwise the flag is quietly forced off.

Validation failures and their details

Two validation layers reject a write, with different codes and different `details` shapes.

Request-body validation — the schema check on the DTO itself — fails with `422` and code `VALIDATION_FAILED`. Its details are per-violation objects of `{path, code, message, info}`; an enum violation, for instance, reports `code: "enum"`, the offending `path`, and `info.allowedValues` (see `InvalidEnumValue`, which mirrors exactly this shape).

Entry-value validation — the platform validating `value` against the Entity's fields — fails with `400`. Each single failure is code `ENTRY_VALIDATION_ERROR` with details `{fieldName, reason}`, where `reason` is a human-readable sentence such as `"Field is required!"`. When *every* failure in a write is a field-validation failure, the writer groups them into one code `ALL_ENTRY_VALIDATION_ERROR` whose details are the **array** of those per-field objects; a mixed failure returns only the first error. A grouped failure arrives like this:

```
// HTTP 400
{
  "success": false,
  "result": {
    "errorCode": "ALL_ENTRY_VALIDATION_ERROR",
    "errorMessage": "Entry validation errors",
    "service": "data-service",
    "details": [
      { "fieldName": "email", "reason": "Field is required!" },
      { "fieldName": "skills", "reason": "the value of skills should be an array" }
    ],
    "retryable": false
  },
  "error": {
    "code": "ALL_ENTRY_VALIDATION_ERROR",
    "message": { "en": "Entry validation errors", "ar": "أخطاء في تحقق من المدخل" },
    "traceId": "…"
  }
}
```

Note where the details are: `error` itself carries only `code`, `message` and `traceId`. The field-level `details` ride on `result`, which on a failure holds the serialized platform exception — populated, not `null`. That placement comes from a response-body factory the code itself flags for refactoring, so branch your logic on `error.code` and the HTTP status, and read `result.details` defensively — errors relayed from another service can carry a different residue.

Since 1.8 — a write that would add, remove or reorder the rows of a repeatable block containing a sensitive field [Build Apps › Protect a field] the caller cannot read is rejected with `400` and code `SENSITIVE_FIELD_WRITE_NOT_ALLOWED`: stored sensitive values re-associate with rows by position only, so a row-count change cannot be reconciled safely.

Repeatable-block rows. A repeatable block's rows are written as one array that replaces the stored array, and each row carries a server-owned `__id` (`Since 1.10`). Echo the ids you read: two rows with the same `__id` in one array fail with `400 ENTRY_VALIDATION_ERROR` (reason `duplicate __id in <field>: <ids>`). Through a **view** that grants only some of the block's columns, the stored values of the other columns are restored for each row (`Since 1.4`), but from release 1.10.1 only for rows whose `__id` matches a stored row; a row sent without an id is a new row and gets nothing restored (`Since 1.10`). The full contract is on Block: saving rows through a view [Build Apps › Block field].

Update conflicts

`Since 1.10` — **Entries now carry an optimistic-concurrency token of their own.** An earlier version of this page said they did not; that is no longer true, and a client written against the old advice is losing updates it could be catching.

`updateIdentifier` is a token the server keeps on every Entry and **rotates on every write**. You do not create it, and you cannot set it: it arrives with the Entry on every read, and the server replaces it each time the Entry changes.

The check is opt-in per request. There is no tenant-wide switch, no global setting and no feature flag to turn on — you opt in on the one write you want guarded, by sending back the token you read with the copy you are changing. Omit it and the write behaves exactly as it always has: last writer wins.

There are exactly three ways to send it, and no header form — no `If-Match`, no ETag:

Operation	Where the token travels
`PATCH /api/v1/entities/{entity}/entries/{id}`	Body property `updateIdentifier`, alongside `value`

Operation	Where the token travels
`PATCH /api/v2/entities/{entity}/entries/{id}`	**Query parameter** `updateIdentifier` — this operation's body is a JSON Patch array and cannot carry it
`PUT /api/v1/views/{id}/entries/{entryId}`	Body property `updateIdentifier`

So the recipe is four steps, and it is the same on all three:

1. **Read the Entry** with whatever call you already make. The token rides on the response — no extra request.
2. **Keep the token** next to the copy you are editing, not next to the form payload you are about to send. It describes the version you read.
3. **Send it back with your change**, in the position the table above gives for your operation.
4. **On a `409`, re-read, re-apply, re-send.** The response tells you someone else wrote the Entry after you read it. Re-read to get the current copy *and* the current token, re-apply your intended change on top of it, and send again. Never retry the same request unchanged — it carries the stale token and will conflict identically every time.

An Entry that has **no stored token yet is never rejected** on that account: the comparison is skipped, the token is stamped on that write, and the Entry is guarded from its next write onward. There is no migration to run and nothing for your client to detect first.

> ⚠ **Two gaps in the coverage — known, and not yet closed**
>
> These are gaps, not design decisions, and neither is something an external client can work around:
>
> - The shipped web application does **not** send the token on the `v2` JSON-Patch inline-edit path, even though the query parameter exists on that route. Inline edits made in the platform's own UI are therefore not guarded this way.
> - Three internal write paths inside `data-service` reach the Entry collection through the raw MongoDB driver rather than through the repository that stamps the token, so those writes do **not** rotate it. An Entry changed only by one of them keeps a token that no longer describes its contents.
>
> Plan on the guard covering the three operations above, and not more.

Definition objects have worked this way for longer. Entities, Views, Queries, relations and collections all carry an `updateIdentifier` that you read with the object and must send back unchanged; if it no longer matches what is stored — someone else saved in between — the write fails with `409` and code `UPDATE_CONFLICT` (`UpdateIdentifierConflictException`). Entry conflicts return the identical body:

```
// HTTP 409
{
  "success": false,
  "result": {
    "errorCode": "UPDATE_CONFLICT",
    "errorMessage": "Update conflict updateIdentifier value not match with the one in the database",
    "service": "data-service",
    "retryable": false
  },
  "error": {
    "code": "UPDATE_CONFLICT",
    "message": { "en": "UPDATE_CONFLICT", "ar": "" },
    "traceId": "…"
  }
}
```

The recovery is always the same: re-read the object, reapply your change on the fresh copy, and resubmit with the new `updateIdentifier`. Do not retry the stale request — it will fail identically every time. A definition write can also fail with a locked-object error when another user holds an edit lock on the object; that one clears when the lock does.

A client that also holds a socket connection can hear about the change before it tries to save: the `entries/value-updated` event now carries the Entry's current `updateIdentifier`, so a page can compare it against the token it holds and warn the user rather than waiting for the `409`. That is a live, best-effort notice and not a substitute for the check — see Real-Time.

Unique-value collisions

When a write violates a unique field, the storage layer surfaces a MongoDB duplicate-key error (code `11000`). The data plane normalises it — in the writer's create, update and upgrade paths, with a final backstop in the shared reject provider — into a `400` with code `UNIQUE_VALUE` and a `details.field` describing the collision:

```
// HTTP 400
{
  "success": false,
  "result": {
    "errorCode": "UNIQUE_VALUE",
    "errorMessage": "email is unique",
    "service": "",
    "details": {
      "field": {
        "path": "email",
        "originalPath": "value.email",
        "value": "jane@example.com"
      }
    },
    "retryable": false
  },
  "error": {
    "code": "UNIQUE_VALUE",
    "message": { "en": "UNIQUE_VALUE", "ar": "" },
    "traceId": "…"
  }
}
```

`path` is the field as you know it — the storage prefix is stripped from the index key, so `value.email` becomes `email` — while `originalPath` keeps the raw key and `value` is the offending value (`UniqueValueException`). Treat

`UNIQUE_VALUE` as terminal for that payload: retrying without changing the value fails identically. In bulk operations the same collision appears as a per-row failure instead of failing the batch.

The human-verification gate on anonymous writes

Since 1.8 — Entry create and both update operations assert human verification before writing. The gate is narrower than it sounds, and for most headless clients it never fires — all three conditions must hold:

- the request is **view-scoped**, targeting a Data View whose anonymous configuration has `humanVerification.enabled: true`, **and**
- the caller is **anonymous** — no authenticated user on the request.

A signed-in user, an application token, or any OAuth-authenticated client is never challenged. The gate exists for the remaining case: driving a **public anonymous form** programmatically. There the client must solve an Altcha proof-of-work challenge — fetch it from `GET /api/human/challenge`, compute the solution, and send it in the `x-altcha-token` header with the write. The gateway verifies the solution and injects `x-human-verification: true` upstream, stripping any client-supplied copy of that header first so it cannot be spoofed. An anonymous write without a valid solution fails with `403` and the message `Human verification required to perform this action`. Configuring the feature is an administrator concern — see Human Verification [Administration › Human verification (bot protection)].

Write refusals added in 1.10 `Since 1.10`

Five refusals a write client can newly meet on the 1.10 line. Each is a stable code — none is worth retrying unchanged except the `503`s:

Status	`error.code`	When	What to do
`400`	`MISSING_REQUIRED_PARAMETER`	An entry or draft create (`POST /api/v1/entities/{entity}/entries`, `…/entries/draft`, and the two `/api/v1/views/{id}/entries` forms) sent **with no body at all**. The create body is now required; a request with a body is validated as before.	Send the `CreateDataEntryDto`.
`403`	`ENTRY_CAP_EXCEEDED`	Creating an entry — single, batch, draft or through a proxy — when the tenant's plan limit is reached and no overage is granted. Updates, archives and reads are never refused. `details`: `{cap, hardLimit, current}`.	Stop creating; the tenant's administrator must raise the plan or archive entries.
`400`	`FILE_EXPIRED`	The save references an uploaded temporary file that expired before the save promoted it. `details` lists `{fileId, fieldName?, path?, reason}` per file.	Upload the file again and re-save with the new id.
`503`	`STORAGE_SERVICE_UNAVAILABLE`	Storage could not be reached and the save adds a file that is not already on the entry. A save that adds no new file goes ahead.	Retry with backoff.
`503`	`TENANT_IN_MAINTENANCE`	Multi-tenant deployments: the tenant's data is being restored, or the tenant is being moved. Comes with `Retry-After`.	Wait `Retry-After` seconds, then retry.

Executing a workflow action that is hidden on the entry is now `422 WORKFLOW_ACTION_EXECUTION_NOT_ALLOWED`, and taking ownership of a step that already has an owner is `422 CURRENT_STEP_ALREADY_OWNED` — both used to be `500`. The complete list is under Typed refusals and Plan limits.

Working with Files

Files are the one part of the data plane that does not speak the platform's usual dialect. They are owned by `storage-service`, a plain Express application — no LoopBack conventions, no generated OpenAPI document, and a response envelope that differs from the rest of the API in ways your error handling must know about. This page covers what an external client actually does with files: upload them, attach them to entries, download them, and understand their lifecycle.

The service's internals — storage providers, upload validation, the metadata model, temp-file routing — are owned by File Storage & Attachments. This page stays on the client's side of the wire and links there for depth.

The storage surface

Two operations are what a client builds on. The router deliberately disables everything else of the classic file API — the list, get-by-id, delete and update routes are commented out because they were unused and carried no authorization:

Method	Path	Purpose
`POST`	`/api/storage/v2/files`	Upload one or more files
`GET`	`/api/storage/v2/files/download/{id}`	Download a file's bytes by id

Both go through the gateway with your normal credential — the same `x-app-token` or bearer token as any other call. You never handle the service's internal identity headers: the gateway strips any `x-shared-key` a client sends and injects its own (`request-transformer` in `api-gateway/config/kong.yml`), and requests relayed by the gateway are classified as *external*, so the full upload validation applies to you.

The service has other routes, and none of them is client surface. The signed-link routes (below) serve browsers. `GET /api/storage/v2/exports/{id}/content` `Since 1.10` downloads a tenant export only through a single-use link the platform mints — you never construct one. The `internal routes are not routed` by the gateway, and admit only named platform services. Details in File Storage & Attachments.

Because there is no OpenAPI document for this service, no generated client covers it (see Generating a client). Integrate these two endpoints by hand; the surface is small enough that this costs a few lines.

Uploading a file

Send a multipart request whose form field is named `file`. The route accepts multiple parts under that one name (`multer.array('file')`), so one request can carry several files:

```
curl -s -X POST "$BASE/api/storage/v2/files" \
  -H "x-app-token: $TOKEN" \
  -F "file=@invoice.pdf"
```

A success is `201` with the envelope, and `result` is always an **array** — one descriptor per uploaded part, even when you sent a single file:

```
{
  "success": true,
  "result": [
    { "id": "5f0e9b2a-…-uuid", "name": "invoice.pdf", "mimeType": "application/pdf", "size": 20481 }
  ]
}
```

The `id` is the only handle you will ever get. Store it: it is what an entry field references and what the download endpoint takes. Storage internals (where the bytes physically live) are never exposed.

Two query parameters matter to a client:

- `temp=true` stores the file as **temporary** — it is auto-deleted after one day (a TTL index on `createdAt`) unless a save promotes it first. `Since 1.10` this is the recommended way to upload a file you are about to attach to an entry (see Temp upload, then save); it is also right for staging artifacts such as a package ZIP you are about to import.
- `ownerType` and `ownerId`, supplied **together**, record which platform object the file belongs to and nest the stored object under that owner. Use them when the file is destined for an entry attachment.

Uploads from outside are validated: an extension allowlist, magic-byte content verification for binary types, and a PDF embedded-JavaScript scan. A rejection is a `400` with `error.code` of `INVALID_FILE_TYPE`, `INVALID_FILE_CONTENT` or `PDF_JAVASCRIPT_NOT_ALLOWED`. The full rule set, including which extensions are checked and which rely on the allowlist alone, lives in File Storage & Attachments. Note that the service configures **no maximum size per file** of its own; any per-request cap comes from proxies in front of the deployment. What it does enforce, `Since 1.10`, is the tenant's **storage limit**: an upload that would take the tenant past it is refused with `403 STORAGE_CAP_EXCEEDED` (see Errors when saving files).

Attaching a file to an entry

An attachment field on an entity is a field whose data type is `blob` (`isBlobField`). Its value is not bytes — it is the descriptor the upload returned. The canonical shape is the shared `FileModel`:

```
{
  "id": "5f0e9b2a-…",        // required — the storage handle
  "name": "invoice.pdf",
  "mimeType": "application/pdf",
  "size": 20481,
  "thumbnail": { /* optional, same shape */ }
}
```

So the flow from an external client is: upload first, then write. Upload the file (with `ownerType`/`ownerId` pointing at the owning object when you know it), take `result[0]` from the response, and set it — or an array of such descriptors, for multi-file fields — as the field's value in an ordinary entry create or update. The write mechanics are owned by Writing data.

On the read side, the entry-details response can include an `attachments` block: the server walks every `blob` field of the entity, collects the descriptors from the entry's values (including inside repeatable blocks), and groups them by field (`getEntryAttachments`). Whether the block is present depends on the view's *show information* configuration, so do not treat its absence as "no files" — the field values themselves are the source of truth.

Temp upload, then save `Since 1.10`

`Since 1.10` storage keeps a ledger of which saved record references which file. When an entry save names a file id, data-service registers it before writing — and if the id belongs to a **temp** upload, storage **promotes** it to permanent

storage under the same id. So the robust pattern for a client is the one the web portal now uses:

1. Upload with `?temp=true`.
2. Save the entry with the returned descriptor within the day.
3. The save promotes the file; from then on it is permanent and referenced by that entry.

What this changes for an API client:

- **Every new file id on a save must exist.** Descriptors in `blob` fields are checked — objects with an `id`, `name`, `mimeType` and numeric `size`. A descriptor whose id storage cannot find — a temp upload older than a day, or an id that never existed — fails the save with `400 FILE_EXPIRED`, naming the field. Ids the stored entry already held are not re-checked, so re-saving an old record never fails on them.
- **A save can wait on storage.** If storage cannot be reached while a save adds a new file, the save fails with a retryable `503 STORAGE_SERVICE_UNAVAILABLE` instead of writing a reference nothing tracks.
- **Permanent uploads still work** exactly as before. They are simply not tracked by the ledger, so they are never cleaned up automatically.

Downloading

Downloads are the canonical example of a response that **bypasses the envelope structurally**: the handler writes to the HTTP response itself — content type, disposition, then the bytes — so there is no `{success, result}` wrapper and nothing to JSON-parse. A client that pipes every response through a JSON parser will fail exactly here.

```
curl -s "$BASE/api/storage/v2/files/download/5f0e9b2a-…" \
  -H "x-app-token: $TOKEN" \
  -o invoice.pdf
```

What comes back (`downloadFile`):

- `Content-Type` is derived from the stored file **name's extension** (a `mime-types` lookup, falling back to `application/octet-stream`) — not from the `mimeType` recorded at upload.
- `Content-Disposition: attachment` with the original file name is always set. Nothing is ever served as inline, executable content.
- The body is the raw bytes.

The same endpoint serves temporary files — the handler checks the permanent collection first, then the temp one — so a `temp=true` upload is downloaded by plain id like any other. A missing id is a `404` — and so, `Since 1.6`, is an id whose metadata exists but whose stored bytes are gone (it used to be an unhandled `500`).

Signed links for browsers `Since 1.6`**.** If a browser needs to show a file with a plain `<img src>` (no bearer token), ask storage for a short-lived signed URL with `POST /api/storage/v2/files/links` and use the returned `url`. The link is a **bearer capability**: anyone holding the link can fetch the file until it expires (one to two hours by default). Keep such links out of logs and shareable URLs. Details: Signed, cacheable content links.

One asymmetry to internalize: **only the success path is binary.** A failed download — unknown id, backend trouble — is still a JSON error body with the appropriate status. Branch on the status code (or `Content-Type`) before deciding how to read the body.

Lifecycle: what exists, for how long

The public API creates and reads files, and never destroys them. Clean-up, where it happens at all, is the platform's job:

- **Permanent uploads persist indefinitely.** There is no delete endpoint on the public surface, so a file uploaded straight to permanent storage outlives any entry that referenced it; removing an attachment from an entry edits the entry's value, not the stored file. `lastAccessedAt` is updated on each download of a permanent file, giving operators a signal for offline cleanup — but cleanup is an operational task, not an API call.
- **Promoted files can be swept** `Since 1.10.` A file that a save promoted from temp is tracked. If every record that referenced it releases it (for example the record is deleted for good) and a grace period passes, a background sweeper deletes it — but only after each former owner confirms it no longer lists the file. An entry keeps every file it ever held in its history, so its files are not swept while the entry exists. The platform's chart ships the sweeper switched on but in a log-only dry run, until an operator lets it delete; see Orphan file sweeper.
- **Temporary uploads self-destruct after one day** via the TTL index. This is also how the platform treats its own transient artifacts: an exported solution package, for example, is uploaded as a temp file, so the `fileId` an export job hands you is valid for roughly a day — download it promptly (see Long-running work). A published package version's ZIP is stored permanently instead, and deleting the version or an export removes its record only: the ZIP stays in storage.
- `createdBy` **is recorded from your authenticated identity** on permanent uploads, and `ownerType` / `ownerId` metadata ties a file to its owning object when you supplied them.

Errors from the storage service

Storage errors share the platform's `{success, result, error}` outline but **not** its exact shape. The Express services assemble the envelope by hand, and the divergence is deliberate, preserved for wire compatibility:

```
// storage-service failure body (middleware.error.ts)
{
  "success": false,
  "result": {
    "code": "NOT_FOUND",
    "message": "File is not available!",
    "serviceName": "storage",
    "traceId": "…",
    "details": { "error": "File not found." }
  },
  "error": {
    "code": "NOT_FOUND",
    "message": "File is not available!"   // a plain string — NOT {en, ar}
  }
}
```

Three differences from the LoopBack error body your handler already knows from Calling the HTTP API:

1. `error.message` is a **plain single-language string**, not a localized `{en, ar}` object.
2. `error` carries **no** `traceId` — the trace id lives in `result.traceId` here.
3. `result` is a populated diagnostic object on failure, not `null`.

Also, a `401` from the service's own authentication check is emitted before the error middleware runs and is **not enveloped at all** — the raw error object is sent as is. Write the storage error path as its own small branch: match on HTTP status and `error.code`, and read the trace id from wherever it is. The retry policy itself is the same as everywhere else — see Errors and retries.

Errors when saving files `Since 1.10`

Three codes concern files. Note which service answers, because that decides the body shape (storage's own, described above, or the platform's usual one from Errors and retries):

Status / code	Raised by	Meaning	What to do
`403 STORAGE_CAP_EXCEEDED`	storage-service, on upload	The tenant reached its storage limit (its plan's `maxStorageGb`). Storage error shape; `result.details` carries `capBytes` and `usedBytes`. An upload that crosses the limit mid-stream is removed and refused the same way.	Not retryable — the limit has to be raised for the tenant before uploads succeed again.
`400 FILE_EXPIRED`	data-service (and reaction-service), on save	A new file id in the saved value cannot be found. Platform error shape with a localized message; `result.details` lists `{fileId, fieldName}` (or `{fileId, path}`) per missing file, each with a localized `reason`.	Upload the file again and re-save with the new descriptor.
`503 STORAGE_SERVICE_UNAVAILABLE`	data-service (and reaction-service), on save	Storage could not be reached while the save added a new file.	Retryable: back off and repeat the same save.

Subscribing to Events

Polling is the wrong tool for "tell me when something changes". The platform's push channel is the **public socket** — the `web-socket-service`, a Socket.IO server that turns domain events into room-scoped pushes. Real-Time (Sockets & RTC) owns that service's full story: the event catalogue, the RabbitMQ bridge behind it, scaling, and the browser integration. This page covers the part that guide does not: connecting from a **non-browser client** — a worker, a daemon, your own backend — where the credential question and the reconciliation discipline look different.

One framing rule before anything else: the channel is **push-only and best-effort**. You subscribe and receive; you never write through it, and you never treat it as a complete log. Every write, and every authoritative read, is an HTTP call.

The channel and the protocol

The endpoint speaks **Socket.IO v4** (`socket.io ^4.8.1`), which is its own protocol on top of WebSocket/HTTP long-polling — a bare WebSocket client cannot connect. From a non-browser process, use a Socket.IO v4-compatible client; official and community implementations exist for JavaScript, Python, Java, Go, .NET and others, and everything on this page is client-library-agnostic: an auth payload at handshake, named events in both directions.

The service is a standalone Express + Socket.IO process, listening on port `3017` by default. It is deployed and exposed **separately from the `/api` gateway path** — the gateway's route table (`api-gateway/config/kong.yml`) carries no socket route — so the socket host is a property of your deployment. Obtain it from whoever operates your environment, alongside your base URL. CORS on the server is `origin: '*'`, which is irrelevant to a non-browser client anyway.

Authenticating the handshake

Authentication happens once, in a handshake middleware, and the credential goes in the Socket.IO `auth` payload under the key `token` (`setupSocketAuthMiddleware`):

```
import { io } from "socket.io-client";

const socket = io("https://<socket-host>", {
  auth: { token: accessToken },
});
```

Here is the trap for a headless client: **this must be an OAuth 2.0 access token, not an API token.** The server validates the credential by posting it — with its own `CLIENT_ID` / `CLIENT_SECRET` — to the identity provider's token-introspection endpoint and requires an `active: true` answer. An API token is a different credential validated by a different endpoint (the gateway's `app-token-auth` plugin); it does not introspect as an active OAuth token, so it cannot open the socket. A service that authenticates its HTTP calls with `x-app-token` therefore needs a second credential for the socket: **a user's** OAuth token, which for a service means the password grant for a dedicated service account. That account must be an ordinary user (`NORMAL_USER`): an `APPLICATION_USER` is held only as an API token, has no sign-in record, and the password grant answers `invalid_grant` for it. A `client_credentials` token is not a substitute. It introspects as active, so the socket opens, but it has no `sub`: `socket.userId` stays unset, and `subscribeToNotifications` answers `error {message: 'User ID is required'}` and disconnects. Choosing a credential covers the trade-offs.

On success the server binds your identity: `socket.userId` is set from the token's `sub`, and that identity — not anything you send later — decides which user room you may join. An invalid or inactive token rejects the connection with `Invalid token`, surfaced to your client as a `connect_error`.

Two lifecycle consequences of "validated once, at handshake":

- **Every reconnection is a new handshake.** A long-lived client whose token has meanwhile expired will fail to reconnect. Refresh the token before the retry — with `socket.io-client`, assign `socket.auth = { token: freshToken }` inside a `reconnect_attempt` (or `connect_error`) handler.

Rooms: subscribe before anything arrives

A freshly connected socket receives nothing. Delivery is scoped to **rooms**, and you opt in by emitting a subscribe event; each subscribe is answered by an acknowledgement event confirming the join (`setupSocketHandlers`):

Room	Join by emitting	Payload	Acknowledgement
`user:{userId}` — your notifications	`subscribeToNotifications`	*(none — identity comes from the handshake)*	`subscribedToNotifications`
`entries/{entityId}_{entryId}` — one entry	`entries/subscribe-to-entry`	`{entityId, entryId}`	`subscribedToEntry`
`job:{messageId}` — one async job	`subscribeToJobExecution`	`{messageId}`	`subscribedToJobExecution`

Matching `unsubscribe` events exist for the entry and job rooms; missing payload fields produce an `error` event instead of a join. **Wait for the acknowledgement before assuming you are in the room** — some join failures are swallowed server-side and produce no `error` at all, a caveat Real-Time documents. What arrives in each room — entry value updates, workflow and comment events, notifications, job completions — is the event catalogue that guide owns; do not hardcode assumptions beyond the event names you actually consume.

Two things arrive **without** a room join, and a headless client should expect both. A few events are broadcast to every socket of the tenant — the icon-set and request-entry notices, and `Since 1.10` `packages/changed` when a package's review or store state moves (a signal to re-read, carrying only the `packageId`). And notifications are **not filtered by space** on the socket: your user room receives notifications raised in every space you belong to. `Since 1.10` the shipped web client hides those from other spaces itself, but that filter is client-side only — filter on the notification entry's `value.space` if your client works in one space (Real-Time).

Following a job you started over HTTP

The subscription most useful to a headless client is the job room, because it replaces a polling loop. The pattern, end to end:

```
import { io } from "socket.io-client";

const socket = io(SOCKET_HOST, { auth: { token: oauthToken } });

socket.on("connect", () => {
  // messageId came back from an async HTTP operation, e.g. a bulk update
  socket.emit("subscribeToJobExecution", { messageId });
});

socket.on("subscribedToJobExecution", ({ messageId }) => {
  console.log("watching job", messageId);
});

socket.on("osos/infrastructure/job-executed", (evt) => {
  // evt: { messageId, userId, messages, result: { success, error?, result? } }
  if (evt.messageId !== messageId) return;
  finish(evt.result);
});
```

The completion event is emitted to **both** the job room and your user room (`sendJobExecutionFinished`), and its payload carries the job's outcome object. Treat it as a wake-up call, not as the record: confirm the terminal state over HTTP with the job-status endpoint, which is the authoritative source — the division of labour is described in Long-running work.

Subscribe **before** or immediately after kicking off the work, not when you get bored of waiting: if the job finishes before your join completes, the room emit has already happened and nothing is replayed.

Delivery guarantees and reconciliation

Design your client around what the transport actually promises, which is little:

- **Best-effort, no replay.** Events are emitted to whoever is in the room at that moment. Entry-room emits are skipped entirely when the room has no subscribers (`sendDataToEntry` returns early), and nothing is queued for you while you are disconnected.
- **Reconnection is a gap.** After any disconnect, assume you missed events: re-subscribe to your rooms (joins do not survive a new connection) and reconcile state with an HTTP read — refetch the entry, re-poll the job status.
- **At-least-once handling.** A job completion can reach you twice — once via the job room, once via your user room. Key your handling on `messageId` and make it idempotent.
- **Ordering is not contractual.** Treat each event as "something changed, verify over HTTP" rather than as a delta to apply blindly.

This is the same posture the platform's own UI takes: the socket makes it fast; HTTP makes it true.

What this channel is not

- **Not a write path.** No client-to-server event mutates data; anything that looks like a command over the socket is ignored by this service. Writes go through the HTTP API.
- **Not a service-to-service channel.** `web-socket-service` is the platform's only socket. Older releases also ran an internal relay between backend services (`socket-service`); it was removed in 1.10 `Removed 1.10` , and backend services now follow jobs over the message bus. Pages written for 1.8 may still describe it.
- **Not a message queue.** If you need durable, replayable integration events, poll the HTTP surface or build on webhooks and integrations — do not try to make the socket into one.

Long-Running Work

Some operations are too heavy to answer inline: bulk writes over thousands of entries, solution-package export and import, a full dependency-graph rebuild, a background file export. For these the platform answers immediately with a **handle**, and your client's job changes from "read the response" to "run the handle to a terminal state". This page covers the four handle families an external client meets, their polling endpoints and terminal states, what can and cannot be cancelled, and the responses that come back as **file downloads rather than JSON**.

The general shape — kick off, poll, terminal state — is introduced in Calling the HTTP API. This page is the practitioner's layer on top: the exact contracts, verified against the services that implement them.

The pattern

Every family follows the same loop, differing only in the handle and the status vocabulary:

1. **Kick off.** The operation returns a handle — a `messageId` or a `jobId` / `sessionId` — instead of a result.
2. **Poll** the family's status endpoint with that handle, at a modest interval with backoff.
3. **Stop on a terminal state**, then read the result from the status payload — which may be a value, a per-row report, or a `fileId` to download.

Two cross-cutting rules:

- **Handles are tenant-checked.** Polling a job that belongs to another space returns `403`; status endpoints verify ownership before answering.
- **Handles expire.** Each family retains status for a limited time (detailed below). Poll promptly and persist results you care about — a handle is not an archive.

For a push-based alternative to step 2, see Progress without polling.

Data-plane jobs: the `messageId` handle

Heavy `data-service` operations dispatch a command onto the message bus and return a fresh UUID — the `messageId` — as the enveloped result. The ones an external client most commonly triggers:

Operation	Route
Bulk update	`PATCH /api/v1/entities/{entity}/bulk-update`
Bulk upgrade	`PATCH /api/v3/entities/{entity}/bulk-upgrade`
Restore entries	`PATCH /api/v1/entities/{entity}/restore/entries`
Background template export	`GET /api/v1/entities/{entity}/entries/export/{templateId}` (background-type templates)
Execute a custom-action button `Since 1.8`	`POST /api/v1/system-buttons/{id}`

Poll the single shared status endpoint:

```
curl -s "$BASE/api/v1/job/$MESSAGE_ID" -H "x-app-token: $TOKEN"
```

```
{
  "success": true,
  "result": {
    "messageId": "1c9a…",
    "status": "InProgress",            // Scheduled | InProgress | Success | Failed
    "message": { "en": "…", "ar": "…" },
    "progress": 40,                     // 0–100
    "result": null                      // populated on completion, e.g. a per-row report
  }
}
```

The contract (`JobStatusResponseDto`):

- `status` walks `Scheduled → InProgress → Success | Failed` (`JobStatusEnum`). `Success` and `Failed` are terminal.
- On completion, `result` carries the operation's report — for a bulk update, the totals-and-failures DTO that Writing data describes; a partial success is a `Success` whose report lists failures, so always read the report.
- **Status lives in a cache with a 24-hour expiry** (`JOBS_CACHE_EXPIRY`). An unknown or expired handle answers `404` with `error.code: JOB_STATUS_NOT_EXIST` — indistinguishable from a wrong id, so persist anything you need before it ages out.
- **There is no cancellation** for this family: no cancel endpoint exists on the job controller. Once dispatched, the command runs to its own conclusion.

Packaging jobs: BullMQ handles and cancellation

`packaging-service` runs its export and import work on BullMQ queues with dedicated workers, and its status vocabulary is BullMQ's, normalized at the API boundary.

Export. `POST /api/packaging/v1/export` enqueues the job and returns `{jobId}` ; poll `GET /api/packaging/v1/export/{jobId}/status` :

```
{
  "success": true,
  "result": {
    "status": "completed",   // waiting | waiting-children | active | prioritized |
                             // delayed | completed | failed | unknown
    "result": {              // present when completed
      "fileId": "…",
      "name": "package-….zip",
      "resolvedResources": [ /* what actually went into the package */ ]
    },
    "failedReason": { "message": { "en": "…", "ar": "…" }, "code": "…" }  // when failed
  }
}
```

`completed` and `failed` are terminal. `unknown` means the state did not map to the modeled set — **keep polling**, do not fail (`normalizeExportJobStatus`). On completion, download the package promptly: the ZIP is uploaded to storage as a **temporary file** and self-destructs after about a day — see Working with files. `Since 1.10` an export also leaves an **export record** you can list and read later (`GET /api/packaging/v1/exports` , `GET /api/packaging/v1/exports/{id}`); deleting one (`DELETE …/exports/{id}`) removes the record only, not the ZIP in storage (Export records).

Import. Import is session-based: upload the package ZIP to storage, validate it into a session (`POST /api/packaging/v1/import/validate` with the `fileId`), then execute — `POST /api/packaging/v1/import/{sessionId}/execute` returns `{jobId, sessionId}` and the poll target is the **session**: `GET`

`/api/packaging/v1/import/{sessionId}/status`, answering `{status, importResult, errorMessage, resourceMappings}`. A failed or partially failed session can be re-run with `POST …/{sessionId}/retry`. The session model and configuration surface are packaging-cluster territory — see Packaging.

Cancellation exists here, with honest semantics (`POST /api/packaging/v1/import/{sessionId}/cancel`):

- Only a session that is currently `importing` or `validated` can be cancelled; anything else is a `400`.
- The session is atomically transitioned to `failed` with a `Cancelled:` marker; a concurrent state change surfaces as `409`.
- The response tells you what actually happened to the underlying job: `removed_from_queue` (it never started), `cancel_requested` (it was running — a running worker cannot be hard-killed; it aborts at its next periodic check of the session status), or `not_found` (already finished or cleaned up).

So treat cancellation as *requested*, not *instantaneous*: after a `cancel_requested`, keep polling the session until it settles.

The dependency rebuild

A full dependency-graph rebuild is the family's simplest member, with one distinctive behaviour — **single concurrency**:

- `POST /api/dependency/v1/rebuild` answers `202` with `{jobId, status: "queued"}`.
- If a rebuild is already in flight, you get `409` — and the error message names the in-flight `jobId`. The correct reaction is to **poll that job**, not to retry the kick-off.
- `GET /api/dependency/v1/rebuild/status/{jobId}` answers `{jobId, state, result?, failedReason?}`; an unknown or aged-out id is `404`.

Tenant backups, restores and exports `Since 1.10`

In a multi-tenant deployment a tenant's own system administrator can back up, restore and export the tenant's data through `data-service`. It is an **API only** — there is no web screen for it in 1.10 — and in a single-tenant deployment every route answers `501 BACKUP_NOT_SUPPORTED_IN_SINGLE_MODE`. The operator side (the engine, retention, the per-day quotas) is described on the setup section's backup and restore page; the full HTTP contract and refusal codes are in Tenant backups, restores and exports.

Kick off	Answers	Poll
`POST /api/v1/admin/backups` (no body)	`202` with `backupId`	`GET /api/v1/admin/backups/{backupId}`
`POST /api/v1/admin/restores` (`{backupId}` or `{pointInTime}`, plus `confirmationToken`)	`202` with `restoreId`	`GET /api/v1/admin/restores/{restoreId}`
`POST /api/v1/admin/exports`	`202` with `exportId`	`GET /api/v1/admin/exports/{exportId}`

What is different about this family:

- **A restore is two steps.** First `POST /api/v1/admin/restores/preview` with the same target: it answers what the restore would do, with warnings, and a single-use `confirmationToken` that expires. Then start the restore with that token. A bad, expired or already used token is `400 CONFIRMATION_INVALID` — ask for a new preview.

- **A restore puts the tenant under maintenance.** From the start until it finishes, every request for the tenant answers `503 TENANT_IN_MAINTENANCE` with `Retry-After` — **except** `GET` on `/v1/admin/restores…`, `/v1/admin/backups…` and `/v1/admin/exports…`. Poll the restore with those; do not expect anything else to work, and do not start a second one (`409 RESTORE_IN_PROGRESS`).
- **An export is downloaded through a single-use link.** When the export is done, `POST /api/v1/admin/exports/{exportId}/link` answers `{url, expiresAt}`: a link on the tenant's own host, usable once, for up to 10 minutes. At most five links per export (`linksLeft` on the export; then `409 EXPORT_LINKS_EXHAUSTED`). The archive holds the tenant's databases **without user accounts**.
- **Quotas, not load.** Starts count against per-day quotas (`backupsPerDay`, `restoresPerDay`, `dumpDownloadsPerDay`), reported with their reset time by `GET /api/v1/admin/backups`. Over a quota is `429` with a `Retry-After` that runs to the reset.
- **No cancellation** endpoint exists for this family.

Results that are downloads, not JSON

Two different shapes of "the result is a file" exist, and a client must handle both:

Inline downloads. Some operations write the file straight into the HTTP response — the template export endpoint is the canonical case. When the template is configured for inline export, `GET /api/v1/entities/{entity}/entries/export/{templateId}` sets the file's content type and an attachment disposition and streams the bytes. These responses **bypass the response envelope structurally** — there is no `{success, result}` wrapper and nothing to JSON-parse. The same endpoint returns an enveloped JSON `messageId` when the template is background-typed, and the request looks identical either way — so **branch on the response's `Content-Type` header**, not on what you expected: `application/json` means an envelope; anything else is the file itself.

Handle-then-download. Background work that produces a file hands you a `fileId` instead, and the bytes come from the storage download endpoint as a second request. The packaging export above is one case. The background template export is another, with one indirection: when the file is ready, a notification entry is created for you whose value carries the file descriptor (`value.attachment`, with `reference` pointing at the file id), and a `notification` event is pushed to your user room on the socket. Either way, the final hop is `GET /api/storage/v2/files/download/{fileId}` — a raw-bytes response with the caveats Working with files spells out, including the one-day lifetime of temporary artifacts.

Progress without polling

Every `messageId` job doubles as a socket subscription: join the `job:{messageId}` room and receive the completion event instead of discovering it on the next poll, plus progress updates where the operation reports them. The handshake, the room mechanics and the credential caveat (the socket takes an OAuth token, not an API token) are covered in Subscribing to events.

Use the two channels in their natural division: the socket to *wake up*, the status endpoint to *conclude*. The HTTP status is the authoritative record; a socket event is a hint that it is worth reading now.

Where jobs actually run

Knowing the machinery helps interpret what you observe:

- **Data-plane commands** travel the message bus (RabbitMQ) and execute on BullMQ workers inside `data-service` (bulk update, upgrade, restore each split into parent, chunk jobs). Completion is written to the Redis-

backed job status you poll, then published on the bus as the job-executed event, which the public socket forwards to the browser. (Until 1.10 it travelled through an internal socket relay, since removed `Removed 1.10`.)

- **Packaging and dependency work** runs on each service's own BullMQ queues and workers.
- **Tenant backups, restores and exports** `Since 1.10` are run by the multi-tenant cell's own controller service; `data-service` only resolves the calling tenant and relays the request and the status.
- **Scheduled and recurring work** belongs to `scheduler-service`, an Agenda-based internal service driven entirely by bus commands — it exposes no client-facing HTTP routes at all (its only HTTP route is a health check). A client cannot schedule work through it directly; recurring behaviour is configured through platform features such as reactions and scheduled reports, which then use it internally.

The practical consequence of all of this: job execution is asynchronous, distributed and at-least-once by design. Build your client to poll with backoff, to treat handles as perishable, and to make acting on a result idempotent.

Errors and Retries

Every failure your client will ever see from OSOS is either a rejection at the gateway, a typed exception from one of the platform services, or an error from one of the Express edge services. This page is the reference for all three: the shape each one arrives in, the complete `error.code` catalogue grouped by the exception class that raises it, a written retry policy, and how to use `traceId` when you escalate. The mechanics the whole API shares — the response envelope, routing, versioning, pagination — are owned by Calling the HTTP API and are not restated here; the Quickstart walks through handling one error end to end.

Codes and class names below are reproduced exactly as the platform emits them, including their misspellings — match on them verbatim.

Where errors originate

A request crosses up to three error-producing layers, and each has its own body shape. The first thing an error handler should do is work out which layer it is talking to.

The gateway. The gateway's plugins authenticate, resolve the tenant and check access before any service is reached, and they reject with a shared helper whose shape looks like the platform envelope but differs in two tells: `error.message` is a plain string, not a localized object, and `error.serviceName` is always `"API Gateway"`:

```
{
  "success": false,
  "error": {
    "code": "SPACE_FORBIDDEN",
    "message": "User does not have access to the specified space",
    "serviceName": "API Gateway",
    "traceId": "5b2f…",
    "details": null
  }
}
```

Gateway *rate-limit* rejections are the one exception: they come from Kong's stock `rate-limiting` plugin (`api-gateway/config/kong.yml`) as a `429` with Kong's own small `message` body, not this shape.

The platform services. Every LoopBack service — the entire data plane — funnels errors through one reject provider and one body factory. The enveloped failure body:

```
{
  "success": false,
  "result": {
    "errorCode": "UNIQUE_VALUE",
    "errorMessage": "email is unique",
    "service": "",
    "details": {
      "field": {
        "path": "email",
        "originalPath": "value.email",
        "value": "a@example.com"
      }
    },
    "retryable": false
  },
  "error": {
    "code": "UNIQUE_VALUE",
    "message": { "en": "…", "ar": "…" },
    "traceId": "5b2f…"
  }
}
```

Four facts about this shape that the code guarantees and a client must know:

- **`error` carries exactly `code`, `message` and `traceId`.** Branch on `code`; it resolves as `errorCode ?? code ?? name`, so an error that was never given a platform code surfaces its JavaScript class name instead — `UnauthorizedError`, `ForbiddenError`, `NotFoundError`, and on a genuine server bug even `TypeError`.
- **`error.message` is a per-service dictionary lookup.** The data service's dictionary localizes 262 codes; for any code outside it the factory returns `{ "en": "<the code itself>", "ar": "" }`. Never rely on `message.ar` being non-empty, and never parse `message` at all.
- **`result` is not `null` on failure.** The factory leaves a remnant of the internal exception there: for typed exceptions the serialized object above — which is where the structured `details` payload actually lives, e.g. the failing `field` of a unique-value violation or the per-violation array of a body-validation failure — and for untyped errors just the message string. The factory carries an unresolved `@fixme` on this method; treat `result`-on-failure as observed behaviour, useful for diagnostics, not as a contract to build on.
- **Relayed errors pass through verbatim.** When a service forwards a failure from another service, the upstream `error` object is reused as-is and may carry extra fields (`details`, `serviceName`). Tolerate unknown properties. `Since 1.8` the **status** follows it too: a 4xx from the other service is answered with that 4xx, not flattened to `500` (`relayedClientError`). A non-JSON reply from the other service still reads as a transport failure.

The Express edge services. `users-service` (authentication) and `storage-service` are not LoopBack and do not envelope. Token and OIDC endpoints fail with standard OAuth 2.0 error objects (`error`, `error_description`). The step-up unmask endpoints for sensitive fields *(Since 1.8)* return a flat `{ "error": "<CODE>" }` body — their codes are catalogued below.

How the HTTP status is resolved

The reject provider resolves the status in a fixed order — `statusCode`, then `status`, then the typed exception's `requestCode`, then `404` if the code is `ENTITY_NOT_FOUND`, else `500` — as documented in Calling the HTTP API. What that page does not spell out is where `requestCode` comes from and what happens when it is missing, and that is the single most important thing on this page for retry logic.

Each service defines a small set of abstract exception families, and only the family sets the status (note the file name):

Family	`requestCode`	Retry?
`ValidationException`	400	Never — the request is wrong; fix it.
`UnauthorizedException`	401	Refresh the credential, retry once.
`ForbiddenException`	403	Never — a permission or view-access decision.
`NotFoundException`	404	Never — re-check the identifier.
`ConflictException`	409	Re-read state, then re-attempt deliberately.
`UnprocessableEntityException`	422	Never — the payload is semantically invalid.
bare `DataException`	**500** (none set)	**Never — it is deterministic, not transient.**

The last row is the trap. A concrete exception class that extends `DataException` directly — without passing through a family that sets `requestCode` — surfaces as HTTP `500`, and the data service alone has **101** such classes for purely semantic failures: `ACTION_NOT_ALLOWED`, most workflow-configuration errors, duplicate-name violations. A `500` from OSOS therefore does **not** imply a transient server fault. Check `error.code` against the catalogue before deciding to retry: a `500` carrying a catalogued code will fail identically on every attempt. Several of these refusals moved to a proper family — executing a hidden workflow action is `422` (`Since 1.8`), re-archiving an archived entity is `422` and a view saved with no supported edit operation is `400` (`Since 1.10`) — and new refusals were added in the right family from the start; the rows are marked in the table below and summarised in Calling the HTTP API. Code that special-cased one of these codes under `500` must now look for it under its new status.

Two normalizations happen inside the reject provider before serialization:

- A MongoDB duplicate-key error (driver code `11000`) becomes `UniqueValueException`: HTTP `400`, code `UNIQUE_VALUE`, with the offending field path and value under `result.details.field`.
- An authentication failure inside the pipeline (`USER_PROFILE_NOT_FOUND`) becomes a plain `401` whose `error.code` is `UnauthorizedError`.

One more thing the base class reveals: `DataException` declares a `retryable` field, and it is on the wire in the `result` remnant. Only a handful of `503` refusals set it to `true` — storage unreachable (`STORAGE_SERVICE_UNAVAILABLE`), the dependency check of an entity purge, the functions service's own unavailability and, `Since 1.10`, the backup API's `503`s; everything else leaves it `false`. Do not branch on it; derive retryability from status and code as this page describes.

The error-code catalogue

The catalogue below groups every concrete typed exception the services and gateway define under the family that determines its status. It is the complete *typed* error surface. Three honest caveats:

- A class being defined does not prove every operation can raise it; some fire only in background jobs, where the failure surfaces through a job-status poll rather than an HTTP error (see Long-running work).
- The same code string can carry a *different status in a different service* — `ENTITY_FIELD_NOT_EXIST` is a `404` from the data service but a `500` from the automation service, because the two services' classes chose different families. Branch on `code`; treat the status as the coarse signal.
- Three constants are exported under one name but emit a *different string on the wire*. The tables list wire values; grep the backend for the constant name if you need the source:

Constant (source)	Wire value (`error.code`)
`FILTER_FIRST_FIELD_CAN_NOT_BE_OF_THIS_TYPE`	`FILTER_FIRST_FIELD_CAN_NOT_BE_OF_THIS_TYPE_EXISTS`

Constant (source)	Wire value (`error.code`)
`ROLE_MUST_BE_JOB_POSITION`	`ROLE_MUST_BE_JOP_POSITION`
`INVALID_MANAGER_STEP_OWNER_SELECTION_CRITERIA`	`INVALID__MANAGER_STEP_OWNER_SELECTION_CRITERIA` (double underscore)

Gateway errors

Raised by the gateway's plugins before any service is reached, in the gateway shape shown above. The `500` s in this table are infrastructure failures (the gateway could not reach its own dependencies) and are the *only* `500` s in this catalogue worth an automatic retry.

Status	`error.code`	Plugin	When
400	`DOMAIN_ERROR`	`domain-resolver`	The request `Host` does not resolve to a tenant.
404	`SPACE_ERROR`	`space-resolver`	No space matches the host (`Since 1.10` ; it was `400`).
404	`APP_ERROR`	`app-resolver`	In a multi-tenant deployment: the host maps to no tenant (`Since 1.10`).
400	`SPACE_ID_MISSING`	`user-space-checker`	The `x-space-id` header is absent where required.
400	`MISSING_DOMAIN_ID`	`osos-access`	Domain id missing from the resolved context.
400	`MISSING_USER_ID`	`user-delegation`	Delegation requested without a user id.
400	`INVALID_APP_TOKEN_API_ERROR`	`app-token-auth`	The `x-app-token` API token is unknown, expired, deactivated or archived. The plugin passes on the validation endpoint's `400` . Do not retry.
401	`AUTHENTICATION_ERROR`	`oauth2-introspection`	The bearer token failed introspection.
403	`INVALID_ACCESS`	`osos-access`	The access object forbids this route/view for this caller.
403	`SPACE_FORBIDDEN`	`user-space-checker`	The user is not a member of the requested space.
403	`DELEGATION_EXPIRED`	`user-delegation`	The delegation window has lapsed.
429	*(Kong body)*	`rate-limiting`	Over 100 requests/second or 10,000/hour to one service.
429	`RATE_LIMITED`	`tenant-rate-limit`	In a multi-tenant deployment: the tenant exceeded its per-second or per-minute budget; honour `Retry-After` (`Since 1.10`).
503	`RATE_LIMIT_UNAVAILABLE`	`tenant-rate-limit`	The limiter's store failed and the operator chose to refuse; `Retry-After: 1` (`Since 1.10`).
500	`REDIS_ERROR` , `REDIS_CONNECTION_ERROR`	several	The gateway could not reach or read its Redis.
500	`API_REQUEST_ERROR` , `API_DECODE_ERROR` , `API_LOGIC_ERROR`	`user-space-checker` , `osos-access`	An internal lookup call failed or returned garbage.
500	`AUTHENTICATION_ERROR`	`oauth2-introspection`	Introspection *infrastructure* failed (distinct from the 401).
500	`APP_TOKEN_VALIDATION_REQUEST_FAILED`	`app-token-auth`	The token-validation call could not complete.

Status	error.code	Plugin	When
500	USER_PROFILE_FETCH_ERROR, ACCESS_OBJECT_DECODE_ERROR	osos-access	Profile or access-object retrieval failed.
500	DELEGATION_API_REQUEST_FAILED, DELEGATION_API_DECODE_ERROR, DELEGATION_API_LOGIC_ERROR, DELEGATION_DATE_PARSE_ERROR	user-delegation	The delegation lookup failed.
500	INTERCEPTOR_DECODE_ERROR	interceptor	The interceptor could not decode an internal response.
500	KNATIVE_REQUEST_FAILED	knative-functions	A function invocation could not be dispatched.

Framework errors (any LoopBack service)

Raised by the request pipeline itself, in any LoopBack service, before or around your operation. The request-validation codes come from `@loopback/rest`; the rest surface a class name because no platform code was attached.

Status	error.code	When
400	INVALID_PARAMETER_VALUE	A query/path parameter failed coercion or validation — e.g. malformed `filter` JSON.
400	MISSING_REQUIRED_PARAMETER	A required parameter was omitted — `Since 1.10` also an entry or draft create sent with no body at all, and `Since dev` a view save (`PATCH /api/v1/entities/{entity}/views/{id}`) sent with no body.
400	SPACE_CONTEXT_MISSING	A service was reached with no usable `x-space-id` and no space scope (`Since 1.10` ; see Tenancy and identity). Not retryable as is.
400	SPACE_NOT_RESOLVED	The `x-space-id` header conflicts with the space already in scope (`Since 1.10`).
400	UNIQUE_VALUE	The normalized Mongo duplicate-key violation; the failing field is under `result.details.field` .
401	UnauthorizedError	Missing or invalid credential — including the normalized `USER_PROFILE_NOT_FOUND` case.
403	ForbiddenError	An untyped forbidden — notably the human-verification gate on anonymous entry creation.
404	NotFoundError	No route matched, or an untyped miss such as an unknown job id.
404	TENANT_NOT_RESOLVED	In a multi-tenant deployment: the host maps to no active tenant (`Since 1.10`).
415	UNSUPPORTED_MEDIA_TYPE	The `content-type` is not accepted.
422	VALIDATION_FAILED	The request body failed schema validation; the per-violation breakdown (JSON path, keyword, allowed values) is under `result.details` .
503	TENANT_IN_MAINTENANCE	In a multi-tenant deployment: the tenant's data is being restored from a backup, or the tenant is held for a move to another cell (`Since 1.10`). Comes with `Retry-After` (1–3600 s, default 30); also answered by `users-service` and `storage-service` . Written directly, so `traceId` is `NO-TRACE-ID` and `message.ar` is empty. Wait and retry.
500	TypeError , other JS class names	An unhandled server bug. Report it; never retry-loop on it.

The data plane — `data-service`

The largest surface. The data service defines about **520 concrete exception classes**; resolving each to its family gives roughly 311 validation (`400`), 2 unauthorized (`401`), 34 forbidden (`403`), 46 not-found (`404`), 10 conflict (`409`), 21 unprocessable (`422`), and 95 on the bare `DataException` — most of those surface as `500` , a few set their own status (the `503` s and the backup API, whose refusals carry the backup engine's status). The table below is not exhaustive (the 1.10 additions and re-classifications are marked `Since 1.10`), so a code absent from it is not proof that it does not exist.

Points worth pulling out of the table before you scan it:

- **The sensitive-field group** *(Since 1.8)*: `SENSITIVE_FIELD_NOT_ALLOWED` , `SENSITIVE_FIELD_CANNOT_BE_MAIN_TITLE` , `SENSITIVE_FIELD_QUERY_NOT_ALLOWED` and `SENSITIVE_FIELD_WRITE_NOT_ALLOWED` are `400` validation rejections raised when a definition, query or write touches a sensitive field in a way the platform forbids.
- **The lock family is `409`** : `LOCKED_OBJECT` , `LOCKED_OBJECT_ALREADY_OWNED` , `NOT_ALLOWED_TO_LOCK_OBJECT` — another actor holds the object. Waiting and re-reading is the correct response; blind retry is not.
- **`UPDATE_CONFLICT` (`409`)** means your `updateIdentifier` no longer matches the stored one — the entry changed under you. Re-fetch, re-apply your change, resubmit. See Writing data.
- **Permission-configuration errors** live (spelled exactly so) and are `400` s about entitlement configuration — e.g. `ENTITLED_ENTRY_RESTRICTED_PERMISSION` , `RESTRICTED_LEVEL_SYSTEM_ONLY` — not runtime permission denials, which are the `403` family.
- **Two `404` s are special-cased**: `DataSourceEntityNotExist` and `EntitiesNotFound` extend the bare base, and reach `404` only because the reject provider maps the code `ENTITY_NOT_FOUND` explicitly.
- **The `401` s are unusual.** `DashboardConfigurationNotEnabledException` is a dashboard-configuration state, not a credential problem, despite its status; `UserNotAuthenticatedException` (`USER_NOT_AUTHENTICATED`) is a user-scoped endpoint called with an anonymous token.
- **Plan limits are `403` s** `Since 1.10` — `ENTRY_CAP_EXCEEDED` , `ENTITY_CAP_EXCEEDED` , `SPACE_CAP_EXCEEDED` (and `STORAGE_CAP_EXCEEDED` from `storage-service`). They refuse creates only, and retrying will not help until the tenant's plan or usage changes — see Plan limits.
- **The backup API relays the backup engine's codes with their own status** `Since 1.10` — `400` , `404` , `409` , `429` (quota, with `Retry-After`), `501` (single-tenant deployment), `502` , `503` (with `Retry-After: 60`). The full list is in Tenant backups, restores and exports.

Status	`error.code`	Originating exception class
400	`ALL_ENTRY_VALIDATION_ERROR`	`AllEntryValidationExceptions`
400	`ARCHIVE_ENTRIES_MUST_PROVIDE_IDS_OR_SELECT_ALL`	`ArchiveEntriesMustProvideIdsOrSelectAllException`
400	`ASSIGNMENT_LEVEL`	`OrgUnitAcceptJustMainLevelException`
400	`ATTACHMENT_EXTENSIONS_NOT_SET`	`ViewDiscussionsAttachmentExtensionsException`
400	`BULK_UPDATE_ENTRIES_MUST_PROVIDE_IDS_OR_SELECT_ALL`	`BulkUpdateEntriesMustProvideIdsOrSelectAllException`
400	`BULK_UPGRADE_ENTRIES_MUST_PROVIDE_IDS_OR_SELECT_ALL`	`BulkUpgradeEntriesMustProvideIdsOrSelectAllException`
400	`CAN_NOT_ARCHIVE_DOMAIN_IN_USE`	`CanNotArchiveDomainInUse`
400	`CAN_NOT_ARCHIVE_TEMPLATE_IN_USE`	`CanNotArchiveTemplateInUse`

Status	`error.code`	Originating exception class
400	`CAN_NOT_CHANGE_CONDITIONAL_INCREMENT_CONFIGURATIONS`	`CanNotChangeConditionalIncrementConfigurations`
400	`CASCADED_FIELD`	`CascadedFieldException`
400	`CIRCULAR_RELATION_NOT_ALLOWED`	`CircularRelationNotAllowedException`
400	`COLLECTION_IS_IMPLICIT`	`ImplicitDeleteException`
400	`COLLECTION_ITEM_IS_IMPLICIT`	`ImplicitItemUpdateException`
400	`COMPONENT_GROUP_ID_NOT_EXIST`	`ComponentGroupIdNotExistException`
400	`COMPONENT_GROUP_ID_NOT_UNIQUE`	`ComponentGroupIdMustBeUniqueException`
400	`COMPONENT_ID_NOT_UNIQUE`	`ComponentIdMustBeUniqueException`
400	`COMPONENT_INCOMPATIBLE_RESTRICTED_PERMISSION`	`IncompatibleRestrictedPermissionException`
400	`COMPONENT_VISIBILITY`	`ComponentVisibilityForAnonymousException`
400	`COUNTER_VIEW_MISSED_MINICARD_VIEW_TYPE`	`CounterViewMissedMiniCardException`
400	`CURRENT_USER_FILTER_CAN_NOT_BE_ENABLED`	`CurrentUserFilterCanNotBeEnabledException`
400	`CURRENT_USER_GROUP_FILTER_CAN_NOT_BE_ENABLED`	`CurrentUserGroupsFilterCanNotBeEnabledException`
400	`CUSTOM_EVENT_INVALID_CONFIGS`	`CustomEventInvalidConfigsException`
400	`CUSTOM_VALIDATION_EXPRESSION_FAILED`	`CustomValidationExpressionFailed`
400	`DASHBOARD_ELEMENT_DUPLICATE`	`DashboardElementDuplicateException`
400	`DASHBOARD_ELEMENT_USED_MORE_THAN_ONCE`	`DashboardElementUsedMoreThanOneException`
400	`DATA_QUERY_UPDATE`	`DataQueryUpdateException`
400	`DATA_VIEW_INVALID_ALLOWED_OPERATIONS`	`DataViewAllowedOperationNotFoundException` (was `500`) `Since 1.10`
400	`DATA_VIEW_INVALID_ARCHIVED_OPERATION_FIELDS_VALUE`	`DataViewArchivedOperationFieldsExistException`
400	`DATA_VIEW_MISSED_REQUIRED_FIELDS`	`DataViewMissedRequiredFieldsException`
400	`DATA_VIEW_OPERATION_FIELDS_REQUIRED`	`DataViewOperationFieldsRequiredException`
400	`DATA_VIEW_SOURCE_ENTITIES_MAPPING_INVALID_SOURCE`	`DataViewSourceEntitiesMappingInvalidSourceException`
400	`DATA_VIEW_SOURCE_ENTITIES_MAPPING_NOT_SUPPORTED_FOR_NON_AGGREGATOR_ENTITY`	`DataViewSourceEntitiesMappingNotSupportedForNonAggregatorEntityException`
400	`DELETE_IMPLICIT_RESOURCE_RESTRICTION`	`DeleteImplicitResourcesException`
400	`DENY_CIRCULAR_RELATION_WITH_MULTI_VALUE_OR_REPEATABLE`	`DenyCircularRelationWithMultiValueOrRepeatableException`
400	`DETAILS_META_DATA_FILTER_CAN_NOT_BE_ENABLED`	`DetailsMetaDataFilterCanNotBeEnabledException`
400	`DRAFT_ENTRY_NOT_BELONG_TO_USER`	`DraftEntryNotBelongToUserException`
400	`DUPLICATE_FIELD_UNIQUE_ERROR`	`DuplicatedFieldCanNotBeUniqueError`
400	`DUPLICATE_PERMISSIONS`	`ViewAnonymousAndRestrictedException`
400	`DUPLICATED_ASSIGNMENT`	`AnotherAssignmentExists`
400	`DUPLICATED_SHOW_AS_FIELD_OPTION`	`DuplicatedShowAsException`
400	`EMBEDDED_PAGE_INVALID_URL`	`EmbeddedPageInvalidURLException`
400	`EMBEDDED_PAGE_MUST_HAVE_URL`	`EmbeddedPageMustHaveURLException`

Status	error.code	Originating exception class
400	EMPTY_ARRAY	ArrayCanNotBeEmptyException
400	EMPTY_LOCALIZED_FIELD	EmptyLocalizedException
400	END_USER_SPACE_ALREADY_ACTIVATED	EndUserSpaceAlreadyActivatedException
400	END_USER_SPACE_ALREADY_DEACTIVATED	EndUserSpaceAlreadyDeactivatedException
400	ENTITLED_ENTITY_NOT_ACTIVE	EntitledEntityNotActiveException
400	ENTITLED_ENTITY_NOT_EXIST	NotExistEntitledException
400	ENTITLED_ENTRY_RESTRICTED_PERMISSION	RestrictedPermissionEntitledEntriesEmptyExcecption
400	ENTITLEMENT_TYPE_REQUIRED	EntitlementTypeRequiredException
400	ENTITY_TYPE_NOT_REQUEST	EntityNotOfTypeRequestException
400	ENTRIES_CONFIGURATION_NOT_ENABLED_FOR_VIEW	ViewEntriesNotAccessibleException
400	ENTRIES_CONFIGURATIONS_MUST_BE_ENABLED	EntriesConfigurationMustBeEnabled
400	ENTRY_COMPLETED	SpecifiedEntryCompletedException
400	ENTRY_NOT_IN_WORKFLOW	EntryNotInWorkflowException
400	ENTRY_VALIDATION_ERROR	EntryValidationException
400	ENTRY_VALIDATION_ERROR	ValidatorFunctionEntryValidationException
400	ENTRY_VERSION_REQUIRED	EntryVersionRequiredException
400	EXTERNAL_STEP_CONFIG_REQUIRED	ExternalStepConfigRequiredException
400	EXTERNAL_STEP_CUSTOM_PAYLOAD_REQUIRES_POST	ExternalStepCustomPayloadRequiresPostException
400	EXTERNAL_STEP_MANUAL_EXECUTION_NOT_ALLOWED	ExternalStepManualExecutionNotAllowedException
400	EXTERNAL_STEP_OWNER_MUST_BE_USER	ExternalStepOwnerMustBeUserException
400	FAIL_TO_ADD_FIELD	FailToAddFieldException
400	FAILURE_ACTION_REQUIRED	FailureActionsRequiredException
400	FIELD_MUST_ENABLE_CIRCULAR_RELATION	FieldMustEnableDenyCircularRelationException
400	FILE_EXPIRED	FileExpiredException Since 1.10
400	FILTER_FIRST_CAN_ENABLE_AUTOFILL_WITHOUT_ENABLING_ADD_ENTRIES	FilterFirstCanNotEnableAutofillWithoutEnablingAddEntries
400	FILTER_FIRST_CAN_NOT_BE_ENABLED_WITH_DASHBOARD	FilterFirstCanNotBeEnabledWithDashboard
400	FILTER_FIRST_CUSTOM_QUERY_NOT_PROVIDED	FilterFirstCustomQueryNotProvidedException
400	FILTER_FIRST_FIELD_CAN_NOT_BE_OF_THIS_TYPE_EXISTS	FilterFirstFieldCanNotBeAppliedToFieldType
400	FILTER_FIRST_FIELD_NOT_EXISTS	FilterFirstFieldNotExists
400	FILTER_FIRST_FIELDS_MISMATCH	FilterFirstFieldsMismatch
400	FILTER_FIRST_MAX_FIELDS_LENGTH	FilterFirstMaxFieldsLength
400	FILTER_FIRST_MIN_FIELDS_LENGTH	FilterFirstMinFieldsLength
400	FILTER_FIRST_SHOULD_BE_ENABLED_WITH_QUERY_PARAMETER	FilterFirstShouldBeEnabledWithQueryParameter
400	FONT_FORMAT_NOT_SUPPORTED	FontFormatNotSupportedException
400	GENERATING_COUNTER_ERROR	GeneratingCounterException

Status	error.code	Originating exception class
400	GENERATING_COUNTER_SEQUENCE_ERROR	GeneratingCounterSequenceException
400	GENERIC_VIEW_NORMALIZED_NAME_DUPLICATE	GenericViewNormalizedNameDuplicateException
400	GENERIC_VIEW_RELATED_TO_DATA_ENTITY	GenericViewRelatedToDataEntityException
400	GET_SYSTEM_BUTTONS_STATUS_FOR_ENTRIES_MUST_PROVIDE_IDS_OR_SELECT_ALL	GetSystemButtonsStatusForEntriesMustProvideIdsOrSelectAllException
400	IMPLICIT_DELETE_RESTRICTION	ImplicitEntityDeleteException
400	IMPLICIT_DELETE_RESTRICTION	ImplicitRelationDeleteException
400	IMPLICIT_ENTRY_UPDATE	ImplicitEntryException
400	IMPLICIT_FIELD_RESTRICTION	ImplicitFieldException
400	IMPLICIT_UPDATE_RESTRICTION	ImplicitEntityUpdateException
400	IMPLICIT_UPDATE_RESTRICTION	ImplicitRelationUpdateException
400	IMPLICIT_UPDATE_RESTRICTION	ImplicitResourceUpdateException
400	INLINE_EDIT_NOT_ALLOWED	ViewDoesNotSupportInlineEdit
400	INTERNAL_ENTITY_NOT_FOUND	InternalEntityNotFoundException
400	INVALID__MANAGER_STEP_OWNER_SELECTION_CRITERIA	InvalidManagerStepOwnerSelectionCriteria
400	INVALID_CATEGORY_OBJECT	InvalidCategoryObject
400	INVALID_DATA_VIEW	InvalidDataViewException
400	INVALID_EXPORT_FORMAT	InvalidExportFormatException
400	INVALID_FIELD_MAPPING	InvalidFieldMappingException
400	INVALID_FIELD_SOURCE_FOR_DENY_CIRCULAR_RELATION	InvalidFieldSourceForDenyCircularRelationException
400	INVALID_GEOJSON_OBJECT	InvalidGeojsonObject
400	INVALID_LOCALIZATION	InvalidLocalizationException
400	INVALID_LOCK_END_DATE	InvalidLockEndDate
400	INVALID_OR_DISABLED_TRIGGER	InvalidOrDisabledTriggerException
400	INVALID_POINTER	InvalidPointerException Since 1.10
400	INVALID_QUERY_EXPRESSION_TYPE	DataQueryExpressionTypeValidationException
400	INVALID_QUERY_EXPRESSION_VALUE	DataQueryExpressionValueNotStringException
400	INVALID_RELATED_FIELD	InvalidRelatedFieldException
400	INVALID_STATISTIC_QUERY_SOURCE	StatisticQuerySourceNotSameAsViewException
400	INVALID_STATISTIC_QUERY_VALIDATION	StatisticQueryValidationExcpetion
400	INVALID_STEP_HAS_MULTIPLE_MAIN_ACTIONS	InvalidStepHasMultipeMainActionsException
400	INVALID_STEP_ID	WorkflowStepIdNotExistException
400	INVALID_STRING_DATE	InvalidStringDateException
400	INVALID_SYSTEM_LINK_TOKEN	InvalidViewTokenExption
400	INVALID_THRESHOLD_VALUE	InvalidStepOwnerThresholdException
400	INVALID_TOKENS_MAP	InvalidTokensMapException

Status	error.code	Originating exception class
400	JOINER_PROXY_MAX_FIELDS_LIMIT_EXCEEDED	JoinerProxyMaxFieldsLimitExceeded
400	JOINER_PROXY_MAX_SOURCES_LEVELS_LIMIT_EXCEEDED	JoinerProxyMaxSourcesLevelsLimitExceeded
400	JOINER_PROXY_MAX_SOURCES_LIMIT_EXCEEDED	JoinerProxyMaxSourcesLimitExceeded
400	LANGUAGE_MAX_LENGTH	LanguageMaxLengthException
400	LANGUAGE_MIN_LENGTH	LanguageMinLengthException
400	LANGUAGE_NOT_SUPPORTED	LanguageNotSupportedException
400	LOCALIZATION_DUPLICATE	DuplicatedLocalization
400	MAIN_TITLE_CANNOT_BE_REPEATABLE_OR_MULTIPLE	MainTitleCannotBeRepeatableOrMultipleException
400	MAX_LESS_THAN_MIN	MinMaxValidationException
400	MAX_VALUE_SHOULD_BE_NUMBER	MaxValueShouldBeNumberException
400	MAX_VALUE_SHOULD_NOT_BE_SMALLER_THAN_MIN_VALUE	MaxValueShouldNotBeSmallerThanMinValueException
400	META_DATA_FILTER_CAN_NOT_BE_ENABLED	MetaDataFilterCanNotBeEnabledException
400	MIN_VALUE_SHOULD_BE_NUMBER	MinValueShouldBeNumberException
400	MISSING_FIELDS_FOR_FILTER_FIRST	MissingFieldsForFilterFirstException
400	MISSING_FILTER_FIRST_FIELD_NAME	MissingFilterFirstFieldName
400	MISSING_REQUIRED_STEP_ACTIONS	MissingStepActionsException
400	MODULE_HAS_ENTITIES	ModuleHasEntities
400	MULTIPLE_MANAGER_ROLE_NOT_ALLOWED	MultipleManagerRoleNotAllowed
400	MUST_AUTOMATE_AT_LEAST_ONE_STEP	MustAutomateAtLeastOneStepException
400	MUST_ENABLE_AUTHENTICATED	AuthenticatedMustEnabledExcecption
400	MUST_ENABLE_DISCUSSIONS_TO_ENABLE_DISCUSSIONS_ATTACHMENTS_INSIDE_ENTRY_INFO	MustEnableDiscussionsToEnableDiscussionsAttachmentsInsideEntryInfo
400	MUST_ENABLE_ENTRIES_LIST_OR_ENTRY_DETAILS_TO_ENABLE_SYSTEM_BUTTONS_ON_VIEW	MustEnableEntriesListOrEntryDetailsToEnableSystemButtonsOnView
400	MUST_ENABLE_RESTRICTED	RestrictedMustEnabledExcecption
400	MUST_PROVIDE_ITEMS_TO_EXPORT_PROPERTY	MustProvideITemsToExportProperty
400	NESTED_REFERENCE_MAIN_TITLE_REFERENCED_BY_OTHER_ENTITY	NestedReferenceMainTitleReferencedByOtherEntity
400	NESTED_REFERENCE_MAIN_TITLE_SOURCE_IS_REFERENCE	NestedReferenceMainTitleSourceIsReference
400	NEW_FIELDS_UNIQUE_ERROR	NewFieldCantBeUniqueError
400	NORMAL_QUERY_TO_PARAMETERIZED_QUERY_RESTRICTION	DataQueryParameterizationChangeException
400	NOT_ALLOWED_TO_PATCH_UPDATE_FIELDS	NotAllowedToPatchUpdateFields
400	NOT_ENOUGH_STORE_SPACES_SELECTED	NotEnoughStoreSpacesSelectedException
400	OBSOLETED_SAME_VERSION	EntrySameVersionException
400	OPERATION_NOT_SUPPORTED_FOR_THIS_STORE_TYPE	OperationNotSupportedForThisType
400	ORIGINAL_OWNER_NOT_FOUND	OriginalOwnerNotFoundException

Status	error.code	Originating exception class
400	PARALLEL_STEP_OWNER_ALREADY_EXECUTED_STEP	ParallelStepOwnerAlreadyExecutedStepException
400	PATCH_UPDATE_DUPLICATE_REFERENCE_ENTRY_ID	PatchUpdateDuplicateReferenceEntryIdException
400	PATCH_UPDATE_SELF_MODE	PatchUpdateSelfModeException
400	PERMANENT_DELETE_ENTRIES_MUST_PROVIDE_IDS_OR_SELECT_ALL	PermanentDeleteEntriesMustProvideIdsOrSelectAllException
400	PERMISSIONS_ANONYMOUS_OR_RESTRICTED_NOT_ENABLED	AnonymousOrRestrictedNotEanabledException
400	PERMISSIONS_ANONYMOUS_RESTRICTED_ENABLED	AnonymousAndRestrictedEnabledException
400	PERMISSIONS_NOT_SET	ViewPublishedWithoutPermissionsException
400	PERSON_CAN_HAVE_ONE_MAIN_ASSIGNMENT	UserCanHaveOnlyOneAssignment
400	PLATFORM_NOT_SUPPORTED	PlatformNotSupportedException
400	POST_TEMPLATE_NOT_EXIST	PostTemplateNotExistException
400	QUERY_ALREADY_IN_USE	QueryInUsedException
400	QUERY_ENGINE_EXPRESSION_ERROR	ExpressionParsingException
400	READONLY_ENTITY_OPERATION	ReadOnlyEntityOperationException
400	RELATED_FIELDS_REQUIRED	RelatedFieldsRequiredException
400	RELATION_NOT_FOUND_INSIDE_VIEW	RelationNotFoundInsideViewException
400	RELATIVE_VALUE_NOT_SUPPORTED	RelativeValueNotSupportedException
400	REPORT_DATA_SOURCE_NOT_SUPPORTED	ReportDataSourceSupportedException
400	REPORT_EXISTS	ReportExistException
400	REPORT_FIELDS_LENGTH	ReportFieldsLengthException
400	REPORT_FIELDS_NOT_EXIST	ReportFieldsNotExistException
400	REPORT_SENDING_TIME	ReportSendingTimeCanNotBeInThePastException
400	REQUEST_CANNOT_EDITED	CanNotEditEntryException
400	REQUEST_CANNOT_PUBLISHED	ServiceHasNoWorkflowException
400	RESTORE_ENTRIES_MUST_PROVIDE_IDS_OR_SELECT_ALL	RestoreEntriesMustProvideIdsOrSelectAllException
400	RESTRICTED_AUTHENTICATED_VIEW_OPTIONS_REQUIRE_PERMISSIONS	RestrictedAuthenticatedViewOptionsRequirePermissionsException
400	RESTRICTED_LEVEL_SYSTEM_ONLY	RestrictedLevelSystemOnlyException
400	RESTRICTED_NORMALIZED_NAME	RestrictedNormalizedNameError
400	RESTRICTED_VIEW_CAN_NOT_BE_DEFAULT	RestrictedViewCanNotBeDefaultException
400	ROLE_MUST_BE_JOP_POSITION	RoleMustBeOfTypeJobPosition
400	SAVE_QUERY_NOT_ALLOWED	SaveQueryNotAllowedException
400	SELECTED_SPACES_SHOULD_NOT_HAVE_DUPLICATION	StoreSelectedSpacesDuplicationNotAllowedException
400	SENSITIVE_FIELD_CANNOT_BE_MAIN_TITLE	SensitiveFieldCannotBeMainTitleException
400	SENSITIVE_FIELD_NOT_ALLOWED	SensitiveFieldNotAllowedException
400	SENSITIVE_FIELD_QUERY_NOT_ALLOWED	SensitiveFieldQueryNotAllowedException
400	SENSITIVE_FIELD_WRITE_NOT_ALLOWED	SensitiveFieldWriteNotAllowedException

Status	error.code	Originating exception class
400	SERVICE_AUDIENCE_REQUIRED	ServiceAudienceIsRequired
400	SERVICE_CATEGORY_NOT_EXIST	ServiceCategoryNotExistException
400	SERVICE_CATEGORY_REQUIRED	ServiceCategoryRequiredException
400	SERVICE_ENTITY_NOT_ACTIVE	CanNotRequestServiceException
400	SINGLE_ROLE_ASSIGNED_MULTIPLE_TIMES	SingleRoleAssignedMultipleTimes
400	STEP_NOT_EXECUTED_BEFORE	StepNotExecutedBeforeException
400	STORE_WITH_SAME_WEIGHT_ALREADY_EXIST	StoreWithSameWeightExistsException
400	SUB_WORKFLOW_STEP_MUST_NOT_HAVE_ACTIONS	SubWorkflowStepActionsException
400	SYSTEM_BUTTON_CONFIRMATION_DIALOG_VARIABLES_NOT_ALLOWED_FOR_MULTI_SCOPE	ConfirmationDialogVariablesNotAllowedForMultiScopeException
400	TABLE_VIEW_CONFIG_NOT_FOUND	TableViewConfigNotFoundException
400	THIS_ROLE_CAN_ONLY_BE_ASSIGNED_ONE_ORG_UNIT	ThisRoleCanOnlyBeAssignedToOneOrgUnit
400	TIMELINE_ATTRIBUTE_MAPPING_NOT_MATCHING_ANY_CRITERIA	TimelineAttributeMappingNotMatchingAnyCriteriaException
400	TIMELINE_ATTRIBUTE_NOT_MATCH_POST_CATEGORY_ATTRIBUTES	TimelineAtrributeNotMatchPostCategoryAttributes
400	TIMELINE_CORE_ENTITY	TimelineCoreEntityExcecption
400	TIMELINE_DESCRIPTION_FIELD_INVALID	TimelineDescriptionFieldInvalidFieldTypeExcecption
400	TIMELINE_FIELD_NAME_NOT_EXIST	TimelineFieldNameNotExistException
400	TIMELINE_FIELD_VALUE_MUST_BE_DEFINED	TimelineFieldValueMustBeDefined
400	TIMELINE_FIELD_VALUE_MUST_BE_HIDDEN	TimelineFieldValueMustBeHidden
400	TIMELINE_MISSING_REQUIRED_FIELDS	TimelineMissingRequiredFieldsException
400	TIMELINE_PUBLISH_CONFIG_CONFLICT	TimelinePublishConfigConflictException
400	TIMELINE_PUBLISH_WORKFLOW_EMPTY_STEPS	TimelinePublishWorkflowEmptyStepsException
400	TIMELINE_PUBLISH_WORKFLOW_INIT_STEP	TimelinePublishWorkflowInitStepException
400	TIMELINE_PUBLISH_WORKFLOW_LAST_STEP	TimelinePublishWorkflowlastStepException
400	TIMELINE_PUBLISH_WORKFLOW_STEP_NOT_EXIST	TimelinePublishWorkflowStepNotExistException
400	TIMELINE_TITLE_FIELD_INVALID	TimelineTitleFieldInvalidFieldTypeExcecption
400	TO_ENABLE_CLICK_TO_BROWSE_MUST_ENABLE_VIEW_CONFIGURATIONS	ToEnableClickToBrowseMustEnableViewConfiguration
400	TRANSFORMATION_REQUIRED	TransformationModelRequiredException
400	TRIGGER_TIMER_OPTIONS_CONFLICT	TriggerTimerOptionsConflict
400	TRIGGERS_CAN_NOT_AUTOMATE_LAST_STEP	CanNotAutomateLastStepException
400	TRIGGERS_LINKED_TO_NON_EXISTING_WORKFLOW	TriggersLinkedToNonExistingWorkflow
400	TRIGGERS_LINKED_TO_NON_EXISTING_WORKFLOW_STEP	TriggerLinkedToNonExistingWorkflowStep
400	TRIGGERS_LINKED_TO_NON_EXISTING_WORKFLOW_STEP_ACTION	TriggerLinkedToNonExistingWorkflowStepAction
400	UNDEFINED_STEP_OWNER_THRESHOLD	UndefinedStepOwnerThresholdException
400	UNKNOWN_OPERATOR_ERROR	UnknownOperatorExpression

Status	error.code	Originating exception class
400	UNSUPPORTED_CONDITIONAL_INCREMENT_CALCULATION_METHOD	UnsupportedConditionalIncrementCalculationMethod
400	UNSUPPORTED_FIELD_EXPRESSION	UnsupportedFieldExpression
400	UPDATE_COLLECTION_NO_ITEMS	UpdateCollectionNoItemsException
400	UPDATE_DRAFT_ENTRY_NOT_ALLOWED	UpdateDraftEntryNotAllowedException
400	UPDATE_IMPLICIT_RESOURCE_RESTRICTION	UpdateImplicitResourcesException
400	UPDATED_ENTRY_EVENT_CONFIG_INVALID_FIELD	UpdatedEntryEventConfigInvalidFieldException
400	USER_ACCOUNT_TYPE	UserIsNotOrganizationAccount
400	USER_ALREADY_OWN_STEP	UserAlreadyOwnException
400	USER_COLLECTION_DELETE	UserCollectionsDeleteException
400	USER_COLLECTION_REPLACE	UserCollectionReplaceException
400	USER_COLLECTION_SAVE_NOT_ALLOWED	SaveUserCollectionNotAllowedException
400	USER_COLLECTION_UPDATE	UserCollectionsUpdateException
400	USER_GROUP_ALREADY_HAS_HIGHER_PERMISSION_LEVEL	UserGroupAlreadyHasHigherPermissionLevelException
400	USER_MUST_HAVE_VIEW_ACCESS_FIRST	UserMustHaveViewAccessFirstException
400	USER_TYPE_CHANGED	UserTypeCannotBeUpdated
400	VIEW_ALLOW_SAVED_QUERIES_CAN_NOT_BE_ENABLED	ViewAllowSavedQueriesCanNotBeEnabledException
400	VIEW_ALLOW_USER_COLLECTIONS_CAN_NOT_BE_ENABLED	ViewAllowUserCollectionsCanNotBeEnabledException
400	VIEW_CONFIGURATIONS_MUST_BE_ENABLED_WITH_FILTER_FIRST	ViewConfigurationMustBeEnabledWithFilterFirst
400	VIEW_CREATE_DRAFT_DATA_ENTRY_NOT_ALLOWED	ViewCreateDraftDataEntryNotAllowedException
400	VIEW_INVALID_PERMISSIONS	ViewInvalidPermissionsException
400	VIEW_MUST_BE_BASED_ON_QUERY_TO_ENABLE_OVERRIDE_SOURCE_FOR_STATISTIC_QUERY	ViewMustBeBasedOnQueryToEnableOverrideSourceForStatisticQuery
400	VIEW_NOT_PUBLISHED	ViewNotPublishedException
400	VIEW_URL_TOKEN_LENGTH	ViewUrlTokenLengthException
400	VIEW_URL_TOKEN_REQUIRED	ViewUrlTokenRequiredException
400	VIEW_URL_TOKEN_USED_BEFORE	ViewUrlTokenUsedBeforeException
400	VIEW_URLS_ITEMS_EMPTY	ViewUrlsItemsEmptyException
400	VIEW_USER_COLLECTIONS_CAN_NOT_BE_ENABLED	ViewUserCollectionsCanNotBeEnabledException
400	VIRTUAL_ENTITY_OPERATION	VirtualEntityNotAllowedOperation
400	VISIBILITY_CUSTOM_SETTINGS_ENABLED	VisibilityCustomSettingsEnabledException
400	WORKFLOW_CAN_NOT_MOVE	CanNotMoveFromLastStepException
400	WORKFLOW_CAN_NOT_RESET	CanNotResetStepException
400	WORKFLOW_CAN_NOT_RETURN	CanNotReturnStepException
400	WORKFLOW_NORMALIZED_NAME_DUPLICATE	WorkflowNormalizedNameDuplicateException
400	WORKFLOW_OWNER_COLLECTION_NOT_EXIST	WorkFlowOwnerCollectionNotFound

Status	`error.code`	Originating exception class
400	`WORKFLOW_OWNER_GROUP_NOT_EXIST`	`WorkFlowOwnerGroupNotFound`
400	`WORKFLOW_RESET_REQUIRED`	`WorkflowResetRequired`
400	`WORKFLOW_STEP_NORMALIZED_NAME_DUPLICATE`	`WorkflowStepNormalizedNameDuplicateException`
400	`WRONG_DATA_TYPE`	`ValidationValueWrongTypeException`
401	`DASHBOARD_CONFIGURATION_NOT_ENABLED`	`DashboardConfigurationNotEnabledException`
401	`USER_NOT_AUTHENTICATED`	`UserNotAuthenticatedException` `Since 1.8`
403	`ACCESS_OBJECT_NO_AUDIENCE`	`AccessObjectNoAudienceException` `Since 1.10`
403	`ACCESS_REQUEST_FORBIDDEN`	`ForbiddenToAccessRequestException`
403	`ARCHIVED_ENTRY_NOT_ACCESSIBLE_THROUGH_VIEW`	`ArchivedEntryNotAccessibleThroughViewException`
403	`BATCH_CREATE_ENTRIES_QUICK_SAVE_NOT_ENABLED`	`BatchCreateEntiresQuickSaveNotEnabledException`
403	`BATCH_CREATE_ENTRIES_RELATION_NOT_ALLOWED`	`BatchCreateEntriesRelationNotAllowedException`
403	`CANNOT_EXPORT_ENTITY`	`ForbiddenToExportEntity`
403	`CANNOT_EXPORT_MODULE`	`ForbiddenToExportModule`
403	`COUNTER_VIEW_NOT_MATCHED`	`CounterViewNotMatchedWithMiddlewareException`
403	`EDIT_MODULE_DASHBOARD_NOT_ALLOWED`	`EditModuleDashboardNotAllowedException`
403	`ENTITY_CAP_EXCEEDED`	`EntityCapExceededException` `Since 1.10`
403	`ENTITY_DASHBOARD_EDIT_NOT_ALLOWED`	`EntityDashboardEditNotAllowedException`
403	`ENTRY_CAP_EXCEEDED`	`EntryCapExceededException` `Since 1.10`
403	`ENTRY_NOT_ACCESSIBLE_THROUGH_VIEW`	`EntryNotAccessibleThroughViewException`
403	`ENTRY_OPERATION_NOT_ALLOWED`	`EntryOperationNotAllowedException`
403	`ENTRY_READ_FORBIDDEN`	`ForbiddenToReadEntryException`
403	`INVALID_ENTRY_NOT_ACCESSIBLE_THROUGH_VIEW`	`InvalidEntryNotAccessibleThroughViewException`
403	`INVALID_REQUIREMENTS_ENTRY_NOT_ACCESSIBLE_THROUGH_VIEW`	`InvalidRequirementsEntryNotAccessibleThroughViewException`
403	`LICENSE_ADMIN_FEATURE_REQUIRED`	`LicenseAdminFeatureRequiredException`
403	`OBSOLETE_ENTRY_NOT_ACCESSIBLE_THROUGH_VIEW`	`ObsoleteEntryNotAccessibleThroughViewException`
403	`PERMISSION_GRANT_EXCEEDS_CALLER_ACCESS`	`PermissionGrantExceedsCallerAccessException` `Since 1.8`
403	`RESOLVE_VALUE_PATH_NOT_STORED`	`ResolveValuePathNotStoredException` `Since 1.8`
403	`SPACE_CAP_EXCEEDED`	`SpaceCapExceededException` `Since 1.10`
403	`SYSTEM_BUTTON_IS_DISABLED`	`SystemButtonIsDisabled`
403	`SYSTEM_BUTTON_NOT_UI_CUSTOM_ACTION`	`SystemButtonNotUiCustomAction`
403	`SYSTEM_DASHBOARD_EDIT_NOT_ALLOWED`	`SystemDashboardEditNotAllowedException`
403	`VIEW_ACCESS_DENIED`	`ViewAccessNotAuthorizedException`
403	`VIEW_DASHBOARD_CONFIGURATION_NOT_ENABLED`	`ViewDashboardConfigurationNotEnabledException`
403	`WORKFOLOW_ADD_ADHOC`	`UserCannotAddAdhocException`
404	`ACCESS_OBJECT_NOT_EXIST`	`AccessObjectNotExistException`
404	`ACTION_NOT_EXIST`	`ActionNotExistException`
404	`BLOCK_NOT_EXIST`	`BlockNotExistException`

Status	`error.code`	Originating exception class
404	`CATEGORY_NOT_EXIST`	`CategoryNotExistException`
404	`COLLECTION_NOT_EXIST`	`CollectionNotExistValidationException`
404	`COUNTER_NOT_FOUND`	`CounterNotFoundException`
404	`DATA_COLLECTION_NOT_EXIST`	`DataCollectionNotFoundException`
404	`DATA_QUERY_NOT_EXIST`	`DataQueryNotFound`
404	`DATA_VIEW_NOT_EXIST`	`DataViewNotExistException`
404	`ENTITIES_NOT_EXIST`	`EntitiesNotExistException`
404	`ENTITY_DATA_SOURCE_NOT_EXIST`	`EntityDataSourcesNotExistsException`
404	`ENTITY_FIELD_NOT_EXIST`	`EntityFieldNotExistException`
404	`ENTITY_NOT_EXIST`	`EntityNotExistException`
404	`ENTITY_NOT_EXIST_VALIDATION`	`EntityNotExistValidationException`
404	`ENTITY_NOT_FOUND`	`DataSourceEntityNotExist`
404	`ENTITY_NOT_FOUND`	`EntitiesNotFound`
404	`ENTRIES_SOURCE_NOT_FOUND`	`EntriesSourceNotFoundException`
404	`ENTRY_NOT_EXIST`	`EntryNotExistException`
404	`FALLBACK_OWNER_REQUIRED`	`FallbackOwnerRequiredException`
404	`FIELD_SOURCE_NOT_FOUND`	`FieldSourceNotFoundException`
404	`FILTER_FIRST_CUSTOM_QUERY_NOT_FOUND`	`FilterFirstCustomQueryNotFoundException`
404	`GROUP_NOT_EXIST`	`GroupNotExistValidationException`
404	`INVALID_DASHBOARD_ID`	`InvalidDashboardId`
404	`INVALID_MERGE_STEP`	`MergeStepValidationException`
404	`INVALID_MODULE_ID`	`InvalidModuleId`
404	`ITEM_NOT_EXIST`	`DataNotExistException`
404	`JOB_STATUS_NOT_EXIST`	`JobStatusNotExistException`
404	`LOCKED_OBJECT_NOT_FOUND`	`LockedObjectNotFound`
404	`MERGE_STEP_ID_REQUIRED`	`MergeStepRequiredException`
404	`MODULE_NOT_EXIST`	`ModuleNotExistException`
404	`ORG_UNIT_NOT_EXIST`	`OrgUnitNotExistException`
404	`OWNER_NOT_EXIST`	`OwnerNotExistValidationException`
404	`QUERY_NOT_FOUND`	`DataSourceQueryNotExistValidationException` (saves naming an unknown query) `Since 1.10`
404	`RELATION_NOT_EXIST`	`RelationNotExistException`
404	`STATISTIC_QUERY_NOT_EXIST`	`StatisticQueryNotFoundException`
404	`SUB_WORKFLOW_ID_NOT_FOUND`	`SubWorkflowIdNotFoundException`
404	`SUB_WORKFLOW_ID_REQUIRED`	`SubWorkflowIdRequiredException`
404	`SYSTEM_BUTTON_NOT_EXIST`	`SystemButtonNotFound`
404	`TAG_GROUP_NOT_FOUND`	`TagGroupNotFoundException`
404	`TEMPLATE_NOT_EXIST`	`TemplateNotExists`

Status	`error.code`	Originating exception class
404	`TRIGGER_NOT_EXIST`	`TriggerNotExistException`
404	`USER_NOT_EXIST`	`UserNotExistException`
404	`WORKFLOW_ACTION_NOT_EXIST`	`WorkFlowActionNotExistException`
404	`WORKFLOW_NOT_EXIST`	`WorkFlowNotExistException`
404	`WORKFLOW_STEP_NOT_EXIST`	`WorkflowStepNotExistException`
409	`LOCKED_OBJECT`	`LockedObjectException`
409	`LOCKED_OBJECT_ALREADY_OWNED`	`LockedObjectAlreadyOwned`
409	`NOT_ALLOWED_TO_LOCK_OBJECT`	`NotAllowedToLockObject`
409	`UPDATE_CONFLICT`	`UpdateIdentifierConflictException`
422	`CURRENT_STEP_ALREADY_OWNED`	`CurrentStepAlreadyOwnedException` `Since 1.8`
422	`ENTITY_ARCHIVE`	`EntityArchiveException` (was `500`) `Since 1.10`
422	`ENTRY_HAS_INCOMING_REFERENCES`	`EntryHasIncomingReferencesException`
422	`ENTRY_NOT_ARCHIVED`	`EntryNotArchivedException`
422	`ENTRY_NOT_ARCHIVED_FOR_DELETION`	`EntryNotArchivedForDeletionException`
422	`ENTRY_NOT_OBSOLETE`	`EntryNotObsoleteException`
422	`OBSOLETED_ENTRY`	`ObsoleteEntryException`
422	`PERMANENT_DELETE_BATCH_LIMIT_EXCEEDED`	`PermanentDeleteBatchLimitExceededException`
422	`REPORT_PERMISSION_NOT_SELECTED`	`ReportPermissionsNotSelectedException`
422	`SYSTEM_SETTINGS_DUPLICATE_PHONE_NUMBER`	`SystemSettingsDuplicatePhoneNumberException` `Since 1.10`
422	`SYSTEM_SETTINGS_FIELD_NOT_SECRET`	`SystemSettingsFieldNotSecretException` `Since 1.10`
422	`SYSTEM_SETTINGS_SPACE_MISMATCH`	`SystemSettingsSpaceMismatchException` `Since 1.10`
422	`SYSTEM_SETTINGS_TENANT_DEFAULT_ARCHIVE`	`SystemSettingsTenantDefaultArchiveException` `Since 1.10`
422	`VALIDATION_FAILED`	`InvalidEnumValue`
422	`WORKFLOW_ACTION_EXECUTION_NOT_ALLOWED`	`WorkFlowActionExecutionNotAllowed` (was `500`) `Since 1.8`
500	`ACTION_DUPLICATE`	`ActionDuplicateException`
500	`ACTION_NOT_ALLOWED`	`ActionNotAllowed`
500	`CHANGE_MODE_NOT_IMPLEMENTED`	`ChangeModeNotImplementedException`
500	`CIRCULAR_DEPENDENCY_DETECTED`	`CircularDependencyException`
500	`DASHBOARD_ACCESS_NOT_ALLOWED`	`DashboardAccessNotAllowedException`
500	`DASHBOARD_STATISTIC_QUERY_NOT_IN_MODULE`	`DashboardStatisticQueryNotInModule`
500	`DATA_COLLECTION_DUPLICATE`	`DataCollectionDuplicateException`
500	`DATA_COLLECTION_NORMALIZED_NAME_DUPLICATE`	`DataCollectionNormalizedNameDuplicateException`
500	`DATA_MODULE_ICON_REQUIRED`	`DataModuleIconIsRequired`
500	`DATA_QUERY_NORMALIZED_NAME_DUPLICATE`	`DataQueryNormalizedNameDuplicateException`
500	`DATA_QUERY_NOT_BELONG_TO_ENTITY`	`DataQueryNotBelongToEntity`
500	`DATA_RELATION_NORMALIZED_NAME_DUPLICATE`	`DataRelationNormalizedNameDuplicateException`
500	`DATA_SOURCE_STORE_REQUIRED`	`DataSourceStoreRequireException`

Status	error.code	Originating exception class
500	DATA_TYPE_NOT_SUPPORTED	DataTypeNotSupportedException
500	DATA_VIEW_AFTER_SAVE_NOT_FOUND	DataViewAfterSaveNotFoundException
500	DATA_VIEW_AFTER_SAVE_URL_EMPTY	DataViewAfterSaveUrlEmptyException
500	DATA_VIEW_AFTER_SAVE_URL_HTTP_METHOD_REQUIRED	DataViewAfterSaveUrlHttpMethodRequiredException
500	DATA_VIEW_EDIT_CONFIGURATION_NOT_SUPPORTED_FOR_VIRTUAL_AGGREGATOR_ENTITY	DataViewEditConfigurationNotSupportedForVirtualAggregatorEntity
500	DATA_VIEW_EDIT_CONFIGURATION_NOT_SUPPORTED_FOR_VIRTUAL_PROXY_ENTITY	DataViewEditConfigurationNotSupportedForVirtualProxyEntity
500	DATA_VIEW_INVALID_ENTITY_ID	ViewNotBelongToEntity
500	DATA_VIEW_NORMALIZED_NAME_DUPLICATE	DataViewNormalizedNameDuplicateException
500	DATA_VIEW_SAVE_MESSAGE	DataViewAfterSaveMessageEmptyExcpetion
500	DUPLICATE_ITEM	DuplicateItemException
500	ENTITY_ARCHIVED	EntityArchivedException
500	ENTITY_BELONG_TO_MODULE	EntityBelongToModule
500	ENTITY_BLOCK_OPTIONS_NOT_SUPPORTED	EntityBlockOptionsException
500	ENTITY_DUPLICATE	EntityDuplicateException
500	ENTITY_FIELD_DUPLICATE	EntityFieldDuplicateException
500	ENTITY_FIELD_NOT_CATEGORY	EntityFieldNotCategoryFieldException
500	ENTITY_FIELD_NOT_REFERENCE	EntityFieldNotReferenceFieldException
500	ENTITY_FIELD_OPTIONS_ERROR	EntityFieldOptionsException
500	ENTRIES_SOURCE_NOT_BELONG_TO_SOURCE	EntriesSourceNotBelongToSourceException
500	ENTRY_ARCHIVED	EntryArchivedException
500	ENTRY_NOT_ACTIVATE	EntryNotActiveException
500	ENTRY_NOT_IN_WORKFLOW	EntryNotInWorkflow
500	EXPORT_ENTRIES_ERROR	DataExportException
500	EXPORT_ENTRIES_ERROR	ExportEntriesErrorException
500	EXPRESSION_INVALID_RETURN_TYPE	ExpressionExecutionReturnedInvalidTypeException
500	EXTRA_ENTRY_VALUE	ExtraEntryValueException
500	FALLBACK_OWNER_LAST_ITEM_OWNER_TYPE	FallbackOwnerLastItemOwnerTypeException
500	FIELD_SOURCE_REQUIRED	FieldSourceRequiredException
500	FILE_KIND_NOT_SUPPORTED	UnsupportedImportFileKindException
500	FILE_NOT_FOUND	FileNotFoundException
500	IMPORT_FILE_IS_INVALID	InvalidImportFileException
500	INVALID_CURRENT_WF_COLLECTION_OWNER	InvalidWorkflowOwnerCollection
500	INVALID_CURRENT_WF_GROUP_OWNER	InvalidWorkflowOwnerGroup
500	INVALID_CURRENT_WF_ORG_UNIT_OWNER	InvalidWorkflowOwnerOrgunit
500	INVALID_CURRENT_WF_OWNER	InvalidWorkflowOwner

Status	`error.code`	Originating exception class
500	`INVALID_DASHBOARD`	`DashboardValidationException`
500	`INVALID_DASHBOARD_GROUP_ID`	`InvalidDashboardGroup`
500	`INVALID_DATA_SOURCE_ACTION`	`DataSourceActionNotSupported`
500	`INVALID_ELEMENT_ID`	`InvalidElementID`
500	`INVALID_ENTRY_STEP_TYPE`	`InvalidEntryStepTypeException`
500	`INVALID_EXIT_STEP_TYPE`	`InvalidExitStepException`
500	`INVALID_ID`	`InvalidIdException`
500	`INVALID_ITEM`	`InvalidItemException`
500	`INVALID_ITEM_VALUE`	`InvalidItemValueException`
500	`INVALID_PROPERTY_VALUE`	`InvalidPropertyValueException`
500	`INVALID_QUERY_CRITERIA`	`QueryCriteriaNotSupported`
500	`INVALID_STEP_OWNER_TYPE_DATA_COLLECTION`	`InvalidStepOwnerTypeDataCollectionException`
500	`INVALID_STEP_OWNER_TYPE_GROUP`	`InvalidStepOwnerTypeGroupException`
500	`INVALID_STEP_OWNER_TYPE_USER`	`InvalidStepOwnerTypeUserException`
500	`INVALID_STRING_ITEMS`	`InvalidStringItemsException`
500	`INVALID_TREE`	`InvalidTreeException`
500	`INVALID_TREE_NODE`	`InvalidTreeNodeException`
500	`INVALID_TREE_NODE_PARENT`	`InvalidTreeNodeParentException`
500	`INVALID_TREE_NODE_VALUE`	`InvalidTreeNodeValueException`
500	`INVALID_VIEW_DEFINITION`	`DataViewExcpetion`
500	`INVALID_WORKFLOW`	`InvalidWorkFlowException`
500	`INVALID_WORKFLOW_OWNER_SOURCE`	`InvalidWorkflowOwnerSourceExption`
500	`IS_DEFAULT_DUPLICATE`	`DataViewIsDefaultException`
500	`MISSING_PROPERTY`	`MissingPropertyException`
500	`MISSING_REQUIRED_STEPS`	`MissingStepsException`
500	`MODULE_DUPLICATE`	`ModuleDuplicateException`
500	`MODULE_IS_IMPLICIT`	`ImplicitModuleException`
500	`NOT_SUPPORTED`	`DataNotSupportedException`
500	`PERMISSION_SCOPE_NOT_SUPPORTED`	`PermissionScopeNotSupportedException`
500	`PREVIOUS_OWNER_STEP_IS_EXIT_STEP`	`PreviousOwnerStepIsExitStepException`
500	`PREVIOUS_STEP_NOT_FOUND`	`PreviousStepNotFoundException`
500	`QUERY_EXPRESSION_ERROR`	`ExpressionExecutionException`
500	`QUERY_NOT_FOUND`	`DataSourceQueryNotExist` (reads of a stored reference)
500	`REDIRECT_TO_ENTRY_ACTION_MUST_ENABLE_SHOW_ENTRY`	`DataViewAfterSaveRedirectToEntryViewActionMustEnableEntriesConfiguration`
500	`RELATIVE_OPTIONS_INVALID_STEP_OWNER_TYPE`	`RelativeOptionsOwnerTypeException`
500	`RELATIVE_STEP_NOT_EXECUTED`	`RelativeStepNotExecuted`
500	`REQUIRED_FIELD`	`DataSourceQueryRequireException`

Status	error.code	Originating exception class
500	STORE_NOT_FOUND	DataSourceStoreNotExist
500	TRIGGER_DUPLICATE	TriggerDuplicateException
500	UNDEFINED_ITEM	UndefinedObject
500	UNDEFINED_NEXT_WORKFLOW_STEP_ACTION	UndefinedNextWorkFlowStepActionException
500	UNDEFINED_WORKFLOW	UndefinedWorkFlowException
500	UNDEFINED_WORKFLOW_ACTION	UndefinedWorkFlowActionException
500	UNDEFINED_WORKFLOW_ACTION_TARGET_STEPS	UndefinedWorkFlowActionTargetStepsException
500	UNDEFINED_WORKFLOW_STEP_ACTIONS	UndefinedWorkFlowStepActionsException
500	UNDEFINED_WORKFLOW_STEPS	UndefinedWorkFlowStepsException
500	VIEW_MUST_BE_DRAFT_FIRST_CREATION	DataViewFirstPublishException
500	WF_CAN_NOT_CHANGE_OWNER	CanNotChangeOwnerException
500	WORKFLOW_ACTION_COMMENT_REQUIRED	WorkFlowActionCommentRequired
500	WORKFLOW_STEP_CAN_NOT_MOVE	CanNotMoveStepException
501	BACKUP_NOT_SUPPORTED_IN_SINGLE_MODE	BackupNotSupportedException Since 1.10
503	BACKUP_SERVICE_UNAVAILABLE	BackupApiException (retryable; the facade also relays the backup engine's own codes with their status) Since 1.10
503	STORAGE_SERVICE_UNAVAILABLE	StorageServiceUnavailableException (retryable; also on saves that add a file Since 1.10)

Automation — reaction-service

51 typed classes. The EXECUTE_CUSTOM_ACTION_INVALID_CONFIGS and EXECUTE_CUSTOM_ACTION_INCOMPATIBLE_TARGET validation codes belong to the Execute Custom Action reaction *(Since 1.8)*. Several codes shared with the data service carry a different status here (ENTITY_FIELD_NOT_EXIST , ORG_UNIT_NOT_EXIST , STORE_NOT_FOUND are 500 in this service).

Status	error.code	Originating exception class
400	BLOCK_FIELD_NOT_EXIST	BlockFieldNotFoundException
400	COLLECTION_NOT_EXIST	CollectionNotExistValidationException
400	CUSTOM_EVENT_INVALID_CONFIGS	CustomEventInvalidConfigsException
400	ENTRIES_CONFIGURATIONS_MUST_BE_ENABLED	EntriesConfigurationMustBeEnabled
400	ENTRY_NOT_IN_WORKFLOW	EntryNotInWorkflowException
400	EXECUTE_CUSTOM_ACTION_INCOMPATIBLE_TARGET	ExecuteCustomActionIncompatibleTargetException
400	EXECUTE_CUSTOM_ACTION_INVALID_CONFIGS	ExecuteCustomActionInvalidConfigsException
400	FIELD_NOT_REPEATABLE	BlockFieldNotRepeatableException
400	FILE_EXPIRED	FileExpiredException Since 1.10
400	GROUP_NOT_EXIST	GroupNotExistValidationException
400	IMPLICIT_CREATE_RESTRICTION	ImplicitReactionCreateException
400	IMPLICIT_DELETE_RESTRICTION	ImplicitReactionDeleteException
400	IMPLICIT_UPDATE_RESTRICTION	ImplicitReactionUpdateException
400	INVALID_FIELDS_SOURCE	FieldSourceWrongTypeException

Status	`error.code`	Originating exception class
400	`INVALID_QUERY_EXPRESSION`	`DataQueryExpressionValueNotStringException`
400	`INVALID_QUERY_EXPRESSION_TYPE`	`DataQueryExpressionTypeValidationException`
400	`INVALID_TOKENS_MAP`	`InvalidTokensMapException`
400	`MUST_AUTOMATE_AT_LEAST_ONE_STEP`	`MustAutomateAtLeastOneStepException`
400	`QUERY_ENGINE_EXPRESSION_ERROR`	`ExpressionParsingException`
400	`RESERVED_HEADER_NAME`	`ReservedHeaderNameException` `Since 1.10`
400	`TRIGGER_TIMER_OPTIONS_CONFLICT`	`TriggerTimerOptionsConflict`
400	`TRIGGERS_CAN_NOT_AUTOMATE_LAST_STEP`	`CanNotAutomateLastStepException`
400	`TRIGGERS_LINKED_TO_NON_EXISTING_WORKFLOW`	`TriggersLinkedToNonExistingWorkflow`
400	`TRIGGERS_LINKED_TO_NON_EXISTING_WORKFLOW_STEP`	`TriggerLinkedToNonExistingWorkflowStep`
400	`TRIGGERS_LINKED_TO_NON_EXISTING_WORKFLOW_STEP_ACTION`	`TriggerLinkedToNonExistingWorkflowStepAction`
400	`UNKNOWN_OPERATOR_ERROR`	`UnknownOperatorExpression`
400	`UNSUPPORTED_FIELD_ERROR`	`UnsupportedFieldExpression`
400	`UPDATE_BLOCK_ITEM_ACTION_ON_FAILURE`	`UpdateBlockItemActionOnFailureException`
400	`UPDATE_ENTRY_MODE_NOT_SUPPORTED`	`InvalidUpdateEntryModeException`
400	`UPDATED_ENTRY_EVENT_CONFIG_INVALID_FIELD`	`UpdatedEntryEventConfigInvalidFieldException`
400	`VIEW_NOT_PUBLISHED`	`ViewNotPublishedException`
403	`REACTION_AUTHORING_FORBIDDEN`	`ReactionAuthoringForbiddenException` `Since 1.8`
404	`ENTITY_NOT_EXIST`	`EntityNotExistException`
404	`EVENT_SCHEMA_NOT_FOUND`	`EventSchemaNotFoundException`
404	`INTERNAL_ENTITY_NOT_FOUND`	`InternalEntityNotFoundException`
404	`REACTION_NOT_FOUND`	`ReactionNotFoundException`
404	`REACTION_SCHEMA_NOT_FOUND`	`ReactionSchemaFoundException`
404	`REFERENCE_FIELD_NOT_EXIST`	`ReferenceFieldNotFoundException`
404	`TRIGGER_CONTEXT_ENTITY_NOT_FOUND`	`TriggerContextEntityNotFoundException`
404	`WORKFLOW_ACTION_NOT_EXIST`	`WorkFlowActionNotExistException`
404	`WORKFLOW_NOT_EXIST`	`WorkFlowNotExistException`
404	`WORKFLOW_STEP_NOT_EXIST`	`WorkflowStepNotExistException`
409	`ANNOUNCEMENT_NAME_ALREADY_EXISTS`	`AnnouncementNameAlreadyExistsException`
409	`REACTION_NORMALIZED_NAME_DUPLICATE`	`ReactionNormalizedNameDuplicateException`
409	`UPDATE_CONFLICT`	`UpdateIdentifierConflictException`
409	`whatsapp_not_configured`	`WhatsAppNotConfiguredException` `Since 1.10`
422	`VALIDATION_FAILED`	`InvalidEnumValue`
500	`DATA_NOT_SUPPORTED`	`DataNotSupportedException`
500	`ENTITY_ARCHIVED`	`EntityArchivedException`
500	`ENTITY_FIELD_NOT_EXIST`	`EntityFieldNotExistException`
500	`EXPRESSION_INVALID_RETURN_TYPE`	`ExpressionExecutionReturnedInvalidTypeException`

Status	error.code	Originating exception class
500	FIELD_SOURCE_NOT_FOUND	FieldSourceNotFoundException
500	ORG_UNIT_NOT_EXIST	OrgUnitNotExistException
500	QUERY_EXPRESSION_ERROR	ExpressionExecutionException
500	STORE_NOT_FOUND	DataSourceStoreNotExist
502	whatsapp_provider_error	WhatsAppProviderException Since 1.10
503	STORAGE_SERVICE_UNAVAILABLE	StorageServiceUnavailableException (retryable) Since 1.10

The smaller services

The workflow, comments, timeline, notification and audit services each define a handful of typed classes on the same family pattern. The functions, logging and integrations services define only the generic families — their errors arrive as framework errors or with inline codes. The audit service is not routed at the gateway (`api-gateway/config/kong.yml` lists no audit route); its errors surface only relayed through other services.

`workflow-service`:

Status	error.code	Originating exception class
400	WORKFLOW_BULK_EXECUTION_NOT_ALLOWED	WorkflowStepsNotSameException
400	WORKFLOW_STEP_OWNER_CHANGE_TYPE_NOT_SUPPORTED	WorkflowStepOwnerChangeTypeNotSupported
403	FORBIDDEN_WORKFLOW_ADMIN_ACTION	ForbiddenWorkflowAdminActionException
404	DATA_ENTITY_NOT_EXIST	DataEntityNotExistException
404	DATA_VIEW_NOT_EXIST	DataViewNotExistException
404	EVENT_SCHEMA_NOT_FOUND	EventSchemaNotFoundException
500	EXPRESSION_INVALID_RETURN_TYPE	ExpressionExecutionReturnedInvalidTypeException
500	ITEM_ARCHIVED	EntityArchivedException
500	QUERY_EXPRESSION_ERROR	ExpressionExecutionException
500	REACTION_SCHEMA_NOT_FOUND	ReactionSchemaFoundException
500	UNDEFINED_ITEM	UndefinedWorkFlowException

`comments-service`:

Status	error.code	Originating exception class
400	COMMENT_ATTACHMENTS_EXTENSION_NOT_ALLOWED	CommentAttachmentExtensionNotAllowed
400	COMMENT_ATTACHMENTS_NOT_ALLOWED	CommentAttachmentNotAllowed
400	COMMENT_ATTACHMENTS_SIZE_NOT_ALLOWED	CommentAttachmentSizeIsNotAllowed
400	COMMENT_TEXT_NOT_ALLOWED	CommentTextNotAllowed
404	DATA_ENTITY_NOT_EXIST	DataEntityNotExistException
404	DATA_VIEW_NOT_EXIST	DataViewNotExistException
500	COMMENT_NOT_FOUND	CommentNotFound
500	COMMENT_UPDATE_NOT_ALLOWED	CommentUpdateNotAllowed
500	INVALID_COMMENT_AUDIENCE	InvalidCommentAudience

`timeline-service`:

Status	`error.code`	Originating exception class
400	`INVALID_PUBLISH_DATE`	`InvalidPublishDateException`
400	`INVALID_UNPUBLISH_DATE`	`InvalidUnpublishDateException`
400	`POST_MANUAL_PIN_SAME_PIN_STATE`	`DuplicatePinnedStatusPostException`
400	`POST_MANUAL_PUBLISH_IS_NOT_ALLOWED`	`PublishNotAllowedException`
400	`POST_MANUAL_UNPUBLISH_IS_NOT_ALLOWED`	`UnpublishNotAllowedException`
400	`TIMELINE_PUBLISH_DATE_CONFLICT`	`TimelinePublishDateConflictException`
400	`UNABLE_PINNED_MANUALLY`	`UnablePinnedPostManuallyException`
400	`UNABLE_SCHEDULING_ACTION`	`UnableScheduleManualPublishUnpublishPostException`
400	`UNABLE_UNPINNED_MANUALLY`	`UnableUnPinnedManuallyPostException`
404	`DATA_VIEW_NOT_EXIST`	`DataViewNotExistException`

`notification-api` (note: the constant `NOTIFICATION_MARK_AS_READ_EXCEPTION` emits the wire value `NOTIFICATION_MARK_AS_READ_ERROR`):

Status	`error.code`	Originating exception class
500	`NOTIFICATION_MARK_AS_READ_ERROR`	`MarkAsReadRequestEmptyException`
500	`NOTIFICATION_TEMPLATE_NOT_EXIST`	`NotificationTemplateNotExist`

`audit-service`:

Status	`error.code`	Originating exception class
404	`DATA_ENTITY_NOT_EXIST`	`DataEntityNotExistException`
404	`DATA_VIEW_NOT_EXIST`	`DataViewNotExistException`
500	`ITEM_ARCHIVED`	`EntityArchivedException`

Packaging and dependency

`packaging-service` and `dependency-service` define **no typed platform exceptions** (their error dictionaries are empty). They throw framework `HttpErrors` directly from their controllers, so `error.code` is usually a framework class name (`BadRequestError`, `ForbiddenError`, `NotFoundError`, `ConflictError`, `PayloadTooLargeError`) and the human-readable explanation is the message string in `result`. The marketplace and store routes are the exception: they attach their own codes to the framework error (for example `422 PUBLISHER_NAME_INVALID`, `502 MARKETPLACE_REFUSED`), documented on the packaging pages. `Since 1.10` a package install that would pass the tenant's entries limit is refused up front with `403 ENTRY_CAP_EXCEEDED`. Treat that string as diagnostic display text; branch on the status plus the endpoint you called:

Status	Service	Situation
400	both	Invalid parameters: an empty search string, an out-of-range `limit` (dependency search caps it at 1000), a mal-formed override payload, an invalid value-reference merge.
403	packaging	An export/import session polled from a different space than the one that owns it.
404	both	Unknown session, mapping profile, or rebuild job id.
409	packaging	The import-session state machine: executing a session that is not `validated`, retrying one that is not `completed` / `failed`, or a session modified concurrently.

Status	Service	Situation
409	dependency	A full rebuild is already in progress — the message names the in-flight job id to poll instead.
413	packaging	An import payload override exceeds the configured byte ceiling.

The packaging and dependency `409`s are the one place where a retry *after following the message's own instruction* is designed in: re-read the session or job status, then act on the state you find.

Authentication and step-up — `users-service`

Token, OIDC and account endpoints fail with standard OAuth 2.0 error objects — `invalid_grant`, `invalid_client` and friends, per the `oidc-provider` library — so handle them with an OAuth client library, not with envelope logic. See Token lifecycle.

The step-up unmask endpoints for sensitive fields *(Since 1.8)* return `{ "error": "<CODE>" }` with an explicit per-code status map:

Status	Code	When	Retry?
400	`INVALID_DURATION`	The requested unmask duration is not one of the allowed values.	No
400	`MFA_NO_METHOD`	The account has no usable MFA method.	No
400	`USER_NOT_FOUND`	The account could not be resolved.	No
400	`NO_PENDING_CHALLENGE`	Verify was called with no challenge outstanding.	No
403	`EXTERNAL_USER_FORBIDDEN`	External users may not open step-up sessions.	No
429	`MFA_RATE_LIMITED`	Too many challenge attempts.	Back off, then yes
503	`STEP_UP_UNAVAILABLE`	The unmask session store is unreachable — availability, not a client mistake.	Back off, then yes
500	`INTERNAL`	Unexpected failure.	Report

The retry policy

OSOS barely marks anything as retryable on the wire — the `retryable` flag is set only on a few `503`s — and, as shown above, most `500`s are deterministic domain failures. So the policy has to be written client-side, from status and code. This is the policy the catalogue supports.

What to retry, by status

Status	Policy
`400`, `403`, `404`, `415`, `422`	**Never retry unchanged.** The request, the credential's permissions, or the target is wrong; retrying reproduces the failure exactly. Two `400`s are fixed by redoing an earlier step rather than the request itself: `FILE_EXPIRED` (upload the file again) and `CONFIRMATION_INVALID` on a restore (ask for a new preview).
`401`	**Refresh once, retry once.** An expired token is the normal cause; run the refresh flow, then retry once. A second `401` means the credential is revoked or the API token invalid — stop and escalate.
`409`	**Re-synchronize, then re-attempt deliberately.** `UPDATE_CONFLICT`: re-fetch the entry and re-apply your change. Locks: wait for the holder. Packaging sessions and dependency rebuilds: poll the state or job the message names, then act on what you find. Never blind-retry a `409` on a timer.

Status	Policy
`429`	**Back off and slow down.** The gateway allows 100 requests/second and 10,000 requests/hour *per service*, counted across **all** callers of that service (`limit_by: service`, `api-gateway/config/kong.yml`) — so a `429` can be caused by aggregate load, not just your client. Honour a `Retry-After` header when one is present. A sustained bulk job that must stay inside the hourly window should pace itself to roughly 2–3 requests per second per service. The backup API's `429`s are daily quotas, not load: their `Retry-After` runs to the quota's reset, so do not retry sooner `Since 1.10`.
`500`	**Retry only infrastructure-shaped codes.** The gateway `500`s in the catalogue (`REDIS_ERROR`, the `*_REQUEST_FAILED` group, …) merit up to two spaced retries. A platform `500` carrying a catalogued code is deterministic — do not retry; report it. A `500` with a JS-class-name code (`TypeError`) is a server bug — report it with its `traceId`.
`502`, `503`, `504`	**Transient by nature** — a service is restarting or unreachable. Retry with full backoff. `STEP_UP_UNAVAILABLE` and `STORAGE_SERVICE_UNAVAILABLE` (`503`) are explicitly this category. `Since 1.10` `TENANT_IN_MAINTENANCE` (`503`) is too, but wait at least its `Retry-After` — a restore can take a while, and the same refusal repeats until it ends.

Network-level failures deserve the same split the platform applies internally — the classification is written into its own bus-retry logic: a connection that failed *before the request was sent* — refused, reset, DNS — never reached the server and is always safe to retry. A **timeout after sending is not**: the operation may have completed. For reads, retry freely. For writes, reconcile first — query for the entry you tried to create before re-creating it, or re-fetch the `updateIdentifier` before re-updating — and for bulk writes use the per-row result DTOs to resubmit only the rows that actually failed (Writing data).

Backoff shape

Exponential with full jitter, the standard shape:

```
delay = random(0, min(cap, base × 2^attempt))
base  = 1s     cap = 60s     max attempts = 5
```

- Apply it to `429`, `502` / `503` / `504`, pre-send network failures, and the two-attempt infrastructure-`500` case.
- When a `Retry-After` header is present (rate limiting, tenant maintenance, the backup API), treat it as a floor under the computed delay.
- `401` gets exactly one retry after a refresh — no backoff loop.
- `409` is event-driven, not timer-driven: the "delay" is however long the re-read or the lock release takes.
- Cap concurrency as well as rate: the per-service limits are shared, so ten workers backing off in sync can re-trip the limit together. The jitter is what prevents that.

When to stop and escalate

Stop retrying and open a report when any of these holds:

- The attempt budget is exhausted and the same status persists.
- A `500` carries a catalogued (deterministic) code, or any JS-class-name code.
- A `401` survives a successful token refresh.
- `UNIQUE_VALUE` appears for a value your system believes is unique — the conflict is real data, not a transient.
- A `403` appears on an operation that previously worked with the same credential — a permission or view configuration changed server-side; see The security boundary for what is enforced where.

Using `traceId` with support

Every request through the gateway carries a correlation id in the `X-CORRELATION-ID` header. The gateway's `osos-correlation-id` plugin **keeps a client-supplied value** and only generates a UUID when the header is absent — and it

echoes the id back on the response (`echo_downstream: true` in `api-gateway/config/kong.yml`). Downstream, every platform service reads the header case-insensitively and stamps it into `error.traceId`.

Use that deliberately:

- **Send your own id.** Generate a UUID per logical operation and send it as `X-CORRELATION-ID`; your logs and the platform's then share a join key you chose — even for requests that never produced an error body.
- **Log `error.traceId` on every failure**, together with the full error body, the HTTP status, the method and path (never the credential), the tenant host, and a UTC timestamp. That tuple is what support needs to find the request in the platform's logs.
- A `traceId` of `NO-TRACE-ID` is the services' fallback sentinel for a request that arrived without the header — if you see it, the request did not travel the normal gateway path. The one exception is `503 TENANT_IN_MAINTENANCE`, whose body is written directly by the tenant gate and always carries `NO-TRACE-ID`.
- If the error carries `serviceName`, it names the producer: `"API Gateway"` for edge rejections, a service name for relayed platform errors. Its absence means the error was produced by the service that answered you.

With the catalogue and the policy above, an error handler needs only three inputs — the status, `error.code` and `traceId` — and every one of them is stable, machine-readable and traceable end to end.

Generating a client

The OSOS platform ships a large, typed HTTP surface, and hand-writing request and response models against it is a losing proposition — the domain types are deep and there are hundreds of operations. The supported path is to **generate a client from each service's OpenAPI specification**. This page is about doing that correctly: where the specs come from, why a client generated from a *current* spec is right where an older one is subtly wrong, and the handful of responses a generated client will never model on its own.

It assumes you have read Calling the HTTP API, which owns the routing, versioning and envelope mechanics. This page does not restate them; it explains what they mean for a code generator.

Where the specifications come from

Each LoopBack service produces its **own** OpenAPI 3 document from its controllers. There is no merged, cross-service document — you generate one client per service and compose them in your own code. The two ways to obtain a spec are covered in The OpenAPI specification: the runtime endpoint (`{basePath}/openapi.json`, served when `ENABLE_SPEC` is `'true'`) and the offline export entry point (which boots the application and writes the document to a file).

Prefer a **committed snapshot** over hitting a live endpoint. Generating from a file you control means your build does not depend on a running cluster, and — the point of this page — it makes the spec's *provenance* something you can inspect and pin. A spec pulled live from whatever cluster happens to answer is a spec whose age you cannot see.

The current spec tells the truth

Every JSON response is enveloped as `{success, result}`, with no version condition — apart from `/v3` reads, raw again `Since 1.10` (see below). A code generator cannot infer that from an operation's schema, so the platform documents the envelope **in the spec itself**, through a spec enhancer. A client generated from a spec that carries the enhancer's output is generated against the real contract.

What the enhancer publishes

The enhancer augments each generated document in three machine-readable ways:

- A document-root extension key `x-osos-response-envelope`, so tooling can discover the contract without parsing prose. It names the envelope schema and the `result` / `error` property names.
- Two named component schemas, `OsosResponseEnvelope` and `OsosResponseEnvelopeError`. The names are deliberately namespaced rather than reusing the bare model name `ResponseBody`, because three distinct classes with that name exist in the codebase and a controller contributing its own would otherwise collide.
- A `## Response envelope` section appended to the document's `info.description`, stating the invariant in prose for a human reading Swagger UI.

Why operations still type the payload, not the envelope

Here is the nuance that decides whether your generated client is correct. The enhancer **does not** rewrite each operation to wrap its response schema in the envelope. An operation's documented 2xx schema still describes the `result` **payload — not the whole body**. The published `OsosResponseEnvelope` schema is deliberately never referenced by any operation; it exists to be *described*, not pointed at.

That design is intentional, and it is what keeps generation clean: referencing the envelope from every operation would change the generated result type of every method. Because the platform's own codegen configs ignore unused models, leaving the envelope schema unreferenced means the enhancer produces **zero change** in generated client output — the generated method signatures type the payload directly.

The consequence for you: a client generated from a current spec is **correctly typed at the payload level**, but its transport layer must still unwrap `result` for successful responses (and read `error` on failures). `result` is `null` when an operation returns no payload. Whether you write that unwrapping by hand or configure your generator to do it, the contract to code against is the single envelope — the same for every version of every operation, with the one `Since 1.10` exception of `/v3` reads described below.

Telling whether a snapshot predates the enhancer

The envelope became universal in a refactor that also added the enhancer. Before that, the wrapper was version-conditional and the spec said nothing about it at all. A client built from a **pre-enhancer snapshot** is therefore doubly unsafe: it documents no envelope, and it may have been wired against the old "`v3` is raw" behaviour. (`Since 1.10` that behaviour is back for `/v3` **reads** only — see The `/v3` read exception — so check which shape each operation actually answers with rather than trusting either snapshot.)

You can tell a snapshot's provenance by inspecting it for the enhancer's fingerprints. A **current** spec has all three; a **pre-enhancer** snapshot has none:

- the `x-osos-response-envelope` key at the document root;
- the `OsosResponseEnvelope` and `OsosResponseEnvelopeError` schemas under `components.schemas`;
- the `## Response envelope` heading inside `info.description`.

If a snapshot is missing them, do not generate against it — refresh the snapshot from a current backend first. A generated client is only as truthful as the spec it came from, and a stale one describes response shapes that the platform no longer produces.

The `/v3` read exception `Since 1.10`

On the 1.10 line a `GET` on a `/v3/` route is sent **raw** again — the bare payload, no {success, result} ; `/v3` writes and every `v1` / `v2` route keep the envelope (Calling the HTTP API). The enhancer's `## Response envelope` prose lists "`GET` on `/v3`" among the unwrapped cases, and for those operations the documented 2xx schema **is** the whole body. So a generated client must not unwrap `result` unconditionally: either special-case `GET` on `/v3`, or — the robust option, and what the portal does — unwrap only when the body carries `success: true`. The robust form also covers 1.8 deployments, which still envelope `/v3` reads up to 1.8.21.

Why the OpenAPI document is exempt from the envelope

The universal envelope has exactly one non-structural exception: the OpenAPI document itself is served **raw**, never wrapped (matched on a trailing `/openapi.json` or `/openapi.yaml` path segment). This matters directly to client generation, because your generator reads that document.

The reason is mechanical. Wrapping the spec in `{success, result}` would produce a document whose top-level `paths` object is buried one level down under `result` — which is no longer a valid OpenAPI document. Swagger UI would render nothing, and every generator configured to read the spec URL (the platform's own clients use `ng-openapi-gen`) would fail to find any operations. The exception is matched on a *trailing* path segment, so an ordinary API route that merely contains `openapi.json` somewhere in the middle is still enveloped like any other response.

Practically: point your generator at the spec URL and it receives a normal OpenAPI document. Point a normal API call at a data endpoint and it receives the envelope. The two never blur.

A narrower source: the per-view document `Since 1.10`

If your integration works against **one stable view** rather than the whole data plane, there is now a second, much smaller document to generate from. Turning on **Swagger document** in a view's display options publishes it for that view:

```
GET /api/v1/views/{viewId}/openapi.json
```

It is an ordinary OpenAPI 3.0 document — `ng-openapi-gen`, `openapi-generator` or any other OpenAPI-3-aware generator consumes it exactly like a per-service one. What makes it a better input for a scoped integration is what it leaves out. It describes exactly **five operations** — read the entity schema, list entries, read one entry, create an entry, patch an entry — with the view id and the entity id already **baked into each path** rather than left as template placeholders. The result is a client whose surface is the integration's actual surface, and whose paths do not need a template-substitution layer you write yourself. It is also more stable: the five operations are lifted from real controllers, so the document changes when those operations change, not every time an unrelated corner of `data-service` grows a route.

> ⚠ **The exemption applies to the delivery, not to the contract the document describes**
>
> The section above is about *this* endpoint too — and here the distinction bites, because both halves land in your generated client:
>
> - **The response is not enveloped.** `GET /v1/views/{viewId}/openapi.json` matches the same trailing-suffix rule, so your generator receives a bare OpenAPI document. Do not point it at `result`; there is no `result`.
> - **The document it hands you does describe the envelope.** It is run through the same enhancer as a per-service specification, so it carries `OsosResponseEnvelope`, `OsosResponseEnvelopeError`, the `x-osos-response-envelope` root key and the `## Response envelope` prose. Every one of the five operations it describes answers inside `{success, result}` at runtime, and your transport layer must unwrap `result` exactly as it does for a per-service client.
>
> Getting this backwards produces a parser that is wrong twice over: it fails to read the specification, then fails to read every response generated from it.

Two further constraints to design around:

- **Serving the document is gated per view.** With the toggle off, a non-privileged caller gets `403` — so a build step that fetches the document needs the toggle on for that view, and needs to keep it on.
- **The entity-schema operation always requires a bearer token**, even on a view that otherwise allows anonymous reads. A generated client that assumes one credential policy across all five operations will fail on that one.

The full endpoint contract — status codes, the `/v1/v/{viewId}` gateway prefix the documented paths carry, and what the entry `value` schema is built from — is in The per-view OpenAPI document.

Which responses bypass the envelope

Some responses are never `{success, result}` at all, and a generated client — which assumes JSON bodies — will not model them. These bypass the envelope **structurally**: the handler writes to the response itself, so the send pipeline returns early (`result === response || response.headersSent`) rather than wrapping anything.

- **File and binary downloads and streams.** Data exports (XLSX, PDF), theme CSS, translation files and fonts return binary or streamed content. Handle them as downloads; do not attempt to parse them as JSON. These are the same responses flagged under long-running operations.
- **The health, readiness and liveness routes** (`/health`, `/ready`, `/live`).

For these, drop out of your generated client and read the raw HTTP response. A client that assumes every response is enveloped JSON will fail precisely on the operations — like an export download — that an integration reaches for first.

Services with no spec

Generation only covers the LoopBack services that publish a specification. Three edge services are plain Express and emit **no** OpenAPI document at all: `users-service`, `storage-service` and `scheduler-service`. (`shortener-service` is no longer part of the backend — the short-link route is served by a separate shared function.) Between them they own authentication and file storage — the two surfaces an external client needs first — so a generated client cannot cover them, and they must be integrated by hand from their own guides. Keep a clear line in your codebase between the generated clients and these hand-written integrations, so a regeneration never silently overwrites hand-maintained code.

Four further services are LoopBack but have **no offline spec export**: `packaging-service`, `logging-service`, `dependency-service` and `integrations-service`. No committed snapshot contains their routes, so neither does the Endpoint reference — notably the packaging and store routes and the WhatsApp log and webhook routes. Their inventories are in Calling the HTTP API. Remember too that every service other than `data-service` sits under its own base path (`/api/reaction`, `/api/packaging`, …), and the paths in its specification are relative to it — configure each generated client's base URL accordingly (details).

Checklist

Before you trust a generated client:

- One client per LoopBack service, generated from that service's own spec — or, for a view-scoped integration, one client from that view's own per-view document `Since 1.10`.
- Generate from a **committed snapshot**, not a live endpoint, so provenance is inspectable.
- Confirm the snapshot carries the three enhancer fingerprints before generating; refresh it if not.
- Unwrap `result` (and read `error`) at the transport layer — the generated types describe the payload, not the envelope.
- Route downloads, streams and the health endpoints around the generated client; read their raw responses.
- Give each generated client its service's own base path — `/api` only for `data-service`.
- Integrate the three Express services, and the routes of the four LoopBack services with no export, by hand and keep them out of the regeneration path.

Environments and promotion

An external client is written once and run against more than one deployment — a development space, a staging space, production. This page is about what actually differs between those deployments from a client's point of view, so that promoting a client from one to the next is a change of configuration and never a change of code. It stays close to what the OSOS platform genuinely exposes; the generic parts of CI hygiene are kept brief on purpose.

What actually changes between environments

Less than you might expect. The API contract — routing, versioning, the response envelope, the error model — is identical across deployments; that is the whole point of documenting it. What changes per environment is narrow:

- the **base URL** the client talks to;
- the **credential** the client presents.

Everything a client holds should reduce to those two values. If anything else has to change to move between environments, treat that as a defect in the client's configuration boundary, not a fact of the platform.

The base URL is the tenant

You do not address services individually. Every call goes to the gateway, and the gateway resolves *which* tenant you are talking to from the **request host** — the `domain-resolver` and `space-resolver` plugins look the incoming `Host` up against the registered domains and stamp the tenant's `x-space-id` onto the request before any service sees it.

The practical consequence: the base URL is not merely an address, it *is* the tenant selector. A development space and a production space are reached at different hosts, and pointing a client at a different host silently changes which space's data it reads and writes. So:

- The base URL belongs in configuration, never hard-coded.
- There is no `x-space-id` for a client to set — the gateway derives it from the host, and a client that tries to set tenancy headers itself is both ineffective and a red flag (see The security boundary).
- Confirm the host before a write. Promoting a job that was pointed at staging to run against a production host is a data incident, not a config typo.

Credentials are per-environment

A credential minted in one space does not — and must not — work in another. Credentials are the second and last thing that changes on promotion.

Application tokens

The simplest headless credential is an **application token**, minted for a dedicated application user and presented as the `x-app-token` header. A token is issued per application user per space, so each environment has its own. The token carries no permissions of its own; its access is exactly the owning application user's roles, membership and view permissions in *that* space (see Choosing a credential). Provision a separate application user, with least privilege, in each environment.

OAuth client credentials

When the client acts on behalf of an interactive user, it uses the OAuth 2.0 / OIDC path served by `users-service`, with a client id and secret registered for that deployment. Those, too, are per-environment values: a staging client secret is not a production client secret. They belong in the same configuration boundary as the base URL and are subject to the same rule below about never being baked into an artifact.

What stays server-side

Some secrets belong to the platform's *internal* trust and must never reach a client at all. The **shared key** (`SHARED_KEY`) is the preshared secret the gateway uses to authenticate itself to downstream services; it is required in any non-development environment and the process refuses to start without it (`getSharedKey`). A development fallback value exists for local runs only.

A client never holds the shared key, never sets identity headers such as `x-user-id` or `x-space-id`, and never talks to a service port directly — all of that is the gateway's job, and a client in a position to do any of it is a boundary that has already failed. The security boundary covers why in full. When you promote a client, the only material it carries between environments is a base URL and its own credential; the server-side trust material is configured on the platform and stays there.

Never bake a credential into an artifact

An application token is bearer material: whoever holds it acts as its application user. Treat it exactly as you would a password.

- Never commit a token or client secret to source control, and never bake one into a built artifact — a container image, a compiled bundle, a published package. An artifact is copied, cached and shared; a credential inside it leaks with it.
- Inject credentials at deploy time from the environment or a secrets manager, keyed by environment. The base URL travels the same way.
- Rotate on a schedule and on any suspected exposure. Application tokens support rotation with a grace window so an in-flight token keeps working while callers pick up the new one — see Token lifecycle; design your deployment so a rotation is a config push, not a redeploy.

A promotion pattern

A clean setup keeps code identical across environments and varies only configuration:

- One build artifact, promoted unchanged from staging to production. If the artifact differs per environment, a credential or URL has leaked into it.
- Base URL and credential supplied as environment configuration or mounted secrets, selected by the target environment at deploy time.
- A pre-promotion check that the resolved host is the intended one, so a client cannot run against the wrong tenant.
- Distinct, least-privilege application users per environment, so revoking or rotating one never touches another.

Checklist

- Exactly two things change between environments: the base URL and the credential.

- The base URL selects the tenant — verify the host before any write.
- Separate, least-privilege credentials per environment; never reuse across spaces.
- The shared key and all identity headers stay server-side; a client never holds or sets them.
- No credential in source, in an image, or in any built artifact — inject at deploy time.
- Rotate credentials as configuration, using the grace window to avoid downtime.

The security boundary

This page describes a real trust boundary in the OSOS platform's architecture, not a vulnerability. Understanding it correctly is what lets an operator deploy the platform safely and a client author reason about what their code may and may not hold. The short version: **authentication happens once, at the gateway. Everything behind the gateway trusts identity headers.** Get the boundary right and the design is sound; expose a service past the gateway and the same design becomes an open door.

Where tokens are checked, and where they are not

An external caller presents a bearer token (OAuth 2.0 / OIDC) or an application token. That credential is validated **at the edge** — the gateway's `oauth2-introspection` and `app-token-auth` plugins do the validation (`api-gateway/kong/plugins/`).

Downstream services do **not** re-validate the token. They never see it. A LoopBack service's authentication strategy resolves the caller's identity entirely from **request headers** that the gateway sets after it has authenticated the caller, and it authorises the request only when a valid preshared key accompanies those headers. There is no second, independent check of the original token behind the gateway. This is a deliberate performance and separation-of-concerns choice — the edge authenticates, the services authorise — and it is correct **as long as nothing can reach a service except through the edge.**

The identity headers services trust

After the gateway authenticates a caller, it forwards identity as a set of headers. The service's strategy reads them directly:

Header	Meaning
`x-shared-key`	The preshared trust key between gateway and service. The request resolves to a user profile **only** if this matches the service's configured key.
`x-user-id`	The authenticated subject. The loaded profile, roles and permissions are all derived from this id.
`x-space-id`	The tenant / space. The same operation returns different data per space.
`x-client-id`	The OAuth client id of the caller.
`x-is-delegation` , `x-delegatee-id` , `x-delegation-id`	Act-on-behalf-of (delegation) context.
`x-service`	The calling service's name. A service can treat `API Gateway` here as proof that the request came through the gateway.
`x-view-id`	The view the request is made through. A service authorises a request carrying it as made through that view.

The consequence is the whole of this page: to a downstream service, **identity is whatever these headers say it is.** A request that arrives carrying `x-shared-key` set to the service's key and `x-user-id` set to any user's id is, as far as that service is concerned, that user — with that user's full permissions, in the space named by `x-space-id` . No token is required, because tokens are an edge concern. This is exactly why the boundary must never be exposed: the header pair `x-shared-key` + `x-user-id` is sufficient to act as anyone.

How the gateway protects the headers

The design is safe because the gateway makes those headers un-spoofable from the outside. Two mechanisms do this, and understanding them is what makes the deployment rule below non-negotiable.

The shared key is stripped and re-injected

The gateway **removes any client-supplied** `x-shared-key` from every inbound request and then re-adds the correct value itself (`request-transformer` plugin in `config/kong.yml`). Crucially, it only injects the *real* key when the request authenticated: an unauthenticated request has its identity headers cleared and a marker set, so the value re-injected as `x-shared-key` is a placeholder rather than the real key, and the downstream service — seeing a key that does not match its configured one — resolves to **no user** and rejects the call (`checkSharedKey`). A caller therefore cannot smuggle in a valid shared key, and cannot obtain one by simply omitting a token.

Identity headers are cleared and re-set from the validated token

The `oauth2-introspection` plugin **clears** `x-user-id`, `x-client-id` and `x-session-id` on every request and re-sets them from the introspected token's own claims. A client-supplied `x-user-id` is discarded before it can be trusted; the id that reaches the service is the one the token actually authenticated as. The application-token plugin behaves the same way, setting `x-user-id` from the validated token's application user. So the identity a service receives is always the gateway's word, derived from a credential it checked — never the client's assertion.

The deployment rule

Everything above holds on one condition, and it is the single most important operational fact on this page:

> **A downstream service port must never be reachable except through the gateway.**

Firewall every service so that only the gateway can connect to it. The headers that carry identity — above all `x-shared-key`, `x-user-id`, `x-space-id`, `x-service` and `x-view-id`, and every other internal header the gateway sets — are trusted implicitly by the service. If any network path lets a caller reach a service directly, that caller can set those headers itself and impersonate any user in any space, bypassing authentication entirely. The gateway's stripping and re-injection protect requests that go *through* it; they protect nothing about a request that goes *around* it. Name these headers in your network policy and block them at any ingress that is not the gateway.

What a client may hold, and what stays server-side

The boundary defines a clean split of what belongs where.

A client legitimately holds only:

- its **base URL** (the gateway host);
- its **credential** — a bearer token, or an application token sent as `x-app-token`.

A client must **never** hold or set:

- the **shared key** (`x-shared-key`) — it is the gateway↔service trust secret and lives only server-side (`SHARED_KEY`; see Environments and promotion);
- any **identity header** — `x-user-id`, `x-space-id`, `x-client-id`, `x-service`, `x-view-id`, and any other internal header. These are the gateway's to set from the validated credential and the request path. A client that sets

them is either being ignored (through the gateway, they are overwritten) or is talking to an exposed service (around the gateway, they are a breach).

If you find client code setting `x-shared-key` or `x-user-id`, that is the signal to stop: either it is dead code, or it is evidence that a service is reachable off-gateway.

Delegation

Delegation lets a request act as someone other than the immediate caller, and it is strictly gateway-mediated for the same reason as the rest.

Delegation (`x-is-delegation`, `x-delegatee-id`, `x-delegation-id`) lets a caller act on behalf of another user, with an audit trail. The client names the delegation with `x-delegation-id`; the gateway's `user-delegation` plugin validates the delegation record — that it exists, that the caller is its delegatee, and that the current time falls within its active window — and on a validated delegation sets `x-delegatee-id` to the caller, rewrites `x-user-id` to the delegator and sets `x-is-delegation: true`. A delegation that belongs to another user, or is expired or not yet active, is rejected at the edge with a `403`.

Summary

- The gateway authenticates; services authorise from trusted headers. Tokens are never re-checked downstream.
- `x-shared-key` + `x-user-id` is sufficient to impersonate any user — to a service, identity is whatever the headers say.
- The gateway strips client-supplied `x-shared-key` and identity headers and re-injects them from the validated credential, which is what makes the design safe.
- That safety depends entirely on services being unreachable except through the gateway. Firewall the identity headers at every non-gateway ingress.
- A client holds a base URL and a credential — nothing else. The shared key and all identity headers stay server-side.

Workflow model and API reference

This page is for **integrators and developers** who read or write workflow definitions through the API, drive workflow transitions from another system, or build an external system that a workflow step hands entries to. It documents the stored model, its numeric enums, the runtime contracts and the endpoints.

Designing a workflow in the UI?

Builders designing approvals in the entity designer should start with Workflows & approvals [Build Apps] — it covers the same features in the labels the designer shows. Runtime execution as end users experience it (who may act, moving, returning, taking ownership) is in Executing workflow actions [Using OSOS › Acting on Workflow Steps].

Where a workflow lives

When an entry advances, the workflow engine in `data-service` and `workflow-service` reads that embedded definition to decide the next step and its owner, then records progress in per-entry state (`_metaCurrentSteps` , `_metaWorkflowHistory`).

> **Enums are numeric.** Almost every discriminator on a workflow (step type, owner type, source type, target-step type, …) is a numeric enum persisted as an integer. This page lists the integer value next to each name. When you build a definition through the API you send the **number**, not the label.

Workflow **definitions** are edited as part of the Data Entity (its `workflows` array); there is no standalone "create workflow" endpoint. See Developer API surface for the query and runtime routes.

Workflow definition (`DataEntityWorkflow`)

A workflow is an ordered sequence of steps that models the stages an entry passes through. In the designer you add one on the entity's **Workflows** tab; two steps are created by default (named *Initial Step* and *Complete Step*), and you insert the steps you need between them before saving.

Field	Type	Required	Notes
`id`	UUID	yes	Inherited from `Identified<UUID>` .
`name`	`LocalizedValue`	yes	Display name, localized.
`normalizedName`	string	yes	Machine name; validated against `NormalizedNameValidation` (min/max length + pattern).
`isDefault`	boolean	yes	Marks the entity's default workflow.
`steps`	`WorkflowStep[]`	yes	The ordered set of steps (see below).

An entity can hold multiple workflows; an entry tracks which workflow instance is active via the `active` flag on its `_metaCurrentSteps` entries (see `getActiveWorkflowInstance`). Legacy entries that predate the `active` flag fall back to the first current-step entry.

Steps (`WorkflowStep`)

A step is the fundamental element of a workflow: a stage where an owner is assigned and actions are executed. Steps form an **ordered array**, and their order is what drives relative navigation (first, previous, current, next, last).

Field	Type	Notes
`id`	UUID	Step identifier. Referenced by action target steps, owner sources, etc.
`name`	`LocalizedValue`	Localized display name.
`normalizedName`	string	Machine name, same validation as workflow.
`description`	nullable localized	Optional.
`type`	`StepType` (number)	See step-type table below.
`actions`	`WorkflowAction[]`	Transitions out of this step. Defaults to `[]`.
`owner`	`WorkflowStepOwner`	Owner-assignment configuration (optional).
`allowAddAdHoc`	boolean	Whether ad-hoc steps may be inserted after this step. Defaults to `false`.
`inMainFlow`	boolean	Whether the step is part of the main / happy path. Defaults to `true`.
`mainFlowIndex`	number	Position in the main flow (used for the "main-flow status" progress views).
`fallbackOwner`	`WorkflowStepOwner[]`	Ordered fallbacks used when the primary owner cannot be resolved (e.g. no direct manager, previous step not executed). Defaults to `[]`.
`timeMonitoring`	`WorkflowStepTimeMonitoring`	Per-step SLA / time-to-execute config (optional).
`subWorkflowId`	string	ID of a workflow (in the **same** entity) to run as a sub-workflow.
`mergeStep`	`NullableActionTargetStep`	Step in the **parent** workflow to resume at when the sub-workflow reaches its Exit step.

The Entry (start) and Exit (end) steps are created by default and cannot be deleted; their name, description, and field requirements can still be edited.

The step's display `name` is a multilingual value constrained to a **minimum of 3 and a maximum of 128 characters**; its `normalizedName` follows the machine-name validation noted for the workflow above.

Main path and sub-paths

The steps of a workflow form a **main path** — the primary sequence of steps followed to reach the goal — and, optionally, one or more **sub-paths**. A sub-path is an alternative branch taken when the main-path conditions are **not** met (for example a rejection, or a conditional route to a different step). The `inMainFlow` and `mainFlowIndex` fields on each step record whether the step belongs to the main path and its position along it, which is what drives the main-flow progress views.

`Since 1.8` — when the entry currently sits on a step that is **off** the main path, the progress view predicts the remaining steps by following that step's main action. A **relative** target (Next, Previous, First, Last) on that action is resolved the same way an absolute step id is, so the prediction resumes from the right point. When the target cannot be resolved to a main-path step, the view reports **no** upcoming steps rather than re-listing the whole main path.

Step types (`StepType`)

Value	Name	Meaning
0	`Entry`	Start of the workflow, where newly created entries land. Its owner is always the entry's creator (see the resolution engine).
1	`Exit`	Terminal step that marks the process complete. Owner resolution is relaxed for Exit (no `OriginalOwnerNotFoundException`).
2	`Intermediate`	A normal in-between step, such as a review or approval stage.
3	`AdHoc`	A step inserted at runtime for an additional approval or additional information.
4	`SubWorkflowStep`	A step that delegates execution to another workflow via `subWorkflowId`.
5	`ExternalStep`	A step that hands the entry to an external system and completes only when that system calls back. See External step. `Since 1.8`

In the designer's **Step Type** picker these appear as **Normal** (Intermediate), **Sub-Workflow Step** and **External Step**; Entry and Exit are the two default steps, and ad-hoc steps are only inserted at run time.

`Since 1.8` — an entry sitting on the Exit step is, by itself, proof the workflow is **complete**. A separate `lifeCycleStatus` flag also flips to mark completion, but that update runs asynchronously; the status endpoints do not wait for it.

A workflow must start with an Entry step and end with an Exit step. Relative movement (First, Previous, Current, Next, Last) is resolved from a step's position in the sequence, so step order is significant.

Sub-workflows `Since 1.4`

A **Sub-Workflow step** delegates an entry into a separate, nested workflow and then resumes the parent flow when that nested workflow finishes. This lets you reuse a common process (for example a standard "budget approval") inside many larger workflows without rebuilding it each time. A `SubWorkflowStep` references another workflow in the **same** entity through `subWorkflowId`; the referenced workflow runs in its entirety, and when it reaches its `Exit` the engine resumes the parent at `mergeStep`.

Design rules for a sub-workflow step

Because a Sub-Workflow step hands the entry off to another workflow rather than doing work itself, the builder enforces special rules:

- It has **no actions of its own** — the transition happens automatically when the nested workflow completes. Adding actions to a Sub-Workflow step is rejected.
- It must reference an existing **sub-workflow** defined on the same entity, and that sub-workflow's first step must be an Entry step.
- It must define a **merge step**: the step in the parent workflow the entry returns to when the sub-workflow completes. The merge step can be a specific step or a relative one (First, Previous, Current, Next, Last).

How an entry crosses a sub-workflow boundary

- **Entering — the parent suspends.** When an entry transitions onto a Sub-Workflow step, the parent workflow is suspended and the entry starts the nested workflow at its Entry step. The parent's Sub-Workflow step stays pending until the nested workflow finishes.

- **Completing — the parent resumes at the merge step.** When the nested workflow reaches its Exit step, the entry returns to the parent at the configured merge step, the Sub-Workflow step is marked executed, and the merge step's owner is calculated at that moment. The nested workflow is then closed and removed.
- **Chaining and cascading.** If a merge step is *itself* a Sub-Workflow step, the next sub-workflow starts automatically; and if completing a sub-workflow lands the entry on the Exit step of a parent that also belongs to a higher workflow, completion cascades upward until it reaches a workflow that is not nested.
- **Reactions on the boundary** `Since 1.8`**.** The step being left when a sub-workflow starts, and the sub-workflow's own step when an action inside it exits back to the parent (for example a "Return" or "Reject"), both report their own step-exit progress, in the standard order, so a reaction configured on the action or step being left at either boundary fires as designed.

Design a Sub-Workflow step's SLA with the whole nested duration in mind: the parent's countdown on that step keeps running for the entire time the entry is inside the sub-workflow, and is satisfied only when the parent resumes.

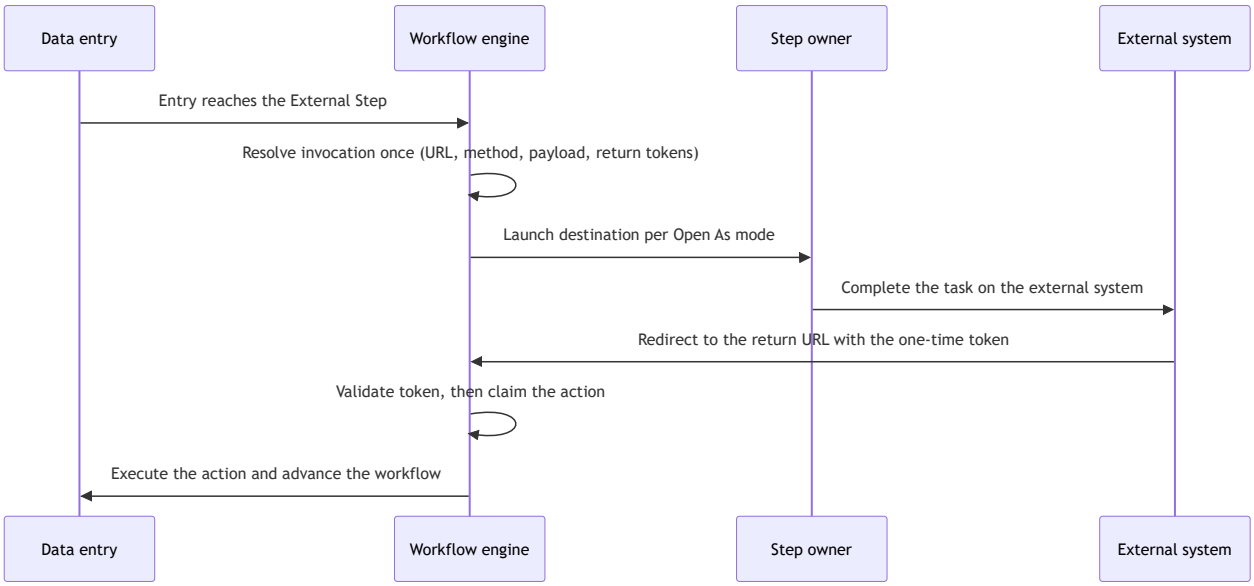
Owner inheritance at the sub-workflow entry step

Normally an Entry step is owned by the entry's creator. Inside a sub-workflow this is different: the sub-workflow's Entry step **inherits the owner already computed for the parent Sub-Workflow step**, not the creator. If the parent Sub-Workflow step is configured for parallel approval or for manual ("take ownership") selection, the sub-workflow Entry step is instead left **unassigned** so that an eligible user can claim it.

`Since 1.8` — this inheritance also holds on a **later transition back** onto the sub-workflow's own Entry step (for example a "Return" taken from inside the sub-workflow), not only when the sub-workflow first starts. The owner falls back to the entry creator only when the parent linkage can't be resolved or the inherited owner isn't usable (for example an automated/system-triggered return with nothing configured to inherit).

External step `Since 1.8`

An **External Step** (`StepType.ExternalStep`, value `5`) hands the entry off to an outside system — an e-signature service, a payment gateway, an external portal — and resumes only when that system calls back. None of its actions can be completed by clicking a button in the portal: the return callback (or an automation reaction) is the only way the step advances.



Configuration (`ExternalStepConfig`)

The step's external behaviour is configured on `WorkflowStep.externalStepConfig`, which is required whenever `type = ExternalStep`.

Field	Type	Required	Notes
`url`	string	yes	The destination. May contain `{{token}}` placeholders resolved when the entry reaches the step. Labelled **URL** in the builder.
`method`	`ExternalStepHttpMethodEnum`	yes	`0 = GET`, `1 = POST`. Labelled **Method**.
`openAs`	`ExternalStepOpenAsEnum`	yes	How the destination is presented — see Open As modes. Labelled **Open As**.
`payloadEnabled`	boolean	no	Default `false`. Turns on the **Custom Payload**; only meaningful with `POST`.
`payload`	string	no	The custom payload: JSON text with `{{token}}` placeholders, parsed into an object once, at resolution time.
`tokenBindings`	`Record<string, TokenSource>`	no	One `TokenSource` (`type` + `source`) per `{{token}}` placeholder used in `url` / `payload`. Defaults to `{}`.

A `POST` step with no custom payload sends a default body of `{entry: <entry details>, actions: [{id, name, returnUrl}, …]}` — the same entry shape the entry-details read endpoint returns, not the raw stored document.

Open As modes

`openAs` decides how the owner is taken to the destination. There are five modes:

Value	Label	What the owner sees
0	Full Screen	The browser navigates straight to the external destination.
1	Iframe	The external page loads inline, in a resizable frame on the entry itself.
2	Sidebar	The external page opens in a side drawer panel.
3	Dialog	The external page opens in a modal dialog.
4	Popup Window	The external page opens in a new browser window or tab.

The web portal supports all five. The **mobile app supports Full Screen and Iframe only**, and falls back to Full Screen for Sidebar, Dialog, and Popup Window — so a step that must look the same on both clients should be designed for one of the two supported modes.

Return URL and value substitution

The `url` (and, for `POST`, the `payload`) is resolved **exactly once**, when the entry reaches the step. Two kinds of placeholder are substituted:

- **Return-URL placeholders.** `{{@returnUrl}}` embeds the return link for the step's single action; `{{@returnUrl:actionId}}` embeds the return link for one named action. A bare `{{@returnUrl}}` is auto-substituted only when the step has exactly one action — on a step with none or several it is left unresolved rather than guessing which action is meant. The builder shows the hint *"Use `{{@returnUrl}}` in the URL or payload above to embed the return link (or `{{@returnUrl:actionId}}` if this step has more than one action)."* Each action's **Action Id** is shown read-only on the action form with a copy-to-clipboard control, so you can paste it into the placeholder.

- **Token bindings.** Every other `{{token}}` in the URL or payload is resolved from `tokenBindings` against the entry. The two reserved return-URL placeholders are deliberately filtered out of the bindable-token list so they cannot be bound as ordinary tokens by mistake.

Values substituted into the **URL** are URL-encoded before insertion, unless the placeholder occurrence *is* the whole trimmed URL (you configured the return link as the entire destination rather than as a value nested inside a larger URL). This keeps a substituted value that is itself a URL from corrupting the outer query string. Payload substitution stays unencoded, because a payload is raw text or JSON, not a URL.

The return contract

Every action on every external step returns through one endpoint, `POST /external-step/return`, whose body is `{ token }`. The endpoint is **intentionally unauthenticated** — the caller is an external system redirecting the user's browser, not a signed-in portal session, and the opaque one-time token is itself the credential. Integrations redirect the user back to the frontend route `{APP_BASE_URL}/external-step-return?token=…`, which calls the endpoint on their behalf.

One token is minted **per action**, once, when the invocation is resolved. It is an opaque, unsigned string: a base64url prefix identifying the entry, workflow, instance, step, and action, followed by a random secret. The secret is compared in constant time against the value persisted on the entry's current step, so a token cannot be guessed or replayed from another action.

Return-token TTL `Since 1.8` — a return token expires a fixed period after the step was launched. The window is configurable per environment and defaults to **24 hours**. A missing or unreadable launch timestamp **fails closed**: the return is rejected rather than accepted with an unbounded lifetime.

A return resolves to one of four outcomes: it succeeded and the action was queued; it was already processed (the entry has moved on, or a near-simultaneous return already claimed the action — an idempotent no-op, not an error); it was invalid (malformed, expired, mismatched, or the entry/step is no longer eligible); or the action was claimed but dispatching the workflow action failed.

Rules the builder enforces

Saving a step of type `ExternalStep` is validated both in the builder and on save:

Rule	Rejected with
`url` (non-blank), `method`, and `openAs` are all required.	`ExternalStepConfigRequiredException`
The step's owner must resolve to exactly one specific user — owner type **User**, source **Fixed**, pointing at the `USER` entity. Group, dynamic, and expression owners are rejected at save time.	`ExternalStepOwnerMustBeUserException`
A custom payload (`payloadEnabled = true`) is only allowed when `method = POST`.	`ExternalStepCustomPayloadRequiresPostException`

The builder mirrors these rules client-side: switching a step to External Step defaults `method` to `GET` when unset, requires URL / Method / Open As, clears the custom payload when the method is not `POST`, and restricts the owner picker to a single fixed user.

Because that fixed user is a **required** owner reference, it is also protected when a workflow travels between environments: a package import that would clear a required fixed step-owner reference is stopped by a pre-flight check rather than failing mid-import — the same invariant that makes a `null` fixed primary owner fail entity save.

Invalidation and guard rules

- **Manual completion is blocked.** Executing an external step's action from the portal is refused unless the command comes from the return callback or from an automation reaction. The end-user portal never renders the step's own action buttons.
- **Archived, invalid, and obsolete entries cannot complete the step.** The return path requires the entry to be Active and not invalid, and re-checks that condition a second time as part of the write that claims the action — closing the window in which an entry could be archived or invalidated between the two.
- **The live step type is re-asserted on return.** Even with a valid, unclaimed token, the return looks up the entry's *current* workflow definition and requires the step to still be an External Step. Retyping the step (back to Intermediate, say) while a token is outstanding stops that token from executing anything.
- **A stale invocation is cleared on every transition.** The resolved invocation is deleted whenever the entry leaves a step — not only when it leaves through this step's own actions — so an external step reached later through a cross-workflow jump or an administrative move always gets a freshly resolved URL, payload, and set of return tokens.

Transitions and actions (`WorkflowAction`)

Actions are the buttons a step owner sees; each action routes the entry to a target step. Every non-Exit, non-Sub-Workflow step needs at least one action, and a step may have at most one primary (main) action.

Field	Type	Notes
`id`	UUID	Action identifier.
`name`	`LocalizedValue`	Button label.
`description`	nullable localized	Optional.
`isMainAction`	boolean	Marks the primary / happy-path action. Used by group-decision voting (see `mainActionExecutionCondition`).
`dialogOptions`	`WorkflowActionDialogOptions`	Comment / confirmation dialog behaviour.
`targetStep`	`ActionTargetStep`	Where the entry goes when this action is executed.
`message`	`WorkflowActionMessage`	Optional message shown after execution (`enabled` + localized `value`).
`visible`	`FieldHiddenOption`	Optional visibility rule — the action only appears when its condition is met.
`conditionalTargetSteps`	`ConditionalTargetStep[]`	Expression-driven overrides of `targetStep`. Optional.
`stepUp`	`StepUpRequirement`	Optional — makes the executor re-confirm their identity before this action runs, for a single entry or — `Since 1.10` — for a whole gated bulk selection. See Step-up verification before executing an action. `Since 1.10`

Target step (`ActionTargetStep`)

Field	Type	Notes
`isRelative`	`TargetStepTypeEnum` (number)	`0 = Relative`, `1 = Absolute`, `2 = Dynamic`.
`targetStep`	`RelativeStepEnum` | UUID	For Absolute: a step UUID. For Relative: a relative token.

A **fixed (absolute)** target names a specific step by id (for example “direct manager approval”). A **relative** target names an unknown step by position. Relative tokens (`RelativeStepEnum`) are sentinel strings resolved at runtime:

Token	Value
Next step	`___NEXT_STEP___`
Previous step	`___PREVIOUS_STEP___`
First step	`___FIRST_STEP___`
Last step	`___LAST_STEP___`
Current step	`___CURRENT_STEP___`

Built-in Main Action, Return and Reject

The **Create New Step** dialog creates the step’s first actions for you:

- **Main Action** (required) — an action named *Main Action* with `isMainAction = true` , targeting the step you pick (a relative step or an existing step).
- **Return** (checkbox) — an action named *Return* whose target is the relative **First step** (`___FIRST_STEP___`), i.e. back to the Entry step.
- **Reject** (checkbox) — an action named *Reject* whose target is the relative **Last step** (`___LAST_STEP___`), i.e. the Exit step, ending the workflow.

All three are ordinary `WorkflowAction` s once created; rename them or retarget them like any other action. Return and Reject also appear as `failureActions` options for group-decision voting.

Conditional target steps (`ConditionalTargetStep`) `Since 1.4`

Beyond a single fixed or relative target, an action can route to **different next steps depending on a condition** evaluated at runtime. When present these are evaluated **in** `order` ; the **first** condition that evaluates to `true` supplies the target step, overriding the action’s static `targetStep` . If none match, the action falls back to its default `targetStep` .

Field	Type	Notes
`order`	number	Evaluation order (ascending).
`condition`	string	An expression that must evaluate to boolean.
`targetStep`	`ActionTargetStep`	Target used when the condition is true.

Dialog options (`WorkflowActionDialogOptions`)

Field	Type	Default	Notes
`commentValidationMode`	`WorkflowActionCommentValidationModeEnum`	`Optional (0)`	`0 = Optional` , `1 = Required` , `2 = Disabled` (labelled **Hidden** in the designer).
`showConfirmationDialog`	boolean	`false`	Prompt before executing.
`title`	nullable localized	—	Dialog title.
`message`	nullable localized	—	Dialog body.

Step-up verification before executing an action `Since 1.10`

An action can be configured to make the person executing it **confirm who they are, again**, in the moments before it runs — independently of the session they are already signed in under. The setting lives on the **action** (`stepUp`), not on the step and not on the workflow, so two actions on the same step can differ: gate *Approve payment*, leave *Request more information* alone.

It has two parts: **turn it on**, and, optionally, **give an expression** that decides per entry.

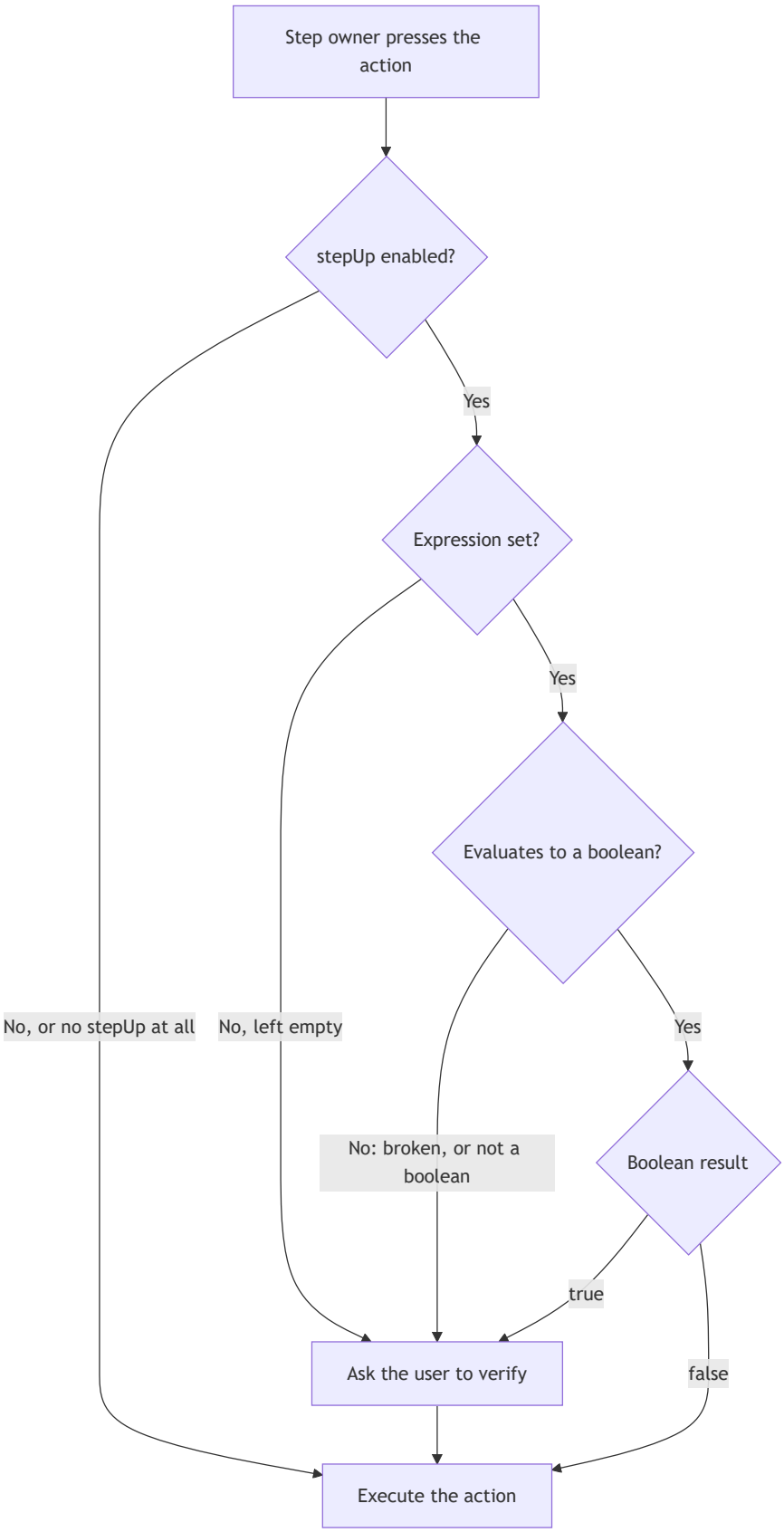
Field	Type	Default	Notes
`enabled`	boolean	`false`	Turns the requirement on for this action.
`expression`	string | null	`null`	Optional condition, evaluated against the entry being acted on.

The three behaviours, stated plainly:

How it is configured	When the verification is required
Off — `enabled` is `false`, or the action carries no `stepUp` at all	**Never.**
On, expression left empty	**Always.**
On, with an expression	**When that expression is true** for the entry being acted on.

An action with no `stepUp` behaves exactly as it did before this feature existed: nothing to migrate, nothing to switch off.

> ⚠ **A broken expression requires the verification — it does not skip it**
>
> An expression that cannot be evaluated, or that returns something other than a boolean, makes the verification **required**. That is deliberately the opposite of how an action's `visible` expression fails: a broken visibility rule fails *open* into "hidden", while a broken step-up rule fails *closed* into "verify first". Write the expression so it always yields a boolean — a malformed one is not ignored, it locks the action behind the prompt for every entry.



What the person can verify with. Step-up reuses the verification methods the account already has; it adds no new mechanism. What actually works here is:

- a code from an **authenticator app**, or
- a **one-time code sent by SMS**, or
- a **one-time code sent by email**, or

- the **account password**, but only when the account has none of the above set up — a last resort the platform picks, never an option the user chooses.

A passkey cannot be used for this, even by someone who normally signs in with one. The check asks for a code that is typed in, so a passkey-preferring account falls back to the first code method it has. Say so in whatever instructions you write for your users: the passkey prompt they expect will not appear.

The proof is deliberately narrow. A successful verification mints a proof that is **single-use**, **expires after two minutes**, and is **bound to that user, that entry and that action** — all three are compared when it is redeemed, and redeeming it consumes it. It is not a session: it does not make the next action easier, and it does not reveal masked sensitive fields. (Revealing a sensitive value is a separate check with its own credential; passing one never satisfies the other, even though both ask for the same kind of code.) The proof travels as the `stepUpProof` field on the execute-action call — see Developer API surface.

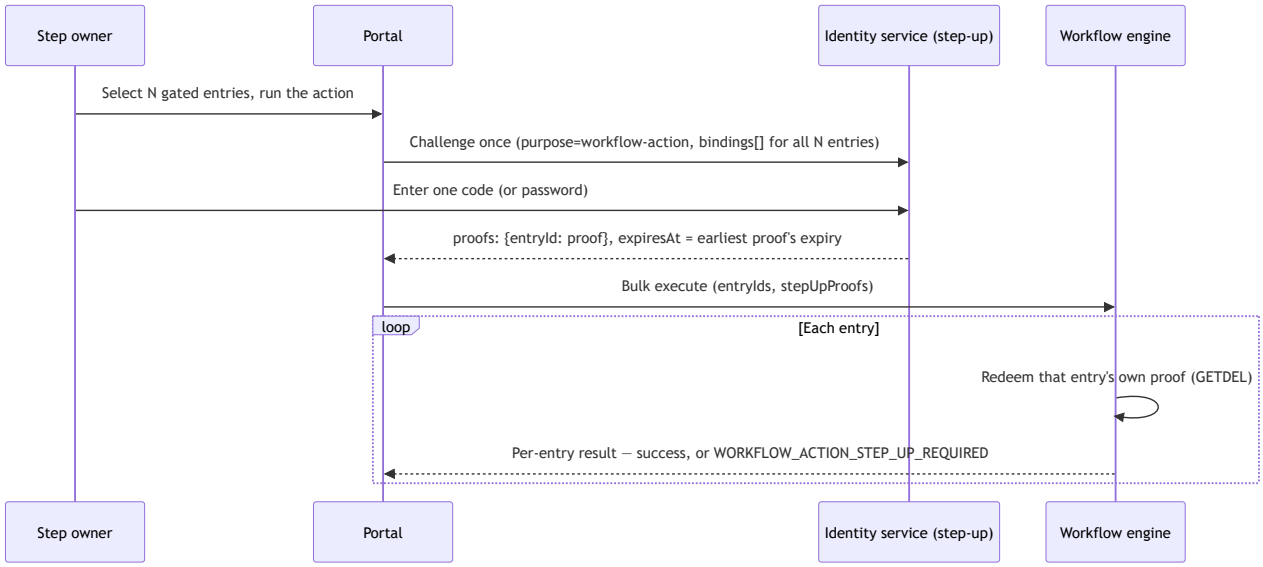
Bulk execution `Since 1.10` — one verification covers the whole gated selection. Bulk execution of a gated action is fully supported: the executor verifies **once** for the whole selection, and the batch runs behind a **map of single-use proofs, one per gated entry** — never a single shared proof, and never no proof at all.

The bulk endpoint's request body (`POST /v1/entities/{entity}/{workflowId}/{actionId}/bulk` , `BulkExecuteWorkflowActionRequestDto`) gained an optional field:

```
class BulkExecuteWorkflowActionRequestDto {
  entryIds: string[];
  notes: string;
  // entryId -> single-use step-up proof, one per entry, all minted by one
  // verification. Absent when the action needs no step-up.
  stepUpProofs?: Record<string, string>;
}
```

The server reads `stepUpProofs?.[entryId]` for each entry in the batch and forwards it as that entry's own proof to the same per-entry execution path a single-entry call uses — so a bulk execution is indistinguishable, from the engine's point of view, from N independent single-entry executions, each carrying its own proof (or none). **An entry with no matching key in the map is refused exactly as an unproven single-entry call would be** (`WORKFLOW_ACTION_STEP_UP_REQUIRED`), independently of its siblings — an incomplete map can never widen access, it can only leave some entries failing.

Minting the batch. The client challenges once with `purpose: 'workflow-action'` and a `bindings[]` array instead of a single `binding` — one `{entityId, entryId, actionId}` per selected entry, all sharing the same `entityId` / `actionId` (one action, many entries — never a mixed batch), capped at **100** bindings, no duplicate `entryId` . A single successful verify mints **one proof per binding** and returns `{proofs: {entryId: proof}, expiresAt}` — `expiresAt` is the **first** minted proof's expiry, not the last, since the batch is only usable while its earliest member still is. The full three-purpose minting/redemption mechanism — shared with the `Decrypt` step-up purpose added the same cycle — is documented once, in Sensitive Fields: Step-up unmask, Decrypt, and WorkflowAction [Build Apps › Protect a field], to avoid this page and that one drifting apart.



What your users will see, so you can design around it:

- Selecting **several entries** and running a gated action opens **one** verification dialog for the whole selection — not one prompt per row.
- On a successful verification, **every** selected row gets its own proof minted for it, and the batch runs behind that per-entry map. See Executing Workflow Actions: Verifying your identity before an action runs [Using OSOS › Acting on Workflow Steps] for the end-user-facing flow.
- An individual entry whose proof lapses or mismatches between minting and that entry's own execution (for example the entry changed step in the meantime) comes back as **that entry's own failure** in the bulk result map — it does not fail the rest of the batch.
- Selecting **exactly one** entry still works exactly as before: the single verification dialog, prompted just as it would be from the entry's own page.
- Someone acting through a page that grants **execute as admin**, or from the administration area where there is no end-user page in context, is **never prompted** — for bulk exactly as for a single entry. Automated executions (a reaction, an external step's callback) are never gated either.

If a step's work is genuinely done in batches, leave the gate off that action and put it on another one: a gated action and a batch-approved action cannot be the same action.

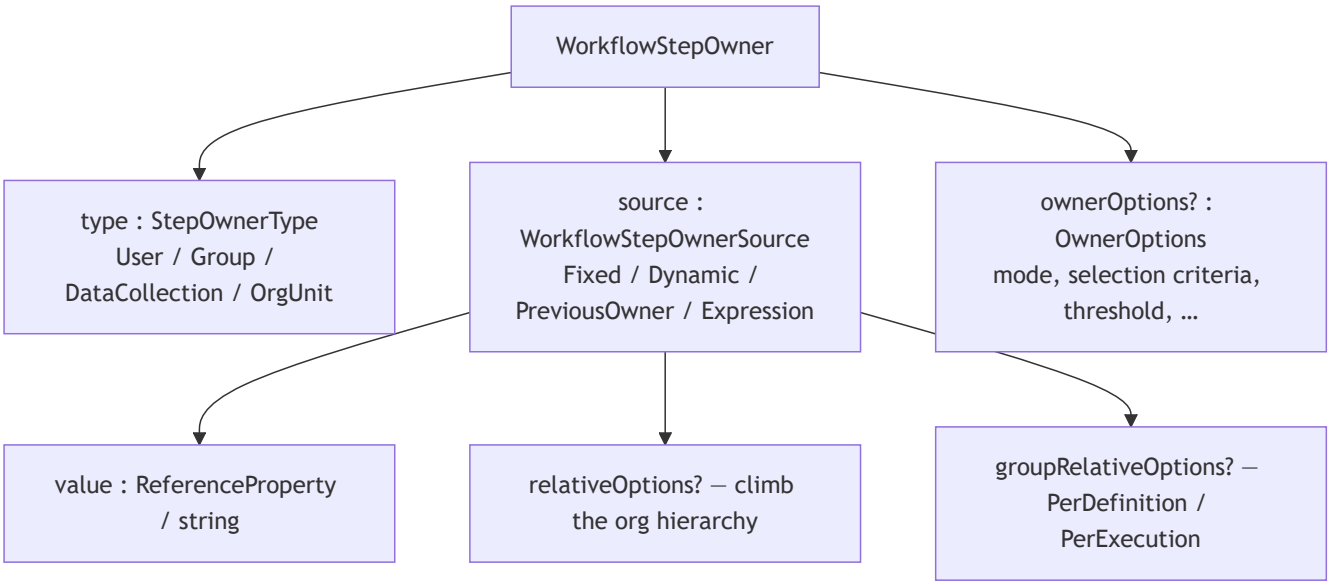
The refusal is identified by its error code — contract on the code, not on the HTTP status. An execution that reaches the server without a valid proof is refused with the error code `WORKFLOW_ACTION_STEP_UP_REQUIRED`, whose details carry the `actionId`. The HTTP status that refusal arrives with is **inherited from a generic error base rather than chosen for this case**: it is not the `401` or `403` an integrator would reasonably guess, nothing in the platform pins it, and a client that branches on the status will break. Branch on the error code — that is what the portal itself does. This holds for bulk too: an individual entry's refusal inside a batch result carries the same error code, not a distinct HTTP status of its own.

A successful or failed verification is recorded independently of the workflow action's own audit trail — see Monitoring: Logs & Audit [Administration › Monitoring: logs & audit] for the audit record an administrator sees.

Owner assignment

A step owner is responsible for monitoring and performing the tasks associated with a step: making decisions, taking actions, and moving the entry forward. The owner is described by `WorkflowStepOwner`, which has three parts: the **type**

(what kind of principal owns the step), the **source** (how the reference is found), and **owner options** (behaviour once resolved).



Owner type (`StepOwnerType`)

Value	Name	Resolution behaviour
0	`User`	The reference is a user; used directly. Designer label **User**.
1	`Group`	Expanded to group members. First member is picked unless parallel / manual / manager criteria apply. Designer label **Group**.
2	`DataCollection`	First item of the referenced data collection becomes the owner. Designer label **List**.
3	`OrgUnit`	Organizational-unit reference, optionally its manager. Not offered in the designer's **Step Owner** picker (which lists User, Group and List); reachable through the API.

Owner source (`WorkflowStepOwnerSource`)

`source.type` is a `StepOwnerSourceType` — it decides *how* the owner reference is found:

Value	Name	`value` holds	Resolved by strategy
0	`Fixed`	A `ReferenceProperty` (or array) to a user/group chosen at design time	`FixedOwnerStrategy`
1	`Dynamic`	A field normalized-name / path on the entry (may follow a relation)	`DynamicOwnerStrategy`
2	`PreviousOwner`	The **step id** the owner is taken from	`PreviousOwnerStrategy`
3	`Expression`	An expression returning a reference (or array)	`ExpressionOwnerStrategy`

Designer labels: **Fixed**, **Field**, **Previous Step Owner**, **Expression**. The designer offers all four for a **User** or **Group** owner, and every source except **Field** for a **List** owner. The **Change Current Step Owner** reaction action uses the same form without **Previous Step Owner**.

`source.value` is typed as `ReferenceProperty | string` and, per the model comment, means: a reference (Fixed), a field normalized name (Dynamic), or a step id used as the relative anchor (Previous owner).

> **Design-time UI caveat.** In the step-owner form the source-entity picker (`entityIdForSourceType`) supports only **User** and **Group** sources; **DataCollection** and **OrgUnit** sourcing has narrower design-time UI support.

Relative owner (`relativeOptions`)

When enabled, the resolved owner is replaced by their manager, climbing the hierarchy. This only applies when the owner is a **User**.

Field	Type	Notes
`enabled`	boolean	Turn relative resolution on.
`relationType`	`StepRelativeRelationTypeEnum`	Currently only `0 = Manager`.
`relationGrade`	number	Levels to climb, starting at 1.

The `OwnerTypeProcessor` implements this by calling `userService.getUserDirectManager(userId, relationGrade)`; if no manager is found the owner is cleared and the chain falls through to the next candidate/fallback. For example, if "Mohammad" is the primary owner and the relation is Manager at grade 1, "Salma" (Mohammad's direct manager) becomes the step owner.

Group-relative mode (`groupRelativeOptions`)

Only `mode` (`GroupRelativeOptionsModeEnum`): `0 = PerDefinition`, `1 = PerExecution`. Used by `PreviousOwnerStrategy` for Group / DataCollection owners:

- **PerDefinition (By Definition)** — take the owner definition as configured on the relative step (works even if that step hasn't executed yet).
- **PerExecution (By Execution)** — take the **actual executors** of the relative step's execution(s); the full executor list becomes `originalOwner`. If the relative step is the current step, the current user is appended. If the resolved owner is the system user, the owner is cleared. By-execution is what applies when a manager intervened, the group membership changed, ownership was taken, or the primary assignment fell through to a fallback.

Owner options (`OwnerOptions`)

Field	Type	Notes
`allowChangeOwner`	boolean	Permit reassigning the owner at runtime.
`mode`	`StepOwnerMode` | null	`0 = SingleUser`, `1 = AllUsers`.
`returnTo`	`TakeOwnershipReturnToModeEnum` | null	For take-ownership steps: `0 = Group`, `1 = LastOwner`.
`accessPermission`	`StepOwnerMode` | null	Who may access the step. Default `SingleUser (0)`.
`selectionCriteria`	`StepOwnerSelectionCriteria` | null	`0 = First`, `1 = Manual`, `2 = Manager`.
`mainActionExecutionCondition`	`MainActionExecutionConditionEnum`	Group-decision rule: `0 = AllUsers`, `1 = SomeUsers`.
`failureActions`	`StepOwnerFailureActions[]`	Action(s) to take when SomeUsers voting fails the threshold.
`threshold`	`StepOwnerThreshold` | null	Voting threshold (see below).
`isParallel`	boolean	Assign to all members simultaneously (parallel step).

Designer labels for these options: `mode` is **Select User Type** (**Single User**, **All Users**); `isParallel` is **Multi Assignment Type** (**Sequential**, **Parallel**); `selectionCriteria` is **Choose Selection Type** (**First one**, **Take Ownership**, **Org Unit Manager**); `accessPermission` is **Access Permission**; `returnTo` is **Return to** (**Group**, **Last owner**); `mainActionExecutionCondition` is **Main Action Execution Condition** (**Some Users**, **All users**); `failureActions` is

Action On Failure (including **Most Chosen Secondary Action**); `threshold` is **Proceed Threshold**; `allowChangeOwner` is **Enable Reassign**; `relativeOptions` is the **Relative** switch with **Relation** and **Relation Grade**.

Selection criteria drives post-resolution behaviour:

- `First (0)` — the first eligible member is used.
- `Manual (1)` — the auto-resolved owner is **cleared**; an eligible user picks up the step by clicking **Take Ownership** in the entry's details view.
- `Manager (2)` — for a Group owner, the owner becomes the **org-unit manager** of the group rather than the first member.

Access permission decides visibility: with `SingleUser` only the owning user sees the step, with `AllUsers` every member of the group sees it. A step whose `selectionCriteria = Manual` and `returnTo = LastOwner` restores its most recent owner on re-entry; with `returnTo = Group` it returns to the group.

`Since 1.8` — whether the current owner may reassign a step (`allowChangeOwner`, designer label **Enable Reassign**) always reflects the **live** workflow design whenever that design has an owner configuration, so turning it off also applies to entries already sitting on the step. The `ownerDefinition` snapshot stamped onto the entry is consulted only when the live step has no owner configuration at all.

Unowned group-owned steps `Since 1.8`

A step whose owner is a **Group** or **DataCollection** with `mode = SingleUser` and `selectionCriteria = Manual` is designed to sit **unowned**: the engine clears the auto-resolved owner and keeps the eligible group on the step's `originalOwner`, so any member can claim it later with **Take ownership** (see Executing workflow actions [Using OSOS › Acting on Workflow Steps]).

That state can also be **written onto a step that already had an owner**. The internal change-owner command accepts a `null` owner, and the handler that applies it skips the owner lookup and validation and clears the current step's `owner` instead of rejecting the request.

The step's owner definition is kept in sync with the owner. The `ownerDefinition` snapshot on the step is written alongside the new owner, because the take-ownership eligibility check **prefers** that snapshot for `mode` and `selectionCriteria`, and only falls back to the workflow's static design when the step has no snapshot at all.

Where a `null` owner comes from. The runtime change-owner endpoint (`PATCH /v2/…/change-owner`, below) never sends one: omitting `owner` from the request body puts the call in take-ownership mode instead, which always resolves to the calling user. The verified caller is the reactions/automation action **"Change current step owner"** — it runs the same owner-resolution pipeline described below and, when that resolves to a group `originalOwner` with no single owner (the Manual / parallel case), leaves the step unowned rather than skipping the entry.

The reassign-owner record persisted on the step-owner-changed event allows **both** its old and new owner to be `null`.

Group-decision voting and thresholds

When a step is owned by **all users** of a group, the transition to the next step requires the main action from enough of them, per `mainActionExecutionCondition`:

- `AllUsers` — if any user takes a non-main action, the result is a failure action.
- `SomeUsers` — count main-action executions against the `threshold`; if below threshold, the `failureActions` apply. Once the threshold is met, the action is no longer sent to the remaining members.

`failureActions` can be the most-chosen secondary action, the main action, a Return, or a Reject; if the condition fails and no valid failure action exists, the first secondary action chosen by the first user is executed.

Threshold (`StepOwnerThreshold`)

Field	Type	Notes
`type`	`ThresholdValueTypeEnum`	`0 = Percentage` (value 1–100), `1 = Number` (count of users). The designer's **Proceed Threshold Type** currently offers only **Fixed Number**; a percentage threshold can be set through the API.
`value`	number	Percentage or user count per `type` .

Transition condition	Users performing the main action	Executed action
Approval from all users	Some users only	Failure action
Approval from all users	All users	Main action
Approval from some users	Threshold or more users	Main action
Approval from some users	Fewer than the threshold	Failure action

Alternative (fallback) owners

`fallbackOwner` is an ordered list of backup owner definitions tried in turn when the primary owner cannot be assigned — for example the reference is ineligible, a user account is deactivated, or a relative manager cannot be found. Each fallback is configured with the same structure as the primary owner and can be repeated until it resolves to a fixed user. In the designer these appear as **Fallback Assignment 1**, **2**, … sections; one is added automatically, and required, whenever the primary owner is anything other than a fixed user without **Relative** (a field, previous-step, expression, group, or list owner, or a fixed user climbing to their manager). The chain ends at the first fallback that is a fixed **User**. If a step ends up with no valid owner, the transition fails.

Step-owner resolution (observable behaviour)

The engine that turns this configuration into an owner is documented in full in Engine: Workflow & Owner Resolution (handler chain, source strategies, post-processing, eligibility validator). What an integrator can observe:

- **Order of precedence.** A **permanent** reassignment (`permanentChange` on change-owner) wins; then a take-ownership step with `returnTo = LastOwner` restores its last owner; then the remaining owners of an all-users step; the **Entry** step is always owned by the entry's creator; otherwise the step's `owner` is tried, then each `fallbackOwner` in order, and the first that yields a valid owner is used.
- **Relative manager.** With `relativeOptions` on a User owner, the resolved user is replaced by their manager `relationGrade` levels up; if no manager exists the candidate is dropped and the next fallback is tried. It is not applied on the Exit step.
- **Expression source** is evaluated with `entry` , `dataEntity` , `currentUser` , `currentStepId` , `history` and `workflow` in scope; only valid references are kept.
- **Previous Step Owner** on a take-ownership step inherits the user who actually claimed it (`Since 1.8`), not the group it was offered to.
- **Eligibility.** A candidate is skipped (and the next fallback tried) when the referenced user, group or org unit is archived, the user is deactivated, or the reference is an obsolete record. A **List** owner is checked against the list itself. A **Group** or **List** that expands to **zero eligible members** falls through to the fallback owners — except with **Org Unit Manager** selection, which takes the manager rather than the members.

- **No owner.** Unless the step is the Exit step, a step that resolves no owner fails the transition (`OriginalOwnerNotFoundException`). Exit steps may be ownerless.

Eligible owners for an ownerless step `Since 1.6`

The change-owner picker is fed by `eligible-owners` (see Developer API surface), which scopes a non-admin caller's candidates to the step's `originalOwner`. For a step owned by a fixed group **with no members** — `originalOwner` cleared to `null` (an unowned group-owned step), or an empty group reference — the step is treated like a single-user owner: when `originalOwner` is null, or the group or data collection has no members, the picker lists every **active** user — exactly the set the change-owner write accepts. A group whose members exist but are all ineligible still yields an empty list. Also delivered to the 1.6 line as an update.

`Since 1.10` — four step-owner design errors (`FallbackOwnerLastItemOwnerType`, `PreviousOwnerStepIsExitStep`, `PreviousStepNotFound`, `RelativeOptionsOwnerType`) answer **400**, a validation error naming the problem.

Group ownership on a group-owned step

Applies to the 1.4, 1.6 and 1.8 lines alike — a correctness fix, not a milestone feature.

`originalOwner` is described above as the full eligible owner set, and for a Group or DataCollection step that is normally the group reference itself. But for every non-parallel, non-Manual group step the engine **expands the group into its individual members** and keeps one of them as the working owner — so by the time a completed step is read back, `originalOwner` may hold that single member and carry no group reference at all.

The platform therefore recovers the group from the step's **configured** owner: when `originalOwner` no longer references a group, user group, role, org unit, space or data collection, resolution falls back to the owner definition's `source.value` — the group the step was designed with.

That fallback only works for a `Fixed` owner source. A `Dynamic` or `Expression` source stores a field path or an expression string in `source.value`, not a group reference, so there is nothing to recover and no fallback is attempted for those.

This is what keeps non-acting group members listed in the workflow panel for a completed group-owned step.

SLA and step-time monitoring (`WorkflowStepTimeMonitoring`)

Each step can carry an **SLA**: a time limit after which the step is considered overdue, so you can monitor how long steps take and react to delays. Per-step timing is configured on `WorkflowStep.timeMonitoring`.

Field	Type	Notes
`enabled`	boolean	Turn monitoring on for the step.
`workingTimeReference`	nullable `ReferenceProperty`	Reference to the working-time / calendar definition used to count elapsed time.
`monitoringUnit`	`TimeMonitoringUnitEnum` | null	`0 = Minutes`, `1 = Hours`, `2 = Seconds`.
`value`	number | string | null	The time-to-execute; a number, or a source expression / field per `mode`.
`fallbackValue`	number	Used when `value` cannot be resolved.
`mode`	`TokenSourceType` | null	How `value` is obtained: `0 = Fixed`, `1 = Field`, `2 = Expression`, `3 = Pointer`. Default `Fixed`.

The due date is computed by an external **clock service** that honours the configured working hours and holidays, so the countdown excludes non-working time (working hours and holidays are themselves configured as data in the platform). At the platform level, `workflow-service` persists an SLA record per monitored step (`WorkflowSlaConfig`) that copies the `timeMonitoring` (`timeToExecute`) alongside `module`, `entity`, `workflowId`, and `stepId`. There is no public SLA endpoint: an entry's breached due dates appear in its history as `SlaOverdue` items — see History and timeline API.

`Since 1.8` — closing a step's SLA record on exit is scoped to the **specific entry that exited**, not just the step.

`Since 1.10` — how SLA records are kept (neither point changes what *Pending*, *Achieved* and *Overdue* mean for a step you designed):

- **An achieved SLA stays achieved.** The overdue check moves one record at a time and only while that record is still pending, and raises a breach only when that transition actually happened — so a record that became *Achieved* during a check is never written back to *Overdue*, and no false breach is raised.
- **An entry with no active step is skipped,** in both the periodic overdue check and the handler that starts a step's clock, without affecting other entries.

Ad-hoc steps

Ad-hoc steps let a runtime user insert an extra approval/information step after a step where `allowAddAdHoc = true`. The inserted step is a sub-step of the current one; the workflow pauses there until the ad-hoc owner acts, then resumes the original step automatically. Its owner is chosen with the same owner-selection options described above.

`CreateAdhocWorkflowStepDto` (mirrored identically in `data-service` and `workflow-service`, the latter being the public request body for `POST …/adhoc-step` below):

Field	Type	Notes
`adhocStepType`	`AdhocStepTypeEnum`	`0 = AdditionalApproval`, `1 = AdditionalInformation`.
`owner`	`WorkflowStepOwner`	Required owner definition for the inserted step.
`fallbackOwner`	`WorkflowStepOwner[]`	Optional fallbacks.
`notes`	string	Optional.
`description`	string	Optional free-text note explaining why the step was added. `Since 1.8`

`description` `Since 1.8`. The single free-text value is stamped onto the created step and duplicated into both `en` and `ar` of its `NullableLocalizedValue` (the create dialog only collects one value; it is not resolved per viewer locale), and left `undefined` when none is supplied. The per-step tooltip renders it. See Adding an ad-hoc step [Using OSOS › Acting on Workflow Steps] for the end-user-facing behavior.

Workflow field requirements

At each step you set how the entity's fields behave, choosing one display option per field:

1. **Hidden** — the field does not appear at this step at all.
2. **Locked** — the field appears but cannot be modified.
3. **Editable** — the field can be modified during this step.
4. **Mandatory** — the field must be filled before the workflow can proceed.

For example, in a report workflow the “Creation Date” field might be Hidden at data entry, Locked at review, and the “Submit Report” field Mandatory at the final confirmation step.

The mapping is stored on the **field**, not on the step: each field carries an optional per-workflow, per-step table (`RequirementSettingEnum`: `Hidden`, `Locked`, `Editable`, `Required` — labeled **Mandatory** in the builder). A step the field has no entry for behaves as Editable. A **Block** field’s own per-step setting can instead be `PerBlock`, which defers to its **internal entity’s** own field-level requirement for that same step (falling back to Editable if the internal entity has no mapping for the active workflow at all).

Which step’s requirements apply to a viewer

Requirements are resolved **per viewer**, not only per entry, because two people looking at the same entry can be standing on different steps of it — an owner acting on the live step versus someone reading the entry back later. The step used is worked out with this precedence:

1. The previous-owner source of an **ad-hoc** step, if the viewer’s currently owned step is itself ad-hoc.
2. The step the viewer (or their group/collection) **currently owns**.
3. The last **non-ad-hoc** step the viewer executed (or could access through their group/collection).
4. `Since 1.8` — failing all of the above, the entry’s **actual current step**, regardless of who owns it.

Step 4 is what applies requirements — Hidden included — to a viewer who never owned or executed a step on the workflow, such as a requester reading back their own entry after it moved into a **sub-workflow** they only triggered.

Hidden fields stay hidden after Return or Reject `Since 1.8`

Two step types never carry field requirements at all — the requirements picker excludes the **Exit** step and **Sub-Workflow** steps, since neither is a place an admin can meaningfully set Hidden, Mandatory, or Locked. But both **Reject** and a sub-workflow-closing **Return** can park an entry on exactly one of those steps: Reject sends it to the workflow’s main Exit step, and a Return that closes a sub-workflow leaves it on that sub-workflow’s own Exit step. Resolved directly, a step like that has no requirement mapping, so the platform applies a **Hidden-only** fallback: when the entry’s actual step has no requirement mapping, a field stays Hidden if it was Hidden on **any** step the entry actually passed through in that workflow — including a sub-workflow that has since closed. It deliberately recovers only `Hidden`: a completed or rejected entry never has a stale `Mandatory` / `Locked` setting applied this way.

List-shaped reads report per-row hidden fields via `hiddenFields` `Since 1.8`

Everything above resolves Hidden by mutating the **entity definition** the response is serialized against — that works for a single-entry read, where one entity definition describes exactly one row. It cannot express a **per-row** fact: a list-shaped read (Table, Badge, MiniCard, Kanban, Tree) serves many rows against **one shared entity definition**, and two rows sitting on different workflow steps can legitimately disagree about which fields are Hidden for that row.

`DataEntryService` reports the per-row fact separately, on a new response-only `hiddenFields?: string[]` array carried by both list-read models (`DataEntry`, `DataEntryV2Body`) — the `fullNormalizedName`s the current reader must not be shown **for this row**. It covers every list-shaped read path (`findWithResolves`, `findKanbanWithResolves`, `findTree`), is skipped for `viewAsAdmin` / `editAsAdmin` views and system callers, and fails closed: if a row’s hidden fields can’t be resolved, that row reports everything as hidden rather than nothing.

Concurrency guard on workflow writes `Since 1.10`

Every workflow write is **conditional on the workflow state it was validated against** — executing an action (including the branches that start or complete a sub-workflow), inserting an ad-hoc step, changing the step owner and changing

the step. Two calls racing on one entry therefore record one transition and publish `WorkflowActionExecuted` / `WorkflowStepExited` / `WorkflowStepReached` once; the loser is refused rather than silently overwritten. The guarded state is the current step, the workflow instance, the owner, the executed-by count and the step-history length, for every workflow instance on the entry. What an integrator sees:

- **A lost race answers `409 UPDATE_CONFLICT`**, and nothing is recorded or published for it. The 409 is passed through to the caller (an upstream failure that names no status is answered 502), so a client can tell "someone moved this first" from a real error: reload the entry and decide again. The web client shows its update-conflict dialog.
- **The write is partial.** Only the keys the transition changed are set or unset, so a concurrent edit to a field outside the workflow is not lost.
- **The guard is the workflow state, not `updateIdentifier`.** Unrelated writes legitimately re-stamp `updateIdentifier` (the External Step return and reaction snapshots both do), so it cannot detect a workflow race on its own; change-owner additionally matches the `updateIdentifier` it loaded.
- **A step-up proof survives a lost race.** The proof is redeemed before the transition is computed; when the guarded write loses, it is put back for its remaining lifetime so the caller can retry. A proof that does not match is still burned.
- **Event order.** The change-step branch that targets a sub-workflow publishes its step events after the guarded write.
- **Archive and restore** use the same pattern on the entry's lifecycle state: archive writes only while the entry is Active and restore only while it is Archived, so two racing archives publish `DataEntryArchived` once. The loser — like any archive of an entry that is already archived — answers **422** `ENTRY_ARCHIVED`; a lost restore answers `ENTRY_NOT_ARCHIVED`, and an archived entity is likewise reported as 422.
- **A stale entry snapshot cannot undo a step move** `Since 1.4`. A save that may not override workflow meta leaves `_metaCurrentSteps`, `_metaWorkflowHistory`, `_metaAdhocSteps`, `_metaOrder` and `lifeCycleStatus` as stored, and a step move (for example a **Move Workflow To Step** reaction) starts from the entry's stored workflow state rather than from the triggering event's snapshot. As a consequence, such a save rejects a value key that contains `.` or starts with `$` (`EXTRA_ENTRY_VALUE`). An **ad-hoc step** still computes from its snapshot; it fails with a conflict rather than reverting.
- **Two refusals answer 422** `Since 1.8`. Executing an action that is hidden on the entry (`WORKFLOW_ACTION_EXECUTION_NOT_ALLOWED`) and taking ownership of a step that already has an owner (`CURRENT_STEP_ALREADY_OWNED`) are refusals, answered **422** with those error codes. Match on the `errorCode`, not the status.

Worked example as a definition: a two-step expense approval

The same process the builder guide walks through in the designer, expressed in model terms — a minimal expense-approval workflow on an `Expense` entity (fields: amount, description, a `submitter` reference, a department reference).

1. **Entry — "Submitted".** Owner source **Dynamic**, pointing at the entry's `submitter` field, so each entry is owned dynamically by whoever raised it. One main action **Submit** → relative **Next step**.
2. **Intermediate — "Manager review".** Owner is the submitter's **Manager**: a source with `relativeOptions.enabled = true`, `relationType = Manager`, `relationGrade = 1`, so the engine climbs one level up the org chart. Set the action's `commentValidationMode = Required` so the reviewer must leave a note, and add an **SLA** of **2 working days** on the step (`timeMonitoring`, counted against a working-time reference so weekends and holidays are excluded). Actions: **Approve** → Next; **Reject** → a Return action (relative Previous step, back to "Submitted").

3. **Exit — "Approved".** Terminal step; no owner required.
4. **Escalation on breach.** Wire a workflow reaction on the **SlaTimeExceeded** event for the "Manager review" step that escalates to the **manager's manager** — for example a notification, optionally reassigning the step owner (relative Manager at grade 2).
5. **Parallel N-of-M variant (optional).** To require several reviewers instead of one, make "Manager review" a **Group** step with `mode = AllUsers`, `isParallel = true`, `mainActionExecutionCondition = SomeUsers`, and a `threshold` (for example 2 of 3). Add a `failureActions` entry (the most-chosen secondary action) for when the threshold is not met.

Design-time rules enforced on save

When you save a workflow, these rules are checked:

- A workflow must have **at least two steps**, the **first of type Entry** and the **last of type Exit**.
- **Every step needs at least one action**, except the Exit step and Sub-Workflow steps.
- **A step can have at most one primary (main) action.**
- **A Sub-Workflow step must reference an existing sub-workflow and define a merge step, and must not have any actions of its own.**
- **Step names and workflow names must be unique** within their scope.
- For workflow-automation reactions: an automation must automate at least one step action, and the last (Exit) step cannot be automated.

Developer API surface

Design-time workflow definitions are edited as part of the **Data Entity** (the `workflows` array); there is no standalone "create workflow" endpoint. The endpoints below are the query and runtime surfaces — the runtime execution semantics (movement rules, who may act) are covered on Executing workflow actions [Using OSOS › Acting on Workflow Steps].

`workflow-service` — `WorkflowController`

The user-facing runtime transitions (owner-context):

Method & path	Purpose
`PATCH /v1/entities/{entity}/entries/{entry}/{workflowId}/execute-action`	Execute a workflow action (transition) on an entry. Carries the `stepUpProof` field when the action requires step-up verification. `Since 1.10`
`PATCH /v1/entities/{entity}/entries/{entry}/workflow/{workflowId}/change-step`	Move the entry to a different step (`ChangeWorkFlowStepDto`).
`PATCH /v2/entities/{entity}/entries/{entry}/{workflowId}/change-owner`	Reassign the step owner (`ChangeStepOwnerDto`).

Method & path	Purpose
`POST /v1/entities/{entity}/entries/{entry}/{workflowId}/adhoc-step`	Insert an ad-hoc step (`CreateAdhocWorkflowStepDto`).
`GET /v1|v2/entities/{entity}/entries/{entryId}/main-flow-status`	Main-flow progress of an entry.
`GET /v1/entities/{entity}/workflow/{scope}`	Workflow (definition / preview) for a scope.
`POST /v1/entities/{entity}/{workflowId}/{actionId}/bulk`	Execute one action on every entry in `entryIds` (`BulkExecuteWorkflowActionRequestDto` : `{entryIds, notes, stepUpProofs?}` — `stepUpProofs` a `{entryId: proof}` map, one single-use proof per gated entry, all minted by a single verification, `Since 1.10` , see Step-up verification before executing an action); all entries must currently be on the same step, else `WorkflowStepsNotSameException` . Returns a per-entry result map.

Change step (`ChangeWorkFlowStepDto`) — `stepId` (UUID or empty string) plus `changeStepMode` (`ChangeStepModeEnum`): `0 = Normal` , `1 = ReturnToStep` , `2 = MoveToStep` , `3 = ResetWorkflow` , `4 = EndWorkflow` , `5 = AdHoc` ; optional `notes` .

Change owner (`ChangeStepOwnerDto`) — `owner` (`UserReferenceProperty`), `note` (default `''`), and `permanentChange` (boolean). Setting `permanentChange` is what `PermanentChangedOwnerHandler` later reads to keep the reassignment sticky across re-entries.

Bulk execute (`BulkExecuteWorkflowActionRequestDto`) — one action applied to every entry in `entryIds` . Each entry is dispatched onto the service bus and awaited independently, so one entry's outcome never blocks another's; the response is a per-entry result map. `Since 1.8` — dispatch settles each entry's reply independently, so a failure or timeout on one entry is reported only as that entry's own result and never affects other in-flight workflow requests. See Acting on many entries at once (bulk) [Using OSOS › Acting on Workflow Steps] for the usage-facet write-up.

`Since 1.10` — the bulk route is allow-listed in every Data View's **execution plan**, so a step action can be executed on entries selected in a list through any view. Separately, a step-up-gated action is always refused here without a valid proof — see Step-up verification before executing an action.

Execution-plan routes added to every view `Since 1.8`

Each Data View carries an **execution plan** — the allow-list of routes a request made through that view may call. These routes are in every plan (existing plans were backfilled; a view whose recompute failed gains the route on its next save):

- **Ask User field data** `Since 1.8` (delivered on the 1.8 line as an update). The Ask User form of a system button bound to a custom action is addressed as entity `TRASK_<entityId>_<triggerId>` , and its reference fields load their options from the field-data endpoints (`POST /v2/entities/TRASK_…/fields/{fieldId}/data` and `…/data/count` , `GET /v1/entities/TRASK_…/fields/{fieldId}/data`). They are allowed for every `TRASK_` id on every view — only those field-data endpoints — so the popup's reference pickers load; the handler still rejects a missing, invalid or disabled trigger.
- **Export-template status** `Since 1.10` . `POST /v1/entities/{entity}/templates/status/entries` returns, per requested entry, the status of each conditioned Single Entry export template, so the export buttons know which templates an entry's condition disables or hides. It is added only to views that expose an export template. Through a view, only that view's templates are reported, and conditions are evaluated only when the view can

read every requested entry — otherwise each template is reported as not met, so the answer reveals nothing about rows outside the view.

Version notes

Capability	Availability
Sub-workflow steps (`subWorkflowId` , `mergeStep`)	Since 1.4
Conditional target steps (`conditionalTargetSteps`)	Since 1.4
External steps (`StepType.ExternalStep` , `ExternalStepConfig` , return callback)	Since 1.8
Unowned group-owned steps (nullable change-owner command, `ownerDefinition` kept in sync)	Since 1.8
Group ownership recovered on a group-owned step (fallback to a `Fixed` owner source)	Correctness fix — current behavior on 1.4, 1.6 and 1.8
Step owner falls back to `fallbackOwner` when a group has no eligible members	Since 1.8
Inherited owner preserved on a sub-workflow Entry-step return	Since 1.8
Reassign permission (`allowChangeOwner`) reflects the live step design, not a frozen snapshot	Since 1.8
SLA close scoped to the exiting entry, not just the step	Since 1.8
Bulk action dispatch: one failing entry no longer crashes the worker	Since 1.8
Sub-workflow boundary step-exit/executed events publish reliably and in order	Since 1.8
An entry parked on the Exit step is reported as completed	Since 1.8
Field requirements: final current-step fallback for viewers with no step history	Since 1.8
Hidden field requirements survive landing on Exit / Sub-Workflow steps (Return/Reject)	Since 1.8
List-shaped reads (`findWithResolves` / `findKanbanWithResolves` / `findTree`) report per-row `hiddenFields`	Since 1.8
Ad-hoc step gains an optional free-text `description`	Since 1.8
Step-up verification on a step action (`stepUp` , `stepUpProof`), single entry	Since 1.10
Step-up verification extended to bulk execute-action (`stepUpProofs` map, one verification per selection)	Since 1.10
Bulk execute-action route allow-listed in Data View execution plans	Since 1.10
An SLA achieved inside the checker's window is no longer recorded as a breach	Since 1.10
A null active step no longer aborts an SLA pass or blocks a step's SLA clock	Since 1.10
Ask User (`TRASK_*`) field-data routes allowed in every view's execution plan	Since 1.8
Change-owner picker lists every active user for an ownerless step	Since 1.6
Workflow writes conditional on the validated workflow state; a lost race answers 409 `UPDATE_CONFLICT`	Since 1.10
Export-template status route in view execution plans	Since 1.10
Step-owner design errors answer 400; archiving an archived entry answers 422 `ENTRY_ARCHIVED`	Since 1.10
Executing a hidden action and taking ownership of an owned step answer 422 (`WORKFLOW_ACTION_EXECUTION_NOT_ALLOWED` , `CURRENT_STEP_ALREADY_OWNED`)	Since 1.8

Capability	Availability
Entry saves keep the stored workflow state; a step move starts from the stored state, so a stale snapshot cannot undo it	Since 1.4
Ordered fallback owners, parallel steps, take-ownership return-to, per-step time monitoring, group-decision voting, group-relative modes, ad-hoc steps	Long-standing
Step-owner resolution engine (Chain + Strategy, Expression source)	Long-standing configuration surface; the engine was reorganized internally in a later release

Data views API reference

This page is for **integrators and developers** who read or write data views through the API, call the kanban, quick-filter or export routes, or need the exact rules the server applies when a view is saved. It also keeps the change history of the view designer. Sections follow the order of the designer.

> **Designing a view in the UI?**
>
> Builders should start with Data Views & Faces [Build Apps] — it covers the same features in the labels the view designer shows.

Views and Grafana

A view's reach extends past the portal: `Since 1.6` it is also the mandatory unit a **Grafana** panel executes against, because a view's execution plan is what the gateway authorizes. If data is to appear on a Grafana dashboard, it needs a view first — see Grafana Dashboards.

Vocabulary

Some vocabulary used throughout this page:

Term	Meaning
Data view	A view scoped to one data entity, rendering its entries.
Face / display method	One visual presentation of the entries list, typed by its kind (table, badge, ...).
Default face	The single face shown first when the view opens.
Field mapping	Which entity fields feed a face's title, subtitle, picture, date, or category slots.
Display information	The per-field configuration that makes a field available to map into a face.

Proxy views

> **Proxy views · Since 1.10**
>
> A view can also exist as an internal **Proxy view** (`kind: Proxy`) — auto-created to govern what a Proxy virtual entity [Build Apps › Virtual entities] is allowed to read and write, rather than something you design through the ordinary flow below. Its editor surface is deliberately narrower than an ordinary view's (identity, source, fields, and the create/edit/archive toggles only — no publish state, audience, or permissions of its own), and a view's kind can never change once set. The rest of this page describes the ordinary, user-designed view kind; see Virtual Entities [Build Apps › Virtual entities] for the Proxy view's own governance model, including how narrowing a Proxy view's field list cascades back onto the entity's own field picker.

Designer labels and stored names

Designer label	Stored as / meaning in this page
Card view	the Badge face
Counter view (under **Other options**)	the Mini-card face
Face slots **Main title**, **Sub title**, **Image/file**, **Main reference**, **Sub category**, **Main number**, **Metadata**	Primary Title, Secondary Title, Image / File, Main Reference, Subcategory, Main Number, "show additional information"
Kanban columns: **All columns** / **Used columns** / **Or select specific columns**	`columnsMode`: `all` / `used` / `specific`
Allow users to change this setting (kanban)	`allowUsersToChangeColumns`
Tree reference, **Track parents line**	the tree's self-reference field, trace-parents
Open entry in: **System Default Behavior** / **View Entry Details** / **Edit Entry**	table row-click mode
Quick filters › **Placement In status list** / **Standalone toggle**; groups **Standalone dropdown**; **Default**	`displayMode`, `isAppliedByDefault`, `order`
Aggregators; **Grouping** › **Add Group By**, **Collapse groups by default**, **User change grouping**	per-view aggregates; grouping levels, start collapsed, let end users group
Allow export	`viewConfiguration.allowExport`
Repeatable reorder, **Add category options**, **Edit category options**, **Link create and edit card settings**	`allowRepeatableReorder`, `allowAddCategoryOptions`, `allowEditCategoryOptions`, `linkCategoryOptionSettings`
Buttons display layout, **Apply to workflow actions only**	`actionButtonsConfig.layout`, `actionButtonsConfig.applyWorkflowActionsOnly`
Default landing screen	screen order
Web settings (tab)	per-view / per-entity scripts

Apply to workflow actions only is offered only for the Horizontal and Vertical layouts. It lays out the current step's workflow actions (or, when the step has none, the primary action) and moves every other action under a **More** overflow (`libs/feature/entry-content/.../entry-header.component.ts`); it does not hide them. An earlier version of the builder page said it "restricts the view's action buttons to workflow actions, hiding the rest".

Faces at a glance

One entity can have many views, and a single view can enable more than one face. The table summarizes each face and where it is available:

Face	Best for	Web	Mobile
Table	Dense data and admin work; column reorder, inline edit, quick-add, selection, bulk actions	Yes	Yes
Kanban	Status-driven work; cards grouped into columns by a classification field, drag between columns	Yes	Yes
Calendar	Date-centric data; month / week / day / list layouts	Yes	Yes
Timeline	Posts and feeds; entries shown pinned-first, then by date	Yes	Yes
Tree	Hierarchical / org-chart data; lazy-expand nodes	Yes	Yes
Badge	Compact badge / gallery chips	Yes	Yes
Mini-card	Compact card list with counters	Yes	No

Note: Selecting the **Mini-card** face on the mobile app renders an empty area — use another face on mobile. Ordering / reordering of records is available on Table, Badge, and Kanban; on mobile you can drag a Kanban card to another column (persisted to the server), but reordering whole Kanban columns is not yet supported.

Designing each face

Each face reads a **field mapping** that tells it which entity fields feed its title, subtitle, picture, date, or category slots. For a field to be mappable, it must first be exposed in the view's **display information**. Server-side validation runs only when the entries-list screen is enabled, and only for the Table, Kanban, Calendar, Timeline, and Tree faces — **Badge** and **Mini-card** have no server-side face validation.

Table

Displays entries as rows and columns. To design it:

1. Enable the entries-list screen.
2. Enable the **Table** face.
3. Configure:
    - **Table Columns** (required) — the fields shown as columns; at least one must be selected. Every column field must have its **show-in-table-view** option enabled on the entity, otherwise the field cannot be used as a column.
    - **Allow Field Changes** (optional) — lets the end user toggle which columns are visible.
    - **Column reordering** (optional, off by default) and a cap on how many values a multi-value cell displays.

The list-level **row-click behavior** controls what happens when a user clicks a row — system default, open the entry details, or open the entry for editing. **Since 1.6**

Badge

Displays entries as compact badge / gallery chips, suitable for condensed lists. Map:

- **Primary Title** (required)
- **Secondary Title** (optional)
- **Image / File** (optional)

A **show additional information** toggle (on by default) surfaces metadata such as creation / update dates. On the mobile app the badge face offers selectable sub-layouts (badge / gallery). There is no server-side badge validation.

Calendar

Plots date-centric entries on a month / week / day / list calendar. Map:

- **Primary Title** (required)
- **Secondary Title** (optional)
- **Image / File** (optional)
- **Date fields** (required) — the dates plotted on the calendar; every date field must resolve to a `Date` or `DateTime` entity field, otherwise the calendar face is rejected.

Kanban

Displays entries as cards grouped into columns by a classification field. Map:

- **Primary Title** (required)
- **Category** (required) — the field whose values become the board columns: a single-value **category** field or, `Since 1.10`, a single-value **reference** field (see Column field and column modes).
- **Secondary Title**, **Image / File**, **Main Reference**, **Subcategory**, **Date**, and **Main Number** (optional).
- **Kanban columns** — which values become columns. Before 1.10 this was a required hand-picked list (uncategorized entries were hidden whenever a list was given); `Since 1.10` it is a **mode** — *All columns*, *Used columns* or a specific selection — described below.

A **show additional information** toggle (on by default) surfaces entry metadata. Validation requires the field mapping and a category field to be present.

The classification field may live **inside a block** — including a repeatable block — and the board still resolves it, builds its columns, and places the cards. Previously only top-level fields were scanned, so a board mapped to a nested category field rendered with no columns. `Since 1.8`

Dragging a card to a **different column** is a category change, not a reorder — it now works regardless of whether the view is **orderable**. Previously this cross-column move was rejected outright on any board where orderable was off, even though orderable is meant to gate only reordering **within** a column. Reordering cards inside a column is still controlled by the orderable setting; moving a card between columns no longer is. `Since 1.8`

Column field and column modes `Since 1.10`

Which fields can split a board. The column field may be:

- a **category** field — but never a *multi-value* one: a card cannot sit in two columns, and a card move would overwrite the whole array. The save is refused with `MULTI_VALUE_CATEGORY_FIELD_NOT_ALLOWED` (`details.usage: 'kanban'`); the same applies to the **Subcategory** slot.
- a **reference** field — to records of other entities, to a **List of records**, or to a **List of values**. The reference must be single-value, not repeatable, not inside a repeatable block, and must not point at the view's own entity; otherwise the save fails with `KANBAN_REFERENCE_FIELD_NOT_ALLOWED` (`details.reason`: `multiValue`, `repeatable`, `repeatableBlock` or `selfReference`). The guard also works in reverse: turning a reference field a Kanban view uses into a multi-value or repeatable one is refused with `KANBAN_REFERENCE_FIELD_IN_USE`, whose `details.usages[]` lists the views to change first.

Column modes. Under **Kanban columns** the designer offers:

Mode (`columnsMode`)	Columns shown	Offered for
All columns (`all`)	Every current option — or, for a reference field, every record it may point to — **including values added later**.	Category and reference fields.
Used columns (`used`)	Only values that at least one card holds.	Reference fields only (saving it on a category field → `KANBAN_COLUMNS_MODE_NOT_ALLOWED`). It is the **only** mode when the field's entries source is a query with parameters, because the candidate set then depends on a value the board cannot know.
Specific columns (`specific`)	The values you tick (stored in `categories`).	Category and reference fields.

The designer always saves the resolved mode. A view saved before 1.10 has no `columnsMode`; it behaves as **specific** when it has a column list, otherwise **used** for a reference field and **all** for a category field — so existing boards render exactly as before.

Cards with no value sit in a **Not Set** column (labelled *Category Not Set* on the board). A board loads at most **20 columns** at a time; beyond that the end user sees a "some columns are hidden" notice suggesting another face. A reference value whose record (or List item) was deleted keeps its column, flagged *Deleted record*, so its cards stay reachable.

Allow users to change this setting (`allowUsersToChangeColumns`) lets end users pick their own columns from *View Settings → Columns* in the gallery, with per-column counts and a reset to the view's setting. The choice is session-only and is discarded when you change the view, its column field or the offered mode.

API. `GET /v1/entities/{entity}/entries/kanban/{fieldId}` accepts a category field or a Kanban reference field. For a reference field the caller must name the columns to load in `columns` (at most 20, no duplicates; missing → `KANBAN_COLUMNS_REQUIRED` , invalid → `KANBAN_COLUMNS_INVALID`); `categoryId` still loads a single column, and `CATEGORY_NOT_SET` is the Not Set column's id. The new `GET /v1/entities/{entity}/entries/kanban/{fieldId}/columns` returns the column values with their counts (`values[]` with `id` , `label` , `archived` , `deleted` , `count` ; `notSetCount` ; and, for a reference field, `parameterized`). Both routes, and their `/draft` twins, are on a Kanban view's execution-plan allow-list, and both accept `quickFilterParameters` (see Queries [Build Apps › Queries & filters]).

Per-card move checks `Since 1.10`

Dropping a card on another column calls `PUT /v1/entities/{entity}/entries/{id}/position` . On a board split by a **reference** field the server checks the target before it writes:

1. The target must be an item of the field's List, or a record of one of the field's source entities — otherwise `KANBAN_COLUMN_NOT_IN_SOURCE` .
2. The target record must not be archived — otherwise `KANBAN_COLUMN_ARCHIVED` .
3. If the reference is **cascaded** (filtered by another field of the same record) or sourced from a **parameterized query**, the target must be one that the card's own reference picker would offer — otherwise `KANBAN_COLUMN_NOT_ALLOWED_FOR_ENTRY` . A drop inside the card's own column (a reorder) and a drop on **Not Set** are never checked.

The move then writes the column value (`{id}` for a List of values, `{dataEntity, dataEntry}` for a record, `null` for Not Set) **without re-running the field's own validation**, and is guarded by the entry's `updateIdentifier` , so a move that races another edit fails with the usual update conflict instead of overwriting it. A refused move snaps the card back with a message. This replaces an interim 1.10 pre-release rule that refused *every* move on a cascaded or parameterized reference board. Reordering within a column still requires **orderable**.

Timeline

Displays entries as a chronological feed, pinned-first then by date. **Prerequisite:** timeline contribution must be enabled on the entity first (see the entity's timeline settings); otherwise the timeline face is rejected. Options:

- **Allow publishing / unpublishing posts** — lets anyone with access to the view publish or hide an entry.
- **Allow pinning** — keep an important post fixed at the top.

Timeline is the one face whose validation checks only the entity, not the face configuration.

Tree

Organizes entries in a hierarchy resembling a tree. **Prerequisite:** add a **reference field pointing to the same entity** and enable its loop-prevention (deny-circular-relation) option. Map:

- **Reference Field** (required) — the self-referencing relation; it must have circular-relation prevention enabled.

- **Primary Title** (required) — must be a valid title data type.
- **Secondary Title**, **Image / File** (icon or image), and **Main Reference** (optional).

Tree has the strictest validation: the view's data source must be **all entries** or a **query**, and the tree face **cannot be combined with filter-first**. The **trace-parents** option walks and displays a node's ancestors. **Since 1.2**

When an end user narrows the tree by **status or validity alone** — archived-only, obsolete-only, and so on, with no search term, no filter criteria, and no quick filter — the tree now expands so matching entries deeper in the hierarchy are fetched and shown. Previously a status-only selection was not counted as an active filter, so only root entries were returned and the face reported "no matching results" even when matches existed further down. `Since 1.8`

Portal fix: the incoming-relation sidebar now resolves correctly `Since 1.8`**.** Clicking a card's incoming relation on the Tree face opens a sidebar showing the related entries through that relation's own render view. The sidebar's requests run under the **main** view's access object, but it previously asked for the relation's render view through the unscoped view-by-id route, which the main view's execution plan never whitelists — the gateway rejected it as `INVALID_ACCESS` and the portal showed its "no access" page. It now resolves the view through the relation-scoped route instead, keyed by the main view's entity. The sidebar's own rendering was also fixed: it previously used a bespoke, ~780-line list that did not reliably re-render once its data arrived; it now reuses the same relation-list component the entry-details page's own Relations tab (above) uses, so the two look and behave consistently.

Mini-card

Displays a compact card list built around counters. Map:

- **Primary Title** (required)
- **Secondary Title** (optional)
- **Image / File** (icon) (optional)
- **Date** (optional)

The mini-card face is **not available on the mobile app** and has no server-side validation. To add counters, see the entity's counters configuration.

> To fill the mapped slots in any face, the fields must be specified in the view's display information first.

Designing the entries-list screen

The entries-list screen hosts the faces plus the controls end users use to find entries. Enable **Display entries list** to turn it on; the options below layer on top.

Filtering and search

- **Allow search** (on by default) and **Allow filters** — filters require search to be enabled first.
- **Entry meta-data filters** — filter by Created By, Creation Date, Updated By, Update Date, Entry ID.
- **Entry meta-data details filters** — filter by the fuller set of Created-By detail fields.
- **Current-step filters** and **Previous-step filters** — workflow-step fields such as current / previous step, step owners, arrival date, execution duration.
- **Save queries** — lets end users save their filters (requires filters enabled).
- **Create lists (collections)** — lets end users build personal entry lists based on their permissions.

If you build view or quick-filter criteria through the API, note that the criteria field-path dialect is surface-specific — view criteria expect `value.`-prefixed field paths, and a wrong path silently returns an empty list rather than an error.

See Criteria and field-path dialects for the per-surface reference.

Sorting, quick filters, and classification styles

- **Default sorting** — choose the fields and direction applied on open; the order field ascending is the starting default.
- **Quick filters** — expose the entity's quick queries as one-click filters, choose where each appears, and optionally apply some by default. `Since 1.10` this grew into a fuller configuration, described in Quick filters, groups and ordering.
- **Entry classification styles** — pick a style applied on open, optionally let end users switch styles, and choose which styles are available. End users then see a **Styles** menu in the view.

Where the sorting option sits `Since 1.10`. In the View Settings tab, the option that lets end users change the sorting now sits directly beneath **Default sorting**, so the two sorting decisions read as one block; the aggregate and grouping options follow below it. That option and the two grouping toggles are checkboxes rather than switches. Nothing about sorting itself changed.

Quick filters, groups and ordering `Since 1.10`

A gallery offers a quick filter **if and only if it is listed** in the view's quick-filter settings (`viewConfiguration.quickFilters[]`). For each listed quick query you set:

- **Placement** (`displayMode`) — **In status list** (the default: it appears inside the gallery's Status menu) or **Standalone toggle** (its own toggle placed just before Status; toggles that do not fit wrap onto a row below).
- **Default** (`isAppliedByDefault`) — pre-applied when the gallery opens. Several defaults are allowed. A quick query that contains **Parameter** conditions can never be a default (`QUICK_FILTER_PARAMETERIZED_DEFAULT`): it asks the user for its values each time it is turned on. Only a change is checked — a newly added query or a newly switched-on default — so an unrelated edit to an older view is not refused.
- **Order** (`order`) — drag the rows; lower renders first. A stored view without `order` keeps its array order.

The label end users see is the query's **Quick filter name** (localized), set on the query itself; when left blank the admin form copies the query name, and the two are independent afterwards.

Groups. Quick queries can be classified into **quick filter groups**, managed on the entity's *Queries* tab (*Quick filter groups* sub-tab — see Queries [Build Apps › Queries & filters]). A query belongs to at most one group. Enable a group on a gallery through `viewConfiguration.quickFilterGroups[]`, with the same **Default**, **Order** and **Placement** settings (a standalone group renders as a **Standalone dropdown**). A default group pre-applies every member. In the Status menu a group is a tri-state heading over its members; applying a group simply applies its member queries, so the entries request is unchanged.

Through a gallery, `GET /v1/entities/{entity}/queries` returns the gallery's quick filters plus the members of its enabled groups, so end users need no extra route. A quick filter whose query was deleted is skipped instead of failing the list. The mobile app honours the same names, placement, order and defaults.

Aggregates and grouping on the entries-list card `Since 1.10`

Two more entries-list options are configured here but **modelled on the entity**, so this page covers only the view side of them. For the model itself — the five functions, which field types accept them, which fields can be grouped and why the rest cannot, how group blocks are ordered, and how protected fields behave — see Entity Aggregates & Entry Grouping [Build Apps › Totals & grouping].

- **Aggregates** — a checkbox list over the aggregates defined on the entity; tick the ones this view should compute and show. **A view that ticks none shows none.** It does not inherit the entity's set, and it does not acquire one later because somebody added an aggregate to the entity. That intersection is resolved on the server, so it is a boundary rather than a display filter — and the cost is that a new aggregate stays invisible until you enable it on each view that should carry it.
- **Grouping** — an ordered list of grouping levels (the first is the outermost block), plus **start collapsed** and **let end users group**. Both are off by default. A grouping level must be one of the view's own listed fields; grouping by a field the view hides is refused.

Turning the entries-list card off hides both settings without clearing them — they come back untouched when you turn it back on.

A view's totals arrive from a **view-scoped summary call**, made separately from the entries request and computed over everything that view's own scope matches, its filters, search, quick filters and filter-first included. Calling the view-scoped route rather than the entity-wide one is precisely what limits the result to the aggregates this view enabled.

An enabled aggregate is only rendered when its field is one of the view's displayed columns. Enable *Sum of Amount* on a view whose column list does not include Amount, and the figure is left out of the grand total and of every subtotal rather than parked in an unrelated column. Whether the view *may* compute an aggregate is still a server-side decision; this rule only governs where a figure can be drawn. So if an expected total is missing, check the view's columns first.

Filter-first

Set the view to **filter-first** (search-first) so no entries load until the user fills a filter form. Configure up to four filter fields, and optionally **auto-fill** those values into new entries. Filter-first adds a human-verification (captcha) step for anonymous visitors and is **mutually exclusive with the tree face and with dashboards**.

Two panel fixes `Since 1.8.` Returning to a filter-first view whose search a user had already submitted — for example, navigating back from an entry's details page — used to redraw the full filter prompt on top of the results already fetched, instead of collapsing to the compact "Results for… / Change your search" bar; the same defect affected the view designer's own preview. And the designer's Query-type ("match") dropdown showed raw i18n keys (`OPERATORS.And`, `DATA_VIEW.CUSTOM_QUERY`) instead of their translated labels. Neither changed how filter-first searching itself works — both are display fixes.

Obsolete and archived entries

- **Show obsolete entries** — surface entries that need updating.
- **Show archived entries** — surface entries that have been archived.

Action-button layout

Choose how action buttons render inside the view — **Default**, **Horizontal**, or **Vertical** — with the primary button treated distinctly in each layout. You also enable which published buttons appear and drag to order them; a button linking to a related view lets you pick the related entity's view. An **apply-workflow-actions-only** option restricts the view's action buttons to workflow actions, hiding the rest. **Since 1.8**

Implicit system buttons are reconciled on save `Since 1.8`

Five system-button ids are **implicit**: `VIEW`, `EDIT`, `ARCHIVE`, `CREATE`, and `DUPLICATE`. They are not stored buttons — they are derived from the view's own configuration (whether the entries-list screen is enabled, which create / edit / archive operations the edit configuration allows, and whether duplicate is allowed).

When you save a view whose payload still carries an implicit button the current configuration no longer permits — the classic case is an `EDIT` or `DUPLICATE` button left behind after you switch the matching operation off — the save **succeeds and the disallowed implicit button is pruned from the payload** before the view is stored. It used to be rejected outright with a "system button not found" error, because the button-existence check looked the implicit id up in the stored-button repository, where it never exists.

Only those five ids are eligible for pruning. **Custom system buttons and template buttons are never pruned**: they are still resolved against the stored button, and a reference to a button that does not exist is still an error.

Exporting entries

End users can export a view's entries in three ways:

1. **XLSX export** — export the current view's entries to an Excel (`.xlsx`) file; the exported columns follow the view's configured fields (`GET /v1/entities/{entity}/views/{id}/entries/export/xlsx` , web and mobile).
2. **Template export** — render selected entries into a print / export **template** (authored as **DOCX**, **HTML**, or **XLSX**) and, where configured, convert the result to **PDF**.
3. **Quick export** `Since 1.10` — the gallery's **Export Entries** button, offered only when you turn on **Allow export**; it produces **XLSX or CSV**. See Gallery quick export.

CSV exists only in the quick export (and its admin twin, the entity page's *Export Entries*). The XLSX export, template exports (in their template's format, optionally converted to PDF) and reports remain **XLSX / PDF only**. Large template jobs run in the background so the user can keep working; smaller jobs run in the foreground and download immediately. Sensitive-field permissions are respected in the output. A template export does not apply quick filters that carry parameter values; the user is told so when it starts.

Gallery quick export `Since 1.10`

Allow export (`viewConfiguration.allowExport` , default off) adds the orderable system button **Export Entries** (`EXPORT`) to the entries list. It is only meaningful while the entries-list screen is on: saving a view with the list off forces it back to `false` , and the button is pruned when export is off — the same reconciliation as the implicit system buttons. When on, the view's execution plan allow-lists the export route.

End users pick **Excel (XLSX)** or **CSV** in a small dialog, and the file contains **every entry matching the list as they see it** — search, filters, quick filters (with their parameter values), tag filters, filter-first values and grouping order — in the table's current columns and column order (or the view's fields, in order, when the gallery has no table face). The route is `GET /v3/entities/{entity}/views/{id}/entries/export` with `format=xlsx|csv` ; it reuses the list's own scope, so filter-first verification, the view's query and the sensitive-field guards all apply. Callers need ordinary access to the view's entries; with the list or **Allow export** off the route answers `EXPORT_NOT_ALLOWED` .

Matching entries	What happens
Up to `MAX_ENTRIES_RESULT_LIMIT` (default **10,000**)	The file is returned directly.
More, **signed-in** user	The export is queued; the user gets a *file ready* notification. The background file holds at most `MAX_BACKGROUND_EXPORT_ROWS` (default **100,000**) rows and is named `…_first-<n>` when cut.
More, **anonymous** visitor	Refused (`EXPORT_TOO_LARGE_FOR_ANONYMOUS`) unless the dialog's **Export first 10,000** is chosen (`truncate=true`).

The CSV is UTF-8 with a byte-order mark, comma-separated with CRLF line ends; a text cell starting with `=` , `+` , `-` , `@` , tab or carriage return gets a leading `'` unless it is plain numeric text (formula-injection guard). A block field becomes

`Block › Field` columns in CSV and a merged two-row header in XLSX. The file carries entries only — no group headers or totals.

Administrators get the same export on the entity page (**Export Entries**, `GET /v3/entities/{entity}/entries/export`): signed-in users who may list that entity's entries, every entity field in definition order by default, named after the entity. The **mobile app has no quick export**; it keeps the XLSX and template exports.

Designing the entry-details screen

The entry-details screen shows a single entry. Design it by enabling **Entry details view** and selecting the **fields** to display (they must be set in the display information first). The entries-list screen must be enabled before the entry-details screen — end users reach an entry's details by selecting it from the list first. Additional panels:

- **Discussion** — let end users add and react to comments. You can allow **mentioning people** (and restrict the mention scope to all users, view-permission users, or a custom static / dynamic scope). When you enable **attachments**, you choose which file types are permitted **for that view** and set a maximum attachment size (around **1 MB** by default) — the allowed types are configured per view, not a fixed list.
- **Entry history** — show modifications, who made them, reactions, and **reaction details** (the reactions plus who reacted and when), each toggled separately.
- **Workflow path** — show the entry's workflow diagram, optionally controlling how step-owner names are displayed (all steps vs. only steps the user owns).
- **Emails** — show UI-integration and background-integration emails related to the entry, scoped to all / sent-by-user / received-by-user.
- **Relations** — show related entries as a **tab** or an inline **sub-view**, depending on the chosen display method. When you choose sub-view, a **Display Mode** sub-option appears (**Show as Block**).
- **Navigation between entries** — let the end user page between entries without returning to the list.

Default workflow-history view (Path/History) `Since 1.8`

When the workflow panel lets the end user reach the execution/history view in addition to the main path, you can also choose which one the tab **opens on** by default: **Path** or **History**. Previously the tab always opened on Path, with no way to configure this.

The default only takes effect once history access itself is turned on for the panel — with it off, History stays unreachable and the tab opens on Path regardless of what the default says, so turning this setting on for a view that doesn't expose History has no effect. Web and mobile both honor the same admin-configured default. See Executing Workflow Actions [Using OSOS › Acting on Workflow Steps] for how end users read and switch between Path and the execution view at runtime — not duplicated here.

Designing the create, edit, and archive screens

The edit configuration controls how end users add, edit, and archive entries. Enable the relevant operation and, for each:

- **Create entries** — select the fields the end user may fill (at least one required), then set the **after-save action**: show a message, return to the entries list, clear the fields for another entry, or go to the entry details.
- **Edit entries** — same field selection and after-save choices; edit also exposes an **Administrative Permissions** section for entities with a workflow, letting authorized users edit, execute a workflow action on behalf of a step owner, end / move / return / reset the workflow, or change a step owner temporarily or permanently.

- **Archive entries** — set the after-save action after archiving.

Other edit options include inline edit, quick create, save-as-draft, and duplicate. Note that entry meta-data fields cannot appear in an operation's field list, and edit configuration is not supported for virtual-proxy entities.

Granting only some columns of a repeatable block. An operation may list just a few of a repeatable block's sub-fields. The block's rows are saved as one array that replaces the stored one, and the server then restores, row by row, the stored values of the columns the operation does not grant, so they are no longer wiped on save `Since 1.4`. From release 1.10.1 a row is matched by its `__id` only, and a row without one is a new row with nothing restored `Since 1.10`. Granting any sub-field of the block also lets the user add and remove rows. The client contract is on Block: saving rows through a view [Build Apps › Block field].

Repeatable reorder `Since 1.8`

Repeatable reorder (`allowRepeatableReorder`, default off) is a **per-operation view-configuration option**: it sits on each entry in the edit configuration's supported-operations list, next to that operation's field selection and after-save action. It is **not** an entity-field option and is not stored on the field definition, so the same field can be reorderable on the Create screen and fixed on the Edit screen of the same view — or reorderable on one view and not on another.

The option becomes selectable only once the operation's field list contains at least one **repeatable** target — a repeatable field, or a repeatable block. Matching is on the field's full name, so two nested fields that merely share a name segment (`blockA.title` and `blockB.title`) are never confused for one another. If you later remove the last eligible field from the operation, the option resets itself to off.

What the end user gets, when it is on: on the create or edit form, a drag handle appears next to each repetition of a repeatable field's values and next to each repetition of a repeatable block. Dragging a handle reorders the repetitions in place and marks the form dirty, so the new order is saved with the entry. The handle only appears when there is **more than one** repetition — with zero or one, there is nothing to reorder and dragging is disabled.

> **Both clients honour this setting** `Since 1.8`. The mobile app reads the same per-operation `allowRepeatableReorder` flag off the view configuration and renders its own drag handle on each repetition beyond the first, so a view configured for reorder behaves the same in the web portal and on the phone. (An earlier version of this page described repeatable reorder as web-only; that is no longer the case.)

Category options from the entry card `Since 1.10`

You can let end users **add** or **edit** a category field's options while they create or edit an entry, without an administrator editing the entity. Three settings must line up, and the server enforces the first two:

Setting	Where	Meaning
Allow users to manage options (`allowUsersToManageOptions`)	The category field's settings on the entity	The field accepts user-managed options at all.
Add category options / **Edit category options** (`allowAddCategoryOptions`, `allowEditCategoryOptions`, default off)	Each create / edit operation of the view	Whether that card may add, or edit, options. The field must also be editable in that operation.
Link create and edit card settings (`linkCategoryOptionSettings`, default on)	The view's edit configuration	The create and edit cards share their two toggles. Enforced by the designer only.

The new or edited options travel with the entry write itself (`categoryOptionChanges` on the create / update body, keyed by the field's full name), so they are saved together with the entry. Labels are per language and the default colour is `#81C784`. The save is refused with `CATEGORY_OPTION_CHANGE_NOT_ALLOWED` (not permitted here), `INVALID_CATEGORY_OPTION_CHANGE` (malformed, unknown field, empty label), `CATEGORY_OPTION_CONFLICT` (someone else

changed or deleted that option) or `CATEGORY_OPTIONS_CHANGED` (the field's option list moved since the card loaded) — the user reloads and retries.

Multi-value category fields. A category field can now hold several values. Such a field cannot be a Kanban column or subcategory field, a default-sort field, a dashboard statistic's group-by, a timeline category or the entity's main / sub category; see Multi-value category fields [Build Apps › Totals & grouping] for the full rule.

Ask User For

Enable **Ask User For** to select a set of fields the view prompts the end user to fill when a UI event (such as an action button) is executed. At runtime those fields are presented in a dialog, letting the event collect extra input before it runs.

Dashboard screen

Enable the **dashboard view** so end users can see summaries and statistics for the entity, and optionally make the dashboard the view's default landing screen. The dashboard **cannot be combined with filter-first**. Click-to-browse on the dashboard requires the entries-list screen to be enabled, and overriding the source for a statistic query requires a query-sourced view.

Summaries on the entity overview respect "match the view source" `Since 1.8`. With **Show only summaries that match the view source** set, the Entity Overview panel used to list every summary of the entity anyway, and each incompatible one then failed to run with "Invalid summary source". The list (`GET /v1/entities/{entity}/statistic-queries` on a data-view request) now applies the same check that running a summary already did — the summary is on the view's entity, the view's dashboard is enabled, and its source matches when the view restricts it — and pages **after** that filter, so `skip` and `limit` count only the summaries the view can show. Requests without a view, and requests through a page, are unchanged.

Pinning a view to the sidebar

Pin a designed view to the sidebar with **Pin to Sidebar** so end users can reach it directly. The **sidebar-display behavior** controls the pinned view's open state — **closed**, **open**, or **retain** (keep the last state).

Using and switching between faces

At runtime the view opens on its **default face** — the single face marked as default when the view was designed. The end user switches to any other enabled face with the display switcher. A view's **screen order** determines which screen a user is redirected to (entries list, create, single-entry details, dashboard, or details-with-navigation). **Since 1.4**

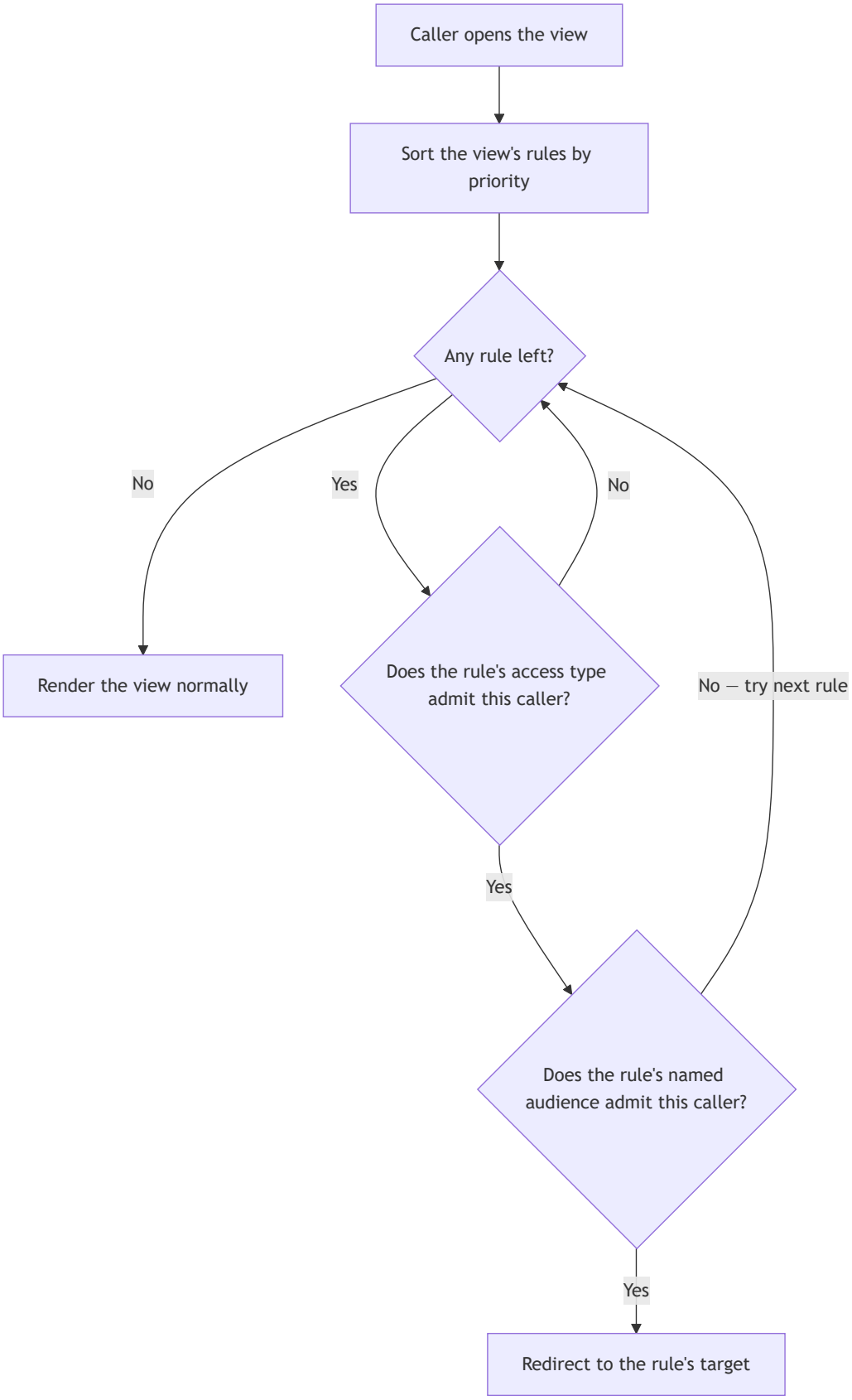
Record ordering / reordering is available on the Table, Badge, and Kanban faces. On mobile, dragging a Kanban card to another column is persisted to the server, though reordering whole Kanban columns is not yet supported.

Redirection rules

A view can carry **redirection rules** that send an end user straight to a different target — a fixed URL, another data view, a page, or a report — before the view itself ever renders. Rules are evaluated in **priority order** (lowest first); the first rule that matches the caller wins, and evaluation stops there. A caller no rule matches sees the view normally.

Each rule combines two conditions that must **both** hold for it to match:

- **Access type** — `Anonymous`, `Authenticated`, `Everyone`, or `Restricted`. This is the coarse gate: an `Anonymous` rule only ever considers a signed-out visitor, an `Authenticated` rule only a signed-in one, `Everyone` always qualifies, and `Restricted` qualifies everyone at this first gate — it exists purely to be narrowed by the second.
- **Named audience** — specific users and/or groups attached to the rule. A rule naming nobody matches everyone its access type already admitted, **except** a `Restricted` rule: naming nobody there matches nobody at all, since an empty `Restricted` rule that matched everyone would defeat the point of choosing `Restricted` in the first place. A rule that does name users or groups matches only a caller in that list — for every access type, not `Restricted` alone.



`Since 1.10`, both halves of this match are checked on every rule, for both data views and pages. Before this fix, only a `Restricted` rule ever consulted its named users/groups — an `Authenticated` rule that also named a group matched *every* signed-in caller regardless of membership, so the named group was silently dead configuration. A related crash was fixed alongside it: a page carrying redirection rules used to render **completely blank** for anonymous visitors, because the evaluator choked on an anonymous caller's missing group list instead of treating it as empty. Both fixes are also backported onto the two most recent release lines as patch-level fixes, not as a new capability on those lines.

Redirection rules are the same mechanism on a data view and on a page [Build Apps › Pages & Widgets] — see that page for how a page's default-page setup and its own redirection rules fit together.

Previewing a view while you design

While building a view, use the builder's **preview** to see how it renders against deterministic sample (fake) data — no real entries required. A **mobile preview** tab lets you check how the column / card choices and action-button layout look on a phone before publishing. Preview is a request-scoped mode (driven by a preview header), not a stored view property, and the relaxation it applies to authorization is narrow — it lets a genuinely not-yet-existing resource through so the sample data can render, it is not a blanket permission bypass. See Preview Mode [Build Apps] for the request contract, the generated sample data, and the live mobile preview. **Since 1.8**

Per-view styling, help, and scripts

Beyond the display options above, a view can carry its own presentation tweaks, inline help, and in-browser logic — all authored in the view builder, with no deployment.

Custom CSS and element styles

Attach CSS style rules to a view, or to individual elements within it, to fine-tune its appearance (`appCssStyler` ; a view's rules are served through `get-view-config-css-rules`). This is distinct from the **entry classification styles** above — those recolor entries by a classification field, whereas custom CSS lets you style arbitrary parts of the view. Because the rules are broad, test any custom styling in both light and dark themes and in the mobile preview before publishing.

Fix: saved CSS classes show again in the custom-style settings `Since 1.8` **.** Classes saved for the view, its fields, dashboards, buttons and workflow steps were stored and applied, but the settings panel showed them blank after a reload. The panel now shows each saved class, and a class you clear stays cleared. On the 1.10 line this arrives with release 1.10.2.

A related fix (`Since 1.8`) corrected how a **script's** own DOM-driven styling resolves when the same view is embedded more than once on a Page — see Per-view and per-entity scripts below.

Contextual help

Author **per-view and per-field help content** in markdown; OSOS renders it in a help sidebar alongside the view (the `help-and-documentation` feature). Use it to explain a view's purpose or clarify individual fields inline, without sending end users to a separate document.

A field's help note can also be authored **per screen** — the entries list, entry details, create, edit, and "ask user" screens each resolve their own help independently, since a field can appear on some screens and not others. The designer's **Contextual Help** panel is an accordion covering the whole view, then one section per screen, listing every field any of the five screens can resolve — so help for a field that appears in (say) the edit operation but not in the view's own base projection is still reachable and authored against the right screen.

Fix: a block sub-field's help note no longer crashes the view `Since 1.8` **.** Adding contextual help to a field nested inside a block used to make every screen of the view throw and fail to open. It is fixed by resolving a sub-field's help note to the top-level field it belongs to (so the note renders nested under that field, instead of a lookup that never had the sub-field as a key) and by building each screen's help options from that screen's own field list rather than the view's base field projection — so help for a field only present on, say, the edit screen no longer crashes screens that don't have it.

Per-view and per-entity scripts

For logic that runs entirely in the end user's browser — with no function to deploy — you can attach user-configured JavaScript to a view or its entity. On the entry view/edit lifecycle the frontend runs the **view-face** script first, then the **entity-face** script (`scriptAttributes[face]`), via `EntityScriptExecutorService.run`. This is one of the platform's ways to run your own logic without a deploy, alongside user functions and reactions.

Fix: a script's DOM effects now target the right widget instance `Since 1.8`**.** A script commonly styles or toggles an element by id (`document.querySelector('#some-id')`). When the same view is embedded more than once on a Page as a data-view widget, every instance duplicates the same admin-configured element ids, so an unscoped lookup always matched the **first** instance on the page — a script-driven style could land on the wrong widget, or never visibly land on the one actually being viewed. Lookups made from inside a widget's script (`document.querySelector` / `querySelectorAll` / `getElementById`) now search only that widget's own subtree; scripts run outside a widget instance are unaffected.

Saving a view: conflicts and refusals `Since 1.10`

- **A save only lands on the version you read.** A view is saved as a full document, guarded by its `updateIdentifier`: a stale token is `409 UPDATE_CONFLICT`. Before 1.10 only an up-front read compared the tokens, so two saves that both passed that read could overwrite each other. Now the write itself is conditional — the view is claimed only while its stored `updateIdentifier` still matches the one you read — so the second of two concurrent saves loses with a 409 and must re-read. Pages are saved the same way.
- **Entry edits need an allowed operation.** Saving a view that enables entry edits without any supported operation is a `400` (`DATA_VIEW_INVALID_ALLOWED_OPERATIONS`, "Allowed Operation must be provided"). It used to surface as a `500`.
- **A view no audience can open is a 403, not a 500.** A view, page or report that enables **neither** the anonymous nor the authenticated audience is legitimate data — a Proxy view is stored that way on purpose — but asking the platform whether a caller may open it (`GET /v2/access-object/{id}`) used to crash with a `500`. It now answers `403 ACCESS_OBJECT_NO_AUDIENCE`, and the gateway and the audit, comments, timeline and workflow services pass that 403 through unchanged.
- **A view save with no body is a 400** `Since dev`. `PATCH /v1/entities/{entity}/views/{id}` now declares its body required, so a request with no body at all is refused with `400 MISSING_REQUIRED_PARAMETER` before it reaches the controller; it used to fail as a `500`. A partial body is accepted as before.
- **Files are tracked with the view.** Saving a view or page records the files it references before the write and releases the ones it no longer uses afterwards; deleting it releases them all.

Installed views (from a package) `Since 1.10`

A view written by a package **install** is locked by that package: a write is refused with `403 PACKAGE_RESOURCE_READ_ONLY` ("belongs to installed package …, fork it to edit"). The client still owns a few things on it, and an **update of the package keeps them**:

What	Rule
Published / hidden (`isPublished`)	Changed from the entity's views list; never checked against `updateIdentifier`.

What	Rule
Name, description, icon	Editable in the view editor (not the name of an implicit view). The editor shows the package's value beside yours, and the package's own value is recorded in `packageLabels` the first time yours differs; typing the package's value back removes that record. A label change needs the current `updateIdentifier` (`409` otherwise). `description: null` always clears; `icon: null` clears the icon only with `?clearIcon=true`.
Display order, audience, access rights	Kept on update (the client's values are laid back over the incoming view).

A save through `PATCH /v1/entities/{entity}/views/{id}` on a locked view writes **only** those keys — partially, so nothing else in the stored view is touched — and anything else in the body is ignored; a body that changes none of them is refused with the original 403. The rest of the editor is inert (see the packaging screens); relation overrides and scripts stay usable. To change anything else, fork the installation.

Layout references on the end-user route `Since 1.10`

For a view on the **Layout builder** display template, `GET /v2/views/end-user/{id}` resolves the SHELL entries the template names (`layout_list`, `layout_view`, `layout_edit`) into a `layouts` property — `{listEntry, viewEntry, editEntry}`, each the SHELL entry's `{id, name, definition}` or `null` when nothing is linked or the entry is missing, not active or invalid. The template stores only the references, so a layout edit reaches every linked view without re-saving it, and anonymous visitors still receive the definition. The v1 route does not carry `layouts`. **The backend half is in 1.10, but no shipped client uses it yet:** the Layout builder template and the shell builder that read `layouts` are `Since dev`.

Version notes

Foundational pieces — the seven faces, filter-first, dashboards, and the entry / edit screens — are **Long-standing** (present since before 1.2) and carry no badge. The design-facing additions since then:

Feature	Since
Tree trace-parents	1.2
Screen order (redirection between screens)	1.4
Table row-click behavior	1.6
Action-button layout	1.8
Preview mode	1.8
Repeatable reorder (per-operation option)	1.8
Implicit system buttons pruned on save instead of rejected	1.8
Kanban classification field inside a block	1.8
Kanban column changes allowed regardless of the orderable setting	1.8
Tree face expands on a status-only filter	1.8
Default workflow-history view (Path/History)	1.8
Tree face's incoming-relation sidebar resolves correctly	1.8
Filter-first panel restore + Query-type label fixes	1.8
Contextual-help crash on a block sub-field fixed	1.8
View-widget script DOM scoping fixed for a view embedded more than once on a Page	1.8

Feature	Since
Aggregates enabled per view, on the entries-list card	1.10
Entry grouping configured on the entries-list card (levels, start collapsed, let end users group)	1.10
Sorting option regrouped directly under Default sorting in View Settings	1.10
Proxy view kind (`kind: Proxy`), governing a Proxy virtual entity's write capabilities	1.10
Redirection rules: two-gate audience match, and an anonymous-caller crash fix	1.10
Kanban: reference column field, column modes (all / used / specific), per-user column choice, per-card move checks	1.10
Quick filters: name, placement (status list / standalone), explicit order, groups, multiple defaults	1.10
Gallery quick export (**Allow export**, XLSX or CSV) and the admin Export Entries	1.10
Category options added / edited from the entry card; multi-value category fields	1.10
Conditional save: a view or page is written only while its `updateIdentifier` still matches	1.10
Installed view: rename, change the description, clear the icon, kept across package updates	1.10
A view with entry edits but no allowed operation answers 400; a view no audience can open answers 403	1.10
Entity-overview summaries filtered to the view's source when "match the view source" is set	1.8
A save through a view keeps the repeatable-block columns the view does not grant	1.4
Saved CSS classes shown again in the view's custom-style settings	1.8
Repeatable-block rows matched by `__id` on a view save (from release 1.10.1)	1.10
View `PATCH` with no body answers 400 instead of 500	dev
Layout references resolved on `GET /v2/views/end-user/{id}` (backend only; the consuming client is `Since dev`)	1.10

Permissions API

This page is for integrators and engineers who manage access programmatically or need to know exactly what the permission screens in Administration do underneath. The administrator's view of the same model — in plain language, with the screens — is Permissions & access [Administration]. The evaluation engine itself (authorizer, voters, merge, caching, the OpenFGA shadow check) is documented for platform engineers in Engine: Permissions & Voters. The generated, per-operation endpoint list is Access — data-service.

All paths below are relative to the data API (`/api`); every response comes in the standard `{success, result}` envelope described in Calling the HTTP API.

The grant record

Every permission is one `AccessRight` record (collection `access_right`), one record per grant. Its `value` (`AccessRightValue`) holds:

Field	Meaning
`entitled_entity`	Who the grant is for: a reference `{dataEntity, dataEntry}` where `dataEntity` is `USER` , `GROUP` , or `ROLE` and `dataEntry` is the id.
`entitlement`	What the grant applies to: a reference `{dataEntity, dataEntry}` where `dataEntity` is the entitlement type (module, entity, view, …) and `dataEntry` is its id. **Null/absent for system-level grants.**
`level`	The access level (see below).
`permission_type`	Optional. Distinguishes a `NORMAL` grant from a `SENSITIVE_FIELD` read grant. Absent is treated as `NORMAL` .
`scope`	Optional canonical path (e.g. `module/entity/field`) used by sensitive-field grants.
`expiry`	Optional validity window stored as `[startIso, endIso]` ; `null` means the grant never expires.

So "grant role R Edit on module M" is a single record with `entitled_entity = {ROLE, R}` , `entitlement = {DataModule, M}` , `level = EditAccess` .

Related models:

Model	Role
`AccessRight`	`access_right` — the stored permission record.
`AccessRightValue`	`entitled_entity` + `entitlement` + `level` (+ `permission_type` , `scope` , `expiry`).
`EntitledEntityPermission`	Subject reference: `dataEntity` ∈ `{USER, GROUP, ROLE}` .
`EntitlementReferenceProperty`	Resource reference: `dataEntity` ∈ `EntitlementEnum` .
`GroupDto` / `UserDto`	A group/role or user with its list of `AccessRightDto` permissions.
`RoleDataEntry` / `RoleDataEntryValue`	A named role: name, description, org-unit linkage, type, assignment type (see Roles data model).
`GrantAccessDto`	Write body — a single `level` .

Who a grant can target (entitled entities)

Grants target one of three entitled-entity types (`EntitledEntityEnum`):

Value	Meaning
`USER`	An individual user.
`GROUP`	A group. Groups carry a `GroupTypeEnum`: `UserGroup` (0), `Role` (1), `OrgUnit` (2), `Virtual` (3), `QueryGroup` (4).
`ROLE`	A role.

Roles and org units are groups distinguished by their `GroupTypeEnum`, so a single grant record can target a plain user group, a role, a virtual/query group, or an organizational unit. Every role has a virtual role group, created automatically, that contains all users holding the role. In the permission-list APIs the entitled-entity type in the path is `user` or `group` (case-insensitive; it is upper-cased before use).

What a grant can apply to (entitlement scopes)

Scope (`entitlementType`)	Stored `dataEntity`	What it controls
`System`	`System`	The whole tenant. `entitlementId` is null.
`Module`	`DataModule`	A single module. A module grant also cascades to the entities inside it.
`DataEntity`	`DataEntity`	A single data entity.
`DataView`	`DataView`	A single data view.
`GenericView`	`GenericView`	A page.
`Report`	`Report`	A report.

A grant on a module surfaces the entities inside it. A grant on a single entity surfaces its parent module only as a navigation container and does **not** reveal the sibling entities. `Since 1.8` the parent-module ids reached this way are returned to clients as `entityParentModuleIds` on the current-user response (see The current-user response).

Access levels

`AccessRightLevelEnum` is a **numeric, ordered** ladder; several checks compare levels numerically.

Level	Value	Meaning
`None`	`-1`	The "remove" sentinel. Sending it **deletes** the matching grant (see Upsert behaviour); it is never stored by the upsert.
`Restricted`	`-0.5`	System-only "onboarded floor". Grants nothing on its own; it lets the user's resource grants take effect and unlocks read-only access to core infrastructure (users, groups, org units) for a user who holds at least one module or entity grant. `Since 1.8`.
`ViewAccess`	`0`	Read the definition of the object.
`DeveloperAccess`	`0.5`	Builder access to the studio tools (entity designer, views, themes, fonts, languages). It sits *beside* Edit: Edit satisfies every check Developer does, but Developer does not grant record access. Long-standing (predates 1.2).
`EditAccess`	`1`	Read and write, including entry data.
`AdminAccess`	`2`	Full control, including managing permissions. Offered only at **System**; `Since 1.6` an Admin grant on a module or entity opens nothing.

Levels each target offers (`GRANTABLE_LEVELS_BY_ENTITLEMENT`): System — View, Developer, Edit, Admin (plus Restricted); module and entity — View, Developer, Edit; view, page and report — View.

Entry data needs Edit. Listing the entries of an entity requires Edit on the entity, Edit on its module, or system Edit/Admin — View is not enough. `Since 1.8` reading a single entry by id (for example from a relation's quick-view

sidebar) requires the same Edit-tier access; the request is refused with `ENTRY_READ_FORBIDDEN` otherwise. Reads made through a view, page or service follow that object's own access rules instead, and Service-type entities keep their exemption.

A user with no system grant is denied. If the merged system bucket is empty (no direct or group system grant), the top-level authorizer denies every request that is not served through a view context. The users-service grants nothing by default when a user is created, so onboarding must give each user a system level — at least `Restricted` — directly or through a group.

Granting, changing and revoking

Grants are created and changed through an idempotent **upsert**. There are two forms — one addressed by entitlement type and id, one addressed by a system-object id:

```
PUT /v1/permissions/{entitlementType}/{entitlementId}/{entitledEntityType}/{entitledEntityId}
PUT /v1/permissions/{systemObjectId}/{entitledEntityType}/{entitledEntityId}
```

Request body (`GrantAccessDto`):

```
{ "level": 1 }
```

`level` is required and must be one of the numeric `AccessRightLevelEnum` values. Path parameters:

- `entitlementType` — `System` | `Module` | `DataEntity` | `DataView` | `GenericView` | `Report`.
- `entitlementId` — the target object's id. For `System` it is null (the router parses a missing value to the string `"null"`, which is accepted and treated as System).
- `entitledEntityType` — `user`, `group` or `role` (upper-cased and validated against `EntitledEntityEnum`).
- `entitledEntityId` — the id of the user, group or role.

The system-object form looks the object up in the `SystemObject` collection, maps its type to an entitlement (`DataView → DataView`, `Page → GenericView`, `Report → Report`) and performs the same upsert. If the object does not exist or its type is not one of those three, the call fails with `ENTITY_NOT_FOUND` / `INVALID_ENTITILED_ENTITY`. The system-object form and the per-user/per-group permission DTO reads are `Since 1.6`.

Upsert behaviour

`AccessRightsService.upsertPermission` applies these steps in order:

1. **`Restricted` outside `System` is a revoke** `Since 1.8`. `Restricted` is never *stored* on a Module/Entity/View grant. Requesting it there is normalized to `None` and routed through the revoke path (step 4) — revoking a resource grant is exactly what drops a principal back to their system-level `Restricted` floor. (Before 1.8 such a request failed with `RestrictedLevelSystemOnlyException`, which left no way to bring a raised Module/Entity grant back down short of deleting it.)
2. **The entitled entity must be Active** — the controller loads the user/group and rejects with `EntitledEntityNotActiveException` if its `state` is not `Active`.
3. **Existing grant lookup** — it finds a matching non-`Archived` grant for that (entitled entity, entitlement). System grants are matched by the entitlement being absent/null, resource grants by the entitlement reference. Archived records are skipped so a dead grant is never silently reused. `Since 1.8` the system-grant match also excludes `permission_type = SENSITIVE_FIELD` records: both shapes carry a null entitlement, and before the fix granting a system level to a user who held only a sensitive-field grant matched that grant — the call reported success, no system permission was created, and the sensitive-field grant was corrupted with a stray `level`.

4. `level === None` ⇒ **delete** — the matching grant is removed. This applies at every scope, System included. With no matching grant, `None` is a no-op.
5. **Downgrade guard** — if a matching grant exists, the new level is *lower* than the current one, the entitlement type differs from the stored grant's, and the scope is not System, the call throws `UserGroupAlreadyHasHigherPermissionLevelException`.
6. **Otherwise update or create** — an existing grant's level is updated (and `updateInfo` stamped); otherwise a new `Active` `AccessRight` is created in the public store.
7. **Event** — an `AccessRightChangedEvent(entitlementType, entitlementId)` is published so caches and dependents refresh. How the caches are invalidated is in Engine: Permissions & Voters — Generation (epoch) invalidation.

There is no separate delete endpoint for a grant: send `level: -1` to the same `PUT`. Bulk replacement is handled internally by `storePermissionsInAccessRights(..., isUpdate = true)`, which diffs the submitted set against the current grants and adds/removes the difference.

Who may assign

The two upsert forms are guarded differently. Both also need a license that includes administration.

The form addressed by entitlement type and id is guarded by `voteByEntitlementFactory(0, 1)`, which for a `PUT` requires the caller to be able to **edit the entitlement being granted**:

- `System` grants → caller must be a system developer or higher (`isSystemDeveloper`).
- `Module` grants → caller must be able to edit that module (`canEditModule`).
- `DataEntity` grants → caller must be able to edit that entity (`canEditEntity`).

The system-object form carries no entitlement check of its own, so it is open to **system administrators only**: the platform admits a write route that names no allowed roles for a system admin alone. (The entitlement voter this form once carried could never reach a decision and was removed in 1.8.)

A grant cannot exceed the caller's level `Since 1.8`

Unless the caller is a system **Admin**, the form addressed by entitlement type and id refuses:

- a requested level **above the caller's own effective level** on that target, and
- **any change to, or removal of, a grant that is already above** the caller's level.

The refusal is `403 PERMISSION_GRANT_EXCEEDS_CALLER_ACCESS` ("You cannot grant, change or revoke access above your own level"); its `details` name `entitlementType`, `entitlementId`, `callerLevel`, `requestedLevel` and `currentLevel`. The caller's effective level counts their system grants, direct and group grants on the target, and what they inherit from its module or entity; under a delegation, the delegation's permissions count. Internal platform calls (user onboarding, reaction actions) are exempt. Before 1.8 the edit check was the only check, so a system Developer could make themselves system Admin. Engine detail: A permission grant is capped at the caller's own level.

Reactions that set or remove permissions `Since 1.10`

Reactions can carry *Set permission* and *Remove permission* actions. When such a reaction is **saved**, the platform checks that the person saving it already holds what the action would hand out: a *Set* needs at least the granted level on that resource, a *Remove* needs **Admin** on it. Only the levels the permission editor offers are accepted per resource type (System: View, Developer, Edit, Admin; module and entity: View, Developer, Edit; views, pages and reports: View). A failing check refuses the save with `PERMISSION_ACTION_EXCEEDS_SAVER_ACCESS`; if the author's access cannot be read, the save is refused as well. The system user is exempt. When the reaction runs, a *Set* never lowers an existing direct

grant (the principal is reported as skipped, "already higher"); a *Remove* lowers by design. Engine detail: Internal access reads and the reaction ceiling.

Parameter validation

Before any grant or list operation runs, `validatePermissionsParams` checks the request path:

- `entitlementType` must be a valid `EntitlementEnum` — otherwise `PermissionScopeNotSupportedException`.
- `entitledEntityType` must be a valid `EntitledEntityEnum` (`USER` / `GROUP` / `ROLE`) — otherwise `PermissionScopeNotSupportedException`.
- For any non-`System` entitlement, `entitlementId` must be provided — otherwise `EntitlementTypeRequiredException`.

Reading and auditing permissions

List endpoints accept the standard LoopBack `filter` / `where` query parameters and a `/count` variant.

Endpoint	Guard	Returns
`GET /v1/permissions/{entitlementType}/{entitledEntityType}`	`voteByEntitlementFactory`	Users/groups with their permission on a specific system/module/entity/view.
`GET /v1/permissions/{entitlementType}/{entitledEntityType}/count`	`voteByEntitlementFactory`	Count of the above.
`GET /v1/permissions/list/{entitledEntityType}`	module/system voter	All users or groups in the system, optionally scoped by `moduleId`.
`GET /v1/permissions/list/{entitledEntityType}/count`	module/system voter	Count of the above.
`GET /v1/permissions/current_user`	authorization skipped	The caller's own effective permissions.
`GET /v1/permissions/user/{userId}`	`systemLevelVoterFactory`	A user's resource permissions (`UserPermissionsDto[]`).
`GET /v1/permissions/user/{userId}/admin`	`systemLevelVoterFactory`	A user's admin-level permissions.
`GET /v1/permissions/group/{groupId}`	`systemLevelVoterFactory`	A group/role/org-unit's permissions.
`GET /v1/permissions/group/{groupId}/admin`	`systemLevelVoterFactory`	A group's admin-level permissions.
`GET /v1/permissions/anonymous`	`systemLevelVoterFactory`	Permissions applied to the anonymous audience.
`GET /v1/permissions/system-object/{systemObjectId}`	`systemLevelVoterFactory`	Every user/group with access to a given object (`ObjectPermissionsDto[]`). `Since 1.6`.

The `systemLevelVoterFactory` guard requires a system viewer for reads and a system developer for writes. `Since 1.8` it carries that name; it was previously `systemAdminVoterFactory`, a rename only — the check never required Admin. The `user`, `group` and `admin` endpoints each have a `/count` variant.

Query flags on the list endpoints:

- `include_virtual` — virtual (system-synthesized) entities are excluded by default (`virtual: {neq: true}`); pass `true` to include them.
- `only_none_permission` — restrict the result to entitled entities with **no** permission (or a stored `None`) — this is what the **Add** picker uses. `Since 1.8`.
- `entitlementId` — narrow to a specific object when listing by entitlement type.
- `moduleId` — on the `list` endpoints, scope to one module.

The user audit view

`GET /v1/permissions/user/{userId}` returns entries describing not just the object and level but **how** the access was acquired:

Field	Meaning
`object` (`ResourceObjectDto`)	The target: id, name, `type`, its `dataModule` / `dataEntity`, computed `abilities`, and `isAuthenticated` / `isAnonymous` / `isRestricted` flags.
`assignmentType` (`UserPermissionAssignmentTypeEnum`)	`Direct` (0), `group` (1), `authenticated` (2), `delegation` (3), `anonymous` (4).
`acquiredThroughGroupType` (`GroupTypeEnum`)	If via a group, whether it was a UserGroup/Role/OrgUnit/Virtual/QueryGroup.
`acquiredThrough`	The resolved group/role the access came through.
`adminLevel`	The effective admin level, when applicable.
`delegationId`	Set when the access came from a delegation (`getAllDelegateePermissions` resolves what a delegatee holds on the delegator's behalf).

`abilities` (`ResourceObjectAbilitiesDto`) exposes the booleans the client uses to show or hide actions: `create`, `edit`, `archive`, `entriesList`, `entryDetails`, and for shareable objects `audience` and `observer`. In the Administration screens these appear as the **Assignment Type** (**Direct**, **Indirect (Group)**, **Indirect (Authenticated)**, **Indirect (Delegation)**, **Indirect (Anonymous)**), **Acquired Through** and **Available Actions** columns.

`ObjectPermissionsDto` (the `system-object` endpoint) is the inverse view: for a single object it lists each entitled entity with its `type` (`EntitledEntityTypeEnum` GROUP/USER), `assignmentType`, and an `accessType` (`ObjectPermissionAccessTypeEnum`: `Default` (0), `Audience` (1), `Observer` (2)).

The current-user response

`GET /v1/permissions/current_user` builds the caller's own permission list. For a system editor or admin it returns just the system grants (no per-resource enumeration is needed). Otherwise it returns system and module grants, entity grants for entities **not** already covered by a module grant, and all view, page and report grants. It also returns `entityParentModuleIds` (`Since 1.8`) so the client can open a module reached only through an entity grant; the list is empty for system viewers/editors/admins.

Sharing and object visibility

Separate from user/group grants, a **shareable object** (data view, page, report) carries configuration that decides where it appears and who can open it once published. The platform applies both on every read: a user only sees objects that are visible to them **and** that they may open. Reports are web-only; on mobile a report link resolves to an unsupported-link placeholder.

Sharing levels

Every object carries one cross-space **sharing level** (`ApplicationSharingLevelEnum`), chosen when a module or application is created:

Level	Meaning
`PRIVATE`	Visible only inside the object's own space.
`SHARED`	Visible within the set of spaces associated with the object.
`PUBLIC`	Visible in every space (typically system-wide shared items, such as a space's default pages).

At read time a record is shown if its level is `PUBLIC`, or if it is `PRIVATE` / `SHARED` and the current space is among the object's spaces. `GET /v2/access-object/{id}` returns an `AccessObjectDto` with the resolved `authenticationMode`, `domainId`, `spacesId`, `isPublicSpace` (true when `sharing.level` is `PUBLIC`), `executionPlan`, `published`, and the original object. Resolution detail: Engine: Permissions & Voters — Sharing-level resolution.

In-space visibility

Some objects — such as posts and timeline items — also support a **visibility** level within a space:

Visibility	Meaning
Private	Seen only by its owner and the users it is explicitly assigned to.
Organization	Seen by all organization users within the space.
Public	Also surfaced to unauthenticated visitors (subject to anonymous access).

Audience options

A shareable object's sharing panel configures its audience:

- `anonymous` (`AnonymousOption`) — `enabled`: whether unauthenticated visitors can open the object.
- `authenticated` (`AuthenticatedOption`):
  - `enabled` — signed-in users can open it;
  - `isRestricted` — when true, only the users/groups in `entitledEntries` can open it (a per-object allow-list);
  - `entitledEntries` — the allow-list, required only when `isRestricted` is true.

The `Permission` model is a container of `users[]` and `groups[]` plus a legacy `accessType` (`AccessTypeEnum` — `ReadOnly` / `Modify`, deprecated in favour of `AccessRightLevelEnum`). It backs `UpdateResourcePermissionsDto` (permissions + anonymous + authenticated options) for view/page/report sharing, while `UpdateServicePermissionsDto` carries a service's `audience` and `observers` (`PATCH /v1/entities/{id}/meta/permissions`, `Request`-type entities only). Adding a user or group to an object from the **Group Permissions** or **User Permissions** screen writes a View grant on the object and switches the object to restricted mode; for a service it adds the principal to the service's audience allow-list or its observers instead.

Audience validation rules

`PermissionsService.validatePermissions` enforces these invariants when an audience is saved:

Rule	Exception
`anonymous.enabled` **and** `authenticated.enabled` **and** `authenticated.isRestricted` all true	`AnonymousAndRestrictedEnabledException` — an object cannot be open to anonymous visitors while restricting signed-in users.

Rule	Exception
Neither anonymous nor authenticated enabled	`AnonymousOrRestrictedNotEanabledException` — the object must be reachable by someone.
`entitledEntries` provided but `authenticated.enabled` is false	`AuthenticatedMustEnabledExcecption`.
`entitledEntries` provided but `isRestricted` is false	`RestrictedMustEnabledExcecption`.
`isRestricted` true but `entitledEntries` empty	`RestrictedPermissionEntitledEntriesEmptyExcecption`.

Related helpers: `hasEmptyPermissions` (a `Permission[]` is empty when absent, when it has no entries, or when every entry has no users **and** no groups); `anonymusResetEntitledEntries` (enabling anonymous access clears any authenticated allow-list); `getDuplicatedEntitledEntries` (detects duplicate references); `checkUserAndGroupExistence` / `findEntitledEntry` (every referenced user/group must exist and be `Active`, else `NotExistEntitledException`).

Effective object access

`GET /v1/access-object/can-access/{id}` (authorization skipped, so it works for anonymous callers) returns `{result, view}` and is the runtime check for opening a view, page or report:

1. If the object is **not published** → denied.
2. If `anonymous.enabled` and the caller is anonymous → allowed.
3. If `authenticated.enabled`:
    - not restricted and the caller is authenticated → allowed;
    - restricted → allowed only if the caller holds a matching per-object grant.
4. Otherwise → denied.

Anonymous access and sharing links

Two mechanisms expose data to people who are not signed in:

1. **Anonymous permissions** — grant the *anonymous* audience `ViewAccess` on an entity or view (`GET /v1/permissions/anonymous` lists them).
2. **Sharing links** — a share link resolves to a view of a specific object; it can be **anonymous** (no sign-in) or **authenticated** (requires a login).

Service requests

Who can see a service's requests is decided by the service, not by grants:

Who	Rule (`RequestService`)
Can submit (sees the service in the catalogue)	`canCreateRequest`: the service is activated and has an audience; anonymous-only services refuse signed-in users; a restricted authenticated audience requires the user or one of their groups in `entitledEntries`.
The requester	`canAccessMyRequest`: the creator of the request (an anonymous creator only while no step has an owner).
Step owners	`canAccessInbox`: the current step's owner, a member of the owning group or list while the step is unclaimed or set to all users, anyone who executed or owned an earlier step — never the creator.
Watchers	`canAccessObservers`: the user, or one of their groups, is in the service's `meta.observers`. Observers are copied onto each request row, and a change of observers updates existing requests.

Under a delegation, each check is replaced by the delegation's scope for the matching service action (`submit`, `request`, `inbox`, `observe`). Outside a view, the entries of a request entity are readable only with Edit-tier access (see

Access levels).

Sensitive-field grants (data model)

A grant can be typed as a **sensitive-field** read grant via `permission_type = SENSITIVE_FIELD` and a `scope` naming the field path (`module/entity/field`, or a wildcard `module/entity/*` / `module/*/*`). `PermissionTypeEnum` distinguishes these records from `NORMAL` grants; legacy records with no `permission_type` are treated as `NORMAL`. They are maintained per scope by `syncSensitiveFieldAccess`, `getSensitiveFieldGrants` and `removeSensitiveFieldAccess(scopePrefix)`, which reconcile `SENSITIVE_FIELD` records and delete stale ones. **Encrypted** fields `Since 1.10` are authorized by the same `SENSITIVE_FIELD` grant on the same scope; there is no second grant type.

Resolution and the step-up unmask gate are in Engine: Permissions & Voters — Sensitive-field grant resolution; the administrator procedure is Sensitive Permissions [Administration].

Grant expiry and auto-archival `Since 1.8`

`AccessRightValue.expiry` stores a validity window `[startIso, endIso]`; it currently applies to sensitive-field grants.

- A `null` expiry, or a window with no end, never expires; a `null`/missing element leaves that bound open.
- Only currently valid grants count when resolving effective access (`buildExpiryWindowExpr`, `buildActiveExpiryWindowWhere`). A scalar legacy value fails closed.
- An automatic check **archives** any grant whose expiry has passed. When a grant is archived — automatically or by an admin — access is revoked right away: the affected object recomputes whether it is restricted, and the archived grant stops appearing among the principal's active grants.

Roles data model

A role you create is a role data entry (`RoleDataEntry` / `RoleDataEntryValue`) and a group of type `Role` (`GroupTypeEnum.Role`); it can be an entitled entity in its own right (`EntitledEntityEnum.Role = 'ROLE'`). A user assigned to the role inherits its grants, and when the system reports which group a permission came through, a role group is labelled `Role` in `acquiredThroughGroupType`.

Field	Type	Values
`name`	localized string	Display name.
`description`	localized string	Optional.
`linked_with_org_unit`	category (`RoleLinkWIthOrgUnitTypes`)	`NOT_LINKED_WITH_ORG_UNITS`, `LINKED_WITH_ONE_ORG_UNIT`, `LINKED_WITH_MANY_ORG_UNITS`.
`type`	category (`RoleTypes`)	`JOB_POSITION` or `MERE_ROLE`. A role cannot be a job position if it is not linked to any org unit (field validation).
`assignment_type`	category (`RoleAssignmentTypes`)	`UNIQUE` (single holder) or `MULTIPLE` (many holders).

Users are bound to roles through **Assignment** entries (role, assign level Main/Additional, assign type Original/Delegation, user, org unit, is-manager). Each assignment is validated before it is saved (level, uniqueness, role, organizational-unit and organization-account rules), and changing assignments recomputes the org unit's manager when needed.

The system access tiers (`Roles` enum: `SystemAdmin`, `SystemEditor`, `SystemDeveloper`, `SystemViewer`, `Restricted`) are **not** records; they are computed per request from the user's system-scope level. The predicates and the order in which the top-level authorizer applies them are in Engine: Permissions & Voters — Decision flow.

Manager permissions (current limitations)

The **Admin Permissions** screen supports a manager-permissions concept: certain users may grant permissions to other users and groups within limits. Two current behaviours:

- The screen lists only principals that already have a permission record at that scope; a brand-new user or group is reached through the **Add** picker (`only_none_permission`).

Both are being addressed in an ongoing redesign of this model.

Object locking

To keep two users from overwriting each other's edits, the platform supports cooperative **object locks** (pessimistic concurrency control). A record being edited can be locked; other users see the locked state, and a lock **history** is kept. Locks coordinate the editing experience rather than hard-locking the database, so a client that ignores the lock protocol can still write.

Endpoint	Operation
`POST /v1/locks`	Acquire a lock on an object.
`GET /v1/locks`	Get the current lock(s).
`PUT /v1/locks`	Extend (renew) an existing lock.
`GET /v1/locks/{objectId}/history`	The lock history for an object.

Whether a lock can be taken is gated by `getLockedStatusByUser` (is the object locked, and by whom) and a `can-lock` check. In the Administration app, locking is driven by the client `locks` state (`LockEntity`, `UnlockObject`, `ExtendLockObject`, and a discard action), and an entity **lock-history** tab shows the history endpoint. Virtual aggregator entities apply their own lock state through `AggregatorDataEntryService.applyVirtualAggregatorLockState`. Generated reference: Locks — data-service.

Related engine topics

These are documented for platform engineers on Engine: Permissions & Voters:

- How grants from users, groups, roles, org units and query groups merge to one grant per resource — Grants are merged without hiding valid ones.
- Permission caching across services and the generation counters that invalidate it — Caching and Generation (epoch) invalidation.
- The Allow-All delegation and administration rights — Delegation: an Allow-All delegation carries no admin rights.
- The OpenFGA shadow check, its switches and its guarantees — OpenFGA authorization in shadow mode.
- Data repairs shipped with 1.10 — Data repairs.

Endpoint reference

Generated from the committed OpenAPI snapshots — one page per controller, grouped by service. Regenerate with `npm run api-reference`; never edit a generated page by hand.

Generated (from OpenAPI snapshots)

Service	Controller	operations
`comments-service`	Comment	7
`comments-service`	Reaction	4
`data-service`	Access	18
`data-service`	BackupAdmin	10
`data-service`	CounterWidget	4
`data-service`	Counters	1
`data-service`	Dashboard	5
`data-service`	DataCollections	21
`data-service`	DataEntities	20
`data-service`	DataEntityAggregate	6
`data-service`	DataEntityBlocks	6
`data-service`	DataEntityCategories	7
`data-service`	DataEntityFields	7
`data-service`	DataEntries	53
`data-service`	DataEntryTemplateExport	2
`data-service`	DataModule	7
`data-service`	DataQuery	10
`data-service`	DataRelation	20
`data-service`	DataViews	38
`data-service`	Delegations	2
`data-service`	EntityEntriesExport	1
`data-service`	Expression	2
`data-service`	GenericView	10
`data-service`	Groups	1
`data-service`	Icons	1
`data-service`	ImportExport	4
`data-service`	Job	1
`data-service`	Locks	5
`data-service`	OrgConfig	1
`data-service`	QuickFilterGroup	6

Service	Controller	operations
`data-service`	ReportEntry	12
`data-service`	Request	1
`data-service`	SensitiveFieldGrants	8
`data-service`	SideBar	1
`data-service`	Spaces	10
`data-service`	StatisticQuery	11
`data-service`	Stores	4
`data-service`	SystemButton	10
`data-service`	SystemLink	2
`data-service`	SystemObject	1
`data-service`	SystemSettings	1
`data-service`	TagGroup	6
`data-service`	Tags	2
`data-service`	Templates	1
`data-service`	UiSettings	13
`data-service`	User	11
`data-service`	UserPreferences	1
`data-service`	Version	1
`data-service`	ViewApiDocs	1
`data-service`	ViewEntriesExport	1
`data-service`	Workflow	1
`functions-service`	Functions	2
`notification-api`	Notification	3
`notification-api`	NotificationTemplate	1
`notification-api`	PushNotificationsTokens	4
`reaction-service`	Announcement	7
`reaction-service`	DataReaction	12
`reaction-service`	ReactionLog	2
`reaction-service`	SystemReaction	5
`reaction-service`	WhatsAppTemplates	1
`timeline-service`	Timeline	7
`workflow-service`	Workflow	10

Hand-maintained (no machine spec)

- `users-service`
- `storage-service`
- `scheduler-service`

These services are plain Express applications and emit no OpenAPI document. Their surface is documented by hand in the narrative guides — authentication in Choosing a credential and files in Files.

Extracted from the backend

- `shortener-service` → `osos-shortener-fn`
- `altcha-service` → `osos-altcha-fn`

These are **not** backend services any more (Since 1.8). Each is now a standalone, shared per-cluster Knative function consumed as a published image, with its own source outside this monorepo — there is no backend code left to spec or hand-maintain. The public route each one fronts stays frozen at the gateway (e.g. `/api/human/challenge`); only the upstream implementation moved.

Comment — comments-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `comments-service.json` · comments 0.0.1 · 7 operations

`GET /v1/{entity}/{relatedToId}`

`getCommentsRelatedTo`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`relatedToId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	DataEntry model instance

`POST /v1/{entity}/{relatedToId}`

`createComment`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`relatedToId`	path	yes	`string`

Request body: `NewCommentDto`

Responses

Status	Description
`200`	Comment DataEntryValue model instance

`PATCH /v1/{entity}/{relatedToId}/{commentId}`

`updateComment`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Name	In	Required	Type
`relatedToId`	path	yes	`string`
`commentId`	path	yes	`string`

Request body: `UpdateCommentDto`

Responses

Status	Description
`200`	Comment DataEntryValue model instance

`GET /v1/{entity}/{relatedToId}/attachments`

`getCommentsAttachmentsRelatedTo`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`relatedToId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	DataEntry model instance

`GET /v1/{entity}/{relatedToId}/count`

`getCommentsRelatedToCount`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`relatedToId`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	Comment Data Entry model count

`GET /v1/comments/{entityId}/{entryId}/mentions-list`

`getMentionsList`

Parameters

Name	In	Required	Type
`entryId`	path	yes	`string`
`entityId`	path	yes	`string`
`keyword`	query	yes	`string`

Responses

Status	Description
`200`	Comment DataEntry model instance

`GET /v1/comments/audience-list`

`getAudienceList`

Parameters

Name	In	Required	Type
`keyword`	query	yes	`string`

Responses

Status	Description
`200`	Comment DataEntry model instance

Reaction — comments-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `comments-service.json` · comments 0.0.1 · 4 operations

`DELETE /v1/reaction/{entity}/{relatedToId}`

`deleteReaction`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`relatedToId`	path	yes	`string`

Responses

Status	Description
`200`	Delete user reaction on specific entry

`GET /v1/reaction/{entity}/{relatedToId}`

`getReactions`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`relatedToId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	ReactionDataEntryValueDto model instance

`PUT /v1/reaction/{entity}/{relatedToId}`

`createReaction`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`relatedToId`	path	yes	`string`

Request body: `CreateReactionDto`

Responses

Status	Description
`200`	Reaction added

`GET /v1/reaction/{entity}/{relatedToId}/count`

`getReactionsCount`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`relatedToId`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	CountSchema model instance

Access — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 18 operations

`GET /v1/access-object/{id}`

`findAccessObjectById`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Return value of AccessController.findAccessObjectById

`GET /v1/access-object/can-access/{id}`

`canAccessObject`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Return value of AccessController.canAccessObject

`GET /v1/permissions/{entitlementType}/{entitledEntityType}`

`getPermissions` — List users/groups with system, module, entity and view permissions

Parameters

Name	In	Required	Type
`entitlementType`	path	yes	`string`
`entitledEntityType`	path	yes	`string`
`entitlementId`	query	no	`string`
`filter`	query	no	—
`include_virtual`	query	no	`boolean`

Name	In	Required	Type
`only_none_permission`	query	no	`boolean`

Responses

Status	Description
`200`	Array of UserDto model instances

`GET /v1/permissions/{entitlementType}/{entitledEntityType}/count`

`getPermissionsCount` — Get permission count

Parameters

Name	In	Required	Type
`entitlementType`	path	yes	`string`
`entitledEntityType`	path	yes	`string`
`entitlementId`	query	no	`string`
`where`	query	no	—
`include_virtual`	query	no	`boolean`
`only_none_permission`	query	no	`boolean`

Responses

Status	Description
`200`	DataEntry model count.

`PUT /v1/permissions/{entitlementType}/{entitlementId}/{entitledEntityType}/{entitledEntityId}`

`updateAccessPermission` — Update/revoke a access permissions that belongs to the user/group

Parameters

Name	In	Required	Type
`entitlementType`	path	yes	`string`
`entitlementId`	path	yes	`string`
`entitledEntityType`	path	yes	`string`
`entitledEntityId`	path	yes	`string`

Request body: `UpsertPermission`

Responses

Status	Description
`200`	DataEntry PUT success

`PUT /v1/permissions/{systemObjectId}/{entitledEntityType}/{entitledEntityId}`

`updateAccessPermissionBySystemObjectId` — Update/revoke a access permissions that belongs to the user/group

Parameters

Name	In	Required	Type
`systemObjectId`	path	yes	`string`
`entitledEntityType`	path	yes	`string`
`entitledEntityId`	path	yes	`string`

Request body: `UpsertPermission`

Responses

Status	Description
`200`	DataEntry PUT success

`GET /v1/permissions/anonymous`

`getAnonymousUserPermissionsDto` — Get all permissions for the user

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of UserPermissionsDto model instances

`GET /v1/permissions/current_user`

`getUserPermissions` — Get all permissions for the current user

Responses

Status	Description
`200`	Array of UserDto model instances

`GET /v1/permissions/group/{groupId}`

`getPermissionsDtoByGroupId` — Get all permissions for the user

Parameters

Name	In	Required	Type
`groupId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of UserPermissionsDto model instances

`GET /v1/permissions/group/{groupId}/admin`

`getAdminPermissionsDtoByGroupId` — Get all admin permissions for the user

Parameters

Name	In	Required	Type
`groupId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of UserPermissionsDto model instances

`GET /v1/permissions/list/{entitledEntityType}`

`getSystemPermissions` — List all users/groups in the system with permissions

Parameters

Name	In	Required	Type
`entitledEntityType`	path	yes	`string`
`moduleId`	query	no	`string`
`filter`	query	no	—
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	Array of UserDto model instances

`GET /v1/permissions/list/{entitledEntityType}/count`

`systemCount` — Get permissions count

Parameters

Name	In	Required	Type
`entitledEntityType`	path	yes	`string`
`moduleId`	query	no	`string`
`where`	query	no	—
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	DataEntry model count.

`GET /v1/permissions/system-object/{systemObjectId}`

`getObjectPermissionsDtoBySystemObjectId` — Get all permissions for the system object

Parameters

Name	In	Required	Type
`systemObjectId`	path	yes	`string`

Responses

Status	Description
`200`	Array of ObjectPermissionsDto model instances

`GET /v1/permissions/user/{userId}`

`getPermissionsDtoByUserId` — Get all permissions for the user

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of UserPermissionsDto model instances

`GET /v1/permissions/user/{userId}/admin`

`getAdminPermissionsDtoByUserId` — Get all admin permissions for the user

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of UserPermissionsDto model instances

`GET /v1/permissions/user/{userId}/admin/count`

`getAdminPermissionsDtoCountByUserId` — Get count of admin permissions for the user

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	DataEntry model count.

`GET /v1/permissions/user/{userId}/count`

`getPermissionsDtoCountByUserId` — Get count of permissions for the user

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	DataEntry model count.

`GET /v2/access-object/{id}`

`findAccessObjectByIdV2`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Return value of AccessController.findAccessObjectByIdV2

BackupAdmin — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 10 operations

`GET /v1/admin/backups` `Since 1.10`

`list`

Responses

Status	Description
`200`	The tenant's backups and today's quota use

`POST /v1/admin/backups` `Since 1.10`

`start`

Responses

Status	Description
`202`	The backup, running

`GET /v1/admin/backups/{backupId}` `Since 1.10`

`read`

Parameters

Name	In	Required	Type
`backupId`	path	yes	`string`

Responses

Status	Description
`200`	One backup

`GET /v1/admin/exports` `Since 1.10`

`listExports`

Responses

Status	Description
`200`	The tenant's exports

`POST /v1/admin/exports` `Since 1.10`

`startExport`

Responses

Status	Description
`202`	The export, pending

`GET /v1/admin/exports/{exportId}` `Since 1.10`

`readExport`

Parameters

Name	In	Required	Type
`exportId`	path	yes	`string`

Responses

Status	Description
`200`	One export

`POST /v1/admin/exports/{exportId}/link` `Since 1.10`

`linkExport`

Parameters

Name	In	Required	Type
`exportId`	path	yes	`string`

Responses

Status	Description
`200`	A single-use download link

`POST /v1/admin/restores` `Since 1.10`

`startRestore`

Request body: `object`

Responses

Status	Description
`202`	The restore, running

`GET /v1/admin/restores/{restoreId}` `Since 1.10`

`readRestore`

Parameters

Name	In	Required	Type
`restoreId`	path	yes	`string`

Responses

Status	Description
`200`	The restore's status

`POST /v1/admin/restores/preview` `Since 1.10`

`previewRestore`

Request body: `object`

Responses

Status	Description
`200`	What the restore would do, and its confirmation token

CounterWidget — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 4 operations

`GET /v1/counter-widgets`

`getCounterWidgets` — Get counter widgets DTO

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of CounterWidgetDataEntryDto model instances

`GET /v1/counter-widgets/{id}/count`

`getCounterWidgetsCountEndUser`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	CounterWidgetDataEntry model count.

`GET /v1/counter-widgets/{id}/entries`

`getCounterWidgetsEntriesEndUser`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of CounterWidgetDataEntryEndUserDto model instances

`GET /v1/counter-widgets/end-user`

`getCounterWidgetsEndUser`

Responses

Status	Description
`200`	Array of CounterWidgetDataEntry model instances

Counters — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/auto-increment-counters`

`find` — List all auto increment counters

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataCollections model instances

Dashboard — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 5 operations

`GET /v1/dashboards`

`find` — Get all dashboards

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of Dashboard model instances

`GET /v1/dashboards/{id}`

`findById` — Get dashboard by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Dashboard model instance

`PUT /v1/dashboards/{id}`

`updateById` — Update dashboard by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `DashboardPartialExcluding_creationInfo-updateInfo-stateHistory-id_`

Responses

Status	Description
`200`	Dashboard PUT success

DataCollections — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 21 operations

`GET /v1/collections`

`findAll` — Get all collections

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataCollections model instances

`GET /v1/collections/{id}`

`findByIdWithoutEntity` — Get all collections

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataCollections model instances

`GET /v1/collections/count`

`countAll` — Get collections count

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	DataCollections model count.

`GET /v1/collections/my`

`findMyCollections` — List all user collections in a specific view or in all views

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataCollections model instances

`GET /v1/collections/shared`

`findSharedCollections` — List all user shared collections in a specific view or in all views

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataCollections model instances

`GET /v1/entities/{entity}/collections`

`find` — Get all entity collections

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataCollections model instances

`POST /v1/entities/{entity}/collections`

`create` — Create new data collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataCollections`

Responses

Status	Description
`200`	DataCollections model instance

`DELETE /v1/entities/{entity}/collections/{id}`

`deleteById` — Delete specific entity collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataCollections DELETE success

`GET /v1/entities/{entity}/collections/{id}`

`findById` — Get specific entity collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	DataCollectionsPermissionsDto model instance

`PATCH /v1/entities/{entity}/collections/{id}`

`updateById` — Update specific entity collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `DataCollectionsPartialExcluding_id-updateInfo-creationInfo-isImplicit-relatedTo-module_`

Responses

Status	Description
`200`	DataCollections PATCH success

`PUT /v1/entities/{entity}/collections/{id}`

`replaceById` — Update specific entity collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `ReplaceDataCollections`

Responses

Status	Description
`200`	DataCollections PUT success

`POST /v1/entities/{entity}/collections/{id}/duplicate`

`duplicate` — Duplicate data collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `DuplicatedDataCollections`

Responses

Status	Description
`200`	DataCollections model instance

`GET /v1/entities/{entity}/collections/{id}/entries`

`findEntries` — Get specific entity collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	Array of DataEntry model instances

`PATCH /v1/entities/{entity}/collections/{id}/entries`

`updateCollectionItems`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: array of `object`

Responses

Status	Description
`200`	Update collection items

`GET /v1/entities/{entity}/collections/{id}/entries/count`

`findEntriesCount` — Get specific entity collection count

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`where`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntry model count.

`PATCH /v1/entities/{entity}/collections/{id}/share`

`share` — Share collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `DataCollectionsShareModel`

Responses

Status	Description
`200`	DataCollections PATCH success

`GET /v1/entities/{entity}/collections/count`

`count` — Count specific entity collections

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	DataCollections model count.

`POST /v2/entities/{entity}/collections`

`createV2` — Create new data collection

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataCollectionsV2`

Responses

Status	Description
`200`	DataCollections model instance

`GET /v2/entities/{entity}/collections/{id}/entries`

`findEntriesV2` — Get specific entity collection with applied tags

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntriesResponseV3 model instance

`PATCH /v2/entities/{entity}/collections/{id}/entries`

`updateCollectionItemsV2`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `PatchDataCollectionItemsDto`

Responses

Status	Description
`200`	Set or unset collection items with select all ability

`GET /v2/entities/{entity}/collections/{id}/entries/count`

`findEntriesCountV2` — Get specific entity collection count

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`where`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntry model count.

DataEntities — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 20 operations

`GET /v1/entities`

`find`

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Get a list of data entities

`POST /v1/entities`

`create` — Create a new data entity. `isImplicit` and `implicitOptions` are ignored for user callers: the entity is stored with `isImplicit: false`.

Request body: `NewDataEntity`

Responses

Status	Description
`200`	DataEntity model instance

`GET /v1/entities/{entity}/breaking-changes/{version}`

`getEntityVersionChanges` — Get breaking changes for an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`version`	path	yes	`number`

Responses

Status	Description
`200`	Array of DataEntityVersionChanges model instances

`DELETE /v1/entities/{id}`

`deleteById` — Delete a data entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataEntity DELETE success

`GET /v1/entities/{id}`

`findById` — Get a single data entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—
`includeMainTitleFields`	query	no	`boolean`
`includeMetaDataFields`	query	no	`boolean`

Responses

Status	Description
`200`	DataEntityDto model instance

`PATCH /v1/entities/{id}`

`updateById` — Update data entity by id. `isImplicit` and `implicitOptions` are ignored for user callers: the stored values are kept.

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `UpdateDataEntity`

Responses

Status	Description
`200`	DataEntity model instance

`PATCH /v1/entities/{id}/breaking-changes`

`previewBreakingChanges` — Preview breaking changes for a data entity update

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `UpdateDataEntity`

Responses

Status	Description
`200`	BreakingChangesResultDto model instance

`GET /v1/entities/{id}/index-paths` `Since 1.10`

`getIndexPaths` — List the paths a custom index on the entity may use

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Pickable index paths of the entity

`POST /v1/entities/{id}/indexes/{indexId}/retry` `Since 1.10`

`retryIndex` — Re-run the index sync for a custom index of an entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`indexId`	path	yes	`string`

Responses

Status	Description
`200`	Index sync requested; the body is the standard envelope {success: true, result: null}

`PATCH /v1/entities/{id}/meta/permissions`

`updateServicePermissionsById` — Update service permissions for a data entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `UpdateServicePermissionsDto`

Responses

Status	Description
`200`	DataEntity model instance

`PATCH /v1/entities/{id}/module`

`moveToModule` — Move a data entity to a different module

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`force`	query	no	`boolean`

Request body: `object`

Responses

Status	Description
`200`	DataEntity model instance

`DELETE /v1/entities/{id}/permanent` `Since 1.10`

`deleteEntityPermanently` — Permanently delete a data entity, its entries and everything it owns

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`204`	The entity and everything it owned were deleted

`GET /v1/entities/all`

`findAll` — List all entities in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataEntity model instances

`GET /v1/entities/all/count`

`countAll` — Count all entities in the system

Parameters

Name	In	Required	Type
`where`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	DataEntity model count.

`GET /v1/entities/count`

`count` — Get count of data entities in the system

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	DataEntity model count.

`POST /v1/entities/move` `Since 1.10`

`moveEntitiesToModule` — Move several data entities to one module, one at a time; each entity is reported as moved or failed

Request body: `MoveEntitiesRequest`

Responses

Status	Description
`200`	MoveEntitiesResult model instance

`POST /v1/entities/virtual`

`createVirtual` — Create a new virtual data entity

Request body: `NewVirtualDataEntity`

Responses

Status	Description
`200`	DataEntity model instance

`PUT /v1/entities/virtual/{id}`

`updateVirtualEntityById` — Update data entity by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `UpdateVirtualDataEntity`

Responses

Status	Description
`200`	DataEntity model instance

`GET /v2/entities`

`findV2`

Parameters

Name	In	Required	Type
`filter`	query	no	—
`offset`	query	no	`number`

Responses

Status	Description
`200`	Get a bounded page of data entities

`GET /v2/entities/{entity}/breaking-changes`

`getEntityVersionChangesV2` — Get breaking changes for multiple entity versions (v2)

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`versions`	query	no	array of `string`

Responses

Status	Description
`200`	Array of DataEntityVersionChanges model instances

DataEntityAggregate — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 6 operations

`GET /v1/entities/{entity}/aggregates` `Since 1.10`

`find` — List aggregates that belong to a specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataEntityAggregate model instances

`POST /v1/entities/{entity}/aggregates` `Since 1.10`

`create` — Create a new aggregate for a specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataEntityAggregate`

Responses

Status	Description
`200`	DataEntityAggregate model instance

`DELETE /v1/entities/{entity}/aggregates/{id}` `Since 1.10`

`deleteById` — Delete an aggregate by id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataEntityAggregate DELETE success

`PATCH /v1/entities/{entity}/aggregates/{id}` `Since 1.10`

`updateById` — Update an aggregate by id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `DataEntityAggregatePartialExcluding_id_`

Responses

Status	Description
`200`	DataEntityAggregate PATCH success

`GET /v1/entity-aggregates` `Since 1.10`

`findAll` — List all entity aggregates in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataEntityAggregate model instances

`GET /v1/entity-aggregates/{id}` `Since 1.10`

`findByIdWithoutEntity` — Get an entity aggregate by id, without an entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataEntityAggregate model instance

DataEntityBlocks — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 6 operations

`GET /v1/entities/{entity}/blocks`

`find` — Get all blocks for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of InternalEntity model instances

`POST /v1/entities/{entity}/blocks`

`create` — Create block for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewInternalEntity`

Responses

Status	Description
`200`	InternalEntity model instance

`DELETE /v1/entities/{entity}/blocks/{id}`

`deleteById` — Delete block for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	InternalEntity DELETE success

`GET /v1/entities/{entity}/blocks/{id}`

`findById` — Get block for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	InternalEntity model instance

`PATCH /v1/entities/{entity}/blocks/{id}`

`updateById` — Update specific block in specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `InternalEntityPartialExcluding_id_`

Responses

Status	Description
`200`	InternalEntity PATCH success

`GET /v1/entities/{entity}/blocks/count`

`count` — Count blocks for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Responses

Status	Description
`200`	InternalEntity model count.

DataEntityCategories — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 7 operations

`GET /v1/entities/{entity}/categories`

`find` — List internal collections that belongs to specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of InternalCollection model instances

`POST /v1/entities/{entity}/categories`

`create` — Create internal collection inside specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewInternalCollection`

Responses

Status	Description
`200`	InternalCollection model instance

`DELETE /v1/entities/{entity}/categories/{id}`

`deleteById` — Delete specific internal collection that belongs to specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	InternalCollection DELETE success

`GET /v1/entities/{entity}/categories/{id}`

`findById` — Get specific internal collection that belongs to specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	InternalCollection model instance

`PATCH /v1/entities/{entity}/categories/{id}`

`updateById` — Update specific internal collection that belongs to specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `InternalCollectionPartialExcluding_id_`

Responses

Status	Description
`200`	InternalCollection PATCH success

`PUT /v1/entities/{entity}/categories/{id}`

`replaceById` — Update specific internal collection that belongs to specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `InternalCollectionExcluding_id_`

Responses

Status	Description
`200`	InternalCollection PUT success

`GET /v1/entities/{entity}/categories/count`

`count` — Count internal collections that belongs to specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Responses

Status	Description
`200`	InternalCollection model count.

DataEntityFields — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 7 operations

`GET /v1/entities/{entity}/fields`

`find` — Get all fields of specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataEntityField model instances

`POST /v1/entities/{entity}/fields`

`create` — Add field to specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataViewFields`

Responses

Status	Description
`200`	DataEntityField model instance

`DELETE /v1/entities/{entity}/fields/{fieldId}`

`deleteById` — Delete single field of specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`fieldId`	path	yes	`string`

Responses

Status	Description
`200`	DataEntityField DELETE success

`GET /v1/entities/{entity}/fields/{fieldId}`

`findById` — Get single field of specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`fieldId`	path	yes	`string`

Responses

Status	Description
`200`	DataEntityField model instance

`PATCH /v1/entities/{entity}/fields/{fieldId}`

`updateById` — Update single field of specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`fieldId`	path	yes	`string`
`updateIdentifier`	query	no	`string`

Request body: `DataEntityFieldPartialExcluding_id_`

Responses

Status	Description
`200`	DataEntityField PATCH success

`GET /v1/entities/{entity}/fields/count`

`count` — Get total fields of specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Responses

Status	Description
`200`	DataEntityField model count.

`GET /v1/entities/fields/system`

`findSystemFields` — Get all system fields

Responses

Status	Description
`200`	Array of DataEntityField model instances

DataEntries — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 53 operations

`GET /v1/currencies`

`findCurrencies` — List all currencies

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of CurrencyDto model instances

`DELETE /v1/entities/{entity}/archive/entries`

`archiveEntries` — Delete a all data entries that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: array of `object`

Responses

Status	Description
`204`	Data entries status updated to archive success

`PATCH /v1/entities/{entity}/bulk-update`

`bulkUpdateEntries` — Bulk update data entries

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `BulkUpdateEntries`

Responses

Status	Description
`200`	BulkUpdateResponseDto model instance

`PATCH /v1/entities/{entity}/bulk-upgrade`

`bulkUpgradeEntries` — Bulk upgrade data entries

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `BulkUpgradeEntries`

Responses

Status	Description
`200`	BulkUpgradeResponseDto model instance

`GET /v1/entities/{entity}/entries`

`find` — Get a list of data entries that belong to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	Array of DataEntry model instances

`POST /v1/entities/{entity}/entries`

`create` — Create a new data entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataEntry`

Responses

Status	Description
`200`	DataEntry model instance

`DELETE /v1/entities/{entity}/entries/{id}`

`archiveEntry` — Delete a single data entry that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataEntry DELETE success

`GET /v1/entities/{entity}/entries/{id}`

`findById` — Get a single data entry that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`excludeResolved`	query	no	`boolean`
`fields`	query	no	`string`

Responses

Status	Description
`200`	DataEntry model instance

`PATCH /v1/entities/{entity}/entries/{id}`

`updateById` — Update a single data entry that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `UpdateDataEntryDto`

Responses

Status	Description
`200`	DataEntry PATCH success

`GET /v1/entities/{entity}/entries/{id}/editable/{fieldId}`

`isFieldEditable` — Check if field is editable or not

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`fieldId`	path	yes	`string`

Responses

Status	Description
`200`	IsEditableFieldDto model instance

`GET /v1/entities/{entity}/entries/{id}/emails`

`getEmails`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataEntryEmailDto model instances

`GET /v1/entities/{entity}/entries/{id}/history`

`findEntryHistoryById` — return entry history

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	Array of DataEntryHistory model instances

`GET /v1/entities/{entity}/entries/{id}/owner-change-notes` `Since 1.10`

`findOwnerChangeNotesById` — return owner-change notes for an entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	Array of OwnerChangeNote model instances

`PUT /v1/entities/{entity}/entries/{id}/position`

`updateEntryPosition` — update data entry _metaOrder

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `UpdateEntryPositionDto`

Responses

Status	Description
`200`	DataEntry model instance

`PATCH /v1/entities/{entity}/entries/{id}/restore`

`restoreEntry` — Restore an archived data entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataEntry PATCH success

`PATCH /v1/entities/{entity}/entries/{id}/upgrade`

`upgradeEntry` — Upgrad., e data entry and reset the workflow that belongs to the entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `DataEntryPartialExcluding_creationInfo-updateInfo-stateHistory-id-isInvalid-invalidReason-searchIndex_`

Responses

Status	Description
`200`	DataEntry PATCH success

`GET /v1/entities/{entity}/entries/count`

`count` — Counts the number of data entries that belong to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`where`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntry model count.

`GET /v1/entities/{entity}/entries/draft`

`findDraftEntries` — Get a list of draft entries that belongs to the current user and entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`

Responses

Status	Description
`200`	DataEntriesResponseV3 model instance

`POST /v1/entities/{entity}/entries/draft`

`upsertDraftEntry` — Create or update a draft entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `CreateDataEntryDto`

Responses

Status	Description
`200`	DataEntry model instance

`DELETE /v1/entities/{entity}/entries/draft/{id}`

`deleteDraftEntry` — Delete a single draft entry that belongs to the current user and entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataEntry DELETE success

`GET /v1/entities/{entity}/entries/draft/{id}`

`findDraftEntryById` — Get a single draft entry that belongs to the current user

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataEntry model instance

`GET /v1/entities/{entity}/entries/draft/count`

`countDraftEntries` — Counts the number of draft entries that belongs to the current user and entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`

Responses

Status	Description
`200`	DataEntry model count.

`GET /v1/entities/{entity}/entries/kanban/{categoryFieldId}`

`findKanban` — Get a kanban view of data entries that belong to an entity and applied query tags

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`categoryFieldId`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`collectionId`	query	no	`string`
`categoryId`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`
`quickFilterParameters`	query	no	—
`columns`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntryKanbanViewDto model instance

`GET /v1/entities/{entity}/entries/kanban/{categoryFieldId}/draft`

`findKanbanDraft` — Get a kanban view of draft data entries that belong to the user

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`categoryFieldId`	path	yes	`string`
`filterCriteria`	query	no	—
`collectionId`	query	no	`string`
`categoryId`	query	no	`string`
`search`	query	no	`string`
`columns`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntryKanbanViewDto model instance

`GET /v1/entities/{entity}/entries/kanban/{fieldId}/columns` `Since 1.10`

`findKanbanColumns` — Distinct kanban reference values across scoped entries, with card counts

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`fieldId`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`collectionId`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	KanbanColumnValuesDto model instance

`GET /v1/entities/{entity}/entries/kanban/{fieldId}/columns/draft` `Since 1.10`

`findKanbanColumnsDraft` — Distinct kanban reference values across the user's drafts

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Name	In	Required	Type
`fieldId`	path	yes	`string`
`filterCriteria`	query	no	—
`collectionId`	query	no	`string`
`search`	query	no	`string`

Responses

Status	Description
`200`	KanbanColumnValuesDto model instance

`GET /v1/entities/{entity}/entries/summary` `Since 1.10`

`findSummary` — Aggregate totals, and group headers with subtotals, over the entries matching the current scope

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`aggregates`	query	no	array of `string`
`groupBy`	query	no	array of `string`
`groupSkip`	query	no	`number`
`groupLimit`	query	no	`number`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`collectionId`	query	no	`string`
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	EntrySummaryResponse model instance

`GET /v1/entities/{entity}/entries/tree`

`findTree` — Get a list of data entries that childs to another entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`startingItem`	query	no	`string`

Name	In	Required	Type
`field`	query	no	`string`

Responses

Status	Description
`200`	Array of DataEntryTreeDto model instances

`POST /v1/entities/{entity}/entries/versions/count`

`getEntriesVersionsCount`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `SelectAllDto`

Responses

Status	Description
`200`	Get count of entries by version for selectAll criteria

`GET /v1/entities/{entity}/fields/{field}/data`

`fetchReferenceFieldData` — Get a list of users as reference fields for reference field

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`field`	path	yes	`string`
`relatedEntryId`	query	no	`string`
`cascadedParentId`	query	no	`string`
`uiContext`	query	no	`number`
`includeId`	query	no	array of `string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of ReferenceProperty model instances

`PATCH /v1/entities/{entity}/restore/entries`

`restoreEntries`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `ArchiveDataEntryDtoModel`

Responses

Status	Description
`200`	resulting string value

`GET /v1/entities/{entity}/sources/{referenceFieldId}/{referenceDataEntry}`

`getMappedFieldsData` — Get mapped fields related to a reference field

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`referenceFieldId`	path	yes	`string`
`referenceDataEntry`	path	yes	`string`
`entryId`	query	no	`string`

Responses

Status	Description
`200`	Array of MappingFieldSourcesDto model instances

`POST /v1/entities/{entityId}/entries/batch`

`batchCreate` — Create a main entry and related entries in a single operation

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`

Request body: `BatchCreateEntriesRequestDto`

Responses

Status	Description
`200`	BatchCreateEntriesResponseDto model instance

`DELETE /v2/entities/{entity}/archive/entries`

`archiveEntriesV2` — Delete a all data entries that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `ArchiveDataEntryDtoModel`

Responses

Status	Description
`204`	Data entries status updated to archive success

`PATCH /v2/entities/{entity}/bulk-update`

`bulkUpdateEntriesV2`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `BulkUpdateEntries`

Responses

Status	Description
`200`	resulting string value

`PATCH /v2/entities/{entity}/bulk-upgrade`

`bulkUpgradeEntriesV2` — Bulk upgrade data entries v2

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `BulkUpgradeEntries`

Responses

Status	Description
`200`	BulkUpgradeResponseDto model instance

`GET /v2/entities/{entity}/entries`

`findV2` — Get a list of data entries that belong to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filterCriteria`	query	no	—

Name	In	Required	Type
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	Array of DataEntry model instances

`POST /v2/entities/{entity}/entries/{entryId}/fields/{field}/decrypt`

`Since 1.10`

`decryptField`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`field`	path	yes	`string`
`rowIndex`	query	no	`number`

Request body: `object`

Responses

Status	Description
`200`	The decrypted field value

`GET /v2/entities/{entity}/entries/{id}`

`findEntityAndEntryById` — Get the entity and entry information.

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`includeNavigationLinks`	query	no	`boolean`
`relationId`	query	no	`string`
`parentId`	query	no	`string`
`filter`	query	no	—
`collectionId`	query	no	`string`
`search`	query	no	`string`
`parameters`	query	no	—

Responses

Status	Description
`200`	DataEntryV2Body model instance

`PATCH /v2/entities/{entity}/entries/{id}`

`updateByIdV2` — Patch a single data entry that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`updateIdentifier`	query	no	`string`

Request body: array of `JsonPatchOperation`

Responses

Status	Description
`200`	DataEntry PATCH success

`GET /v2/entities/{entity}/entries/{id}/history`

`findEntryHistoryByIdV2` — return entry history

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`lastTimeStamp`	query	no	`string`

Responses

Status	Description
`200`	Array of DataEntryHistory model instances

`DELETE /v2/entities/{entity}/entries/{id}/permanent`

`permanentlyDeleteEntry` — Permanently delete a single archived data entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	Data entry permanently deleted

`GET /v2/entities/{entity}/entries/{id}/references`

`getIncomingReferences` — Get incoming references for a data entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	EntryIncomingReferencesDto model instance

`GET /v2/entities/{entity}/entries/count`

`countV2` — Counts the number of data entries that belong to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`whereCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	DataEntry model count.

`GET /v2/entities/{entity}/entries/draft/{id}`

`findEntityAndDraftEntryById` — Get the entity and draft entry information.

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataEntryV2Body model instance

`GET /v2/entities/{entity}/entries/tree`

`findTreeV2` — Get a list of data entries that child to another entry with applied tags filters

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`startingItem`	query	no	`string`
`field`	query	no	`string`
`appliedTagsFilters`	query	no	array of `string`
`appliedQuickFilters`	query	no	array of `string`
`expandAllChildren`	query	no	`boolean`
`filterCriteria`	query	no	—
`search`	query	no	`string`

Responses

Status	Description
`200`	DataEntryTreeDtoV2 model instance

`POST /v2/entities/{entity}/fields/{field}/data`

`fetchReferenceFieldDataV2` — Get a list of users as reference fields for reference field with Included fields

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`field`	path	yes	`string`

Request body: `GetEntriesForFieldRequestDto`

Responses

Status	Description
`200`	Array of ReferenceProperty model instances

`POST /v2/entities/{entity}/fields/{field}/data/count` `Since 1.10`

`countReferenceFieldDataV2` — Count the options a reference field offers for the given filter

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`field`	path	yes	`string`

Request body: `GetEntriesForFieldRequestDto`

Responses

Status	Description
`200`	DataEntry model count.

`DELETE /v2/entities/{entity}/permanent/entries`

`permanentlyDeleteEntries` — Permanently delete multiple archived data entries

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `PermanentlyDeleteDataEntryDtoModel`

Responses

Status	Description
`200`	Data entries permanently deleted

`PATCH /v3/entities/{entity}/bulk-upgrade`

`bulkUpgradeEntriesV3`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `BulkUpgradeEntries`

Responses

Status	Description
`200`	resulting string value

`GET /v3/entities/{entity}/entries`

`findV3` — Get a list of data entries that belong to an entity and applied query tags

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Name	In	Required	Type
`filterCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`
`groupBy`	query	no	array of `string`
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	DataEntriesResponseV3 model instance

`PATCH /v3/entities/{entity}/entries/{id}`

`patchDataEntryV3` — Patch data entry and related entries

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: array of `PatchDataEntryDtoV3`

Responses

Status	Description
`200`	DataEntry PATCH success

`GET /v3/entities/{entity}/entries/tree`

`findTreeV3` — Get a list of data entries that child to another entry with applied tags filters

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`startingItem`	query	no	array of `string`
`field`	query	no	`string`
`appliedTagsFilters`	query	no	array of `string`
`appliedQuickFilters`	query	no	array of `string`
`expandAllChildren`	query	no	`boolean`
`filterCriteria`	query	no	—
`search`	query	no	`string`

Name	In	Required	Type
`relationId`	query	no	`string`
`relationEntry`	query	no	`string`
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	DataEntryTreeDtoV2 model instance

DataEntryTemplateExport — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 2 operations

`GET /v1/entities/{entity}/entries/export/{templateId}`

`exportEntriesWithTemplateV2`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`templateId`	path	yes	`string`
`mode`	query	no	`number`
`itemsToExport`	query	no	array of `string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`sourceCollection`	query	no	`string`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	File will be processed in background mode; the returned value is the messageId of the queued export command, which correlates with the socket event that later delivers the file.

`POST /v1/entities/{entity}/templates/status/entries` `Since 1.10`

`getTemplatesStatusForEntries`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `GetTemplatesStatusForEntriesRequestDto`

Responses

Status	Description
`200`	Array of TemplateStatusForEntriesResponseDto model instances

DataModule — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 7 operations

`GET /v1/data-module`

`find` — Get all modules

Parameters

Name	In	Required	Type
`filter`	query	no	—
`includeCoreModule`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataModule model instances

`POST /v1/data-module`

`create` — Create new module

Request body: `NewModule`

Responses

Status	Description
`200`	DataModule model instance

`DELETE /v1/data-module/{id}`

`deleteById` — Delete module by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataModule DELETE success

`GET /v1/data-module/{id}`

`findById` — Get module by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	DataModule model instance

`PATCH /v1/data-module/{id}`

`updateById` — Update module by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `DataModulePartialExcluding_id-creationInfo-updateInfo-purgeInfo-purgeStatus_`

Responses

Status	Description
`200`	DataModule PATCH success

`DELETE /v1/data-module/{id}/permanent` `Since 1.10`

`deleteModulePermanently` — Permanently delete a module and everything in it, in the background

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`202`	The delete was accepted; the module goes once its job finishes

`GET /v1/data-module/count`

`count` — Get total modules in the system

Parameters

Name	In	Required	Type
`where`	query	no	—
`includeCoreModule`	query	no	`boolean`

Responses

Status	Description
`200`	DataModule model count.

DataQuery — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 10 operations

`GET /v1/entities/{entity}/queries`

`find` — List all queries for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataQueryWithTagDto model instances

`POST /v1/entities/{entity}/queries`

`create` — Create new query for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataQueries`

Responses

Status	Description
`200`	DataQueryWithTagDto model instance

`DELETE /v1/entities/{entity}/queries/{id}`

`deleteById` — Delete query by id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataQuery DELETE success

`GET /v1/entities/{entity}/queries/{id}`

`findById` — Get query by id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	DataQueryWithTagDto model instance

`PATCH /v1/entities/{entity}/queries/{id}`

`updateById` — Update query by id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `DataQueryWithTagDtoPartialExcluding_id-relatedTo-creationInfo-updateInfo_`

Responses

Status	Description
`200`	DataQueryWithTagDto model instance

`GET /v1/entities/{entity}/queries/count`

`count` — Count all queries for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	DataQuery model count.

`GET /v1/queries`

`findAll` — List all queries in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataQueryWithTagDto model instances

`GET /v1/queries/count`

`countAll` — Count all queries in the system

Parameters

Name	In	Required	Type
`where`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	DataQuery model count.

`GET /v1/queries/my`

`findMyQueries` — List all queries for current user in a view or in all views

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of MyQueryDto model instances

`POST /v1/queries/parser`

`criteriaParser`

Request body: `filterExpressionCriteria`

Responses

Status	Description
`200`	renderedCriteria

DataRelation — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 20 operations

`GET /v1/entities/{entity}/entries/{entryId}/relations/{relationId}`

`findRelatedEntries` — Get all related entries for specific relation that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`relationId`	path	yes	`string`
`filter`	query	no	—
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataEntry model instances

`GET /v1/entities/{entity}/entries/{entryId}/relations/{relationId}/count`

`findRelatedEntriesCount` — Get count of related entries for specific relation that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`relationId`	path	yes	`string`
`where`	query	no	—
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	DataRelation model count.

`GET /v1/entities/{entity}/entries/{entryId}/relations/{relationId}/kanban/{categoryFieldId}`

`findRelatedEntriesKanbanView` — Get all related entries for specific relation that belongs to an entity with applied tag filter as kanban view

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`relationId`	path	yes	`string`
`categoryFieldId`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`categoryId`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`
`columns`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntryKanbanViewDto model instance

`GET /v1/entities/{entity}/entries/{entryId}/relations/{relationId}/kanban/{fieldId}/columns` `Since 1.10`

`findRelatedEntriesKanbanColumns` — Distinct kanban reference values across one relation’s entries

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`relationId`	path	yes	`string`
`fieldId`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`

Name	In	Required	Type
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	KanbanColumnValuesDto model instance

`GET /v1/entities/{entity}/entries/{entryId}/relations/{relationId}/summary`

`Since 1.10`

`findRelatedEntriesSummary` — Aggregate totals, and group headers with subtotals, over the related entries for a relation that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`relationId`	path	yes	`string`
`aggregates`	query	no	array of `string`
`groupBy`	query	no	array of `string`
`groupSkip`	query	no	`number`
`groupLimit`	query	no	`number`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	EntrySummaryResponse model instance

`GET /v1/entities/{entity}/relation`

`find` — Get all relations for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of DataRelationDto model instances

`POST /v1/entities/{entity}/relation`

`create` — Create new relation for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataRelation`

Responses

Status	Description
`200`	DataRelation model instance

`DELETE /v1/entities/{entity}/relation/{id}`

`deleteById` — Delete specific relation that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataRelation DELETE success

`GET /v1/entities/{entity}/relation/{id}`

`findById` — Get specific relation for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	DataRelationDto model instance

`PATCH /v1/entities/{entity}/relation/{id}`

`updateById` — Update specific relation that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `DataRelationPartialExcluding_id_`

Responses

Status	Description
`200`	DataRelation PATCH success

`GET /v1/entities/{entity}/relation/count`

`count` — Count relations for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	DataRelation model count.

`POST /v1/entities/joiner/relations`

`findEntitiesRelations` — Get data entities relations

Request body: `findEntitiesRelations`

Responses

Status	Description
`200`	Array of DataRelationDto model instances

`GET /v1/relations`

`findAll` — List all relations in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—

Name	In	Required	Type
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataRelation model instances

`GET /v1/relations/{id}`

`findOneById` — Get a single relation by ID

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataRelation model instance

`GET /v1/relations/count`

`countAll` — Count all relations in the system

Parameters

Name	In	Required	Type
`where`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	DataRelation model count.

`GET /v2/entities/{entity}/entries/{entryId}/relations/{relationId}`

`findRelatedEntriesV2` — Get all related entries for specific relation that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`relationId`	path	yes	`string`
`filterCriteria`	query	no	—

Name	In	Required	Type
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataEntry model instances

`GET /v2/entities/{entity}/entries/{entryId}/relations/{relationId}/count`

`findRelatedEntriesCountV2` — Get count of related entries for specific relation that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`relationId`	path	yes	`string`
`whereCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	DataRelation model count.

`GET /v2/entities/{entity}/relation`

`findV2` — Get all relations for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Responses

Status	Description
`200`	DataRelationListDto model instance

`GET /v2/entities/{entity}/relation/count`

`countV2` — Count relations for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Responses

Status	Description
`200`	DataRelationCountDto model instance

`GET /v3/entities/{entity}/entries/{entryId}/relations/{relationId}`

`findRelatedEntriesV3` — Get all related entries for specific relation that belongs to an entity with applied tag filter

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`relationId`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`
`groupBy`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntriesResponseV3 model instance

DataViews — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 38 operations

`GET /v1/entities/{entity}/default-view`

`findDefaultView` — Get default view thats belongs to a specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Responses

Status	Description
`200`	DataViews model instance

`GET /v1/entities/{entity}/entries/{entry}/view`

`getAViewForEndUser` — List entries that belongs to a specific public view

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entry`	path	yes	`string`
`notificationId`	query	no	`string`

Responses

Status	Description
`200`	EntryViewModel model instance

`GET /v1/entities/{entity}/views`

`find` — List views for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataViews model instances

`POST /v1/entities/{entity}/views`

`create` — Create views for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataViews`

Responses

Status	Description
`200`	DataViews model instance

`DELETE /v1/entities/{entity}/views/{id}`

`deleteById` — Delete specific view that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataViewsPermissionsDto DELETE success

`GET /v1/entities/{entity}/views/{id}`

`findById` — Get view by entity id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	DataViewsPermissionsDto model instance

`PATCH /v1/entities/{entity}/views/{id}`

`updateById` — Update specific view that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`clearIcon`	query	no	`boolean`

Request body: `DataViewsPermissionsDtoPartialExcluding_id-module-isImplicit-creationInfo-updateInfo_`

Responses

Status	Description
`200`	DataViewsPermissionsDto PATCH success

`GET /v1/entities/{entity}/views/{id}/entries`

`findEntries` — List entries that belongs to a specific view

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—
`filterFirst`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	Array of DataEntry model instances

`GET /v1/entities/{entity}/views/{id}/entries/count`

`entriesCount` — Count of entries that belongs to a specific view

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Name	In	Required	Type
`id`	path	yes	`string`
`where`	query	no	—
`filterFirst`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	DataEntry model count.

`GET /v1/entities/{entity}/views/{id}/entries/export/xlsx`

`exportEntries`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`fileName`	query	no	`string`
`filter`	query	no	—
`filterFirst`	query	no	—
`include_virtual`	query	no	`boolean`
`selectedColumns`	query	no	array of `string`
`collectionId`	query	no	`string`

Responses

Status	Description
`200`	The file content

`GET /v1/entities/{entity}/views/{id}/entries/summary` `Since 1.10`

`entriesSummary` — Aggregate totals, and group headers with subtotals, over the entries of a specific view

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`aggregates`	query	no	array of `string`
`groupBy`	query	no	array of `string`
`groupSkip`	query	no	`number`

Name	In	Required	Type
`groupLimit`	query	no	`number`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`filterFirst`	query	no	—
`include_virtual`	query	no	`boolean`
`collectionId`	query	no	`string`
`appliedQuickFilters`	query	no	array of `string`
`parameters`	query	no	—
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	EntrySummaryResponse model instance

`PATCH /v1/entities/{entity}/views/{id}/permissions`

`updatePermissionsById` — Update specific view permissions that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `UpdateDataViewPermissions`

Responses

Status	Description
`200`	DataViewsPermissionsDto PATCH success

`GET /v1/entities/{entity}/views/{id}/relation/{relationId}`

`findRelationViewById` — Get view by entity id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`relationId`	path	yes	`string`
`relatedEntryId`	query	no	`string`
`mainViewId`	query	no	`string`

Responses

Status	Description
`200`	DataViews model instance

`GET /v1/entities/{entity}/views/count`

`count` — Count views for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`where`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	DataViews model count.

`GET /v1/views`

`findAll` — List all views in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataViews model instances

`DELETE /v1/views/{id}/entries`

`archiveEntriesForView` — Archive multiple data entries that belong to a view

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `ArchiveDataEntryDtoModel`

Responses

Status	Description
`200`	AfterSaveEntryDto model instance

`POST /v1/views/{id}/entries`

`createEntry` — Create a new data entry

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `NewDataEntry`

Responses

Status	Description
`200`	AfterSaveEntryDto model instance

`DELETE /v1/views/{id}/entries/{entryId}`

`archiveEntryForView` — Archive a single data entry using a view

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`entryId`	path	yes	`string`

Responses

Status	Description
`200`	AfterSaveEntryDto model instance

`PATCH /v1/views/{id}/entries/{entryId}`

`patchEntry` — Patch a data entry using a view

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`entryId`	path	yes	`string`

Request body: array of `PatchDataEntryDtoV3`

Responses

Status	Description
`200`	AfterSaveEntryDto model instance

`PUT /v1/views/{id}/entries/{entryId}`

`updateEntry` — Update a data entry using a view

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`entryId`	path	yes	`string`

Request body: `UpdateDataEntry`

Responses

Status	Description
`200`	AfterSaveEntryDto model instance

`GET /v1/views/{id}/entries/{entryId}/after-save/{operation}`

`getAfterSaveDto` — Get the after save dto for an entry

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`entryId`	path	yes	`string`
`operation`	path	yes	`number`

Responses

Status	Description
`200`	AfterSaveEntryDto model instance

`DELETE /v1/views/{id}/entries/{entryId}/draft`

`deleteDraftEntry` — Delete a single draft entry that belongs to the current user and entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`entryId`	path	yes	`string`

Responses

Status	Description
`200`	AfterSaveEntryDto model instance

`POST /v1/views/{id}/entries/draft`

`upsertDraftEntry` — Create or update a draft entry

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `NewDataEntry`

Responses

Status	Description
`200`	AfterSaveEntryDto model instance

`GET /v1/views/{moduleId}`

`findViewsInModule` — List views related to specific module.

Parameters

Name	In	Required	Type
`moduleId`	path	yes	`string`
`filter`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataViews model instances

`GET /v1/views/{moduleId}/count`

`findViewsInModuleCount` — List views related to specific module.

Parameters

Name	In	Required	Type
`moduleId`	path	yes	`string`
`where`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	DataViews model count.

`GET /v1/views/admin`

`getViewsForAdmin` — List views for admin with entity and module info

Parameters

Name	In	Required	Type
`filter`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataViewForAdminDto model instances

`GET /v1/views/admin/count`

`countAllForAdmin` — Count all views in the system

Parameters

Name	In	Required	Type
`where`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	DataViews model count.

`GET /v1/views/count`

`countAll` — Count all views in the system

Parameters

Name	In	Required	Type
`where`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	DataViews model count.

`GET /v1/views/end-user/{id}`

`findViewForEndUser` — Find view by id without entity. This is used for end user to get the view info

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataViews model instance

`GET /v1/views/end-user/{id}/permissions`

`findViewPermissionsForEndUser` — Find view permissions by id without entity.

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	EndUserViewsPermissionsDto model instance

`GET /v1/views/end-user/can-access/{id}`

`canUserAccessView` — returns whether the user can access this view or not

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	the result will be true if the user can access the view false otherwise

`GET /v1/views/end-user/modules`

`findModulesForEndUser` — List the modules holding views the current user can open, each with those views

Responses

Status	Description
`200`	LiteModuleWithViews model instances

`GET /v2/entities/{entity}/views/{id}/entries`

`findEntriesV2` — List entries that belongs to a specific view

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Name	In	Required	Type
`filterCriteria`	query	no	—
`search`	query	no	`string`
`filterFirst`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`

Responses

Status	Description
`200`	Array of DataEntry model instances

`GET /v2/entities/{entity}/views/{id}/entries/count`

`entriesCountV2` — Count of entries that belongs to a specific view

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`whereCriteria`	query	no	—
`search`	query	no	`string`
`filterFirst`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`parameters`	query	no	—
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	DataEntry model count.

`GET /v2/views/{id}`

`findByViewId` — Get view by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	DataViewsPermissionsDto model instance

`GET /v2/views/end-user/{id}`

`findViewForEndUserWithRedirect` — Find view with redirection rules without entity. This is used for end user to get the view info

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	EndUserViewDto model instance

`GET /v2/views/module/{moduleId}`

`findViewsInModuleByModuleId` — List views related to specific module.

Parameters

Name	In	Required	Type
`moduleId`	path	yes	`string`
`filter`	query	no	—
`includeImplicit`	query	no	`boolean`

Responses

Status	Description
`200`	Array of DataViews model instances

`GET /v3/entities/{entity}/views/{id}/entries`

`findEntriesV3` — List entries that belongs to a specific view with applied tags

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`filterFirst`	query	no	—
`include_virtual`	query	no	`boolean`

Name	In	Required	Type
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`
`parameters`	query	no	—
`groupBy`	query	no	array of `string`
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	DataEntriesResponseV3 model instance

Delegations — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 2 operations

`GET /v1/delegations/{id}`

`getDelegation` — Returns a delegation by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	MyDelegationsDto model instance

`GET /v1/delegations/my`

`getActiveDelegations` — Returns my current active delegations

Responses

Status	Description
`200`	Array of MyDelegationsDto model instances

EntityEntriesExport — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v3/entities/{entity}/entries/export` `Since 1.10`

`exportEntries`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`format`	query	no	`string`
`fields`	query	no	array of `string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`
`groupBy`	query	no	array of `string`
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	The file content

Expression — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 2 operations

`POST /v1/entities/{entityId}/{entryId}/tokens/resolve`

`resolveTokens`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`entryId`	path	yes	`string`

Request body: `ResolveTokensRequest`

Responses

Status	Description
`200`	Template fields with {{token}} placeholders replaced. Fields whose tokens could not be resolved are null.

`POST /v1/entities/{entityId}/expressions/resolve`

`resolveExpression`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`

Request body: `ResolveExpressionRequestDto`

Responses

Status	Description
`200`	Resolved expression value

GenericView — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 10 operations

`GET /v1/generic-views`

`find` — List all generic views

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of GenericView model instances

`POST /v1/generic-views`

`create` — Create new GenericView

Request body: `NewGenericView`

Responses

Status	Description
`200`	GenericView model instance

`DELETE /v1/generic-views/{id}`

`deleteById` — Delete specific generic view by Id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	GenericViewPermissionsDto DELETE success

`GET /v1/generic-views/{id}`

`findById` — Return single generic view by Id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	GenericViewPermissionsDto model instance

`PATCH /v1/generic-views/{id}`

`updateById` — Update specific generic view by Id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`clearIcon`	query	no	`boolean`

Request body: `GenericViewPermissionsDtoPartialExcluding_id-creationInfo-updateInfo_`

Responses

Status	Description
`200`	GenericViewPermissionsDto PATCH success

`PATCH /v1/generic-views/{id}/permissions`

`updatePermissionsById` — Update specific generic view permissions by Id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `UpdateGenericViewPermissions`

Responses

Status	Description
`200`	GenericViewPermissionsDto PATCH success

`GET /v1/generic-views/count`

`count` — Count of GenericViews

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	GenericView model count.

`GET /v1/generic-views/default`

`getUserDefaultView` — return the default view for the user

Parameters

Name	In	Required	Type
`relatedTo`	query	no	—
`x-domain-id`	header	no	`string`

Responses

Status	Description
`200`	GenericViewPermissionsDto model instance

`GET /v1/generic-views/end-user/{id}`

`findByIdEndUser` — Return single generic view by Id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	GenericViewPermissionsDto model instance

`GET /v2/generic-views/end-user/{id}`

`findByIdEndUserV2` — Return a dto containing either redirection rule or components based on matching rules

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	GenericViewEndUserDto model instance

Groups — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/group/{groupId}/users`

`getUsersInsideGroup`

Parameters

Name	In	Required	Type
`groupId`	path	yes	`string`
`includeResolved`	query	no	`boolean`

Responses

Status	Description
`200`	Array of ReferencePropertyWithResolved model instances

Icons — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/icons`

`getIcons`

Responses

Status	Description
`200`	Icon manifest keyed by icon name
`304`	Not modified — reuse cached body when If-None-Match matches ETag

ImportExport — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 4 operations

`POST /v1/export/entity`

`exportDataEntities`

Request body: `ExportingEntitiesDto`

Responses

Status	Description
`200`	The file content

`POST /v1/export/module`

`exportDataModel`

Request body: `ExportingDataModulesDto`

Responses

Status	Description
`200`	The file content

`POST /v1/import/{fileId}`

`validateFileToImport`

Parameters

Name	In	Required	Type
`fileId`	path	yes	`string`

Request body: `ImportEntitiesDto`

Responses

Status	Description
`200`	ImportResultDto model instance

`POST /v1/import/{fileId}/confirm`

`confirmFileImport`

Parameters

Name	In	Required	Type
`fileId`	path	yes	`string`

Request body: `ImportEntitiesDto`

Responses

Status	Description
`200`	ImportResultDto model instance

Job — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/job/{messageId}`

`getJobStatusByMessageId`

Parameters

Name	In	Required	Type
`messageId`	path	yes	`string`

Responses

Status	Description
`200`	Get job status by message id

Locks — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 5 operations

`POST /v1/locks`

`createLockObject`

Request body: `CreateLockObjectDto`

Responses

Status	Description
`200`	Create lock

`GET /v1/locks/{objectId}`

`getLocksByObjectId`

Parameters

Name	In	Required	Type
`objectId`	path	yes	`string`

Responses

Status	Description
`200`	Get locks by object id

`PUT /v1/locks/{objectId}`

`patchLockObject`

Parameters

Name	In	Required	Type
`objectId`	path	yes	`string`

Request body: `PatchLockObjectDto`

Responses

Status	Description
`200`	Patch lock object

`GET /v1/locks/{objectId}/history`

`getLocksObjectHistory`

Parameters

Name	In	Required	Type
`objectId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Get locks by object history

`GET /v1/locks/{objectId}/history/count`

`getLocksObjectHistoryCount`

Parameters

Name	In	Required	Type
`objectId`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	Get locks by object history count

OrgConfig — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/configs`

`getOrgConfigs` — Get org configs

Responses

Status	Description
`200`	Get new token

QuickFilterGroup — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 6 operations

`GET /v1/entities/{entity}/quick-filter-groups` `Since 1.10`

`find` — List the quick filter groups of an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Responses

Status	Description
`200`	Array of QuickFilterGroupDto model instances

`POST /v1/entities/{entity}/quick-filter-groups` `Since 1.10`

`create` — Create a quick filter group on an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewQuickFilterGroup`

Responses

Status	Description
`200`	QuickFilterGroupDto model instance

`DELETE /v1/entities/{entity}/quick-filter-groups/{id}` `Since 1.10`

`deleteById` — Delete a quick filter group; its queries become ungrouped

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	QuickFilterGroup DELETE success

`PATCH /v1/entities/{entity}/quick-filter-groups/{id}` `Since 1.10`

`updateById` — Rename a quick filter group or set its quick queries

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `UpdateQuickFilterGroup`

Responses

Status	Description
`200`	QuickFilterGroupDto model instance

`GET /v1/quick-filter-groups` `Since 1.10`

`findAll` — List all quick filter groups in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of QuickFilterGroup model instances

`GET /v1/quick-filter-groups/{id}` `Since 1.10`

`findByIdWithoutEntity` — Get a quick filter group by id, without an entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	QuickFilterGroup model instance

ReportEntry — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 12 operations

`GET /v1/reports`

`find` — Get all reports defined in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of ReportEntry model instances

`POST /v1/reports`

`create` — Create a new report

Request body: `NewReportEntry`

Responses

Status	Description
`200`	ReportEntryDto model instance

`DELETE /v1/reports/{id}`

`deleteById` — Delete report with id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	ReportEntry DELETE success

`GET /v1/reports/{id}`

`findById` — Get report by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	ReportEntryDto model instance

`PATCH /v1/reports/{id}`

`updateById` — Update report entry by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `ReportEntryUpdateById`

Responses

Status	Description
`200`	ReportEntry PATCH success

`GET /v1/reports/{id}/entries`

`executeReport` — Execute Report and returns the result as DataEntry[]

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	Array of ReportEntry model instances

`GET /v1/reports/{id}/entries/count`

`executeReportCount` — Get records count for the report execution

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`where`	query	no	—
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	ReportEntryDto model count.

`PATCH /v1/reports/{id}/permissions`

`updatePermissionsById` — Update report entry by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `UpdateReportEntryPermissions`

Responses

Status	Description
`200`	ReportEntry PATCH success

`GET /v1/reports/count`

`count` — Get report entry count

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	ReportEntryDto model count.

`GET /v2/reports/{id}/entries`

`executeReportV2` — Execute Report and returns the result as DataEntry[]

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Name	In	Required	Type
`filterCriteria`	query	no	—
`include_virtual`	query	no	`boolean`
`search`	query	no	`string`

Responses

Status	Description
`200`	Array of ReportEntry model instances

`GET /v2/reports/{id}/entries/count`

`executeReportCountV2` — Get records count for the report execution

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`whereCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`

Responses

Status	Description
`200`	ReportEntryDto model count.

`GET /v2/reports/{id}/export/{format}`

`exportReportV2`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`format`	path	yes	`string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`include_virtual`	query	no	`boolean`
`selectedColumns`	query	no	array of `string`

Responses

Status	Description
`200`	The file content

Request — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/requests/{entity}/{id}/{mode}`

`canAccessRequest` — Get the implicit view thats belongs to the request if the user can access it.

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`mode`	path	yes	`string`

Responses

Status	Description
`200`	DataViews model instance

SensitiveFieldGrants — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 8 operations

`GET /v1/sensitive-field-grants`

`listSensitiveFieldGrants` — List all active sensitive-field access grants

Responses

Status	Description
`200`	Array of SensitiveFieldGrantDto instances

`POST /v1/sensitive-field-grants`

`createSensitiveFieldGrants` — Create sensitive-field grants for a scope

Request body: `CreateSensitiveFieldGrantsDto`

Responses

Status	Description
`200`	CreateSensitiveFieldGrantsResultDto instance

`DELETE /v1/sensitive-field-grants/{id}`

`deleteSensitiveFieldGrant` — Delete a sensitive-field grant

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`204`	Sensitive-field grant deleted successfully

`PATCH /v1/sensitive-field-grants/{id}`

`updateSensitiveFieldGrant` — Update a sensitive-field grant

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `UpdateSensitiveFieldGrantDto`

Responses

Status	Description
`200`	SensitiveFieldGrantDto instance

`PATCH /v1/sensitive-field-grants/{id}/archive`

`archiveSensitiveFieldGrant` — Archive a sensitive-field grant

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`204`	Sensitive-field grant archived successfully

`GET /v1/sensitive-field-grants/by-group/{groupId}`

`listGroupSensitiveFieldGrants` — List a group's sensitive-field grants

Parameters

Name	In	Required	Type
`groupId`	path	yes	`string`

Responses

Status	Description
`200`	Array of SensitiveFieldGrantDto instances

`GET /v1/sensitive-field-grants/by-user/{userId}`

`listUserSensitiveFieldGrants` — List a user's effective sensitive-field grants (direct + indirect)

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`

Responses

Status	Description
`200`	Array of UserSensitiveFieldGrantDto instances

`GET /v1/sensitive-field-grants/inherited-holders`

`getInheritedSensitiveFieldHolders` — List a sensitive field's inherited (entity/module wildcard) holders

Parameters

Name	In	Required	Type
`moduleNn`	query	yes	`string`
`entityNn`	query	yes	`string`
`fieldNn`	query	yes	`string`

Responses

Status	Description
`200`	InheritedSensitiveFieldHoldersDto instance

SideBar — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v2/side-bar`

`getViewsForSidebarV2` — Get all sidebar views (system and module)

Parameters

Name	In	Required	Type
`x-domain-id`	header	no	`string`
`x-space-id`	header	no	`string`
`include`	query	no	`number`

Responses

Status	Description
`200`	SidebarDto model instances

Spaces — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 10 operations

`GET /v1/spaces`

`getAllSpaces` — Get all spaces

Parameters

Name	In	Required	Type
`filterCriteria`	query	no	—
`search`	query	no	`string`

Responses

Status	Description
`200`	Array of SpaceEntry model instances

`POST /v1/spaces`

`create` — Create a new space

Request body: `SpaceEntry`

Responses

Status	Description
`200`	SpaceEntry model instance

`GET /v1/spaces/{id}`

`getSpace` — Get single space by id

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	SpaceEntry model instance

`POST /v1/spaces/{id}/activate`

`activateSpace`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Return value of SpacesController.activateSpace

`POST /v1/spaces/{id}/deactivate`

`deActivateSpace`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Return value of SpacesController.deActivateSpace

`POST /v1/spaces/{id}/end-user-activate`

`spaceEndUserActivate`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Return value of SpacesController.spaceEndUserActivate

`POST /v1/spaces/{id}/end-user-deactivate`

`spaceEndUserDeActivate`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Return value of SpacesController.spaceEndUserDeActivate

`GET /v1/spaces/count`

`getSpacesCount` — Get spaces count

Parameters

Name	In	Required	Type
`search`	query	no	`string`

Responses

Status	Description
`200`	DataEntry model count.

`GET /v1/spaces/my`

`getSpacesForCurrentUser` — Get my spaces

Responses

Status	Description
`200`	Array of SpaceEntry model instances

`GET /v1/spaces/user/{id}`

`getUserSpaces` — Get user spaces

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Array of SpaceEntry model instances

StatisticQuery — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 11 operations

`GET /v1/entities/{entity}/statistic-queries`

`find` — List statistic queries that belongs to a specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of StatisticQuery model instances

`POST /v1/entities/{entity}/statistic-queries`

`create` — Create new statistic query for specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewStatisticQuery`

Responses

Status	Description
`200`	StatisticQuery model instance

`DELETE /v1/entities/{entity}/statistic-queries/{id}`

`deleteById` — Delete statistic query by id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	StatisticQuery DELETE success

`GET /v1/entities/{entity}/statistic-queries/{id}`

`findById` — Get statistic query by id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	StatisticQuery model instance

`PATCH /v1/entities/{entity}/statistic-queries/{id}`

`updateById` — Update statistic query by id

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `StatisticQueryPartialExcluding_id_`

Responses

Status	Description
`200`	StatisticQuery PATCH success

`GET /v1/entities/{entity}/statistic-queries/count`

`count` — Count statistic queries that belongs to a specific entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	StatisticQuery model count.

`GET /v1/statistic-queries`

`findAll` — List all statistic queries in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of StatisticQuery model instances

`GET /v1/statistic-queries/{id}`

`findByIdWithoutEntity` — Get statistic query by id without entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	StatisticQueryDTO model instance

`GET /v1/statistic-queries/by-ids` `Since 1.10`

`findByIdsWithoutEntity` — Get several statistic queries by id without entity

Parameters

Name	In	Required	Type
`ids`	query	no	array of `string`

Responses

Status	Description
`200`	Array of StatisticQueryDTO model instances

`GET /v1/statistic-queries/count`

`countAll` — Count all statistic queries in the system

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	StatisticQuery model count.

`GET /v2/entities/{entity}/statistic-queries/{id}/execute`

`executeStatisticQueryV2`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	execute specific statistic query

Stores — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 4 operations

`GET /v1/entities/{entityId}/stores`

`getAllStores` — Get all stores for an entity including default stores

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`filterCriteria`	query	no	—

Responses

Status	Description
`200`	Array of StoresResponseDto model instances

`POST /v1/entities/{entityId}/stores`

`createStore` — Create a new store for an entity

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`

Request body: `CreateDataStoreRequestDto`

Responses

Status	Description
`200`	StoresResponseDto model instance

`GET /v1/entities/{entityId}/stores/{storeId}`

`getStore` — Get a store for an entity

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`storeId`	path	yes	`string`

Responses

Status	Description
`200`	StoresResponseDto model instance

`PUT /v1/entities/{entityId}/stores/{storeId}`

`updateStore`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`storeId`	path	yes	`string`

Request body: `UpdateDataStoreRequestDtoPartial`

Responses

Status	Description
`200`	update a store for an entity

SystemButton — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 10 operations

`GET /v1/system-buttons`

`findAll`

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of SystemButtonDataEntryDto model instances

`POST /v1/system-buttons/{id}`

`executeSystemButton`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `ExecuteSystemButtonDto`

Responses

Status	Description
`200`	resulting string value

`GET /v1/system-buttons/{id}/options`

`getSystemButtonActionWithOptions`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	Array of DataReactionAction model instances

`GET /v1/system-buttons/{relatedToId}`

`find`

Parameters

Name	In	Required	Type
`relatedToId`	path	yes	`string`

Responses

Status	Description
`200`	Array of SystemButtonDataEntryDto model instances

`GET /v1/system-buttons/{relatedToId}/active`

`findForUser`

Parameters

Name	In	Required	Type
`relatedToId`	path	yes	`string`
`entryId`	query	no	`string`

Responses

Status	Description
`200`	Array of SystemButtonDataEntryDto model instances

`POST /v1/system-buttons/{relatedToId}/status/entries`

`getSystemButtonsStatusForEntries`

Parameters

Name	In	Required	Type
`relatedToId`	path	yes	`string`

Request body: `GetSystemButtonsStatusForEntriesRequestDtoModel`

Responses

Status	Description
`200`	Array of SystemButtonStatusForEntriesResponseDtoModel model instances

`GET /v1/system-buttons/by-id/{id}`

`findById`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	SystemButtonDataEntry model instance

`GET /v1/system-buttons/count`

`countAll`

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	SystemButtonDataEntryDto model count.

`GET /v1/system-buttons/raw`

`findAllRaw`

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of SystemButtonDataEntry model instances

`GET /v2/system-buttons/{id}/options`

`getSystemButtonActionWithOptionsV2`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	SystemButtonOptionsDto model instance

SystemLink — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 2 operations

`GET /v1/system-link/generate`

`generateToken` — Generate random token

Responses

Status	Description
`200`	Get new token

`GET /v1/system-link/resolve/{token}`

`getAnonymousView` — Get anonymous view or generic view

Parameters

Name	In	Required	Type
`token`	path	yes	`string`

Responses

Status	Description
`200`	Get new token

SystemObject — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/system-objects`

`find` — Array of system objects as reference with parent and resolved (like ApplicationComponentFetcher at SystemLevel)

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of ReferencePropertyWithParentAndIncludedFields model instances

SystemSettings — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/system-settings/public` `Since 1.10`

`getPublic`

Responses

Status	Description
`200`	Public system settings

TagGroup — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 6 operations

`GET /v1/entities/{entity}/tag-groups` `Since 1.10`

`find` — List the tag groups of an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Responses

Status	Description
`200`	Array of TagGroupDto model instances

`POST /v1/entities/{entity}/tag-groups` `Since 1.10`

`create` — Create a tag group on an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewTagGroup`

Responses

Status	Description
`200`	TagGroupDto model instance

`DELETE /v1/entities/{entity}/tag-groups/{id}` `Since 1.10`

`deleteById` — Delete a tag group; its tags become ungrouped

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`200`	TagGroup DELETE success

`PATCH /v1/entities/{entity}/tag-groups/{id}` `Since 1.10`

`updateById` — Rename a tag group or set its tags

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `UpdateTagGroup`

Responses

Status	Description
`200`	TagGroupDto model instance

`GET /v1/tag-groups` `Since 1.10`

`findAll` — List all tag groups in the system

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of TagGroup model instances

`GET /v1/tag-groups/{id}` `Since 1.10`

`findByIdWithoutEntity` — Get a tag group by id, without an entity

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	TagGroup model instance

Tags — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 2 operations

`GET /v1/tags/{entityId}/all`

`getTags` — Get all tags under data query

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`

Responses

Status	Description
`200`	Get new token

`GET /v1/tags/{entityId}/groups`

`getTagGroups` — Get tag groups

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`

Responses

Status	Description
`200`	Get new token

Templates — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/templates/{id}/used`

`isTemplateUsed` — returns whether template is used or not

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	the result will be true if the user can access the view false otherwise

UiSettings — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 13 operations

`GET /v1/fonts/files/{fontId}/{format}`

`getFontFile`

Parameters

Name	In	Required	Type
`fontId`	path	yes	`string`
`format`	path	yes	`string`

Responses

Status	Description
`200`	The file content
`204`	The font entry exists but this format was never uploaded. Empty body.

`GET /v1/formats/date`

`getAvailableDateFormats`

Responses

Status	Description
`200`	Get available date formats

`GET /v1/formats/date-time`

`getAvailableDateTimeFormats`

Responses

Status	Description
`200`	Get available date time formats

`GET /v1/formats/time`

`getAvailableTimeFormats`

Responses

Status	Description
`200`	Get available time formats

`GET /v1/language`

`getAvailableLanguages` — return all available languages

Responses

Status	Description
`200`	Array of languages model instances

`GET /v1/language/files/{langCode}/{platform}`

`getLanguageTranslationFile`

Parameters

Name	In	Required	Type
`langCode`	path	yes	`string`
`platform`	path	yes	`string`

Responses

Status	Description
`200`	The file content
`204`	The language exists but has no translation file uploaded for this platform. Empty body — callers should fall back to bundled assets.

`GET /v1/settings/user`

`getUserUiSettings`

Responses

Status	Description
`200`	return the ui settings for the current user

`GET /v1/settings/user/anonymous`

`getAnonymousUserUiSettings`

Responses

Status	Description
`200`	return the ui settings for anonymous user

`GET /v1/themes`

`getAvailableThemes`

Responses

Status	Description
`200`	Get available themes

`GET /v1/themes/{themeId}`

`getThemeDetails`

Parameters

Name	In	Required	Type
`themeId`	path	yes	`string`

Responses

Status	Description
`200`	Get theme details

`GET /v1/themes/anonymous`

`getAnonymousAvailableThemes`

Responses

Status	Description
`200`	Get available themes for anonymous user

`GET /v1/themes/files/{themeId}/css`

`getThemeCssFile`

Parameters

Name	In	Required	Type
`themeId`	path	yes	`string`
`if-none-match`	header	no	`string`

Responses

Status	Description
`200`	The file content
`204`	The theme exists but has no general_css configured. Empty body — never a JSON envelope, which would be invalid CSS if injected as a stylesheet.
`304`	The stylesheet named by If-None-Match is still current. Empty body.

`GET /v1/time-zone`

`getAvailableTimeZones`

Responses

Status	Description
`200`	Get available time zones

User — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 11 operations

`POST /v1/users/{userId}/activate`

`activateUser`

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`

Responses

Status	Description
`200`	Return value of UserController.activateUser

`POST /v1/users/{userId}/deactivate`

`deactivateUser`

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`

Responses

Status	Description
`200`	Return value of UserController.deactivateUser

`POST /v1/users/{userId}/regenerate-app-token`

`regenerateAppToken`

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`

Responses

Status	Description
`200`	Return value of UserController.regenerateAppToken

`POST /v1/users/{userId}/reset-password`

`resetUserPasswordBySystem`

Parameters

Name	In	Required	Type
`userId`	path	yes	`string`

Responses

Status	Description
`200`	Return value of UserController.resetUserPasswordBySystem

`GET /v1/users/direct-manager`

`getUserDirectManager` — Get current user direct manager

Responses

Status	Description
`200`	UserOrgUnitDto model instance

`GET /v1/users/my-groups`

`getMyGroups` — Get user profile info

Responses

Status	Description
`200`	Array of DataEntry model instances

`GET /v1/users/my-orgunit`

`getUserOrgUnitUsers` — Returns the list of users inside the org unit of current person related to the signed in user and the parent org unit and all children org units

Responses

Status	Description
`200`	Array of DataEntry model instances

`GET /v1/users/profile`

`getUserProfile` — Get user profile info

Responses

Status	Description
`200`	UserProfileDto model instance

`GET /v1/users/state`

`getUserState` — return the current user state. If a state doesn't exist for current user, create one with default options and return the newly created state

Responses

Status	Description
`200`	UserStateDataEntryValue model instance

`PUT /v1/users/state`

`updateUserState` — return the current user state. If a state doesn't exist for current user, create one with default options and return the newly created state

Request body: `UserStateDataEntryValueExcluding_user-firstLogin_`

Responses

Status	Description
`200`	UserStateDataEntryValue model instance

`PUT /v1/users/state/reset`

`resetUserState` — Reset the current user state

Responses

Status	Description
`200`	UserStateDataEntryValue model instance

UserPreferences — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`PATCH /v1/user-preferences`

`updateUserPreferences` — Update user preferences for a specific object (deep merge)

Request body: `UpdateUserPreferencesDto`

Responses

Status	Description
`200`	User preferences updated successfully

Version — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/version`

`getVersion` — Get versions

Responses

Status	Description
`200`	Get versions

ViewApiDocs — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v1/views/{viewId}/openapi.json` `Since 1.10`

`getViewOpenApiDocument` — Returns a per-View OpenAPI document covering that View's entity schema and entry CRUD API, scoped via the /v1/v/{viewId}/ wrapped path prefix.

Parameters

Name	In	Required	Type
`viewId`	path	yes	`string`

Responses

Status	Description
`200`	OpenAPI 3.0 document
`403`	The View's displayConfig.swaggerDocument flag is disabled, or the caller fails the end-user View access check (see assertViewAccessibleToEndUser) — system admins bypass both.
`404`	No View exists with that id, or it belongs to an archived entity.

ViewEntriesExport — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`GET /v3/entities/{entity}/views/{id}/entries/export` `Since 1.10`

`exportEntries`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`format`	query	no	`string`
`fields`	query	no	array of `string`
`filterCriteria`	query	no	—
`search`	query	no	`string`
`filterFirst`	query	no	—
`include_virtual`	query	no	`boolean`
`appliedQuickFilters`	query	no	array of `string`
`appliedTagsFilters`	query	no	array of `string`
`parameters`	query	no	—
`groupBy`	query	no	array of `string`
`truncate`	query	no	`boolean`
`quickFilterParameters`	query	no	—

Responses

Status	Description
`200`	The file content

Workflow — data-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `data-service.json` · data-service 1.11.0 · 1 operations

`POST /external-step/return`

`executeExternalStepReturn` — Callback invoked by an external system redirecting a browser back after completing an ExternalStep. Authenticated via the opaque return token itself, not a user session.

Request body: `object`

Responses

Status	Description
`200`	Result of processing the return token

Functions — functions-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `functions-service.json` · functions 0.51.5 · 2 operations

`POST /v1/{functionId}/execute`

`executeFunction`

Parameters

Name	In	Required	Type
`functionId`	path	yes	`string`

Request body: `object`

Responses

Status	Description
`200`	Function executed
`400`	Bad request

`GET /v1/{scope}`

`getFunctions`

Parameters

Name	In	Required	Type
`scope`	path	yes	`string`
`scopeId`	query	no	`string`

Responses

Status	Description
`200`	—

Notification — notification-api

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `notification-api.json` · notification-api 0.0.1 · 3 operations

`GET /v1/my`

`getMyNotifications`

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	DataEntry model instance

`PATCH /v1/my/{id}/mark-as-read`

`markAsRead`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Responses

Status	Description
`200`	NotificationDataEntryValue model instance

`GET /v1/my/count`

`getMyNotificationCount`

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	NotificationDataEntryValue model count

NotificationTemplate — notification-api

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `notification-api.json` · notification-api 0.0.1 · 1 operations

`POST /v1/template/resolve-tokens`

`resolveTokens`

Request body: `object`

Responses

Status	Description
`200`	Token resolved

PushNotificationsTokens — notification-api

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `notification-api.json` · notification-api 0.0.1 · 4 operations

`GET /v1/push-tokens`

`find`

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of PushNotificationToken model instances

`POST /v1/push-tokens`

`create`

Request body: `NewPushNotificationToken`

Responses

Status	Description
`200`	PushNotificationToken model instance

`DELETE /v1/push-tokens/{token}` `Since 1.10`

`deleteOwn`

Parameters

Name	In	Required	Type
`token`	path	yes	`string`

Responses

Status	Description
`204`	No Content

`POST /v1/push-tokens/invalidate` `Since 1.10`

`invalidate`

Request body: `object`

Responses

Status	Description
`200`	Number of deleted PushNotificationToken rows

Announcement — reaction-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `reaction-service.json` · reaction 0.51.5 · 7 operations

`GET /v1/announcements`

`find`

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	List of all Announcement records

`GET /v1/announcements/{entityId}`

`findByEntity`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	List of Announcements for a specific entity

`POST /v1/announcements/{entityId}`

`create`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`

Request body: `AnnouncementExcluding_id-entityId-reactions-creationInfo_`

Responses

Status	Description
`200`	Create announcement for specific entity

`DELETE /v1/announcements/{entityId}/{announcementId}` `Since 1.10`

`deleteById`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`announcementId`	path	yes	`string`

Responses

Status	Description
`204`	Announcement DELETE success

`GET /v1/announcements/{entityId}/{announcementId}`

`findById`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`announcementId`	path	yes	`string`

Responses

Status	Description
`200`	Get a single Announcement by id (scoped to entity)

`PATCH /v1/announcements/{entityId}/{announcementId}`

`update`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`announcementId`	path	yes	`string`

Request body: `AnnouncementPartialExcluding_id-entityId-reactions-creationInfo_`

Responses

Status	Description
`200`	Update announcement for specific entity

`GET /v1/announcements/by-id/{announcementId}`

`findByIdOnly`

Parameters

Name	In	Required	Type
`announcementId`	path	yes	`string`

Responses

Status	Description
`200`	Get a single Announcement by id

DataReaction — reaction-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `reaction-service.json` · reaction 0.51.5 · 12 operations

`GET /v1/reaction/{entity}`

`find`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of Reaction model instances, each with its execution and failure counts

`POST /v1/reaction/{entity}`

`create`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`

Request body: `NewDataReaction`

Responses

Status	Description
`200`	Create a new reaction

`DELETE /v1/reaction/{entity}/{id}`

`deleteById`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Responses

Status	Description
`204`	Reaction DELETE success

`GET /v1/reaction/{entity}/{id}`

`findById`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Reaction model instance

`PATCH /v1/reaction/{entity}/{id}`

`updateById`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`

Request body: `DataReactionPatch`

Responses

Status	Description
`200`	Reaction PATCH success

`GET /v1/reaction/{entity}/{id}/history` `Since 1.10`

`history`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`id`	path	yes	`string`
`outcome`	query	no	`string`
`from`	query	no	`string`

Name	In	Required	Type
`to`	query	no	`string`
`before`	query	no	`string`
`limit`	query	no	`number`

Responses

Status	Description
`200`	A page of the reaction's recorded runs, newest first

`GET /v1/reaction/{entity}/count`

`count`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`where`	query	no	—

Responses

Status	Description
`200`	Reaction model count

`GET /v1/reactions`

`findAll`

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Array of all Reaction model instances

`GET /v1/reactions/{id}`

`findOneById`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Reaction model instance by ID

`GET /v1/reactions/count`

`countAll`

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	Count of all Reaction model instances

`GET /v1/reactions/custom-actions` `Since 1.8`

`findCustomActionCandidates`

Parameters

Name	In	Required	Type
`moduleId`	query	yes	`string`

Responses

Status	Description
`200`	Custom actions callable from the given module via Execute Custom Action

`GET /v1/reactions/module/{moduleId}` `Since 1.8`

`findByModule`

Parameters

Name	In	Required	Type
`moduleId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Array of Reaction model instances defined on entities within the module

ReactionLog — reaction-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `reaction-service.json` · reaction 0.51.5 · 2 operations

`GET /v1/reaction-logs` `Since 1.10`

`find`

Parameters

Name	In	Required	Type
`reactionId`	query	no	`string`
`eventName`	query	no	`string`
`actionType`	query	no	`string`
`status`	query	no	`string`
`from`	query	no	`string`
`to`	query	no	`string`
`doneBy`	query	no	`string`
`search`	query	no	`string`
`order`	query	no	`string`
`skip`	query	no	`number`
`limit`	query	no	`number`

Responses

Status	Description
`200`	A page of recorded reaction runs across the caller's space

`GET /v1/reaction-logs/{id}` `Since 1.10`

`findById`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`reactionId`	query	no	`string`
`eventName`	query	no	`string`
`actionType`	query	no	`string`
`status`	query	no	`string`
`from`	query	no	`string`

Name	In	Required	Type
`to`	query	no	`string`
`doneBy`	query	no	`string`
`search`	query	no	`string`
`order`	query	no	`string`

Responses

Status	Description
`200`	One recorded reaction run, its full record and its neighbours

SystemReaction — reaction-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `reaction-service.json` · reaction 0.51.5 · 5 operations

`GET /v1/schema/{entity}/{trigger}`

`getTriggerSchemas`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`trigger`	path	yes	`string`

Responses

Status	Description
`200`	—

`GET /v1/schema/{entity}/announcement/{announcementId}/source/{sourceType}`

`getAnnouncementSourceEventSchema`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`announcementId`	path	yes	`string`
`sourceType`	path	yes	`number`
`newTriggerId`	query	no	`string`
`relationId`	query	no	`string`

Responses

Status	Description
`200`	—

`GET /v1/schema/{entity}/event/{eventType}/{newTriggerId}`

`getEventSchema`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`eventType`	path	yes	`string`
`newTriggerId`	path	yes	`string`

Responses

Status	Description
`200`	—

`GET /v1/schema/{entity}/event/{sourceType}/{sourceId}/{newTriggerId}/source`

`getSourceEventSchema`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`sourceType`	path	yes	`number`
`sourceId`	path	yes	`string`
`newTriggerId`	path	yes	`string`

Responses

Status	Description
`200`	—

`GET /v1/schema/announcement/{announcementId}`

`getAnnouncementSchema`

Parameters

Name	In	Required	Type
`announcementId`	path	yes	`string`

Responses

Status	Description
`200`	—

WhatsAppTemplates — reaction-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `reaction-service.json` · reaction 0.51.5 · 1 operations

`GET /v1/whatsapp/templates` `Since 1.10`

`listTemplates`

Responses

Status	Description
`200`	Approved WhatsApp templates of the space

Timeline — timeline-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `timeline-service.json` · timeline 0.0.1 · 7 operations

`GET /v1/posts`

`getPosts`

Parameters

Name	In	Required	Type
`filter`	query	no	—

Responses

Status	Description
`200`	Post model instance

`GET /v1/posts/{id}`

`getPostsById`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Post model instance

`PATCH /v1/posts/{id}/pin`

`pinnedPost`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `PinnedBody`

Responses

Status	Description
`204`	pinned/unpinned PATCH success

`PATCH /v1/posts/{id}/publish`

`publishPost`

Parameters

Name	In	Required	Type
`id`	path	yes	`string`

Request body: `PublishBody`

Responses

Status	Description
`204`	publish/unpublish PATCH success

`GET /v1/posts/count`

`getPostsCount`

Parameters

Name	In	Required	Type
`where`	query	no	—

Responses

Status	Description
`200`	Post model count

`GET /v1/posts/entities/{entityId}`

`getPostsByEntity`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`
`filter`	query	no	—

Responses

Status	Description
`200`	Post model instance

`GET /v1/posts/entities/{entityId}/count`

`getPostsByEntityCount`

Parameters

Name	In	Required	Type
`entityId`	path	yes	`string`

Responses

Status	Description
`200`	Post model count

Workflow — workflow-service

Every response below is delivered inside the universal `{success, result}` envelope; downloads and streams bypass it structurally. See Calling the HTTP API.

> Snapshot: `workflow-service.json` · workflow-service 0.0.1 · 10 operations

`POST /v1/entities/{entity}/{workflowId}/{actionId}/bulk`

`bulkExecuteWorkflowAction`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`workflowId`	path	yes	`string`
`actionId`	path	yes	`string`

Request body: `BulkExecuteActionRequestDto`

Responses

Status	Description
`200`	BulkExecuteWorkflowActionResponseDto model instance

`POST /v1/entities/{entity}/entries/{entry}/{workflowId}/adhoc-step`

`addAdHocStep`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entry`	path	yes	`string`
`workflowId`	path	yes	`string`

Request body: `CreateAdhocWorkflowStepDto`

Responses

Status	Description
`200`	Return value of WorkflowController.addAdHocStep

`PATCH /v1/entities/{entity}/entries/{entry}/{workflowId}/execute-action`

`executeWorkflowAction` — Execute action of workflow that belongs to an entity

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`workflowId`	path	yes	`string`
`entry`	path	yes	`string`

Request body: `ExecuteActionModel`

Responses

Status	Description
`200`	workflow PATCH success

`PATCH /v1/entities/{entity}/entries/{entry}/workflow/{workflowId}/change-step`

`changeCurrentStep` — Change current step of a data entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`workflowId`	path	yes	`string`
`entry`	path	yes	`string`

Request body: `ChangeWorkFlowStepDto`

Responses

Status	Description
`200`	workflow PATCH success

`GET /v1/entities/{entity}/entries/{entryId}/main-flow-status`

`getMainWorkflowStatus`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`mode`	query	no	`string`

Responses

Status	Description
`200`	—

`GET /v1/entities/{entity}/workflow/{scope}`

`getWorkFlowOwners`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`scope`	path	yes	`string`

Responses

Status	Description
`200`	find all workflow owners or original owners of an entity depends on the scope

`GET /v2/entities/{entity}/entries/{entry}/{workflowId}/adhoc-step-owners`

`getAdhocStepOwners`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entry`	path	yes	`string`
`workflowId`	path	yes	`string`
`skip`	query	no	`number`
`limit`	query	no	`number`
`search`	query	no	`string`
`ownerType`	query	no	`number`

Responses

Status	Description
`200`	Get eligible owners for adhoc step owner selection

`PATCH /v2/entities/{entity}/entries/{entry}/{workflowId}/change-owner`

`changeCurrentStepOwnerV2` — Change the workflow owner of a data entry

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`workflowId`	path	yes	`string`
`entry`	path	yes	`string`

Request body: `ChangeStepOwnerDto`

Responses

Status	Description
`200`	workflow PATCH success

`GET /v2/entities/{entity}/entries/{entry}/{workflowId}/eligible-owners`

`getEligibleOwners`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entry`	path	yes	`string`
`workflowId`	path	yes	`string`
`skip`	query	no	`number`
`limit`	query	no	`number`
`search`	query	no	`string`

Responses

Status	Description
`200`	Get eligible owners for change owner dialog

`GET /v2/entities/{entity}/entries/{entryId}/main-flow-status`

`getMainWorkflowStatusV2`

Parameters

Name	In	Required	Type
`entity`	path	yes	`string`
`entryId`	path	yes	`string`
`mode`	query	no	`string`

Responses

Status	Description
`200`	—

Requests & services (API and internals)

This page is for integrators and developers who read or drive service requests from outside the portal, or who need to explain exactly why a user can or can't see a request. It covers the data model, the access rules for each access mode, the single access endpoint, the system lists behind the portal's request areas, and how the **Request Status** a user sees is computed.

The end-user view of the same features is Services & Submitting Requests [Using OSOS]. How to design a service's workflow is in Workflows (Design) [Build Apps › Workflows & approvals].

The model

A **service** is a data entity of type `Request` whose `meta.activated` flag is `true`. Only activated services ever reach the portal. Submitting a service creates a **request**: an entry of that entity that carries a workflow.

When the entry is created, the platform starts the entity's active workflow on its first step. A workflow must begin with an **Entry** step, and the owner of a main workflow's Entry step is always the entry's creator. So a newly saved request sits on the requester's own step, with an empty step history, until the requester runs that step's action (the designer names it, typically *Submit*). An approver who "returns" a request with a relative *previous step* action, or with the admin *Return to step* action, sends it back to that Entry step and therefore back to its creator. Inside a sub-workflow the Entry step instead mirrors the owner of the parent's sub-workflow step.

Each request also has a lightweight **tracking record** in the system `REQUEST` entity (see The request tracking record). The portal's request lists read those records, not the original entries.

Access modes

The portal splits request access into four modes. Each has its own portal area, its own access check and its own implicit (system-generated) views.

Mode (`RequestAccessModeEnum`)	Portal area (English label)	Route	Access check	Detail view (`ViewImplicitType`)	List views
`New`	Services Catalog	`/s`	`canCreateRequest`	`MY_REQUEST_DETAILS`	`SERVICE_CATALOG` view of the `SERVICES` entity
`MyRequest`	Outbox	`/my`	`canAccessMyRequest`	`MY_REQUEST_DETAILS`	`ONGOING_MY_REQUESTS`, `COMPLETED_MY_REQUESTS`
`Inbox`	Inbox	`/inbox`	`canAccessInbox`	`INBOX`	`ONGOING_INBOX`, `COMPLETED_INBOX`
`Observer`	Watching	`/rm`	`canAccessObservers`	`OBSERVER`	`ONGOING_OBSERVER`, `COMPLETED_OBSERVER`

The four service actions are `ServiceActionTypes` `submit`, `request`, `inbox` and `observe`. Each maps to a delegation permission pattern of the form `so:service:{serviceId}:{action}` (see Delegation).

*Older pages and the route keys still call Outbox "My Requests" and Watching "Request Monitoring". The English UI labels are **Outbox** and **Watching**; the service list is **Services Catalog**.*

The access endpoint

`RequestController` exposes one authenticated route:

```
GET /v1/requests/{entity}/{id}/{mode}
```

- `entity`: the service entity id.
- `id`: the request entry id, or the new-request placeholder.
- `mode`: a `RequestAccessModeEnum` value: `New`, `MyRequest`, `Inbox` or `Observer`.

It requires the `Auth2` authentication strategy. The handler carries `@authorize.skip()`, because access is resolved by the per-mode checks below rather than by the generic authorizer. It returns the implicit view (`DataViews`) the caller is entitled to for that mode, or `null` if the caller may not access it:

Mode	Check performed	View returned
`New`	`canCreateRequest` (audience)	`MY_REQUEST_DETAILS`
`MyRequest`	`canAccessMyRequest` (you created it)	`MY_REQUEST_DETAILS`
`Inbox`	`canAccessInbox` (owner, group or history)	`INBOX`
`Observer`	`canAccessObservers` (observer lists)	`OBSERVER`

Each implicit view is matched by `{dataEntity, isPublished: true, isImplicit: true, implicitType}` (`ViewImplicitType` values for requests: `INBOX`, `MY_REQUEST_DETAILS`, `OBSERVER`, plus `NONE`). If the target entity is not of type `Request`, the endpoint raises `EntityNotOfTypeRequestException`.

The portal calls this endpoint (as `FetchServiceImplicitViewForRequest`) in two places:

- On a service's page, it builds the **New Request** button (`action.NEW_REQUEST`) only when the `New` call returns a view. No view, no button.
- As a route guard on every request page. With no view the user is sent to `/unauthorized`, except that a request opened from the catalog falls back to the service page.

Who can reach a request

Submit: `canCreateRequest`

When the portal builds a user's navigable objects it calls `getUserServicesCanAccess`, which filters the activated services down to the ones `canCreateRequest` allows. The rule is:

1. If the service is not `activated` or has no `audience`, it is hidden.
2. If anonymous access is enabled (and the caller is not acting under a delegation): an **anonymous-only** service (anonymous enabled, authenticated *not* enabled) is hidden from signed-in users; otherwise it is shown.
3. Any authenticated path needs a signed-in user (`sub`).
4. If the authenticated audience is **restricted**, the user must appear in `entitledEntries`, either directly by user id or through one of their groups.
5. Under a delegation, the result is then filtered by the delegation scope for the `submit` action.

Because this is enforced server-side, a service the user can't submit is simply absent from the catalog. There is no "request access" flow.

Outbox: `canAccessMyRequest`

The caller is the creator (`creationInfo.doneBy` equals their user id), or the anonymous rule below grants access. Under a delegation the decision is the delegation scope for the `request` action.

Anonymous access to a request (`isAnonymouslyAccessibleRequest`): an unauthenticated caller may open a request only if it was created anonymously (its `creationInfo.doneBy` names no user; missing metadata fails closed), and only while **every** current workflow step is still unowned, that is, before the request has been handed to a staff owner. A request with no current steps does not qualify. An authenticated caller never qualifies through this rule.

Inbox: `canAccessInbox`

The **creator is always excluded**. A requester never reaches their own request through the Inbox, not even when its current step is theirs (for example after it was returned to them). They reach it through `MyRequest`, where the Request Status shows *Pending Your Action*.

For everyone else, any one of these is enough:

- The user is the current step **owner** (`owner.dataEntry`), or is in the current step's list of original owners (array form).
- The user belongs to a **group** that is an original owner of a current step, and that step is unassigned or is an "all users" step (`accessPermission === AllUsers`).
- The user belongs to a **collection** that is an original owner under the same unassigned / all-users condition.
- The user **executed** a step earlier in the request's history (`executedBy`), or **was an owner** at a past step.
- The user was inside a **group of owners** on an all-users step in the history.

Under a delegation the decision is the delegation scope for the `inbox` action.

Watching: `canAccessObservers`

The user is listed in the service's `meta.observers.users`, or belongs to a group in `meta.observers.groups`. The portal only surfaces the Watching area for services the user can observe (`getUserServicesCanObserve`), and the sidebar shows it only when the user has ongoing or completed observed requests. Under a delegation the decision is the delegation scope for the `observe` action.

Who can edit a request

The edit route (`/s/:serviceId/r/:requestId/edit`) has its own guard. If the request has a workflow, the user must be the current step owner, be able to take ownership of the step, be able to execute the step's workflow action (this covers members of a group that owns a parallel step), or hold the view's *edit as admin* grant. Otherwise they are redirected to the read-only request page. This is what lets a requester edit their own request only while it sits on their step.

Service audience model (`ServiceMeta`)

Field	Type	Meaning
`audience.anonymous.enabled`	boolean	Anonymous users may submit
`audience.authenticated.enabled`	boolean	Signed-in users may submit
`audience.authenticated.isRestricted`	boolean	Restrict to a named list
`audience.authenticated.entitledEntries`	array	Users/groups allowed when restricted

Field	Type	Meaning
`activated`	boolean	Service is live (required for listing)
`category`	string	Catalog category
`onBehalf`	string	Field path used as the request's subject
`observers.users` / `observers.groups`	refs	Who may observe requests
`showServiceForEveryone`	boolean	Catalog visibility toggle
`showServiceForAuthenticatedUsersOnly`	boolean	Restrict catalog visibility to signed-in users

On-behalf subject. When the tracking record is built, the "on behalf of" party is read from the field named by `onBehalf` on the submitted data (`meta.onBehalf`). If it is empty, it falls back to the request's creator. This is what the portal displays as the subject of the request.

Anonymous forms and human verification. A service whose audience allows anonymous access can be protected, view by view, by a proof-of-work human check. It is off by default. Each challenge is single-use. Signed-in users are never challenged. If the verification service is unavailable, the caller is treated as unverified and a form that requires verification is refused. The wire contract for driving such a form headlessly is in Writing Data; turning it on is in Human Verification [Administration › Human verification (bot protection)].

The request lists

The portal's Outbox, Inbox and Watching tabs are implicit views over the `REQUEST` entity, each backed by an implicit query seeded at install. Two of them show the rules that matter most:

- **Ongoing Outbox** (`ONGOING_MY_REQUESTS_QUERY`): `creationInfo.doneBy = @me` and `status.id != COMPLETED`. A request that is waiting on its creator therefore stays here, with status *Pending Your Action* (or *Draft* if it was never sent on).
- **Ongoing Inbox** (`ONGOING_INBOX_REQUESTS_QUERY`): `creationInfo.doneBy != @me` and the current step is owned by `@me`, or is unowned / all-users and its original owner is in `@myGroups` or `@myCollections`, or is an array of original owners that includes `@me`.

Drafts are a different thing from the *Draft* status. A form saved with **Save as Draft** is an entry in the `Draft` state. It has no workflow yet and is listed only through the Ongoing tab's Drafts switch. The option is offered on a new request (or an existing draft) only when the view's `editConfiguration.allowCreateEntryAsDraft` is on.

Live counts. The catalog subscribes to a socket channel (`subscribeToModifiedRequestsCounts`) and refreshes the Ongoing/Completed counts as requests change. All three lists receive step and status changes over the same real-time connection. The sidebar's service status reports ongoing counts and whether each area has ongoing **or** completed requests, which decides whether the Outbox, Inbox and Watching items are shown.

Delegation filtering of lists. While a user is delegating, an HTTP interceptor rewrites the query on the Service Catalog and on the Outbox / Inbox / Watching entry and count endpoints. See Delegation.

The request tracking record

Each ongoing or completed request is represented by a `REQUEST`-entity entry, `RequestDataEntryValue`, built by `RequestService.buildRequestEntry`. It references the original submission and carries the fields the portal lists show:

- `requestNumber`, `reference` (original entity and entry), `serviceReference`.
- `service` (name, description, icon, category) and `observers`.

- `currentStep` (name, owner, original owner, received-at, owner definition).
- `onBehalf`, `creationInfo`, `updateInfo`, `entityVersion`.
- `lifeCycleStatus` (`LifeCycleStatusEnum`): `Active` (0, in progress), `Completed` (1, finished normally), `Failed` (-1), or undefined (not in a workflow).
- `isInvalid` / `invalidReason` (`DataEntryInvalidReasonEnum`: `Obsolete` = 0, `Invalid` = 1).
- Workflow metadata: `_metaCurrentSteps`, `_metaWorkflowHistory`, `_workFlowStepsHistory`, `_metaAdhocSteps`, `_metaOrder`.

If a request has no active workflow, or no backing service entry can be found, `buildRequestEntry` returns `null` and no tracking record is produced. The record is rebuilt from the entry as currently persisted rather than from an event snapshot, so concurrent updates can't roll a list row back to an older step.

The current step and step history are taken from the **active** workflow instance (`metaCurrentStep.workflowId` / `instanceId`). A request running a nested sub-workflow therefore shows the sub-workflow's step and owner, not the parent's.

The computed Request Status

The `status` field (*Request Status*) is an implicit, virtual category field whose value is computed per viewer. The branches are evaluated in this order:

`status.id`	English label	Condition
`COMPLETED`	Completed	`lifeCycleStatus` is `Completed` (1) or `Failed` (-1)
`OBSOLETE`	Obsolete	`invalidReason` is `Obsolete` (0)
`DRAFT`	Draft	The first workflow history has no executed steps yet
`PENDING`	Pending Your Action	`lifeCycleStatus` is `Active` and the active step's owner is the viewer, or the step is unowned / all-users and the viewer is in its original owners (directly, by group or by collection)
`ON_GOING`	On Going	`lifeCycleStatus` is `Active`
`NONE`	None	Anything else

The Ongoing/Completed split of the lists follows `status.id`: Completed holds `COMPLETED`, Ongoing holds everything else.

Earlier documentation listed the request statuses as Draft, Open, Completed and OnHold, with result values Succeeded, Failed, Rejected, Timedout, Cancelled and Obsolete. None of those values exist in the current code. The table above is the current set.

Notifications when a request reaches someone

When a service entity is created or updated, the reaction service adds an implicit reaction to it: on `WorkflowStepReached`, send an in-app notification to the current step owner, with the title **Action Required** and the text *You have a new request* (notification type `INCOMING_REQUEST_RECEIVED`). This is how a requester learns that a request was returned to their step, and how an approver learns that a request arrived. Email or SMS on the same events are separate reactions the designer adds.

Delegation

When a user acts under a **delegation**, access to submit, Outbox, Inbox and Watching is decided by the delegation's scope instead of by the ordinary ownership rules:

- **Scope** is either `ALLOW_ALL` or `DENY_ALL`.
- **Exceptions** are patterns that carve holes in the scope. Global exceptions are `so:all`, `so:app` and `so:service`. A per-service, per-action exception is `so:service:{serviceId}:{action}`, where the action is `submit`, `request`, `inbox` or `observe`.
- Under `ALLOW_ALL`, a matching exception **denies** the action. Under `DENY_ALL`, a matching exception **allows** it.

The same logic filters the lists fetched while delegating. An HTTP interceptor rewrites the query on the Service Catalog and on the Outbox / Inbox / Watching entry and count endpoints: under `ALLOW_ALL` the excepted services are removed (`nin`), and under `DENY_ALL` only the excepted services are kept (`inq`). All of this happens server-side. The full delegation model is in Delegation [Administration].

Portal routes

Route	Purpose
`/s`	Services Catalog
`/s/:serviceId`	Service page (details and **New Request**)
`/s/:serviceId/r/new`	New request form
`/s/:serviceId/r/:requestId`	View a request
`/s/:serviceId/r/:requestId/edit`	Edit a request (guarded, see Who can edit a request)
`/my`	Outbox (Ongoing / Completed)
`/inbox`	Inbox (Ongoing / Completed)
`/rm`	Watching (Ongoing / Completed)

Request pages opened from a list use the list's prefix, for example `/my/:serviceId/r/:requestId`. A draft is opened with the `?d=true` query parameter.

The Services Catalog requests every entry the user can access in one call (`limit` is set to the maximum safe integer and `activePage` is pinned to `0`), so it has no pagination. Only valid, non-obsolete service entries are shown.

Caching

The set of activated services is cached once per tenant under the `SERVICES` cache key, with the default cache expiry, so repeated catalog loads don't re-query every service. The value doesn't depend on who asks: the per-user filtering (`canCreateRequest`, `canAccessObservers`) runs on the cached list afterwards. A change to an activated service entity deletes the key. (Older releases cached one copy per user, under a key that included the user id.)

Behaviour history

These changes are summarised in Versions [General]. The detail is kept here because it explains behaviour an integrator may still meet on older deployments.

- **Sub-workflow-aware tracking records** `Since 1.4`. The tracking record takes the current step and step history from the active workflow instance, so requests in a nested sub-workflow list the correct step and owner rather than the parent's.
- **Request list corrections** `Since 1.4`. These were also delivered to the 1.4, 1.6 and 1.8 lines as an update (first on 1.4) and are present in 1.10:
  - The Inbox Completed tab matches the user against the history of *every* workflow of the request, not only the one it ended in. An approver inside a sub-workflow of a request that then completed in its main workflow used to lose it from Completed. Known limit: a member of a group that *could* have acted on a group-owned step, but did not, still doesn't see the request there.
  - Tracking records whose current step or finished state had drifted are repaired, and a request that is reset or has just been created stays in step with its record.
  - When a service's observers change, its existing requests are updated too. A new observer sees earlier requests in Watching, and a removed observer stops seeing them.
  - The Watching, Outbox and Inbox sidebar items appear when the area has ongoing **or** completed requests. Before, an area whose requests were all completed could vanish, because its completed count reused the ongoing list.
  - Anonymous requesters can open their request details after submitting.
- **Archiving a service cascades** `Since 1.8`. Archiving a service from the admin Services Catalog archives every active request already submitted for it, so they drop out of Outbox, Inbox and Watching at once. Restoring the service restores them. Before 1.8, archiving or restoring a service only changed the catalog listing, and submitted requests kept showing regardless of the service's state.
- **Implicit request fields self-heal** `Since 1.8`. If a `Request`-type entity's field configuration loses its implicit `requestNumber` or `status` field (legacy data, or a system-driven update that bypassed implicit-field validation), the platform appends the missing fields on the next save, whether create or update. Before 1.8 the repair ran only on create, so an entity that lost these fields stayed broken.

Account security and step-up (internals)

This page is for integrators and developers who need the exact behaviour behind the user's security settings and the reveal of protected values: which method the server picks, what is enforced where, and what the web portal and the mobile app do around each call.

The end-user view is Account Security & Revealing Protected Values [Using OSOS]. The step-up mechanism itself (challenge, verify, session storage, the three purposes `Unmask`, `WorkflowAction` and `Decrypt`) is specified in Sensitive Fields — Step-up unmask session [Build Apps › Protect a field] and three purposes, one mechanism [Build Apps › Protect a field]. OTP delivery, rate limits and passkeys on the server side are in Identity server.

Where the settings live

- **Web:** the **User Preferences** page at `/user/profile`, sections Profile, Account, Security (Password, Two-factor authentication, Passkeys) and Logins (Sessions, Attempts). The former `/user/security` route now redirects to `/user/profile`, so old bookmarks still land.
- **Mobile:** the **User Center**, tabs Profile · Security · Logins `Since 1.10`. The user-options sheet no longer has a separate Security item; the reveal wizard's "open Security" link goes to the tab.

Both clients call the same identity-service API, so a method or passkey set up on one is what the other sees. For a federated (SSO) account the Security section replaces the passkey card with the "managed by your organization's identity provider" message.

Verification methods

Method	How it appears	Removable by the user
Authenticator app (TOTP) `Since 1.8`	Enrolled by the user in Security.	Yes. The only method a user can remove.
Passkey	Registered per device in Security.	Revoked individually in the passkeys card, not through the general remove action.
SMS / email	Provided by the organization from the phone and email on the account, when the domain enables them.	No.
WhatsApp `Since 1.10`	Offered when the organization has turned WhatsApp on **and** the account has a verified phone number.	No. It appears and disappears with the setting and the verified phone.
Password	Never chosen by the user. See Password fallback.	—

The **server** decides which methods are available: TOTP once enrolled, SMS and email once the domain enables them, WhatsApp as above, passkey only while at least one is registered. When the last passkey is revoked, the client drops the passkey option at once, before the server's cached state catches up.

One method is the **preferred (default)** method. The web card shows it with an "Enabled since" date; every other available method offers **Set as default**.

Enrolment

The user scans a QR code (or types the secret) and confirms with a 6-digit code. The web wizard asks for the account password first. The setup secret is short-lived (the mobile app states 10 minutes); if it expires before confirmation, the

client restarts with a fresh secret and QR code instead of reporting a wrong code, since there is nothing left to confirm against.

Passkey rows

Each passkey shows a device label and a device-type icon (phone, laptop or security key). The icon is a **display hint** only. Rows show when the passkey was created and last used, in local time. The web passkey-details dialog that listed a sign-in count is no longer opened from the card. The server's count reflects genuine sign-ins, so a passkey synced across devices is not reported as unused.

WhatsApp ordering `Since 1.10`

The user's saved default wins. With no default and both SMS and WhatsApp offered, the organization's preferred phone channel is shown first. Once WhatsApp is available to a user it counts as an MFA method, so sign-in requires a second step even where the domain did not require one before. The mobile Security list skips any method the app does not know, which today includes WhatsApp. Send failures surface as `MFA_WHATSAPP_SEND_FAILED`. Operator setup: WhatsApp codes.

Recovery codes `Since 1.10`

- Confirming TOTP enrolment generates **8 one-time codes** and returns them **once**. Afterwards only a status summary (generated when, how many remain) is available.
- **Regenerating** replaces the whole set; every unused code of the old set stops working when the new set is issued. It requires fresh proof (a current TOTP code or the account password). An access token alone is not enough, so a hijacked session cannot silently rotate the fallback.
- Two concurrent regenerate requests cannot both win: one succeeds and invalidates the codes the other would have returned. The mobile app reports the loser as "Codes were just regenerated elsewhere. Try again."
- Disabling TOTP also clears the recovery codes.
- On mobile, **Download** opens the platform save dialog; cancelling it is a no-op.

Sessions and sign-in attempts `Since 1.10`

Sessions lists every signed-in device or browser with device, OS, browser, first sign-in, last activity and whether it is the current session.

- Ending **another** session revokes its tokens on the server first, then removes the row, so it stops working immediately.
- Ending the **current** session is a real sign-out, because the token the page runs on is now dead.
- If revocation fails part-way, the row stays, so the list never claims a session is gone while it can still be used.

Attempts is read-only by design: sign-in attempts from the last 60 days, newest first, with device, OS, browser, timestamp and outcome, including failed attempts by others. It pages ("Load more") and shows no location. The underlying store is described in Authentication audit trails.

Password-reset links `Since 1.4`

Requesting a reset **replaces** the previous link with a new one with a fresh lifetime (24 hours by default); the earlier link stops working, and asking again never extends an old link. Links issued before a deployment upgraded to a release with this behaviour (1.10, or the 1.4.118, 1.6.67 and 1.8.25 patches) stop working too. Server side: Password reset.

A browser whose last session belonged to an account that no longer exists signs in normally `Since 1.10`. See Session holding another account's grant.

The sensitive-value reveal (Unmask)

The wizard is the same on web and mobile:

1. **Duration.** 2, 5, 10, 30 or 60 minutes, or 3, 4, 6 or 8 hours. The list mirrors the identity service's allow-list exactly; anything else is rejected with `INVALID_DURATION`. (The 1.8 cycle corrected an inconsistent list some clients showed; mismatched options would indicate drift.)
2. **Challenge.** The client requests a challenge for the chosen duration and the **server picks the method**: normally the preferred one, falling back through the available ones if it is unusable. Then the user either:
    - enters a code: for SMS, WhatsApp or email a code is sent to a masked destination; for TOTP nothing is sent. **The code length is whatever the server reports** for the challenge;
    - enters the password, if the server chose the password fallback (no code, no resend, no method switch);
    - switches method, which restarts the challenge. **Passkey is never offered as a switch target for reveal**, on either client;
    - resends a code for a sendable method once the server-reported cooldown has passed.
3. **Window.** Values render unmasked for the chosen duration until it expires or the user ends it. Ending early applies on the device immediately, before the server confirms.

The web portal opens the same wizard from a masked field's eye button (tooltip **Unmask**) and from the profile menu (**Unmask** / **Disable unmask**).

Scope caveat. A reveal currently covers **every** sensitive field the user is entitled to, not just the field whose eye was selected. Per-field scoping exists internally but nothing in the shipped experience narrows it.

Mobile immediacy `Since 1.8`. A granted field unmasks in place as soon as the reveal succeeds; it no longer needs the entry or list to be reopened. Mechanism: Clearing a stale mask stamp on reveal.

Password fallback

The server may ask for the account password instead of an MFA code when **all** hold: the user has no usable MFA method (`MFA_NO_METHOD`), the domain has opted in (`allowPasswordStepUp`, off by default; **Allow Password Step-Up** in Identity providers [Administration › Sign-in methods & single sign-on]), and the account has a local password. The client never offers it as a choice; it renders a password field when the server selected it.

When reveal is unavailable

If the server reports no usable method, or a federated/SSO account (`EXTERNAL_USER_FORBIDDEN`), the wizard says reveal is unavailable instead of showing a broken challenge. Only for "no method set up yet" does it offer to open Security; for a federated account or an unsupported method it does not, because enrolling there would not help.

Lockout

Too many wrong codes or passwords, or too many challenges too quickly, lock input until a server-reported time. The limit is **account-scoped**: it is the same per-account limiter general MFA uses, so another session gives no fresh allowance. If the server does not say when the lock lifts, the client assumes a short default wait. If the pending challenge expires or is invalidated meanwhile, the wizard restarts the verification step instead of failing.

Lifecycle of the reveal window

Independently of the countdown:

- **Backgrounding and resuming the mobile app.** On resume, values render masked first, then the app re-confirms with the server whether the window is still open. It never assumes the window survived.
- **Signing out, a forced sign-out, switching spaces, or starting or ending a delegated session** closes the window on the device at once, without waiting for the server, so acting as someone else never inherits a window.
- **A masked value seen while the client believed it was revealed** is taken as authoritative: the client closes the window and re-checks.

No reveal state is stored on the device between sessions; every launch starts masked and asks the server. Wiring: Sensitive-value reveal wiring.

Encrypted-field decrypt `Since 1.10`

A field is either **Sensitive** or **Encrypted**, never both. An encrypted value is stored encrypted at rest, so its reveal is a separate path with no duration and no window, and it **requires its own step-up check**. (Encrypted fields arrived on the development line before 1.10 with a grant-only, check-free decrypt; that behaviour never shipped in a release.)

- **Affordance.** With a decrypt grant, the field shows an eye (tooltip **Decrypt**) beside `****`. Without the grant, a lock icon and nothing to select. A field rendered as a password never offers decrypt, even if marked Encrypted, because its stored value is a one-way hash; the decrypt endpoint also rejects that case.
- **Web.** The eye opens the step-up prompt (**Verify your identity**) with the same method-selection rules as the reveal wizard, scoped to this one reveal. On success the value replaces the mask in place. Without entitlement the client does nothing rather than show an error, since the lock icon already said so.
- **Mobile.** Tapping the masked value opens the same verification sheet the app uses for a step-up workflow action; the screen reader label is “Encrypted field, value hidden. Tap to decrypt”. A cancelled or failed prompt leaves the field masked. Implementation: the per-field decrypt cubit.
- **Single-use.** The proof is spent on redemption, valid for about **120 seconds** if unused, and bound to one field (and, inside a repeatable block, one row). A second field, or a reload, needs a fresh prompt.
- **Visibility.** The decrypted value stays while the field is on screen. Navigating away, scrolling it out of view, or reloading re-masks it. Nothing about the reveal is persisted on the device or the server.
- **Audit.** Every successful decrypt is written to the audit trail as *Decrypted Data*; the identity check is recorded separately, success or failure. The value itself is never recorded, only who, when and which field.
- **Failures.** A caller without the grant gets the same refusal every time and is never locked out (there is nothing to guess). A caller with the grant who fails the check repeatedly hits the account-scoped limit described under Lockout.

Endpoint and proof binding: Decrypting a value.

Entity model and API reference

This page is for **integrators and developers** who read or write entity definitions through the API, generate entity payloads, or need the exact rules the server applies when an entity is created, changed, moved or deleted. It documents the stored model, the value shapes entries carry, the lifecycle endpoints and their refusals, and the versioning (breaking-change) system.

> **Designing an entity in the UI?**
>
> Builders should start with Create and design an entity [Build Apps] and Change an entity safely [Build Apps] — they cover the same features in the labels the designer shows.

How entity writes are processed

Writes to entities are **command-driven**. The HTTP controller does not persist directly — it validates the request, then sends a command (`CreateDataEntityCommand`, `UpdateDataEntityCommand`, `ArchiveDataEntityCommand`, and their virtual counterparts) onto the service bus and registers an invoker token. A command **handler** performs validation, preparation, persistence, and event publishing asynchronously, then replies over a socket. Because of this, the `POST` / `PATCH` / `PUT` endpoints return the entity as read back from the repository right after enqueuing; the authoritative result of the operation is delivered via the command-executed socket message.

Entity structure

The persisted document is defined by the `DataEntity` model (`data-entity.model.ts`). Key properties:

Property	Type	Notes
`id`	UUID	Server-assigned (`v4()`) on create; not client-supplied.
`specificationVersion`	number	Schema/spec version of the entity model. Default `1`.
`name`	LocalizedValue	Required. Localized display name. Validated (required, min/max length) and must be unique **within its module**.
`description`	NullableLocalizedValue	Optional.
`normalizedName`	string	Required, **globally unique** (unique index). Must match `NormalizedNameValidation` pattern and length; cannot be a restricted name (see below).
`version`	number	Data/breaking-change version. Default `0`. Bumped only by qualifying breaking changes (see Versioning).
`isImplicit`	boolean	Required. Marks system-provisioned entities with restricted edit/delete rules.
`implicitOptions`	`{allowEdit, isReadOnly}`	Governs whether an implicit entity may be edited.
`virtual`	DataEntityVirtualOptions	Present for pipeline-backed virtual entities; carries `enabled`, `source`, and computed `dependsOn`.
`type`	`DataEntityTypeEnum`	Required. See Entity types.

Property	Type	Notes
`typeOptions`	TypeOptions | VirtualProxyEntityOptions | AggregatorVirtualEntityOptions	Type-specific configuration (e.g. virtual/proxy/joiner/aggregator sources).
`icon`	Icon	Required.
`module`	UUID	Required. Owning module.
`state`	`DataEntityStateEnum`	`Draft` / `Active` / `Archived`.
`stateHistory`	StateHistory[]	Append-only log of state transitions.
`creationInfo` / `updateInfo` / `archiveInfo`	ActionInfo	Who/when, including delegatee. `archiveInfo` is set only on archive.
`fields`	DataEntityField[]	The entity's own fields (see Fields).
`internalEntities`	InternalEntity[]	Backing schemas for **block** fields (nested field groups).
`internalCollections`	InternalCollection[]	Embedded list-of-values collections.
`workflows`	DataEntityWorkflow[]	Attached approval/state workflows.
`validation`	string | null	Entity-level validation expression.
`tags`	string[]	Free-form tags.
`meta`	ServiceMeta | object	For `Request`-type entities: service audience/observers.
`displayOption`	`DataEntityDisplayOptionEnum`	Layout: `KeyValue` (default), `TwoColumns`, `ThreeColumns`, `FourColumns`.
`formSettings`	DataEntityFromSettings	Form-render configuration.
`entryNames`	`{singular, plural}`	Localized labels for a single record vs. many.
`treeStructure`	DataEntityTreeStructureConfig	Config for self-referential tree/hierarchy entities.
`indexes`	DataEntityIndex[]	Index definitions (computed on create/update).
`dataSources`	DataEntityFieldSources[]	Computed field-source mappings.
`storeRules`	StoreRule[]	Data-placement rules; defaulted from `StoreService.defaultStoreRule()` on create.
`sharing`	ApplicationSharing	Inherited from the owning module.
`updateIdentifier`	string	Optimistic-concurrency token (`v4()`), rotated on every write.
`displayTemplate`	EntityDisplayTemplate | null	Optional custom display template. **Since 1.4**
`scriptAttributes`	object	Optional bag of scripting attributes. **Since 1.8**
`deletedFields` / `breakingFields`	arrays	Version bookkeeping. `breakingFields` is a hidden property (`hiddenProperties: ['breakingFields']`).

Fields

Each entry in `fields` is a `DataEntityField` (`data-entity-field.model.ts`):

Property	Type	Notes
`id`	UUID	Field identity; stable across renames and used for breaking-change diffing.
`specificationVersion`	number	Default `1`.
`name`	LocalizedValue	Localized label.

Property	Type	Notes
`normalizedName`	string	Unique field key; matches `NormalizedNameValidation`; cannot be a restricted name.
`dataType`	`DataTypeEnum`	Required. One of the types below.
`block`	FieldBlock	For `Block` fields, links to an `internalEntity` holding the sub-fields.
`options`	FieldOptions	Behavioural flags incl. `repeatable`, `multiValue`, `showAsTitle`, `showAsMainPhone`, `showAsEntryPicture`, `sensitive`, `displayOnly`.
`source`	UUID | null	**Deprecated** single reference target (use `sources`). For `Entity`-typed reference fields.
`sources`	FieldSources[]	List of available reference sources.
`sourceOptions`	FieldSourceOptions	e.g. `denyCircularRelation`.
`dataCollection`	UUID | null	Reference to a data collection (LOV).
`renderOptions`	FieldRenderOptions	Carries `editComponent.component` (e.g. `RelationComponent`, `CategoryComponent`, `SwitchComponent`, `PasswordInputComponent`, `DataQuerySelectorComponent`).
`validations`	FieldValidation	Includes `required.enabled`.
`customValidations`	CustomValidationsRule[]	Optional custom rules.
`locked`	FieldLockedOption	Lock/expression control.
`virtualField`	VirtualField	Marks a computed field (`enabled`).
`isImplicit`	boolean	System field marker (forced `false` for user-supplied fields on create).
`mapping`	DataEntityFieldMapping	Field-source mapping metadata.
`sensitiveAccess`	SensitiveFieldAccess	**Transient**, request-only. Read-access grants for a sensitive field, synced to the `access_right` store then stripped before persist — never stored on the entity document. **Since 1.8**

Supported `dataType` values (`DataTypeEnum`): `Text (0)`, `Number (1)`, `Date (2)`, `Time (3)`, `DateTime (4)`, `Boolean (5)`, `blob (6)`, `Entity (7)`, `Icon (8)`, `Block (9)`, `LocalizedJson (10)`, `GeoJson (11)`, `Json (12)`, `ObjectJson (13)`, `SVGIcon (14)`.

Blocks and nested fields. A field with `dataType = Block` does not store its own sub-fields inline; instead `block.internalEntity` points at an entry in the entity's `internalEntities`, which holds the block's `fields`. The model helper `getAllFields()` flattens this: for each block it emits the sub-fields with a `fullNormalizedName` of `<block>.<field>`, an `entityPath`, and an `isParentBlockRepeatable` flag. This flattened, full-name view is what the versioning and validation code diffs against — so fields are matched by the pair `(id, fullNormalizedName)`, not by name alone.

Show-as / titling helpers. The model exposes convenience selectors used across the service: `getTitledField()` (`options.showAsTitle`), `getPhoneNumberField()`, `getEntryPictureField()`, `getRelationFields()`, `getSensitiveFields()`, `getVirtualFields()`, `getLocalizedFields()`, and `getFieldByPath()`. A field marked `showAsTitle` (the "main title") is subject to extra rules on update (see Update rules).

Beyond the title, a field can carry any of the **semantic view-option roles** (`FieldViewOptionValueTypeEnum`) so card/list/kanban renderers know which field to treat as what: `showAsTitle`, `showAsSubTitle`, `showAsCategory`, `showAsSubCategory`, `showAsRelationField`, `showAsMainNumber`, `showAsMainDate`, `showAsMainPhone`, and `showAsEntryPicture`. Set these when configuring a field — e.g. mark your "Full name" text field `showAsTitle` and your image field `showAsEntryPicture`, and card views render correctly with no extra layout work.

Sensitive fields. Any field can be flagged sensitive (`options.sensitive`). A sensitive field's value is masked (`****`) for every user unless that user holds an explicit read grant for that specific field — a dimension separate from ordinary view/edit permissions. Grants are supplied per request through the transient `sensitiveAccess` property, synced to the `access_right` store, and never persisted on the entity document.

Reference sources that aren't data entities. Not every value a reference (`Entity`-typed) field's `source` / `sources` can carry is a `DataEntity` id — a seeded system entity such as the counter widget gives its `query` / `view` fields the literal platform-source strings `dataQuery` / `dataViews` instead, pointing at a saved query or a data view rather than another entity. **Since 1.8**, the entity designer's reference-source picker recognises these platform sources and renders a plain label instead of trying to fetch them as a `DataEntity` — previously that lookup failed with an "Entity not found" toast over what was otherwise a correctly-configured field.

Field-config panel search `Since 1.8`

The field-config panel's section list and search box (see Create and design an entity [Build Apps]) is a **UI navigation aid only**: it adds nothing to `DataEntityField` / `FieldOptions` and has nothing to do with a field's database indexing.

Stored value shapes

An entry stores each field's value under `value.<normalizedName>`. The stored shape depends on the field kind — and the two reference kinds are discriminated not by `dataType` alone but by the edit component plus whether `dataCollection` is set (`FieldService.isReferenceField` / `isADTField`):

Field kind	Discriminated by	Stored value
Text	`dataType: 0`	Plain string — the text caster coerces with `value?.toString() ?? ''`.
Number	`dataType: 1`	Plain number (`Number(value)`).
Category	`dataType: 12` + `editComponent.component: "CategoryComponent"`	`{"id": "<option id>"}` — only `id` is persisted; `label`, `color`, and `_resolved` (`{label, color}`) are re-injected on read from the matching option.
Entity reference	`dataType: 7` + `"RelationComponent"`, entity source set	`{"dataEntity": "<entity id>", "dataEntry": "<entry id>"}` — both required; an **array** of these when the field is `multiValue`.
Collection-item reference	`dataType: 7` + `"RelationComponent"`, `dataCollection` set	`{"id": "<collection item id>"}` — the id of the item inside the referenced collection.
Block	`dataType: 9` + `"BlockComponent"`	A nested object keyed by the internal entity's field `normalizedName`s (e.g. `value.basic_info.username`); an **array** of such objects when `options.repeatable` is set. Empty values are seeded as `{}` — or `Array(min).fill({})` for repeatable blocks.

A category field's selectable options live at `renderOptions.editComponent.options.options` — an array of `{id, label, color?}` where `id` is a plain **string**: usually a UUID for options created in the builder, a constant string for seeded core fields (e.g. the Space entity's `type` options `STANDALONE` / `SUB_SPACE`). The save-time validator rejects a category value whose `id` is missing or not found in that array.

> **Wrong reference shapes and silent failure.** The two reference shapes are not interchangeable, and the failure mode is direction-dependent:
>
> - Sending the collection shape `{"id": ...}` to an **entity-reference** field is rejected at save with an `Invalid reference property` validation error.

> - Sending the entity shape `{"dataEntity": ..., "dataEntry": ...}` to a **collection-item** field passes the shape check and can save, then **silently resolves to nothing** on read: the collection resolve branch requires `value.id`, so the value falls through and `_resolved` comes back as an empty object — no error is raised at any point.
> - Persistence never repairs a shape: the save-time cleanup keeps whichever of `dataEntity`, `dataEntry`, and `id` you sent.

Naming constraints

- `normalizedName` (entity and field) must satisfy `NormalizedNameValidation` (regex pattern + min/max length) enforced via JSON schema on the model.
- The entity `normalizedName` has a **unique index**; duplicates are rejected on create/update (`EntityDuplicateException`, or a Mongo `11000` `UniqueValueException`).
- Entity `name` must be unique **within the owning module** (not globally) — enforced by `checkDuplicateNamesFromDatabase(..., [{module: {eq: module}}])`.
- Reserved words cannot be used as a `normalizedName` for the entity, its fields, or internal-entity fields. `checkRestrictedNormalizedName()` rejects: `switch`, `new`, the counters name, `default`, `class`, `break`, `private`, `public`, `protected`, `else`, `if`, `return`, `case`, `catch`, `throw`, `while`, `for`, `try` (`RestrictedNormalizedNameError`).

Because the `normalizedName` is the stable machine key that queries and reactions bind to, it is effectively permanent after creation — the builder generates it automatically and lets you tweak it before saving, but you can change the localized display name later without affecting the machine key. It is also used in packaging **import/export** operations to preserve the entity's representation across environments.

Mapped fields `Since 1.8`

A field with `mapping.enabled` mirrors its value from a **source field** on the entity referenced by `mapping.referenceFieldId`. Three rules keep a mapped field's own metadata in step with its source — resolved through the full mapping chain, so a field mapped transitively (C mirrors B, which itself mirrors A) is reached too:

- **Sensitivity inheritance.** A mapped field takes on its source field's sensitive flag — and its sensitive-field access grants — whenever the mapped entity is saved. The reverse direction now propagates immediately too: toggling a source field's sensitivity pushes the change down onto every field mapped from it right away, instead of waiting for the next time the mapped entity happens to be saved. See Sensitive Fields [Build Apps › Protect a field] for how a sensitive field is masked and unmasked at read time.
- `unique` **resyncs one-directionally.** When a source field stops being unique, a mapped field's stale `unique` flag is cleared and the owning entity's index is rebuilt so the physical unique index is actually dropped. A mapped field never *inherits* `unique` from its source, even when the source field newly becomes unique — only a stale `true` is ever cleared, never set.
- **Cascaded auto-first references now fill their mapped fields correctly.** A reference field set to auto-select its first value, whose own source is cascaded on another reference already set on the entry (for example, a reaction that creates an entry with only the cascade parent bound), previously resolved to nothing when computing values for fields mapped through it. Those fields — including a repeatable mapped field the payload never submitted a value for — now fill correctly.

Display and form settings

An entity's records are rendered from a form whose column layout is controlled by `displayOption` (`DataEntityDisplayOptionEnum`):

- `KeyValue` (default) — one field per row.
- `TwoColumns` / `ThreeColumns` / `FourColumns` — two, three, or four fields per row, with the remaining column space filled when a row carries an extra field.

Finer form-render behaviour (field grouping, ordering, section layout) lives in `formSettings` and is edited from the **Form Settings** panel of the entity's Structure tab in the builder.

Entity scripts

An entity can carry **entity scripts** — user-authored JavaScript lifecycle hooks attached to the entity in the designer and persisted in `scriptAttributes`. At runtime the client runs the stored code per **view-face** and then per **entity-face** through `EntityScriptExecutorService.run`, letting a builder react to entry lifecycle events without a server-side reaction.

System metadata fields (`_metaData`)

Beyond an entity's own `fields`, every entry carries a set of **system metadata fields** that OSOS stamps and maintains automatically. They are surfaced through a virtual `_metaData` block so they can be selected inside a view (via `?includeMetaDataFields=true`, see Read-model notes). The block is **locked/read-only** and virtual: it is built by `metaDataBlockField(withWorkflowMetaField)` and backed by an internal entity `_metaDataInternalEntity` (`META_DATA_BLOCK_NORMALIZED_NAME = '_metaData'`); its inner fields flatten to `_metaData.<id>`.

The registry lives in `osos-shared/src/metaFields.ts`, and `META_DATA_PATH_MAP` maps each virtual meta-field id to its actual stored path on the document:

Meta-field id	Stored path	Type
`createdBy`	`creationInfo.doneBy`	ref USER
`createdAt`	`creationInfo.doneAt`	datetime
`updatedBy`	`updateInfo.doneBy`	ref USER
`updatedAt`	`updateInfo.doneAt`	datetime
`entryId`	`id`	text
`entityVersion`	`entityVersion`	number
`invalidReason`	`invalidReason`	number enum (0 = OBSOLETE_ENTRY, 1 = INVALID_REQUIREMENT_ENTRY)

So a view that selects `_metaData.createdBy` actually reads `creationInfo.doneBy` — the registry is what lets the flattened `_metaData.<id>` names resolve to the inner `creationInfo` / `updateInfo` stamps described in the model table above.

Workflow metadata fields. When an entity has workflows, `metaDataBlockField` is called with `withWorkflowMetaField = true`, which adds a further set of selectable `_metaData` fields describing the entry's current workflow step (read from the entry's active current step under `_metaCurrentSteps.*`; history lives under `_metaWorkflowHistory.*` / `_metaAdhocSteps.*`):

Meta-field id	Meaning	Type
`currentStep`	Localized name of the current step	localized text

Meta-field id	Meaning	Type
`currentStepOwner`	Current step owner	ref USER
`currentStepOriginalOwner`	Original owner of the current step	ref USER, GROUP
`currentStepReceivedDate`	When the step was received	datetime
`currentMainStep`	Whether the current step is a main step	boolean
`currentAdhocStep`	Whether the current step is an ad-hoc step	boolean

`CURRENT_STEP_OWNER_META_DATA_FIELDS` = [`_metaData.currentStepOriginalOwner`, `_metaData.currentStepOwner`] identifies the owner-bearing meta fields used for step-ownership checks.

Entity types (core vs custom)

When you create an entity you choose an **Entity Type**. Entities you define yourself are usually `Custom`; the platform ships many other built-in types that you will see but rarely create by hand. The `type` field is a `DataEntityTypeEnum`: the value both categorises the entity and gates what operations are allowed. The **Entity Type is fixed at creation and cannot be changed afterward** — like the `normalizedName`, it is part of the entity's permanent identity. Notable values:

Name	Value	Meaning
`Custom`	0	A regular builder-defined entity.
`ADT`	1	Abstract/primitive data-type entity.
`Service`	2	Service entity.
`Request`	3	Request-type entity; the only implicit type that stays editable and supports service-permission updates.
`Contact`, `TimelinePost`, `CalendarEvent`, `Notification`, `Task`, `User`, `Group`, `OrganizationUnit`, `Configuration`, `Dashboard`, `Report`, `PostCategory`, `Comment`, `AllRequests`, `Person`, `Role`, `Assignment`, …	4–29	Built-in product entity kinds.
`Core`	30	Core platform entity.
`System`	40	System entity.
`VirtualProxy`	50	Virtual entity backed by a single primary source.
`JoinerProxy`	51	Virtual entity joining multiple sources via relations.
`VirtualAggregator`	52	Virtual entity aggregating multiple sources (with field mappings).
`Space`, `SpaceUsers`, `Delegation`, `WorkingHours`, `Interceptors`, `Templates`, `Validators`, `SystemObject`, `Audit`	60–90	Infrastructure/system entity kinds.

The Services Catalog and its related entity `Since 1.8`

A service entry in the Services Catalog carries a pointer to the (typically `Request`-type) entity that actually backs it. **Since 1.8**, archiving or restoring a Services Catalog entry now cascades to that related entity's own entries: archiving the catalog entry also archives its backing entity's active entries, and restoring it restores them — which in turn cascades onward to the mirrored request rows the same way archiving or restoring the related entity directly (from the Data Entities admin page) already did. Before this, archiving a service from the Services Catalog only flipped the state on the catalog row itself, leaving already-submitted requests for that service still visible in My Requests / Inbox.

This is a data-modeling note about the cascade only; the Services Catalog admin workflow itself is covered under Services & Requests [Using OSOS › Services & Submitting Requests], and the request-side effect under Requests & services (API and internals).

Custom entities

Created via `POST /v1/entities` with `type = Custom` (or another non-virtual, non-system type). Their entries are created by hand, and they are fully editable and deletable subject to permission voters.

Core / implicit entities

Entities with `isImplicit = true` are system-provisioned — built-in kinds such as Users, Groups, Organization Units, and Audit, which are often protected or partially read-only. Their editability and deletability are constrained by two service predicates:

```
isEntityEditDenied(e)   = e.isImplicit && !e.implicitOptions?.allowEdit && e.type !== Request
isEntityDeleteDenied(e) = e.isImplicit && e.type !== Request
```

So an implicit entity can only be edited if `implicitOptions.allowEdit` is set (or it is a `Request` entity), and can only be deleted if it is a `Request` entity. Violations raise `ImplicitEntityUpdateException` / `ImplicitEntityDeleteException`. Additionally, on update of an implicit entity the handler preserves the persisted definition and only accepts a **whitelisted** subset of incoming properties (`fields`, `internalEntities`, `displayOption`, `formSettings`, `workflows`, `indexes`, `internalCollections`, `storeRules`, `templates`) — everything else is kept from the stored entity.

The **Core module** (`CORE_MODULE_ID = 00000000-0000-0000-0000-000000000000`) is itself protected: creating an entity in it, or moving an entity into it, raises `ImplicitModuleException`.

Virtual entities

Virtual entities are **read-only**: their entries are computed from other source entities rather than entered by hand. There are three kinds:

- **Proxy** (`VirtualProxy`) — surfaces a single source entity so it can be reached and interacted with from another part or module, without duplicating its data.
- **Joiner** (`JoinerProxy`) — joins the primary source with its related entities through relations, presenting the primary entity's information together with related fields.
- **Aggregator** (`VirtualAggregator`) — computes aggregated values (count, sum, min/max, average) from a source entity grouped by one or more fields, so each entry is a read-only summarized row that is never edited manually.

Virtual entities are **not** created through `POST /v1/entities` — the create and update handlers explicitly reject `VirtualProxy` / `JoinerProxy` / `VirtualAggregator` with `DataNotSupportedException`. They have their own endpoints:

- `POST /v1/entities/virtual` — create.
- `PUT /v1/entities/virtual/{id}` — update.

`typeOptions` for a virtual entity accepts two shapes, handled in `createVirtual`:

1. **Flat / interactive** — `{sourceEntityNormalizedName, sourceQuery}`. This is wrapped into a single `VirtualProxyEntityOptions` with one `Primary` source.
2. **Persisted / packaged** — `{sources: [...]}`. Passed through untouched (used when a packaging import replays a previously-exported virtual entity, including aggregator sources with `fieldMappings`), because rebuilding it would strip aggregator-specific fields.

`VirtualProxyEntitySourceTypeEnum` distinguishes `Primary (0)` from `Additional (1)` sources. `findById` resolves source entities and returns type-specific detail DTOs: `DataEntityAggregatorDetailsDto` for aggregators, `DataEntityJoinerDetailsDto` for joiner/proxy (with resolved relation scopes). Aggregator responses also include `mappedSources`, computed by `getVirtualEntityMappedSources` — one entry per source-entity/view mapping. **Since 1.8**

There is no separate delete endpoint for virtual entities; `DELETE /v1/entities/{id}` (archive) applies.

Lifecycle

States

`DataEntityStateEnum`: `Draft (0)`, `Active (1)`, `Archived (2)`. A Draft entity is still being built; publish it to Active so end users can create entries in it. Every transition is appended to `stateHistory` with `changedAt`, `newState`, `changedBy`, and optional `delegatee`.

Create

`POST /v1/entities` → `CreateDataEntityHandler.handle`. The handler:

1. Rejects if the id already exists (`EntityDuplicateException`) or the type is virtual (`DataNotSupportedException`).
2. If `virtual.enabled`, resets `workflows` to `[]`.
3. Stamps `creationInfo`, `updateInfo`, a fresh `updateIdentifier`, and default `storeRules`.
4. Validation phase: entity validate, duplicate-normalizedName check, name validation & module-scoped duplicate-name check, implicit-field validation (unless `isSystemModification`), field checks (duplicate fields, repeatable settings, source-required, show-as options, per-field validation), timeline-contribution validation, workflow validation, and internal-entity validation.
5. Preparation phase: `defineWorkflowHappyPath`, `handleFieldsOptions`, `appendImplicitConfigs`, index-definition creation, and `calculateEntityMappingFieldSources`.
6. Syncs sensitive-field access rights and strips the transient `sensitiveAccess`.
7. Persists via `repository.create`, publishes `DataEntityCreatedEvent`, and emits a `CommandExecutedMessage` over the socket.

New entities start at `version = 0` (model default) and — barring an explicit `state` in the payload — take their state from the model. Permissions supplied in the create DTO are **not** stored on the entity; the controller strips them and routes them to the access-rights store (`EntitlementEnum.DataEntity`).

Worked example: create payload

The `POST /v1/entities` body is the `DataEntity` document (plus an optional `permissions` array), with the lifecycle fields excluded from the request schema. Below is a minimal-but-valid `fields` + `internalEntities` pair — one text field and one block field whose sub-fields live in a sibling internal entity; the entity-level properties (`name`, `normalizedName`, `type`, `icon`, `module`, …) are documented in the structure table. Every field must carry `dataType`, `options`, `renderOptions`, **and** `validations`, and `renderOptions` requires **both** `editComponent` and `viewComponent`, each a `{component, options}` descriptor whose `component` is a plain string. The owning `module` must be an existing, non-core module — list modules with `GET /v1/data-module` (through the gateway: `/api/v1/data-module`).

```
{
  "fields": [
    {
      "specificationVersion": 1,
      "name": {"en": "Title", "ar": "العنوان"},
      "normalizedName": "title",
      "dataType": 0,
      "options": {
        "hideForEmptyValues": false,
        "showInTableView": true,
        "showAsTitle": false,
        "showAsSubTitle": false,
        "showAsCaption": false,
        "showAsMainDate": false,
        "showAsMainNumber": false,
        "showAsEntryPicture": false
      },
      "renderOptions": {
        "editComponent": {"component": "TextInputComponent", "options": {}},
        "viewComponent": {"component": "ViewTextComponent", "options": {}}
      },
      "validations": {
        "required": {"enabled": false, "expression": null},
        "min": {"enabled": false, "value": null, "valueSourceType": 0},
        "max": {"enabled": false, "value": null, "valueSourceType": 0}
      }
    },
    {
      "specificationVersion": 1,
      "name": {"en": "Basic Info", "ar": "المعلومات الأساسية"},
      "normalizedName": "basic_info",
      "dataType": 9,
      "block": {"internalEntity": "<uuid-A>"},
      "options": {
        "hideForEmptyValues": false,
        "showInTableView": false,
        "showAsTitle": false,
        "showAsSubTitle": false,
        "showAsCaption": false,
        "showAsMainDate": false,
        "showAsMainNumber": false,
        "showAsEntryPicture": false
      },
      "renderOptions": {
        "editComponent": {"component": "BlockComponent", "options": {}},
        "viewComponent": {"component": "BlockComponent", "options": {}}
      },
      "validations": {
        "required": {"enabled": false, "expression": null},
        "min": {"enabled": false, "value": null, "valueSourceType": 0},
        "max": {"enabled": false, "value": null, "valueSourceType": 0}
      }
    }
  ],
  "internalEntities": [
    {
      "id": "<uuid-A>",
      "validation": "",
      "fields": [
        {
```

```
          "specificationVersion": 1,
          "name": {"en": "Username", "ar": "اسم المستخدم"},
          "normalizedName": "username",
          "dataType": 0,
          "options": {
            "hideForEmptyValues": false,
            "showInTableView": true,
            "showAsTitle": false,
            "showAsSubTitle": false,
            "showAsCaption": false,
            "showAsMainDate": false,
            "showAsMainNumber": false,
            "showAsEntryPicture": false
          },
          "renderOptions": {
            "editComponent": {"component": "TextInputComponent", "options": {}},
            "viewComponent": {"component": "ViewTextComponent", "options": {}}
          },
          "validations": {
            "required": {"enabled": false, "expression": null},
            "min": {"enabled": false, "value": null, "valueSourceType": 0},
            "max": {"enabled": false, "value": null, "valueSourceType": 0}
          }
        }
      ]
    }
  ]
}
```

Notes on the shape:

- **`options`** — the eight booleans shown are the `required: true` members of `FieldOptions`; the rest are optional or defaulted. Properties that are required *with* a default (`specificationVersion`, several of these booleans) are auto-filled by model instantiation, but real payloads — including the seeded core entities — include them explicitly.
- **`validations`** — requires `required` (`{enabled, expression?}`) and `min` / `max` (`{enabled, value?, valueSourceType}`, with `valueSourceType` defaulting to `0`).
- **Component descriptors** are plain strings. The seeded entities use e.g. `TextInputComponent`, `LocalizedTextInputComponent`, `BooleanComponent`, `CategoryComponent`, `RelationComponent`, and `BlockComponent` on the edit side, and `ViewTextComponent`, `ViewLocalizedTextInputComponent`, `ViewCategoryComponent`, `ViewRelationComponent` on the view side.
- **Field `id`** is optional (server default: a new v4 UUID). `internalEntities[].id` also defaults to a UUID — supply it explicitly so `block.internalEntity` can point at it; seeded internal entities use `validation: ""`.
- **Category fields** use `dataType: 12` with `"CategoryComponent"` and put their selectable options at `renderOptions.editComponent.options.options`:

```
"renderOptions": {
  "editComponent": {
    "component": "CategoryComponent",
    "options": {
      "options": [
        {"id": "STANDALONE", "label": {"en": "Standalone", "ar": "مستقل"}, "color": "#FF0000"}
      ]
    }
  },
  "viewComponent": {"component": "ViewCategoryComponent", "options": {}}
}
```

Creating an entity requires a license that includes administration plus edit access on the (non-core) module, and runs asynchronously — a `2xx` means the `CreateDataEntityCommand` was enqueued, not that the entity exists yet (see the architecture note at the top of this page).

Update

`PATCH /v1/entities/{id}` → `UpdateDataEntityHandler.handle`. Preconditions checked in the **controller** before enqueuing:

- `isEntityEditDenied` → `ImplicitEntityUpdateException`.
- Client `updateIdentifier` must equal the persisted one, else `UpdateIdentifierConflictException` (optimistic concurrency, see below).
- The entity must not be locked by another user (`LockedObjectException`).

The handler then re-validates (duplicate normalizedName, archived-state guard, implicit whitelist merge, name checks, field/unique/implicit-field validation, workflow validation), recomputes options/indexes/mappings, recalculates deleted fields and entity version, rotates `updateIdentifier`, and performs an **atomic conditional write** (see below). It publishes `DataEntityUpdatedEvent` (with re-index and change flags and a diff) and, if the version was bumped, `DataEntityVersionChangedEvent`.

From the builder's **Edit Entity** flow, the structure you can change on an existing entity includes its display name and icon, its singular/plural entry names, description, fields, reactions, workflows, validation, and timeline; `normalizedName` and `type` stay fixed (and `module` too, on a normal update).

Additional update rules enforced by the handler:

- **Archived entities** cannot be updated unless `skipArchiveCheck` is set (`EntityArchiveException`).
- `normalizedName` is never changed on update — it is forced back to the stored value.
- `module` cannot be changed on a normal update; it is forced back to the stored value unless `allowModuleUpdate` is set (used by the move-module endpoint).
- **Main-title constraints** (`validateMainTitleNotNestedReference`): the `showAsTitle` field may not be `repeatable` / `multiValue` (`MainTitleCannotBeRepeatableOrMultipleException`), and if it is a reference (`Entity`) field it may not create a *nested* reference title — neither the source entity's main title nor any entity referencing this one may itself use a reference field as its title (`NestedReferenceMainTitleSourceIsReference` / `NestedReferenceMainTitleReferencedByOtherEntity`).

Optimistic concurrency (updateIdentifier)

Every write rotates `updateIdentifier` to a new `v4()`. The controller's equality check happens **before** the command is enqueued, leaving a check-then-act window during which a concurrent writer (another client, or one of the many event

subscribers that write back to the entity) can bump the token. To close the lost-update window, the update handler writes with a **conditional** `updateAll` matching on the originally-read identifier:

```
updateAll(dataEntity, {id, updateIdentifier: oldEntity.updateIdentifier})
```

If Mongo updates 0 documents, the write is aborted and `UpdateIdentifierConflictException` is surfaced to the caller. **Since 1.8**

`Since 1.10` — an **entry** now carries a token of its own too. It is a separate, independently rotated `updateIdentifier` on the entry document, not the one on the entity definition described here, and sending it back is opt-in per write request rather than required as it is above. Which endpoints accept it, and the `409 UPDATE_CONFLICT` it produces, are on Calling the HTTP API.

The retry loop, and re-projection onto a fresh read `Since 1.10`

A caller can opt an entry write into automatic conflict retry instead of failing outright on `409 UPDATE_CONFLICT`. Alongside the entry's `updateIdentifier` above, the write payload accepts two further optional fields: `expectedUpdateIdentifier` and `changedPaths` (the specific field paths this write intends to change). When both are supplied:

- On a conflict, the entry is **re-read fresh**, and only the caller's `changedPaths` are re-applied onto that fresh value — never the caller's original snapshot, which could otherwise silently overwrite a concurrent writer's change to a field the caller never touched.
- The write is retried against the freshly-read token, up to **3 attempts total** (1 initial + 2 retries), with **250ms then 500ms** between them.
- Each attempt re-runs the full read → validate → diff → persist sequence, not just the final write, so anything watching for a change (links, aggregators, map fields) is always notified from the current entry, not a stale one.

`changedPaths` cannot be safely reconstructed from a diff after a conflict — a stale snapshot compared against a fresh read conflates the caller's own intent with a concurrent writer's change — so the caller has to declare its intended paths up front. Both fields are optional and purely additive: a write that omits them behaves exactly as it did before this change.

Reaction-driven writes gain the same protection, behind a flag `Since 1.10`

The **Update Data Entry** action in Reactions & Automation [Build Apps › Reactions & automation] can send its writes through this same retry guard, but only when every one of the following holds for that specific write:

- The `REACTION_ENTRY_OPTIMISTIC_LOCKING_ENABLED` environment flag is on for the deployment.
- The target entry actually carries an `updateIdentifier` — a token-less entry is never guarded, even with the flag on, rather than risk sending an unconditional write while an operator believes the guard is active.
- At least one field path actually changed as a result of the write.

This is deliberately narrow: it protects the Update Data Entry reaction action specifically, not reaction writes in general, and it does not change how a human editor's own edit is guarded. With the flag off, or for a write that doesn't meet the conditions above, behavior is identical to before this change.

Move to a different module

`PATCH /v1/entities/{id}/module` with `{module, updateIdentifier}` (and optional `?force=true`). Requires an `updateIdentifier` match and no lock, and refuses the core module. It sends an `UpdateDataEntityCommand` with `allowModuleUpdate = true` and `force` as `skipArchiveCheck`. When the module actually changes, the handler cascades the new module id to the entity's child **views** and **collections**. **Since 1.8** (module cascade)

Who may call it changed in 1.10: the module voter no longer decides; the route is reserved for system administrators (see below).

Admin move, bulk move and what a move repairs `Since 1.10`

Who may move. `PATCH /v1/entities/{id}/module` is now **system-admin only**: the `SystemAdmin` role (the license must include administration). Trusted service calls (a package import) carry no user permissions and are let through; a request bound to an end-user view is refused. Expect these refusals:

Situation	Response
Target is the core module	`400` (`ImplicitModuleException`)
Target unknown or not visible	`404`
Target module is being permanently deleted	`409 MODULE_BEING_DELETED`
Entity is already in that module	`400 ENTITY_ALREADY_IN_MODULE` (a trusted service caller gets a `200` no-op, so a re-run import stays idempotent)

Virtual entities (proxy, joiner, aggregator) can be moved too. Their move changes nothing but the module; before 1.10 it answered `500 "The data not supported"` .

Bulk move. `POST /v1/entities/move` with `{entityIds[], module}` moves **1 to 50** distinct entities into one module, under the same authorisation. The target is checked once — the core module, an unknown module or one being deleted refuses the whole request and moves nothing. The entities are then moved **one at a time, in request order** (the old module's page clean-up rewrites whole pages, so parallel moves would lose removals). The response is always `200` :

```
{
  "module": "<target module id>",
  "moved":  [{ "id": "<entity id>", "repairIncomplete": false }],
  "failed": [{ "id": "<entity id>", "code": "ENTITY_ARCHIVE", "message": { "en": "…", "ar": "…" } }]
}
```

`moved` and `failed` list the outcomes in request order. `repairIncomplete` is `true` only when the move itself landed but a follow-up repair step failed (the failure is logged). The bulk path skips the `updateIdentifier` check (the writer's conditional write still catches a concurrent edit) and never forces, so an archived entity always fails with `ENTITY_ARCHIVE` . Codes without their own message, such as `LOCKED_OBJECT` and `UPDATE_CONFLICT` , fall back to the generic `ENTITY_MOVE_FAILED` text.

What a move rewrites. A move is more than a field change on the entity. The platform re-parents and repairs everything that points at the old module:

What	Effect
Views and collections	Follow the entity into the new module.
Sensitive-field scopes	Re-keyed to the new module in place; module wildcards are untouched.
Sidebar system links	Re-parented; duplicates are archived.
Counter widgets, timeline posts	Re-stamped with the new module.
View access objects, permission and profile epochs	Refreshed.

What	Effect
Old module's pages and dashboards	**Lose** whatever is linked to the moved entity (its id and the ids of its views, charts, queries and reports): matching list items and widgets are removed from that module's pages, and its summaries are pulled from the module dashboard. System pages, other modules' pages and the entity's own dashboard are untouched. A page editor that is open at the time gets an update conflict.
Reactions	`SendNotification` actions are re-stamped with the new module.
Audit	`DataEntityUpdatedEvent` carries `EntityChanges.MODULE_CHANGED` and `previousModule`.

Design consequence: moving an entity out of a module **removes** its widgets and summaries from that module's pages and dashboard — it does not carry them along. Rebuild them in the target module afterwards. The admin UI says so in its warning.

In the admin UI. The single move is on the entity card menu and the entity details page (System Entities and a module's Entities tab), with a target select that can also add a new module. The bulk move is in a module page's header menu: it switches the list into selection mode (the selection survives paging) and disables **Move** above 50 entities.

Update service permissions (Request entities)

`PATCH /v1/entities/{id}/meta/permissions` updates `meta.audience` / `meta.observers` on a `Request`-type entity only; any other type raises `EntityNotOfTypeRequestException`. Subject to the implicit-edit and lock checks.

Archive (delete)

`DELETE /v1/entities/{id}` → `ArchiveDataEntityCommand` → `ArchiveDataEntityHandler`. This route never hard-deletes: the handler sets `state = Archived`, stamps `archiveInfo`, appends to `stateHistory`, persists, and publishes `DataEntityArchivedEvent` (which downstream archives the entity's entries). Delete is blocked for implicit non-`Request` entities (`isEntityDeleteDenied` → `ImplicitEntityDeleteException`) and gated further by `Helper.checkEntityForArchiving`. `Since 1.10` a separate, admin-only route removes an entity for good — see Permanent delete.

Archiving an entity that is already archived answers `422 ENTITY_ARCHIVE` `Since 1.10` (message *The data entity has been archived*). It used to surface as a `500` and reach error tracking; the code and message are unchanged, only the status is now a client error. The same refusal applies to **updating or moving** an archived entity, so those also answer `422`, and a bulk module move no longer logs an archived entity as a server failure. Do not confuse it with `ENTITY_ARCHIVED` (past tense, also `422`), which the permanent-delete route below raises for an entity archived without a purge marker.

Permanent delete (purge) `Since 1.10`

`DELETE /v1/entities/{id}/permanent` irreversibly removes an entity **and everything it owns**. It applies to **Custom, Request and virtual** (proxy, joiner, aggregator) entities, and it is system-admin only (the `SystemAdmin` role; the license must include administration). The ordinary `DELETE /v1/entities/{id}` above is unchanged and still only archives.

Unlink first. The purge refuses to leave dangling references. Before it removes anything it asks the dependency service what *outside* the entity still points at it, and refuses while that list is not empty. The checks, in order:

Check	Refusal
Not a purgeable type, implicit, in the core module, or a restricted / configuration / code-owned entity	`400 ENTITY_PURGE_NOT_ALLOWED`
Archived through the ordinary route (no purge marker)	`422 ENTITY_ARCHIVED`

Check	Refusal
Installed from a package	`403`
Locked by another user	`409 LOCKED_OBJECT`
Something outside the entity still references it	`409 ENTITY_HAS_DEPENDENTS` (`details.count`)
The dependency check cannot answer	`503 DEPENDENCY_CHECK_UNAVAILABLE` (not retried)

In the admin UI the delete dialog lists those references grouped by category, each with an **Open** link, and keeps **Delete** disabled until none remain; with nothing listed, an "I understand…" checkbox enables it.

What is removed. The entity is first archived and marked for purge (without an archive event, so its archive reactions never fire). Then its functions, views, pages, dashboard, reports, buttons, counter widgets, definitions, grants, entries collection and every row keyed by it go — for a Request entity also its catalog row, mirrors, drafts, service system objects and delegation exceptions. Just before the entity document itself is deleted, `DataEntityDeletedEvent` is published (a snapshot plus the ids deleted) and the audit trail records `DATA_ENTITY_DELETED`; downstream services each remove their own rows.

Resumable, not transactional. Every step is idempotent. If one fails, the entity stays archived and marked, and **calling Delete again finishes the job**. Success is `200 {success: true, result: null}` (the OpenAPI document declares `204`).

To remove a whole module with its entities, use the module's own permanent delete — see Modules.

Entity cap and the `isImplicit` flag `Since 1.10`

A tenant can carry a limit on the number of entities. When it is reached, creating an entity or a virtual entity is refused with `403 ENTITY_CAP_EXCEEDED` and `details: {cap, current}` (message *Creating this entity would exceed this application's limit of N entities (it has M).*). What to expect as a builder or integrator:

- **Which limit.** The smaller of two, either of which may be absent: the plan's `maxEntities` `Since 1.10` (see Plan limits) and the per-tenant entity cap an operator can set (the tenant's registry record, or a single-tenant install setting). With neither, nothing is ever capped.
- **What counts.** Every entity that is **not implicit** and **not in the core module**, whatever its type, across every space of the tenant. Virtual entities count. Archived entities are not excluded. A creation in the core module is never checked. The check runs before the entity is persisted, so concurrent creations can overshoot the cap by at most their own number.
- A **package import** counts the entities it would create up front and refuses the whole import before writing any module or entity — design packages with the target tenant's limit in mind.
- **Entries have a limit too.** The plan's `maxEntries` refuses new entries with `403 ENTRY_CAP_EXCEEDED` once usage would pass 110% of it, and a package install that would pass it is refused before it starts (see The entries limit [Administration › Your license & plan limits] and Installing). Entries of core-module entities do not count, and archived entries weigh a quarter.

Because implicit entities are exempt, **`isImplicit` is ignored for user callers**: `POST /v1/entities` stores `isImplicit: false` with no `implicitOptions`, and `PATCH /v1/entities/{id}` keeps the stored values whatever the body says. Package imports, bus commands and seed migrations are unchanged — only they can create implicit entities.

Versioning and breaking changes

Entities carry a numeric `version` (default `0`). This is **not** a per-save revision counter — it is bumped only when a change would break existing data, and only when the entity already has entries. The logic lives in `DataEntityVersionService` and is driven from `UpdateDataEntityHandler.handleEntityVersion`.

When the version bumps

```
if (entriesAtAnyVersion == 0 && currentVersion == 0) → no versioning work
newVersion = (entriesAtAnyVersion > 0 && breakingChanges.length) ? current+1 : current
```

So the version increments **only** when (a) the entity has at least one entry — at **any** version, including entries already obsolete from an earlier unresolved breaking change — and (b) the change set contains at least one breaking change. Non-breaking edits, and edits to an entity that has no entries yet, keep the same version.

What counts as a breaking change

`calculatingBreakingChanges` diffs the old vs. new flattened field sets (matched by `id` + `fullNormalizedName`) plus workflow changes, and records reasons from `BreakingChangesReasons`:

Reason (enum)	Triggered when
`AddedRequiredField`	A brand-new field is required.
`ChangeFieldToRequired`	An existing field becomes required.
`FieldTypeChanged`	`dataType` differs from the old field.
`FieldEncryptionChanged`	The field's **Encrypt** option is turned on or off `Since 1.10`. Added even when another reason already applies to the same field, so the encryption transition is always stamped. See Toggling Encrypt.
`ChangeFieldToRepeated`	Field becomes `repeatable`.
`ChangeFieldToNonRepeatable`	Field stops being `repeatable`.
`ChangeFieldToMultivalue`	Field becomes `multiValue`.
`ChangeFieldToSingleValue`	Field stops being `multiValue`.
`EnableDenyCircularRelationOnField`	`sourceOptions.denyCircularRelation` is turned on.
`FieldRestored`	A previously-deleted field is re-added with a *different* type.
`EntityWorkflowChanged`	Any workflow step added/removed/reordered/changed, or a workflow deleted.

Not currently emitted (commented out in source): `AddedUniqueField`, `ChangeFieldToUnique`. `ReferenceSourceChanged` exists in the enum and has a detector (`isSourceChangedOnReferenceField`) but is **not** wired into the breaking set — so changing a reference field's source does not, by itself, bump the version today. **Deleting a field is not a breaking reason either**: the diff walks only the fields present in the new definition; deleted fields are recorded separately in the version's `deletedFields` (see Version records). A re-added field whose breaking-relevant flags (`dataType`, `repeatable`, `required`) all match its deleted predecessor is treated as a **restore** and is *not* breaking.

> Rule of thumb: additive changes (a new optional field) are safe; type changes, new required fields, repeatable/multi-value changes, Encrypt changes and workflow changes are breaking and call for an upgrade pass.

Impact on existing entries

When a breaking change bumps a **published** entity's version, existing entries stay at their old version and are shown as **obsolete** until they are upgraded to the latest version. If the entity has a workflow, that workflow stops: the manager must decide what to do with the old values — leave them or update them — and the workflow reverts to its initial step. To migrate entries, use **Upgrade entries** (or **Bulk upgrade**) to move them to the new version, and **Bulk update** to mass-set field values across entries.

Preview before saving

`PATCH /v1/entities/{id}/breaking-changes` runs the same diff against a payload **without** persisting — this backs the builder's **Preview breaking changes** action, letting you see the impact on existing entries before publishing. The controller merges the partial payload onto the stored entity (dropping `undefined` keys so omitted fields are not read as deletions), pins id/version, and calls `previewBreakingChanges`, returning a `BreakingChangesResult`:

```
{ version, fields: DataEntityBreakingField[], hasBreakingChanges: boolean, workflowChanges?: [] }
```

Version records

Each version is stored as a `DataEntityVersion` document (collection `DATA_ENTITY_VERSION_NAME`) with: `id`, `dataEntityId`, `version`, `createdAt`, `changes` (`DataEntityVersionChanges[]` — each `{changeType, isBreaking, field}`), a `diff` (from `just-diff` over `fields` / `internalEntities` / `workflows`, kept as a fallback), and `deletedFields`. `upsertDataEntityVersion` updates the current version record in place (pruning breaking changes that no longer apply via `isFieldStillBreaking`, adding new non-duplicate ones) or creates a new one. Deleted fields are recorded separately via `addDeletedFieldsToEntityVersion`.

Reading version changes (API)

- `GET /v1/entities/{entity}/breaking-changes/{version}` — diff between the given old version and the entity's current version (`calculateDiffBetweenChanges`).
- `GET /v2/entities/{entity}/breaking-changes?versions=...` — batch several versions at once; changes already reported by a higher version are filtered out of lower ones (dedup by `field.normalizedName`), and per-version errors are returned inline rather than failing the whole request. **Since 1.4**

API reference

All routes are under the `data-service` and require the OSOS OAuth2 authentication strategy plus the entity authorization voter. Roles noted where the handler declares them.

Method & path	Purpose	Notes
`POST /v1/entities`	Create a (non-virtual) entity	Module voter. Rejects core module & virtual types. Permissions routed to access-rights.
`POST /v1/entities/virtual`	Create a virtual entity	Module voter. Flat or `sources`-shaped `typeOptions`.
`GET /v1/entities/count`	Count entities (access-filtered)	Roles: `SystemViewer`, `Restricted`.
`GET /v1/entities`	List entities as DTOs	Roles: `SystemViewer`, `Restricted`. Adds `entriesCount`, `moduleDetails`, `metaFields`.
`GET /v1/entities/all`	List raw entities	`?includeImplicit=` to include implicit entities.
`GET /v1/entities/all/count`	Count raw entities	`?includeImplicit=`.

Method & path	Purpose	Notes
`GET /v1/entities/{id}`	Get one entity (DTO)	Roles: `SystemViewer`, `Restricted` — **definition only, never entry data**. `?includeMainTitleFields=` and `?includeMetaDataFields=` (adds the locked `_metaData` block). Resolves virtual `typeDetails` and per-request sensitive-field access.
`PATCH /v1/entities/{id}`	Update entity	Requires matching `updateIdentifier`, no lock, not implicit-denied.
`PATCH /v1/entities/{id}/breaking-changes`	Preview breaking changes	No persistence.
`PATCH /v1/entities/{id}/module`	Move to another module	System admin (license must include administration) `Since 1.10` (was the module voter). `?force=`. See Admin move.
`POST /v1/entities/move`	Bulk move 1–50 entities	`Since 1.10`. Same authorisation; per-entity `moved` / `failed` outcome.
`PATCH /v1/entities/{id}/meta/permissions`	Update service audience/observers	`Request` type only.
`PUT /v1/entities/virtual/{id}`	Update a virtual entity	Requires matching `updateIdentifier`, no lock.
`DELETE /v1/entities/{id}`	Archive (soft delete)	Blocked for implicit non-`Request` entities.
`DELETE /v1/entities/{id}/permanent`	Permanent delete (purge)	`Since 1.10`. System admin (license must include administration); refused while anything outside the entity references it. See Permanent delete.
`GET /v1/entities/{id}/index-paths`	Fields that a custom index may use	`Since 1.10`. `[]` for entities that cannot carry custom indexes. See Entity indexes.
`POST /v1/entities/{id}/indexes/{indexId}/retry`	Rebuild a failed custom index	`Since 1.10`. Entity edit access (license must include administration).
`GET` / `POST /v1/entities/{entity}/tag-groups`, `PATCH` / `DELETE …/tag-groups/{id}`	Manage tag groups	`Since 1.10`. See Tag groups.
`GET /v3/entities/{entity}/entries/export`	Quick export of the entity's entries	`Since 1.10`. `format=xlsx` or `csv`. See Quick export.
`GET /v1/entities/{entity}/breaking-changes/{version}`	Diff old→current version	
`GET /v2/entities/{entity}/breaking-changes`	Batch version diffs	`?versions=` array. **Since 1.4**

Read-model notes

- `GET /v1/entities/{id}` deliberately returns **only the entity definition** (fields, types, module, meta) — never entry data — so viewers and Restricted-with-grants users can render reference-field pickers for entities outside their grants. Entity ids are space-scoped UUIDs, so this is not an enumeration vector.
- Each returned DTO carries `sensitiveMaskedFields`, computed **per request** for the current user (never cached, never persisted). **Since 1.8**

- `?includeMetaDataFields=true` appends a locked virtual `_metaData` block (and its backing internal entity) so entry metadata fields can be selected inside a view. **Since 1.8**

Referenced entities on the entity DTO `Since 1.10`

To render reference fields a client used to call `GET /v1/entities/{id}` once per referenced entity. The DTO now carries what it needs in a **response-only** `referencedEntities` array:

```
"referencedEntities": [
  {
    "id": "<entity id>",
    "moduleId": "<module id>",
    "isStatic": false,
    "mainTitleField": {
      "id": "…", "normalizedName": "name", "fullNormalizedName": "name",
      "dataType": "…", "renderOptions": { }
    }
  }
]
```

- **Where it is returned:** `GET /v1/entities/{id}`, `GET /v2/entities/{entity}/entries/{id}` and `GET /v2/entities/{entity}/entries/draft/{id}` — **not** the entity list routes.
- **What it covers:** every reference field of the entity, including block sub-fields, display-only fields and meta fields (both `sources[].entity` and the legacy comma-separated `source`), de-duplicated in first-seen order. A referenced entity that has been deleted is simply left out, so keep a fallback fetch.
- **Static targets.** References to `DataModule`, `DataEntity`, `DataView`, `DataViews`, `Report`, `GenericView` and `System` are flagged `isStatic: true` and described under the core module with a localized title. Entry values that point at one of them now come back with a real `_resolved` (title, module and entity ids) on entry reads that resolve relations and on tree reads. A reader of the host entry sees the names of the modules, entities and views it points at — keep that in mind before storing such references on widely readable entities.

Nothing is persisted and no migration is involved.

Deep-linking to an entity designer tab `Since 1.8`

The entity designer's active tab is reflected in the page URL as a named `?tab=<slug>` query parameter, so a tab is an addressable place rather than transient UI state. In practice this means you can:

- **Bookmark or share a tab.** Copy the URL while the *Queries* tab is open and it reopens on *Queries* for whoever follows the link — useful when handing a colleague a specific part of an entity to look at.
- **Refresh without losing your place.** Reloading the designer keeps you on the tab you were working in.
- **Step back through the tabs you visited.** Opening a tab adds a browser history entry, so **Back** and **Forward** walk through the tabs you opened instead of jumping straight out of the entity.
- **Link into a tab from your own runbooks or tooling** by appending `?tab=<slug>` to the entity's designer URL.

Tab slugs

The slugs below are the complete set, listed in the order the tabs appear in the designer:

Tab	`?tab=` slug
Entries	`entries`
Queries	`queries`

Tab	`?tab=` slug
Views	`views`
Reactions	`reactions`
Announcements	`announcements`
Buttons	`buttons`
Lists	`lists`
Dashboard	`dashboard`
Reports	`reports`
Relations	`relations`
Templates	`templates`
Stores	`stores`
Information	`information`
Lock history	`lock-history`
Functions	`functions`
Permissions	`permissions`
Scripts	`scripts`

Fallbacks. A designer URL with no `?tab=` parameter opens on **Entries**. A slug that is not in the table above is **ignored** — the designer simply opens on the tab it last remembered for that entity, with no error. Note that older numeric deep links (`?tab=1`, `?tab=3`, …) are no longer recognised; replace any you have kept with the named slug.

Quick edit and entity-level validators `Since 1.8`

Inline (quick) edit changes one field on an entry and is applied as a **patch** rather than a full update, so its validation used to be narrower than the entry form's. A patch was blocked only when a validation error named a field the patch actually touched, or when a configured Standalone/Value **function** validator rejected the change.

That left a gap. An **entity-level custom validator** — the business and uniqueness rules an entity enforces across its fields — raises an error that is not tied to any single field, and is not a function validator. Those errors were silently dropped, so a quick edit could save even though the rule had failed. The full create and update paths never had this gap.

Since 1.8 any error raised by an entity-level custom validator **blocks the quick edit**, and the validator's own message is returned to the caller instead of being flattened into the generic list of field-validation errors — so the user sees the rule's real explanation rather than a vague "invalid entry".

Example. An organizational-unit entity enforces "only one manager per org unit". Quick-editing a member's role to *Manager* on a unit that already has one used to save quietly: the rule's error carried no field name, and the patch had touched only the role field. It now blocks the save and surfaces the rule's own "multiple manager role not allowed" message.

Entity indexes (Indexes tab) `Since 1.10`

The entity builder (**Edit Entity**) has an **Indexes** tab, next to Workflows, Timeline and RTC, where an administrator defines extra MongoDB indexes for the entity's entries — typically to speed up a filter or sort that a large entity uses all the time, or to enforce uniqueness across a combination of fields. The tab is offered for the same entities as the

Timeline and RTC tabs, so **not** for core-module, `Request` or virtual (proxy, joiner, aggregator) entities, and it is read-only on an entity installed from a package.

Defining an index. Give it a name, an ordered list of fields with a direction for each (drag or use the keyboard to reorder), optionally **Unique**, and an optional partial filter written as MongoDB JSON. The dialog runs the same checks as the server, so **Save** stays disabled on an invalid draft. The server's rules (`422 ENTITY_INDEX_*` on violation):

Rule	Limit
Custom indexes per entity	at most **10**
Fields per index	**1 to 8** distinct fields, from the pickable paths (`GET /v1/entities/{id}/index-paths`): indexable `value.*` fields plus `state` , `creationInfo.doneAt` , `creationInfo.doneBy.dataEntry` , `updateInfo.doneAt` , `updateInfo.doneBy.dataEntry`
Names and key sets	unique, also against system and migration-defined indexes
Partial filter operators	`$eq` , `$exists: true` , `$gt` , `$gte` , `$lt` , `$lte` , `$type` ; no embedded-document operands
Advanced MongoDB options	not accepted
Unique index	**must carry a partial filter** that excludes entries where the field is empty (`ENTITY_INDEX_UNIQUE_REQUIRES_FILTER`)

The last rule matters because MongoDB indexes a missing value as `null` , so a plain unique index would reject the second entry that leaves the field empty. For example, a unique index on `value.code` needs a filter such as:

```
{ "value.code": { "$type": "string" } }
```

Existing indexes that you do not change are grandfathered, even if they would fail a rule today.

Saving. Index edits are staged in the form and saved with the entity (`PATCH` / `PUT`). A PATCH that omits `indexes` keeps the stored ones, so other integrations that update an entity do not wipe them. Removing a field also removes the custom indexes built on it.

Status and retry. Indexes are built asynchronously by the indexing service, not during the save. The server keeps an `indexStatus[]` per custom index — `building` right after a save, then `ready` or `failed` with MongoDB's error. The tab refreshes while anything is building, shows the error on a failed index (typically duplicate data under a unique index; a field's own **Unique** option is switched back off when its unique build fails) and offers **Retry**, which calls `POST /v1/entities/{id}/indexes/{indexId}/retry` . An edited index that has not been saved shows *Not saved yet*. Custom indexes are named `IXU_<entity>_<12 hex>` ; the entity's system indexes are listed but locked.

Views built on this entity

A view is not modelled on this page — the view builder is documented in Data Views [Build Apps › Data Views & Faces]. Four recent changes belong here anyway, because each of them is about how a view reacts to the *entity's* fields and settings, and each was previously discovered the hard way.

System buttons, per-button order, and the API button

A view's action buttons live in `systemButtons` , an array in which each entry names the button and carries its own `order` . That per-button `order` **replaces** `systemButtonsOrder` , the older array-of-ids property, which is now **deprecated**: it is still read for backward compatibility — a view that still carries one has it honoured, and its sequence

decides the final order — but new configuration should express order per button. When a view carries no legacy order at all, the buttons follow their canonical sequence and the interface applies its own default.

The implicit buttons a view can expose are **View**, **Edit**, **Archive**, **Create**, **Duplicate** and **API**, plus one per document template on the entity. Which of them are allowed follows from the view's own configuration rather than being chosen from a free list — *Create*, *Edit* and *Archive* from the operations the view permits, *View* from its entries configuration, *Duplicate* from its duplicate-entry setting. Buttons the configuration no longer permits are reconciled away when the view is saved, rather than blocking the save.

`Since 1.10` — **the API button appears only when that view's Swagger document is turned on.** It is the button that opens the view's own per-view OpenAPI document; turning the setting off removes the button again, leaving the order of the others intact.

A view is saveable again after one of its fields is deleted

Deleting a field from an entity used to make every view still referencing it **unsaveable** — including a permissions-only change, which offers no field picker with which to clear the dangling reference. A view now prunes those stale references silently and saves, under three rules that keep the pruning from hiding real breakage:

- **Existing views only.** A reference is pruned only if the *stored* view already carried it. A field path a client has just invented still reaches validation and is still rejected, and creating a view never prunes at all.
- **Never down to empty.** If *every* entry in a collection is stale, the feature is broken rather than merely drifted, so validation is allowed to fire and name what an administrator has to fix. Grouping is the single exception, because it has no enabled/disabled switch of its own — an empty level list is itself the valid "no grouping" state.
- **Collections only.** Settings that name a *single* required field are load-bearing and are never dropped: a kanban's category field, a tree's main title, and their kind. Only list-shaped settings are pruned — the view's field list, filter-first fields, grouping levels, a table face's columns and a calendar face's date mappings.

Meta-data (virtual) fields are never mistaken for drift.

Restricted link permissions and navigation order

Two fixes to how a **link** — the navigation or restricted-access entry pointing at a view — reacts to that view being saved:

- **An upgrade no longer empties a Restricted link's audience.** Upgrading a tenant used to clear the users and groups on every active Restricted link, so restricted pages and views silently lost their audience and became unreachable to exactly the people who should have seen them. The link handlers now read the live grants rather than trusting the event that triggered the rebuild, and a guarded repair restores audiences that the old behaviour emptied. Archived Restricted links were never affected.
- **A navigation link keeps its order across a save.** Saving the owning view used to reset the link's position, collapsing the navigation order onto its alphabetical tiebreaker — and an upgrade, which re-saves every view, flattened the whole navigation in one pass. A link now inherits the view's own display order instead. This one **self-heals on the next save** of each affected view; there is nothing to run.

Document templates over a view — current limitation

Exporting a document template **from a view** over an entity containing a block field used to produce a file with **zero records**, while the same template exported from the administration area worked. That is fixed.

What is not fixed, and is worth designing around: a document template that references a field the triggering view does **not display** still renders that field **blank**. The view's field list drives both which relations are resolved and which

values are read, so widening one widens the other — which is what caused the empty exports. Until that is separated, make sure a view used to trigger a template export displays every field the template prints.

Tag groups per entity `Since 1.10`

An entity's admin **tags** — queries with `tag` set, offered to end users as one-click filters — can be organised into **tag groups**. In the UI they live in a **Tag groups** sub-tab of the entity's Queries tab, and the query form's group control is a clearable select with **+ New group**.

A group is a `TagGroup` document with a `name` (3–128 characters, trimmed, unique per entity in each language ignoring case), a `normalizedName` (unique per entity, its identity in packages) and a fixed grey colour. Membership is stored on the tag itself (`DataQuery.tag.tagGroup`), so a tag belongs to at most one group.

Route	Behaviour
`GET /v1/entities/{entity}/tag-groups`	Groups with `queries` : the ids of the entity's enabled admin tags in each.
`POST /v1/entities/{entity}/tag-groups`	Create a group (entity edit access).
`PATCH /v1/entities/{entity}/tag-groups/{id}`	Only `name` and `queries` ; `queries` sets membership to exactly the listed tags, moving them out of any other group.
`DELETE /v1/entities/{entity}/tag-groups/{id}`	Ungroups the tags, then deletes the group. Answers `200 {success: true, result: null}` . Galleries that referenced the group are not rewritten; the UI drops the dangling id.

Errors: `400 TAG_GROUP_NAME_DUPLICATE` (with `details.locale`), `400 TAG_GROUP_INVALID_QUERY` (not an enabled admin tag of this entity), `404 TAG_GROUP_NOT_FOUND` . Note the changed behaviour of `tag.tagGroupName` on query create/update: it now creates a group only when the name is free and otherwise fails with `TAG_GROUP_NAME_DUPLICATE` — it no longer silently reuses an existing group of the same name. Tag groups travel with the entity in packages, and a gallery that points at a missing tag or group imports without that option instead of failing.

Quick export of an entity's entries (admin) `Since 1.10`

The admin entity page has an **Export Entries** action. It is the gallery quick export without a view, backed by `GET /v3/entities/{entity}/entries/export` :

Parameter	Meaning
`format`	`xlsx` **or** `csv` (case-insensitive); anything else is refused.
`fields`	The columns. Without it: every entity field in defined order, each block expanded into its fields.
`filterCriteria` , `search` , `include_virtual` , `appliedQuickFilters` , `appliedTagsFilters` , `groupBy` , `quickFilterParameters`	The same filters as the admin list.

The response is the file, or — when the matches exceed `MAX_ENTRIES_RESULT_LIMIT` (default **10,000**) — a bus `messageId` , and the file is delivered in the background. The file and its sheet are named after the entity.

How it differs from the gallery route `GET /v3/entities/{entity}/views/{id}/entries/export` : rows are scoped like the admin entries list rather than by a view, so there is no `allowExport` , filter-first, human-verification or anonymous

truncation. Access is the right to list the entity's entries, re-checked when a background export runs; anonymous callers are refused, because a background export needs a recipient.

Where CSV is available. CSV exists only in these two quick exports — this admin route and the gallery quick export. Template exports (`/v1/entities/{entity}/entries/export/{templateId}`) produce whatever the template [Build Apps › Export templates] defines (document, XLSX or PDF), and reports remain XLSX and PDF.

Derived entries and entry permanent delete `Since 1.10`

A **derived entry** is a copy that a source entry produces in another entity: a **proxy**, **aggregator** or **joiner** entry of a virtual entity [Build Apps › Virtual entities]. An index links each copy to its source. Permanently deleting a source entry must remove its copies, or they are left orphaned — and once the source's files are released, the orphan sweeper may delete files the copies still show.

- **Bulk permanent delete cascades.** Permanently deleting entries in bulk used to leave their proxy, aggregator and joiner copies behind (only the single-entry delete cascaded). Each of the three cascades now also runs for a bulk delete, for every entry in it, and a failure makes the whole delete event retry.
- **A retried cascade finishes every copy.** If a copy was already deleted but its index cleanup failed, the retry no longer stops on the missing copy: it skips that copy's delete and still cleans its index row.
- **A stale index row does not stop the cascade.** An index row can outlive the derived entity it points to. The aggregator and proxy cascades used to give up on that row and skip every remaining copy (or leave the row behind); now, like the joiner cascade, they delete the stale row, log a warning and carry on. A genuinely missing *source* entity still skips the cascade.

Saving an entry in a custom store

When an entry is saved, the platform records which entity owns its files, so file cleanup knows when they are no longer used. An entry saved in a custom store [Build Apps › Share an entity's records across spaces] that names **no entity at all** (neither the entity id nor the entry's `dataEntity`) used to be registered under a global owner, which the cleanup could not match to any entry — so it could delete files the entry still listed. Such a save now **fails with** `500` (*Cannot record the files of a* `<store>` *entry: it has no entity id*) instead. A normal entry write always carries its entity and is unaffected; the fixed stores keep their global, module and entity scopes.

Notes & caveats for developers

- **Async writes.** A `2xx` from `POST` / `PATCH` / `PUT` means the command was *accepted and enqueued*, not that it succeeded. Validation failures inside the handler are reported via the command-executed socket message, not the HTTP response. Use the invoker token / socket reply to confirm the real outcome.
- **Field identity matters.** Renaming a field keeps its `id` , so it is *not* treated as delete+add by the versioning diff. Reusing a `normalizedName` for a new field (new `id`) after deleting the old one may register as `FieldRestored` .
- `source` **is deprecated** on fields in favour of the `sources` array, though much of the codebase still reads `source` for single-reference fields.
- Internal-entity block fields have their `block.internalEntity` back-filled to the internal entity's `id` during validation, so clients need not set it precisely as long as the block/internal-entity pairing is consistent.

Behaviour history

Changes to behaviour that integrators may still meet in older data or older clients:

- **Custom validation message and async uniqueness (form).** A field's custom validation error message could briefly disappear right after being set, if the field also carried an async **uniqueness** check: the uniqueness result used to overwrite the whole error object, wiping out the custom-rule error even though the field was still invalid. Now the uniqueness errors are merged with the custom-rule errors.
- **Query builder rows loaded from a saved query.** In the query builder used by a data query's criteria and by filter/expression rows, a row loaded from a saved query could stay invalid until hand-edited — blocking Save Query / Apply on a valid row — and Clear Query could leave a stale field selected in a row's field picker. Both are fixed.
- **Platform reference sources.** Before 1.8 the entity designer's reference-source picker tried to fetch `dataQuery` / `dataViews` sources as entities and showed an "Entity not found" toast over a correctly-configured field.
- **Services Catalog archive.** Before 1.8, archiving a service from the Services Catalog only changed the catalog row, leaving already-submitted requests for that service visible in My Requests / Inbox.

Pages, sidebar and share links API reference

This page is for **integrators and developers** who read or write pages through the API, build their own navigation from the sidebar API, resolve anonymous share links, or add widgets to the platform. It documents the stored model, validation, the endpoints and the enums.

> **Building a page in the designer?**
>
> Builders should start with Pages & Widgets [Build Apps] — it covers the page designer, the widgets and their settings in the labels the designer shows.

A **page** — known internally as a **Generic View** — is a composable, entity-agnostic screen: widgets placed into groups on a **12-column grid**, pulling data from many entities at once, unlike a Data View which presents a single entity.

The document covers three layers:

1. **Components (widgets)** — the catalog of reusable UI building blocks, and how the backend definition (`Component`) maps to a frontend renderer via a `componentName`.
2. **Pages (`GenericView`)** — how a page is assembled from components and groups, its layout/visibility/display options, page types, and the CRUD + end-user rendering endpoints.
3. **Navigation & system links** — the sidebar link tree (`LinkType` = Module / View / Page / Report), the `/v2/sidebar` API, delegation and restricted-access filtering, and the anonymous **system-link** token flow.

> Terminology note: in the backend, a "page" is the `GenericView` entity. The route prefix is `/v1/generic-views` and the sidebar/link layer refers to it as `LinkType.Page`. This document uses "page" and `GenericView` interchangeably.

Components (widgets)

The component catalog: `Component`

`Component` (`models/component.model.ts`) is the definition of a reusable UI element — a field control or a page widget. It is a versioned, named catalog entry, not an instance placed on a page.

Field	Type	Required	Meaning
`specificationVersion`	`number` (default `1`)	yes	Schema version of this definition (`ISpecificationVersion`).
`name`	`LocalizedValue`	yes	Human-readable, localized display name.
`type`	`number`	yes	Component type. Values come from `ComponentTypeEnum`: `0 = ViewComponent`, `1 = EditComponent`.
`componentName`	`string`	yes	**Stable identifier** that binds this definition to a frontend renderer and (for field controls) to a backend option **fetcher**.
`dataTypes`	`number[]`	yes	Which field data types this component can render/edit.
`entitySource`	`string`	no	Optional entity the component draws its data from.
`preferredPairedComponents`	`string[]`	yes	Other `componentName`s that pair well with this one (UI hint).

Field	Type	Required	Meaning
`componentOptions`	`object[]`	yes	Declarative option schema exposed to the builder when configuring an instance.
`validations`	`object[]`	no	Optional validation descriptors.

`idInjection` is disabled (`settings: {idInjection: false}`), so a `Component` carries no auto-generated `id` ; the `componentName` is its identity key.

`componentName` : the wiring key

The `componentName` string is the join across the stack. Two consumers rely on it:

- **Backend option fetchers.** Field controls that need server-supplied options (e.g. a user picker) register a `Fetcher` subclass whose static `componentName` matches. `FetcherBooter` (`booters/fetcher.booter.ts`) scans `fetchers/**/*.fetcher.js` and binds each class under `` `${FETCHER_BINDING_PREFIX}${cls.componentName}` `` . `FetcherFactoryService.getFetcher(componentName)` resolves it at request time (optional — returns `undefined` if none is registered). Example: `UserComponentFetcher` declares `static componentName = 'UserReferenceComponent'` .

  Fetchers implement `fetch(spaceId, data, options?)` and receive a rich `FetcherOptions` (filter, `includeIds` , `forceIncludeIds` , cascaded-parent context, search, parameters, user/language). See `fetchers/types.ts` .

- **Frontend registry.** `createComponentMetadata()` (`frontend/libs/shared/plugin-cdk/.../component-metadata-base.ts`) turns a `ComponentModel` (the `Component` shape) into a `ComponentRegisterDataModel` : it maps `componentOptions → options` , `type → componentType` , and applies `displayOptions` / `fieldQueryOptions` defaults, plus `support*` capability flags (`supportUniqueOption` , `supportDefaultValue` , `supportSearchable` , `supportMultipleOptions` , …). This is how a catalog entry becomes a live, configurable widget in the builder.

`FieldComponent` : a component on a field

`FieldComponent` (`models/field-component.model.ts`) is the minimal binding used when a **field** renders through a component:

```
{
  component: string;   // the componentName to render with
  options: object;     // instance configuration for that component
}
```

This is the same `component` + `options` pairing used by page widgets (below); `FieldComponent` is the field-level form of it.

Widgets

The core widgets registered in the web app (`dynamic-widgets-components/metadata.ts` ; groups in `generic-view-content/_utils/widget-group-utils.ts`) are **Section** and **Column** (groups), **Title** (heading levels 1–6), **Text** (multiline or markdown; editable height), **Link** and **Link list** (External, or Internal to a System View, Module View or Data View; link targets support dynamic token filling), **Media** and **Media List**, **Document** (optional preview image) and **Document List**, **Summary** (a statistic query), **View** (a data view; **Count of entries** 3 / 5 / 10, default 5, or 3 / 5 / Unlimited when the face supports paging and search), **TimeLine** (source Full / Module / Entity) and **Navigation** (default 12 columns, auto or fixed row height). Each widget has its own visibility settings. All widgets are registered **in-app** — there is no third-party/remote (federated) widget loading, and adding a new widget is a developer task.

An earlier version of the builder page described widget settings with older labels (Timeline **Page Navigation** Auto / Manual / Page Numbers, **Number of Posts**, **Show Pinned Posts**, **Source** Entity / Unit / General; Summary **Unit** / **Entity** with a locked **View**; View widget locked to the *Table* view with **Number of Entries**; widget **Custom Display** and **Custom Style**). The current designer labels are on Pages & Widgets [Build Apps].

The View widget: client-side behaviour changes

The **View** widget embeds a data view on a page and honours that view's faces, sorting and paging. Three behaviours changed recently; all are client-side, and the page model did not change:

- **Kanban face** `Since 1.6` — a View widget showing the Kanban face used to apply the face to rendering only while still fetching the flat entries list, so the board stayed empty. It now loads the board itself, with the view's column field (including, from 1.10, a reference column field — see Data Views & Faces [Build Apps]) and per-column *load more*; switching the face in the page editor refetches the first page in the new face's shape. Also delivered to the 1.6 and 1.8 lines as an update.
- **Column sort** `Since 1.10` — sorting a column of the widget's table used to flip the header arrow without reordering the rows. The widget now keeps the active sort (seeded from the view's default sorting), returns to page one and refetches on every sort change, and falls back to the default sorting when the sort is cleared.
- **Card-view paging** `Since 1.6` — in card view the paginator no longer sits against the last row of cards (table view unchanged). Also delivered to the 1.8 line.

Plugin widget packs

The optional packs are registered via each plugin's `plugin-module.config.ts` exposes; the working custom widgets live in the core app under `shared/custom-plugins/*`. Which widgets you actually see depends on which packs are installed in your deployment:

- **GATE** — timeline slider, timeline hero-slider.
- **task-management** — hero-slider, quick-links, quick-services, stay-in-touch (announcements), success-stories, welcome-title, who-we-are, wiki, entity-view-summary; plus a **citizen-profile** entry template.
- **Composite summaries** — an aggregated-summaries widget.

Note that the two **sample plugins** that used to ship as an SDK example (`frontend/libs/plugins/gate`, `plugins/task-mgmt`) were empty scaffolds with no implemented behaviour, and `libs/plugins/` was removed outright by the library restructure — the working widget packs above are the ones under `frontend/apps/core/src/app/shared/custom-plugins/*`, which is where they have always lived.

Pages: the `GenericView` model

`GenericView` fields

A page is a `GenericView` (`models/generic-view/generic-view.model.ts`), identified by a `UUID`. It holds the page's identity, its content (components/groups), and its behavior (sharing, anonymous access, refresh, redirects). Selected fields:

Field	Type	Notes
`specificationVersion`	`number` (default `1`)	Page schema version.
`name`	`LocalizedValue`	Required, localized page name.

Field	Type	Notes
`normalizedName`	`string`	URL-safe slug, validated by `NormalizedNameValidation` (min/max length + pattern). Auto-generated from `name` if omitted (see create flow).
`description`	`NullableLocalizedValue`	Optional.
`icon`	`Icon`	Optional.
`anonymous`	`AnonymousViewOption | null`	Anonymous access config incl. share URLs/tokens. Drives system-link creation.
`authenticated`	`AuthenticatedViewOption | null`	Authenticated-access options (e.g. default view, restricted).
`isPublished`	`boolean` (default `false`)	Publish state.
`isPinned`	`boolean` (default `false`)	Pin state.
`relatedTo`	`NullableReferenceProperty`	If set, `{dataEntity:'DataModule', dataEntry:<uuid>}` scopes the page to a module; `null`/absent means a system/space-level page.
`components`	`GenericViewComponent[]` (default `[]`)	Top-level widgets on the page.
`groups`	`GenericViewGroup[]` (default `[]`)	Layout groups; each holds its own `components`.
`sideBarConfig`	`SideBarConfig | null`	Controls sidebar open/close behavior when the page opens.
`displayConfig`	`GenericViewDisplayConfig | null`	Page-chrome flags (transparent, full page, header, spacing, container layout).
`displayOrder`	`number` (default `0`)	Ordering hint.
`uiAttributes`	`object | null`	Free-form UI metadata.
`allowExportPDF`	`boolean` (default `false`)	Enables PDF export of the page.
`domain`	`ReferenceProperty`	Optional domain scoping.
`autoRefresh`	`GenericViewAutoRefresh | null`	`{enabled, interval}` auto-refresh.
`creationInfo` / `updateInfo`	`ActionInfo`	Audit stamps (who/when, delegatee).
`pageType`	`PageTypeEnum` (default `Normal`)	`0 = Normal`, `1 = Embedded`.
`embeddedPageConfig`	`EmbeddedPageConfig`	`{url}` — required and URL-validated when `pageType = Embedded`.
`sharing`	`ApplicationSharing`	Required sharing config (Private/Public/Shared).
`redirectionRules`	`RedirectionRule[]` (default `[]`)	Conditional redirects evaluated for the page — see Redirection rules: evaluation below.
`updateIdentifier`	`string`	Required; regenerated (`v4()`) on create — an optimistic-concurrency / change token.
`help`	`PageHelp | null`	Optional in-page help/documentation. **Since 1.2.**
`executionPlan`	`ExecutionPlan | null`	Computed allowed-routes plan (set by `calculateExecutionPlan`).

The `GenericView` **constructor seeds** `components = []` **and** `groups = []` when absent. This is why a migration must never use `updateById` to patch a page (it would rebuild the entity and blank both arrays); use a raw `$set` instead.

Placing a widget: `GenericViewComponent`

Each widget instance on a page is a `GenericViewComponent` (`generic-view-component.model.ts`):

Field	Type	Required	Notes
`componentId`	`UUID`	yes	Unique per page — enforced (see validation).
`component`	`string`	yes	The `componentName` to render with.
`options`	`{[key:string]: any}`	yes	Instance configuration. May carry `options.groupId` to nest the widget in a group.
`displayOptions`	`ComponentDisplayOptions`	yes	Size/width/alignment/height (below).
`title` / `description`	`NullableLocalizedValue`	no	Per-instance overrides.
`icon`	`Icon`	no	Per-instance icon.
`uiAttributes`	`object | null`	no	Free-form UI metadata.
`visibility`	`ComponentVisibility`	no	Anonymous/authenticated visibility rules (below).

Grid & sizing: `ComponentDisplayOptions`

Field	Type	Default	Notes
`size`	`ComponentSizeEnum`	required	Preset grid footprint: `Row1Col1` (0) … `Row2Col4` (7).
`width`	`number` (1–12)	`1`	12-column width.
`verticalAlginment`	`VerticalAlignmentEnum`	`0 = Stretch`	`Stretch/Start/End/Center` (note: the field name is spelled `verticalAlginment` in source).
`showInNewRow`	`boolean`	`false`	Force the widget onto a new row.
`height`	`ComponentHeightConfiguration`	—	`{advanced, unit (default 'px'), value (default 100, min 0)}` .

Conditional visibility: `ComponentVisibility`

`visibility` (`component-visibility.model.ts`) gates a widget by audience:

- `custom: boolean` (default `false`) — when `false` , the widget inherits the page's default visibility; when `true` , `settings` apply.
- `settings: ComponentVisibilitySettings` — holds `anonymous` (`AnonymousOption`) and `authenticated` (`AuthenticatedOption`) toggles, letting a widget show/hide differently for anonymous vs. logged-in viewers.

Component visibility is validated at create/update time against the page's own anonymous/authenticated configuration (`validateComponentVisibility`), including components nested inside groups. In the designer this surfaces as the **Custom Visibility** toggle in a widget's **General** section.

Grouping & layout: `GenericViewGroup`

A `GenericViewGroup` (`generic-view-group.model.ts`) is a layout container:

- `groupId: UUID` — unique per page.

- `options: GroupConfiguration` — `{groupId?, orientation}` where `orientation` is `LayoutOrientationEnum` (`0 = Horizontal` default, `1 = Vertical`). (`GroupConfiguration` is `strict:false`, so extra layout keys may be present.)
- `components: GenericViewComponent[]` — the widgets inside the group.

A top-level component joins a group by setting `options.groupId` to the group's `groupId`. The create/update flow validates that every referenced group exists.

Page-level display, sidebar & refresh

- **`GenericViewDisplayConfig`** (nullable): `transparent`, `fullPage`, `outerSpacing`, `showPageHeader`, and free-form `pageContainerLayout`.
- **`SideBarConfig`** (nullable): `openMode: SideBarOpenModeEnum` — `0 = Preserve` (default), `1 = Hide`, `2 = Open`, `3 = Close`. Determines what happens to the navigation sidebar when this page is opened.
- **`GenericViewAutoRefresh`** (nullable, `strict:false`): `{enabled (default false), interval (default 0)}`.
- **`LinkConfiguration`** (`link-configuration.model.ts`) — a reusable hyperlink descriptor (used by widgets/config, distinct from sidebar links): `type: LinkTypeEnum` (`0 = External`, `1 = internal`), `url?`, localized `title` / `description`, `icon`, and `openIn: OpenLinkInEnum` (`0 = NewTab`, `1 = SameWindow`, `2 = NewWindow`).

Page types: Normal vs Embedded

`pageType` (`PageTypeEnum`) is `Normal (0)` or `Embedded (1)`. An **Embedded** page renders an external URL via `embeddedPageConfig.url`. On create/update the service enforces (`validateEmbeddedPageConfig`):

- the URL is present → else `EmbeddedPageMustHaveURLException`;
- the URL is valid → else `EmbeddedPageInvalidURLException`.

Designer labels and the stored model

Designer (Details / Content tab)	Stored as
Page Type Normal / **Embedded**, **URL**	`pageType` (`Normal = 0`, `Embedded = 1`), `embeddedPageConfig.url`
Pin to Sidebar	`isPinned`
Display Sidebar Preserve / **Open** / **Close**	`sideBarConfig.openMode` (`Preserve = 0`, `Open = 2`, `Close = 3`; `Hide = 1` has no designer option)
Appearance: **Transparent background**, **Full width page**, **Content padding**, **Page header**, **Contained layout**	`displayConfig.transparent`, `fullPage`, `outerSpacing`, `showPageHeader`, `pageContainerLayout`
Allow Exporting as PDF	`allowExportPDF`
Auto Refresh	`autoRefresh` `{enabled, interval}`
Redirection rules (a JSON textarea)	`redirectionRules`
Help & documentation	`help`
Sharing level, **Spaces**	`sharing` (`ApplicationSharing`)
Widget **Edit width** / **Vertical Alignment** (**Stretch**, **Top**, **Bottom**, **Middle**) / **New row** / **Edit height**	`displayOptions.width`, `verticalAlginment` (`Stretch`, `Start`, `End`, `Center`), `showInNewRow`, `height`
Widget **Custom Visibility**	`visibility.custom` + `visibility.settings`
Section / **Column** widgets	`GenericViewGroup`

Redirection rules: evaluation

`redirectionRules` route users to a specific target page based on conditions — priority-ordered, first match wins, and a rule only matches when **both** its access type (Anonymous/Authenticated/Everyone/Restricted) and its named users/groups admit the caller. `Since 1.10` this two-part match is enforced correctly for a page's own rules (previously a rule's named audience was silently ignored outside the Restricted case, and an anonymous visitor could hit a hard crash instead of a normal redirect-or-render). Pages and data views share the exact same evaluator; see Data Views & Faces § Redirection rules [Build Apps › Data Views & Faces] for the full evaluation semantics. A home/default page is auto-provisioned when a module or space is created.

Page (GenericView) API

Endpoints

All routes are under the `data-service`. CRUD is authorized by `voteByRelatedToFilterFactory('relatedTo', …)` voters; end-user render routes are separate.

Method	Path	Purpose
`POST`	`/v1/generic-views`	Create a page (`NewGenericView` body).
`GET`	`/v1/generic-views`	List pages.
`GET`	`/v1/generic-views/count`	Count pages (`where` filter).
`GET`	`/v1/generic-views/default`	Get the current user's default page.
`GET`	`/v1/generic-views/{id}`	Get a page by id (builder view).
`GET`	`/v1/generic-views/end-user/{id}`	Get a page for rendering (end-user).
`GET`	`/v2/generic-views/end-user/{id}`	End-user render, v2.
`PATCH`	`/v1/generic-views/{id}`	Update a page.
`PATCH`	`/v1/generic-views/{id}/permissions`	Update page permissions only.
`DELETE`	`/v1/generic-views/{id}`	Delete a page.

Create flow: validation & side effects

`POST /v1/generic-views` (`create`) runs, in order:

1. `validateName` on the localized name.
2. `validateGenericView` — e.g. a restricted authenticated view cannot also be the default view (`RestrictedViewCanNotBeDefaultException`); `relatedTo`, if present, must be a valid `ReferenceProperty`.
3. `validateComponentUniqueness` — `componentId`s must be unique across the page (`ComponentIdMustBeUniqueException`).
4. `validateGroupExistence` — every component's `options.groupId` must match an existing group (`ComponentGroupIdNotExistException`).
5. `validateGroupsUniqueness` — `groupId`s must be unique (`ComponentGroupIdMustBeUniqueException`).
6. `validatePermissions` and `validateComponentVisibility` (recursively into group components).
7. If `pageType = Embedded`, `validateEmbeddedPageConfig`.
8. **normalizedName**: scoped uniqueness. Scope is module-level when `relatedTo` is set, otherwise all space-level pages (`relatedTo` null/absent). A provided `normalizedName` must be unique in scope

(`GenericViewNormalizedNameDuplicateException`); otherwise it is auto-generated from `name` with `generateUniqueNormalizedName(..., 'page')`.

9. Audit stamps (`creationInfo` / `updateInfo`), a fresh `updateIdentifier` (`v4()`), and `calculateExecutionPlan` (static allowed routes).
10. Persist, then side effects: if `anonymous.enabled`, create system links via `SystemLinkService.upsertSystemLink`; enforce default-view uniqueness; store permissions into access-rights; publish `GenericViewCreatedEvent`.

Update flow: concurrency and installed pages `Since 1.10`

`PATCH /v1/generic-views/{id}` compares the incoming `updateIdentifier` with the stored one (`409 UPDATE_CONFLICT`) and then writes **conditionally**: the page is claimed only while its stored token still matches the one that was read, and only then replaced with a new token. Two saves that both passed the comparison can therefore no longer overwrite each other — the second one gets a 409 and must re-read. A page save also records the files the page references before the write and releases the ones it dropped afterwards; deleting a page releases them all.

A page **installed by a package** is locked (`403 PACKAGE_RESOURCE_READ_ONLY`). Through the same `PATCH`, and only for a body that changes them, the client keeps:

- **`isPublished`** — never checked against the token.
- **`name`**, **`description`**, **`icon`**. The first change records the package's own value in `packageLabels`, and typing the package's value back removes the record; an update of the package keeps the client's label. A label write needs the current `updateIdentifier`. `icon: null` clears the icon only with `?clearIcon=true`, because the page editor sends `icon: null` for every page that has none; `description: null` always clears.

The write is **partial**: the stored `components` and `groups` stay exactly as they are, the page's existing default-page assignments are kept rather than taken from the payload, and only the chosen keys, the audit stamp and a new `updateIdentifier` are set. Display order, audience and access-right rows (`PATCH /v1/generic-views/{id}/permissions`) are the client's too, and an update of the package keeps them. The same rules for views are on Data views [Build Apps › Data Views & Faces].

Global pages in packages `Since 1.10`

A **global page** — a page with no module or entity scope (`relatedTo` absent) — can be a root of a package, alone or beside modules, and is exported without children. A package save that names a page belonging to a module or entity is `422 PACKAGE_PAGE_NOT_GLOBAL`; publishing refuses a page that has since moved into one (`PUBLISH_PAGE_NOT_GLOBAL`), or that was itself installed or forked from another package (`PUBLISH_PAGE_INSTALLED`, `PUBLISH_PAGE_FORKED`). A statistic-query widget on the page whose data entry points outside the package still installs: the widget is left **unconfigured** — the same state the widget editor produces — instead of failing the install. See the packaging screens for where pages are added to a package.

Navigation: the sidebar link tree

The sidebar is a tree of **links**. A link is a `LinkDataEntry` whose `value` is a `LinkDataEntryValue`; the sidebar API returns them shaped as `GenericLink` nodes.

`LinkDataEntryValue` (stored shape)

Field	Type	Notes
`order`	`number`	Sort order within siblings.

Field	Type	Notes
`parent`	`ReferenceProperty | null`	Parent link (builds the tree).
`icon`	`Icon`	
`published`	`boolean`	Only `published:true`, `state:1` links are returned.
`type`	`CollectionValueModel<LinkType>`	Module / View / Page / Report / Untyped.
`ownership`	`CollectionValueModel<LinkOwnership>`	`System ('0')` or `Manual ('1')`.
`title`	`LocalizedValue`	Required.
`description`	`NullableLocalizedValue`	
`target`	`LinkTarget`	`{url, targetId, targetSectionId, targetViewAction}`.
`authentication`	`LinkAuthentication`	`{type, users[], groups[]}` — controls who can see the link.
`domain`	`ReferenceProperty[]`	Empty array = all domains; otherwise must match request domain.
`sharing`	`ApplicationSharing`	Private/Public/Shared vs. the request space.

`LinkType` (`@osos/shared`): `Module = '0'`, `View = '1'`, `Page = '2'`, `Report = '3'`, `Untyped = '4'`.

`LinkAuthenticationTypeEnum`: `Anonymous = '0'`, `Authenticated = '1'`, `Everyone = '2'`, `Restricted = '3'`.

`GenericLink` (returned shape) & URL derivation

The response node is `GenericLink` (`generic-link.model.ts`): `id`, `order`, `title`, `description`, `icon`, `url`, `target`, `type`, `parent`, `children[]`, plus view extras `userQueries`, `userCollections`, `sharedCollections`. Subclasses set the client `url` from `target.targetId`:

Link type	Class	Derived `url`
Module	`ModuleLink`	`null` (container only)
Page	`PageLink`	`/${targetId}`
View	`ViewLink`	`/v/${targetId}` (also carries saved queries + collections)
Report	`ReportLink`	`/reports/${targetId}` (web-only)
other	`GenericLink`	`target.url` as stored

`GET /v2/side-bar`

`SideBarController.getViewsForSidebarV2` returns a `SidebarDtoV2`.

Headers/params:

- `x-domain-id` (header) — request domain.
- `x-space-id` (header) — request space.
- `include` (query, `SidebarIncludeDataEnum`): `All = 0` (default), `Links = 1`, `OnlyService = 2`. **The `include` selector is Since 1.4.**

`SidebarDtoV2` fields:

Field	Type	Notes
`links`	`GenericLink[]`	The link tree (empty when `include = OnlyService`).
`serviceViewsStatus`	`SidebarServiceStatus | null`	Service-section counters (null when `include = Links`).

Field	Type	Notes
`isRestricted`	`boolean`	Whether the user holds the system-level `Restricted` floor. **Since 1.8.**
`restrictedAccess`	`{hasModuleAccess, hasEntityAccess}`	Resource-level grants for restricted users; each is `true` if any module/entity grant is ≥ `AccessRightLevelEnum.ViewAccess`. **Since 1.8.**

The endpoint is `@authorize({skip: true})` but requires authentication (`Auth2` strategy at the controller). The restricted-access summary is attached to **every** `include` variant so the DTO shape stays stable.

How links are selected & assembled

`LinksService.getLinksForSidebar(domain, spaceId, userId, delegationId?)`:

1. **Audience filter** by `authentication.type`:
   - Anonymous request (`userId` null): types `0` (Anonymous) or `2` (Everyone), plus type `1` (Authenticated) only when the link's own `value.type.id === '0'`.
   - Authenticated request: types `1` (Authenticated) or `2` (Everyone), plus `3` (Restricted) when the user is listed in `authentication.users` or is a member of one of `authentication.groups`.
2. **Base predicate**: `published:true` AND `state:1`; domain matches (empty `domain` array = global, else `domain.dataEntry === domain`); sharing is `Public`, or `Private` / `Shared` scoped to the request `spaceId`.
3. **Order**: `value.order ASC`, then `value.title.en ASC`.
4. **Delegation filter** (when `delegationId` is set): drops links whose target the delegatee lacks permission for — `view` / `report` / `page` targets are checked against delegated `view` / `report` / `genericView` entitlements (`extractResourceInfoFromLink` + `filterLinksByDelegatedPermissions`). Links that don't point to a controlled resource are kept.
5. **View enrichment**: for `View` links, attach the user's saved queries (`userQueries`), owned collections (`userCollections`), and shared collections (`sharedCollections`).
6. **Tree build** (`convertLinksToTree`): parent/child nesting via `value.parent`. A `View` link whose parent is hidden is skipped. Finally, `Module` links with **no children are dropped** (empty modules don't show).

System links (anonymous share tokens)

"System links" are opaque tokens that expose a view or page to anonymous visitors without exposing its id. This is a different concept from sidebar links.

Model & storage

- A share URL is a `ViewUrl`: `{uniqueToken: string, validity?: string | null}`.
- On a page/view, `anonymous` (`AnonymousViewOption`) holds `enabled`, `defaultView`, `urls: ViewUrl[]`, and optional `humanVerification`.
- Persisted tokens live as `SystemLinkEntry` records (`value.relatedObject`, `value.token`, `value.validity`), written by `SystemLinkService`.

Generating & reconciling tokens

- `generateUniqToken(size)` — creates a `nanoid(size)` token and retries on the (rare) collision. `SYSTEM_TOKEN_LENGTH = 32`.
- `upsertSystemLink(relatedObject, links, user, isUpdate?)` — inserts new tokens and, on update, **archives** any previously active token that is no longer in the submitted list (state → `Archived`). Called automatically from

the page create flow when `anonymous.enabled`.

System-link API

Method	Path	Auth	Purpose
`GET`	`/v1/system-link/generate`	authenticated	Returns `{token}`, a fresh unique 32-char token to attach to an anonymous URL.
`GET`	`/v1/system-link/resolve/{token}`	`@authenticate.skip()` (public)	Resolves a token to the entity behind it.

Resolve semantics (`getAnonymousView`):

1. Token length must equal `SYSTEM_TOKEN_LENGTH` (32) → else `InvalidViewTokenExption`.
2. Look up a `DataViews` by token, else a `GenericView` by token → if neither, `DataViewNotExistException`.
3. On success, return a `ReferenceProperty`: `{dataEntity: 'DataViews' | 'GenericView', dataEntry: <view.id>}`. The caller then loads the actual view/page via the normal render route.

Quick reference: enums

Enum	Values
`ComponentTypeEnum`	`ViewComponent=0`, `EditComponent=1`
`ComponentSizeEnum`	`Row1Col1=0` … `Row2Col4=7`
`VerticalAlignmentEnum`	`Stretch=0`, `Start=1`, `End=2`, `Center=3`
`LayoutOrientationEnum`	`Horizontal=0`, `Vertical=1`
`SideBarOpenModeEnum`	`Preserve=0`, `Hide=1`, `Open=2`, `Close=3`
`PageTypeEnum`	`Normal=0`, `Embedded=1`
`LinkType`	`Module='0'`, `View='1'`, `Page='2'`, `Report='3'`, `Untyped='4'`
`LinkOwnership`	`System='0'`, `Manual='1'`
`LinkAuthenticationTypeEnum`	`Anonymous='0'`, `Authenticated='1'`, `Everyone='2'`, `Restricted='3'`
`LinkTypeEnum` (LinkConfiguration)	`External=0`, `internal=1`
`OpenLinkInEnum`	`NewTab=0`, `SameWindow=1`, `NewWindow=2`
`SidebarIncludeDataEnum`	`All=0`, `Links=1`, `OnlyService=2`
`ApplicationSharingLevelEnum`	`Private='PRIVATE'`, `Public='PUBLIC'`, `Shared='SHARED'`

Reaction model and API reference

This page is for **integrators and developers** who read or write reactions through the API, build tooling around them, or need the exact runtime contract — the stored model and its numeric enums, every event and action type with its options, how conditions are evaluated, how runs are executed, chained and recorded, and the endpoints.

> **Building a reaction in the UI?**
>
> Builders adding reactions in the entity designer should start with Reactions & automation [Build Apps] — it covers the same features in the labels the designer shows, organised by goal.

In the API and the code a reaction is called a **trigger**; the stored model is `DataReaction`. A reaction binds to one **event type** on a **defining entity**, optionally narrows down with **conditions** (an event-level matcher plus a criteria/query filter), and then runs its **action**. Actions can create, update or archive data, drive workflows, call external APIs, send notifications, email, SMS or WhatsApp, generate files, run custom functions and publish new events that other reactions listen to — which is what lets reactions **chain**.

Architecture: two services

The automation engine lives in its own microservice, `reaction-service`. The `data-service` does **not** execute reactions itself; its `ReactionServiceProvider` / `ReactionServiceWrapper` (`backend/services/data-service/src/services/reaction.service.ts`) is a LoopBack **service proxy** over the `restds` datasource that forwards `getEntityReactions` / `getReaction` to the reaction microservice, wrapped in `async-retry` (client `4xx` errors `bail` immediately; other errors retry). The `data-entity-action` model folder under `data-service` mirrors the action option DTOs, but the authoritative, executed models are those under `reaction-service/src/models`.

Reactions are stored in MongoDB (the `DataReaction` collection) and are triggered by **domain events consumed off the service bus** (`@node-ts/bus`), not by direct REST calls. Each `EventTypeEnum` has a bus event handler under `handlers/events/**` that resolves affected entries and dispatches into the central `EventsHandler`.

The engine internals behind this — dispatch, chunked fan-out, sagas and recovery — are documented for platform engineers in Engine: Reactions.

The trigger model (`DataReaction`)

Source: `models/data-reaction/data-reaction.model.ts`. Key fields:

Field	Type	Meaning
`id`	UUID	Trigger id
`specificationVersion`	number (default 1)	Schema version of the reaction
`normalizedName`	string	Machine name (pattern/length from `NormalizedNameValidation`)
`name`	LocalizedValue	Display name
`description`	NullableLocalizedValue	Optional description shown on the Reactions tab. `Since 1.10`

Field	Type	Meaning
`definingEntity`	ReferenceProperty	The entity the reaction is attached to (`{dataEntity:'DataEntity', dataEntry:<entityId>}`)
`eventType`	`EventTypeEnum`	The event that fires this reaction (see below)
`eventConfig`	union `DataReactionEventConfig`	Per-event configuration + conditions
`executorType`	`EventExecutorTypeEnum` (`SYSTEM` / `MANAGER` / `EXECUTOR`)	Whose identity/permissions the actions run under
`sensitiveDataAccessPolicy`	`SensitiveDataAccessPolicyEnum`	What the reaction's actions may see of a **sensitive field** on the entries they touch; absent means `Privileged`. See Sensitive-data access policy. `Since 1.10`
`query`	ReferenceProperty | null	Optional saved `DataQuery` whose criteria is AND-merged into the condition
`criteria`	`FilterExpression`	Optional condition filter evaluated against the trigger context
`actions`	`DataReactionAction[]`	Ordered actions to execute
`type`	`DataReactionTriggerTypeEnum` (`NormalTrigger=0` , `SLATrigger=1`)	Normal vs SLA trigger
`sourceEntity`	`DataReactionSourceEntity` | null	Fires from a related/absolute entity instead of self (see Related-entity triggers)
`relatedEntities`	`DataReactionRelatedEntities[]`	Related entities pulled into the context
`spaceScope`	`DataReactionSpaceScope`	Which spaces the reaction executes in (`rootSpace` / `currentSpace` / `otherSpaces`)
`loopThreshold`	number (default 0)	Max re-execution loops per action; `0` = run once (see Loop protection)
`isImplicit`	boolean	System-generated reaction (protected from user CRUD)
`disabled`	boolean	Skipped by the dispatcher when `true`
`validations`	`DataReactionValidation` | null	Validity state; invalid triggers are skipped at dispatch
`updateIdentifier`	string (UUID)	Optimistic-concurrency token (see PATCH)
`creationInfo` / `updateInfo`	ActionInfo	Audit stamps

The `DataReaction` model uses `settings.strict:false`-style loose typing in places because `options` and `eventConfig` are `any`-typed unions resolved at runtime.

Sensitive-data access policy `Since 1.10`

A reaction can declare what its actions are allowed to **see** of a sensitive field [Build Apps › Protect a field] on the entries they touch. The setting is `sensitiveDataAccessPolicy`, picked in the reaction editor from a **Sensitive Data Access Policy** dropdown — the dropdown only appears when the entity actually has a sensitive field, so most reactions never meet it. There are four choices:

Policy	What the automation sees
Privileged — the default when nothing is stored	The **real value**, always, whoever or whatever triggered the run, and regardless of that person's own grants. This is what every reaction written before the setting existed already did, so nothing changed for them.

Policy	What the automation sees
Prohibited	The **mask** (`****`), always — whoever or whatever triggered the run. The reaction can still act on the entry; it just never sees the real value of a sensitive field.
User-Based	Decided **per run, from the grants of the person who actually triggered it**: they see what they would see reading the entry themselves, and someone without the grant gets the mask.
Auto	**System-triggered runs behave as Privileged; user-triggered runs behave as User-Based.** Use it for a reaction that fires both ways and should follow the person whenever there is one.

`User-Based` **cannot be combined with an automatic trigger.** A reaction that fires with no person in the loop — its executor is the **system**, or its event is a **timer** or **SLA time exceeded** — has no "actual executor" whose grants could be consulted, so saving `User-Based` on one is rejected with *"The 'User-Based' policy cannot be selected with an automatic trigger type"*, shown under the dropdown. **Auto** is the setting to reach for there: it resolves to Privileged on exactly those runs. `Privileged` and `Prohibited` stay available on an automatic trigger.

What a masking policy means for a write. Under `Prohibited` — or under `User-Based` / `Auto` when the run resolves to masked — an action must not store a mask over a real value, so a write that would do that fails instead. `Since 1.10` that check is scoped to **the write itself**. An **Update Data Entry** (`UpdateDataEntry`) action against an entity that merely *holds* a sensitive field could previously never run at all under a masking policy, **whatever it actually wrote**: setting one unrelated, non-sensitive field to a constant was enough to fail it, and the only workaround was to switch the reaction to `Privileged`. Now the action fails only if **that particular write** would have stored a masked value; a mask that was already sitting in the entry because the policy masked it on the way in is left alone, and never reaches storage. **Create Data Entry** (`CreateDataEntry`) was never affected — it starts from an empty value, so any mask there is one the action introduced — while **Call API** (`CallApi`) and **Send Email** (`SendEmail`) stay under the stricter whole-value check, because those send the value out of the platform.

The action model (`DataReactionAction`)

Source: `models/data-reaction/data-reaction-action.model.ts` (and the `data-service` mirror at `models/data-reaction-action.model.ts`).

Field	Type	Meaning
`normalizedName`	string	Machine name of the action (used as the key in the trigger context and loop tracking)
`name`	LocalizedValue	Display name
`description`	NullableLocalizedValue	Optional
`actionType`	`AutomatedActionTypeEnum`	Which handler runs (see Action types)
`options`	union `DataReactionActionOptions`	Per-action-type configuration
`resultField`	string	Set at runtime to the result id, so later actions can reference this action's output
`overridableOptions`	`ActionOverridableOptions[]`	Option paths that may be overridden per execution

One action per reaction. The model stores `actions` as an array, but the builder creates and edits exactly one action per reaction, and save-time validation is written for one. Model a sequence as separate reactions chained with `ReactionSucceeded` (see Chaining reactions).

Duplicating in the builder `Since 1.8`. A duplicated reaction opens as a new reaction whose event can be changed (only a reaction added from an event group's **+**, or an existing one being edited, keeps its event locked). If the event is

changed away from *Event Announced* while the source is an **absolute entity** — a source only an announced event can use — the source falls back to **Self** instead of the hidden absolute entity still being saved.

Trigger events

`EventTypeEnum` (`packages/osos-shared/src/models/enums.ts`) — the complete set a reaction can bind to via `eventType`:

#	`EventTypeEnum`	Fires when	Event config type
0	`DataEntryCreated`	An entry is created	`DataEntryCreatedEventConfig`
1	`DataEntryUpdated`	An entry is updated	`DataEntryUpdatedEventConfig` (field-level matching)
2	`DataEntryArchived`	An entry is archived	`DataEntryArchivedEventConfig`
3	`WorkflowActionExecuted`	A workflow action runs	`WorkflowActionExecutedEventConfig` (step/action filter)
4	`WorkflowStepReached`	A workflow step is reached	`WorkflowStepReachedEventConfig` (step filter)
5	`WorkflowStepExited`	A workflow step is exited	`WorkflowStepExitedEventConfig` (step/action filter)
6	`Timer`	A scheduled time / cron fires	`TimerEventConfig`
7	`CustomEvent`	A button fires the custom action, or another reaction's **Execute Custom Action** action invokes it (**Local** or **External** type only)	`CustomEventConfig` (`customActionType` + `contextual` `Since 1.8`; may prompt the user for input)
8	`ReactionSucceeded`	Another reaction completed successfully	`ReactionSucceededEventConfig` (`entityId` + `triggerId`)
9	`FileGenerated`	A file/template was generated	`FileGeneratedEventConfig` (`template`)
10	`ReactionFailed`	Another reaction failed	`ReactionFailedEventConfig` (`entityId` + `triggerId`) `Since 1.8`
11	`EventAnnounced`	An `AnnounceEvent` action published an announcement	`EventAnnouncedEventConfig` (`entityId` + `announcementId`) `Since 1.8`
12	`DataEntryRestored`	An archived entry is restored	(`BaseEventConfig`) `Since 1.8`
14	`WorkflowStepOwnerChanged`	A workflow step's owner changes	`WorkflowStepOwnerChangedEventConfig` (owner-change-type + step filter) `Since 1.8`
20	`ZapierEvent`	An inbound Zapier integration event	(`BaseEventConfig`)
100	`SlaTimeExceeded`	A workflow SLA time is exceeded	`SLATimeExceededEventConfig` (occurrences: `startingAfter` / `repeatAfter` in seconds, `repetitions` a count)

Events 8, 10 and 11 are what make **reaction chaining** possible — see Chaining reactions. `SlaTimeExceeded` is raised by a per-minute background check that marks due-but-open workflow steps as overdue, so you can react to SLA breaches automatically (for example, escalate to a manager or reassign the step owner).

Which events fire — with and without a workflow

Reaction events originate in the `data-service` write paths, and the set that fires depends only on the operation — **attaching a workflow never suppresses an event**. In particular, `DataEntryCreated` is published unconditionally on every entry create; when the entity has workflows attached, the create *additionally* publishes `WorkflowStepReached` for the first step of the active workflow. The same holds on the create-with-related-entries path: `DataEntryCreated` fires for the main entry and for each related entry, plus one `WorkflowStepReached` per workflow-backed entity.

Operation	Fires for every entity	Fires only with a workflow attached
Entry created	`DataEntryCreated`	`WorkflowStepReached` (first step)
Entry updated	`DataEntryUpdated` (immediate, or deferred until the surrounding transaction commits)	—
Entry archived	`DataEntryArchived`	—
Entry restored	`DataEntryRestored`	—
Workflow step moved	—	`WorkflowStepReached` + `WorkflowStepExited`
Workflow action executed	—	`WorkflowActionExecuted` + `WorkflowStepExited`
Step owner changed	—	`WorkflowStepOwnerChanged`
Ad-hoc step inserted	—	`WorkflowStepReached`
Workflow SLA breached	—	`SlaTimeExceeded` (raised by the per-minute SLA check)

Two consequences worth designing around:

- **Workflow step transitions do not publish `DataEntryUpdated`.** The step-change handlers publish only the workflow events, so a reaction that must fire when an entry moves through its workflow has to bind to `WorkflowStepReached` / `WorkflowStepExited` (or the other workflow events) — a `DataEntryUpdated` reaction will not see the step change.
- **A failed event publish on the create path is swallowed.** The publish block is wrapped in a try/catch that logs `Error publishing event` and continues, so the API caller still gets a successful create even when no event went out. If a `DataEntryCreated` reaction unexpectedly did not run for a successfully created entry, check the `data-service` log for that message before debugging the reaction itself.

On deployments older than 1.8, the event types `ReactionFailed` (10), `EventAnnounced` (11), `DataEntryRestored` (12), and `WorkflowStepOwnerChanged` (14) do not exist in `EventTypeEnum` at all — reactions cannot bind to them, and they never fire there.

Timer triggers

`TimerEventConfig.timerOptions` supports either a one-time run (`oneTime`, an ISO timestamp) or a `repeat` schedule. `repeat.expression` is "a human-readable format String, or a cron format String" and `repeat.options` accepts `timezone` (moment-timezone), `startDate`, and `endDate`. Timer executions run through `EventsHandler.handleTimerTrigger`, which builds the context, runs the actions, writes a `TriggerHistory`, and publishes `TriggerExecutedEvent` — the same tail as normal events.

Chunked async fan-out. A timer does not process its whole target set inline. When it fires, the scheduler handler (`trigger-timer-job-triggerd.handler.ts`) loads the active timer trigger, applies its criteria to resolve matching entries, then **splits those entry ids into chunks and enqueues them as BullMQ parent/child jobs** through a

`FlowProducer`, so the actions run asynchronously in batches. Chunk size comes from `TIMER_EVENT_CHUNK_SIZE` (default **100**). Large target sets therefore process batch-by-batch across workers rather than all at once.

Custom actions and "Ask User"

A `CustomEvent` — presented in the builder as a **Custom Action** `Since 1.8`, and still called *Custom Event* in the code and API — is raised from a UI button attached to an entry's options (see the Buttons guide), so a reaction can be invoked on demand rather than by a data change. Two builder-side capabilities extend it:

- **Execution indicator** — for a UI reaction you can control the loading indicator shown while the reaction runs: **Blocking** (show the indicator and block the user until it finishes), **Inform** (show the indicator but let the user navigate away), or **None**. When **Blocking** is selected you can also edit the **On execution started** and **On execution succeeded** messages shown to the end user.
- **Ask User** — enable **Ask User For** to collect input before the actions run. Give the dialog a **Dialog Name** and choose the **Fields Source**: an **Inline Entity** (manually entered JSON) or a **View** (only views with "Ask the User" enabled appear, and their fields populate the dialog). The values the user enters can then be mapped into the selected action's fields. In the code this input is carried into the context as the AskUser entity.

Like timers, custom events **fan their matching entries into BullMQ parent/child chunk jobs** (`custom-event-triggered.handler.ts`) rather than running inline — chunk size `CUSTOM_EVENT_CHUNK_SIZE` (default **100**). The handler additionally emits a socket `JobStarted` event (`JobStartedMessage`) when the batch is enqueued, so the UI can report progress while a large target set processes asynchronously in batches.

Custom action type and Contextual `Since 1.8`

The trigger is named **Custom Action** in the builder, and it is configured with two settings that together replace the old **UI Event** toggle:

- **Custom action level** — a dropdown (`customActionType`) with three values, shown as **UI action**, **Local action** and **External action**:
  - **UI** — the action is invoked from a user interface, i.e. a system button. This is the equivalent of the old *UI Event* toggle being on.
  - **Local** — a module-level action, callable from another reaction's Execute Custom Action action only when the calling trigger's own entity belongs to the **same module**.
  - **External** — callable from an **Execute Custom Action** action anywhere, in any module.
- **Contextual** — a toggle (`contextual`), **on by default**. It governs whether the action expects a **target entry** at all: a contextual action needs one (resolved by its own event, or by the calling **Execute Custom Action** action's Scope — see below); a non-contextual action has none and simply runs once when invoked.

> ### Terminology change — Since 1.8
>
> The custom action type was originally shipped as **UI** / **Module** / **Internal**. A later revision in the same 1.8 cycle **renamed** the two non-UI values to **Local** and **External** — **UI** is unchanged. A one-way migration rewrote every existing reaction's stored value (`Module` → `Local`, `Internal` → `External`); it cannot be reversed automatically. **If you see `Module` or `Internal` anywhere else — an older screenshot, a note, a script — read it as `Local` / `External` respectively; the product and API no longer use the old names.**

Two rules follow from the type you choose:

- The **Show to user** options — the execution indicator and its **On Start** / **On Success** messages described above — apply **only** when the type is **UI**. They are disabled for **Local** and **External**.
- The ask-user field is labelled **Ask User For** when the type is **UI**, and **Parameters** when it is **Local** or **External** — because for a non-UI action those same fields double as the binding target for a calling **Execute Custom Action** action's Parameters (see below). Asking for input no longer depends on the type: a Local or External action can collect input too.
- **System buttons can only fire UI-level actions.** A button can structurally point at a `Local` / `External` custom action, but running it that way is rejected with a 403 — see Buttons: system buttons can only fire UI-level actions [Build Apps › Buttons] for why and what error to expect.

Existing reactions. Every custom-action trigger that existed before 1.8 was migrated to type **UI** with **Contextual** switched on, and the retired `uiEvent` flag was removed — review any trigger that should now be **Local** or **External** and change its type by hand; the original flag cannot be recovered. Triggers that had already been configured with the old **Module**/**Internal** values were carried forward automatically to **Local**/**External** by the terminology-rename migration above — no manual action is needed for that part.

"On owner changed" triggers `Since 1.8`

`WorkflowStepOwnerChanged` (14) fires whenever the owner of a workflow step on an entry changes. The event is published by the `data-service` right after the new owner has been persisted, so a reaction bound to it always sees the committed state.

It covers three kinds of change, and the event says which one it was — so a reaction can react to a hand-off without also firing when somebody simply picks up an unowned step:

`OwnerChangeTypeEnum`	**Fires when**
`ChangeOwner` (0)	The owner is changed to somebody else. This is also the fallback when the caller did not say which kind of change it was.
`TakeOwnership` (1)	A user takes ownership of the step. Resolved server-side whenever the change came through take-ownership mode, regardless of what the caller asked for.
`Reassign` (2)	The step is reassigned.

Alongside the change type, the event carries the workflow and step ids, the localized step name, the previous and new owner, the original owner, the note and permanent-change flag supplied with the change, and whether the step was an **ad-hoc** step.

Builder configuration. The event's own config (`WorkflowStepOwnerChangedEventConfig`) exposes two optional filters, and the reaction builder shows a dedicated panel for them:

- **Owner Change Type** — a multiselect over `ownerChangeTypes` with the three values above. Leave it empty and the reaction fires on any kind of owner change.
- **Choose Step** — the shared step selector, stored as `steps`. Leave it empty and the reaction fires on any step, including steps added to the workflow later. Steps are matched on the workflow **and** step id as a pair, so a step id reused across two workflows cannot cross-fire. The selector's `ADHOC_STEP` entry matches an owner change that happened on an ad-hoc step of the selected workflow.

The reaction's actions can read the change through the event's reaction schema, which exposes the change type as a localized label plus a step-information block: the step id and name, the previous and new owner (resolvable to a user, group, or lookup collection), and who executed the change.

Conditions

A reaction has **two independent filtering stages**, both optional. If neither is configured, every matching event runs the actions.

Event matchers (event-config level)

Before building a context, `EventsHandler.checkEventMatcher` looks up an `IEventMatcher` for the `eventType` (via `EventMatchersFactoryService`). If one exists, its `isEventMatchTrigger(eventConfig, event)` decides whether to proceed. Matchers exist for a subset of events:

- **`DataEntryUpdated`** — `DataEntryUpdatedEventMatcher` matches only if one of the changed fields (`event.valueChanges[].fullNormalizedName`) is in `eventConfig.fields`. **If `fields` and `relatedFields` are both empty, it matches any change.**
- **`WorkflowStepReached` / `WorkflowStepExited` / `WorkflowActionExecuted`** — match against the configured `steps` / `actions`; empty selection matches all.
- **`WorkflowStepOwnerChanged`** — matches against the configured `ownerChangeTypes` and `steps`; either one left empty matches all. **Since 1.8.**
- **`SlaTimeExceeded`** and **`FileGenerated`** — have their own matchers (step selection / template).

Event types **without** a matcher (created, archived, restored, timer, custom, reaction-succeeded/failed, event-announced, zapier) always pass this stage and rely solely on the criteria filter.

Criteria and query filter (entry level)

`criteria` is a `FilterExpression` evaluated against the resolved entry — the same mechanism as a saved query, so a builder can reuse an existing query from the entity under the **Entry Source** option. When a `query` reference is also set, its criteria is **AND-merged**:

```
trigger.criteria = {
  operator: ExpressionTypeEnum.And,
  args: [trigger.criteria, query.result.criteria],
};
```

Evaluation runs the parsed criteria as a real MongoDB query over the candidate entry (`QueryEngineService.parseCriteria` → `DataEntryService.findWithResolvesFactory`), so a criteria behaves like the same filter in a saved query. `Since 1.10` the entry is handed to MongoDB as an in-memory **`$documents`** aggregate source instead of being copied into a temporary `trigger_<random>` collection: nothing is written or dropped per run, and a crashed pod leaves nothing behind. The design consequence is that **operators that need a real collection cannot be used** in a reaction criteria — `$text`, `$geoNear` / `$near` / `$nearSphere`, `$indexStats`, `$collStats`, `$out` and `$merge`. For the standard path, `EventsHandler.getResolvedEntry` merges the criteria with an id filter and only proceeds if the entry still matches; for the reaction-succeeded path it uses `findMatchedCriteriaInTriggerContext`. If the entry does not match, the reaction is skipped (logged as "criteria is not matched with event data").

This means criteria can reference **any field reachable in the trigger context**, including related entities via relation-traversal paths (`value.requester.:relation.value.first_name`) resolved by `TriggerContext`.

Date pointers in criteria. A criteria argument of type Pointer resolves `@`-prefixed date functions before the query runs. Both `@today()` and `@now()` are supported (alongside `@StartOfDay()`, `@StartOfWeek()`, `@StartOfMonth()`, and `@StartOfYear()`), but they return different shapes — and matching depends on the shape:

- `@now()` returns a **full ISO datetime**, so it only suits comparisons against datetime-typed values. Compared against a date-only stored value, an equality or range check never matches — which presents as "`@now()` doesn't work" even though the pointer resolved fine.
- `@today('±Nd')` returns a **date-only string** — use it for date-only fields. (Internally it formats start-of-today plus a hard-coded two-hour offset.)
- The manipulation amount **must be signed**: `@today('+1d')` and `@today('-3d')` work, but an unsigned `@today('1d')` is silently ignored and the base date is returned. Units: `d` days, `w` weeks, `M` months, `y` years, `h` hours, `m` minutes, `s` seconds.
- A pointer that fails to evaluate is not silent — the expression service throws an `ExpressionExecutionException`.

Action types

Actions are dispatched by `AutomatedActionTypeEnum` → an `IActionHandler`. Handlers are auto-registered by `ActionHandlersBooter`, which scans `action-handlers/**/*.action-handler.js` and binds each class under `entity-action-handlers:<actionType>` using the class's `static actionType`. At runtime `EventsHandler.actionHandler` resolves the handler via `ActionHandlerFactoryService.getActionHandler(action.actionType)`; an unknown type throws `DataNotSupportedException`.

The enum declares 26 values; **24 have a concrete handler in `reaction-service`** — three of them new in 1.10: `SetPermission` (31) and `RemovePermission` (32) (see Set / Remove Permission actions) and `SendWhatsApp` (107). The exact `options` body for every type is in the Action options reference. `CreateDataEntity` (0) and `UpdateDataEntity` (1) are declared in the enum and have option interfaces (`CreateDataEntityActionOptions`, `UpdateDataEntityActionOptions`) but never had a handler. `Since 1.10` they are formally retired: the builder no longer offers them, creating or updating a reaction that carries either type is refused with **422**, and an action of either type already stored in an older reaction is **skipped** when the reaction fires — it is recorded as a failed action whose error reads `action type N (<name>) was removed (…); action skipped`, and the actions after it still run. The two enum values stay only because stored reactions may still carry them.

#	`AutomatedActionTypeEnum`	Handler file	What it does
0	`CreateDataEntity`	*(none)*	Retired — refused on save (422), skipped at run time `Since 1.10`
1	`UpdateDataEntity`	*(none)*	Retired — refused on save (422), skipped at run time `Since 1.10`
2	`ArchiveDataEntity`	`archive-data-entity`	Archive a whole data entity (entity-level, distinct from archiving an entry)
10	`CreateDataEntry`	`create-data-entry`	Create an entry on a target entity, mapping values from context
11	`UpdateDataEntry`	`update-data-entry`	Update self / referenced / related / query-selected entries
12	`ArchiveDataEntry`	`archive-data-entry`	Archive entries (self / relation / query source)
13	`CreateBlockItem`	`create-block-item`	Append an item to a block (sub-table) field
14	`UpdateBlockItem`	`update-block-item`	Update block item(s) by index/field/expression criteria

#	`AutomatedActionTypeEnum`	Handler file	What it does
15	`DeleteBlockItem`	`delete-block-item`	Delete block item(s) by criteria
16	`DeleteArchivedEntry`	`delete-archived-data-entry`	Permanently delete already-archived entries
20	`MoveWorkflowToStep`	`move-workflow-to-step`	Move a workflow to a target step
21	`ExecuteWorkflowAction`	`execute-work-flow-step`	Execute one or more workflow actions
22	`ChangeCurrentStepOwner`	`change-current-step-owner`	Reassign the current workflow step owner
30	`UserActivationToggle`	`user-activation-toggle`	Activate/deactivate users (bulk)
31	`SetPermission`	`set-permission`	Grant an access level on a resource to users or groups `Since 1.10`
32	`RemovePermission`	`remove-permission`	Revoke users' or groups' grant on a resource `Since 1.10`
90	`CallApi`	`call-api`	Call an external HTTP API
100	`SendNotification`	`send-notification`	In-app notification from a template
101	`SendEmail`	`send-email`	Email from a template
102	`SendSms`	`send-sms`	SMS from a template
103	`GenerateFileFromTemplate`	`generate-file-from-template`	Render a file from a template
104	`ExecuteCustomFunction`	`execute-custom-function`	Run a custom (Knative) function
105	`AnnounceEvent`	`announce-event`	Publish an announcement that `EventAnnounced` reactions consume
106	`ExecuteCustomAction`	`execute-custom-action`	Run a **Local**- or **External**-level Custom Action reaction against the entry/entries its Scope resolves — or once, with no entry, when the target is non-contextual `Since 1.8`
107	`SendWhatsApp`	`send-whatsapp`	Send an approved Meta WhatsApp template to the resolved recipients `Since 1.10`
200	`NotifyZapier`	`notify-zapier`	Push an outbound event to Zapier

Value mapping and selected options

Options are the `action.options` union `DataReactionActionOptions` (`packages/osos-shared/src/models/trigger-actions.ts`). Highlights a developer needs — the full request body for each type is in the Action options reference:

- **Value mapping (`EntryActionOptionsValue[]`)** — most data actions carry `values: {fieldName, valueType, value}[]`. `value` may be a fixed (manual) value, a **field** from the source entity, or a **pointer**. `@thisReference()` (surfaced in the builder as **This Entry**) refers to the entry that fired the event — write it with the parentheses, see Action options reference; **Current User** refers to the user who triggered the reaction. When mapping into

repeated/multi-value or block fields, only source fields whose data type and repetition limits are compatible with the target are offered.

- **`CreateDataEntry`** — `entitySource: {entityId?}` or `{fieldName?}` (the destination entity may be dynamic from a field) + `values`. The created entry's id is returned as `resultId` and its value plus a `thisEntry` reference is exposed as the action output.
- **`UpdateDataEntry` / block items** — `updateEntryMode` = `Self` (0) / `Reference` (1) / `Relation` (2); `entryToUpdate` selects the target (`mode`: `Fixed` / `Dynamic`, plus `relationId` / `targetEntityId` for relation mode). Block variants add `blockField` and (for update/delete) a `criteria` with `type` `FixedIndex` / `Field` / `Expression` and an `onFailure` policy.
- **`CallApi`** — `apiUrl`, `verb` (`GET` / `POST` / `PATCH` / `PUT` / `DELETE`), optional `headers`, `body` (`{custom, customBody}`), and `tokensMap` for token substitution. Success is `2xx`; the handler returns the full request/response (headers/body serialized) as the action output.
- **`SendEmail`** — `toRecipients` / `ccRecipients` / `bccRecipients` (each a `{sourceType, source:{type, value}}`), `template` (a template UUID or an inline `EmailTemplateValue` with `subject` / `body` / `tokens`), `tokensMap`, `from` (`System` / `ReactionExecutor` / `Custom` via `SendEmailFromModeEnum`), `link` (`{enabled, viewId}`), and `attachments`.
- **`SendNotification`** — `channels` (currently only `InApp=0` in `NotificationChannelTypeEnum`), `recipients`, `template` (UUID or inline `NotificationTemplateValue`), `module`, `tokensMap`, and `customRedirection` (`{enabled, viewId}`).
- **`SendSms`** — `recipients` (sources: `DataCollection` / `Entity` / `Static`), inline `template` (`{content, tokens}`), `tokensMap`, `link`.
- **`AnnounceEvent`** — `announcementId` plus a `schemaMapping` that maps trigger-context values into the announcement's inline/external entity fields.
- **`ChangeCurrentStepOwner`** — `type` / `source` / `ownerOptions`, a `fallbackOwner`, and `permanentChange` (default `false`).
- **`ArchiveDataEntry` / `DeleteArchivedEntry`** — `sourceMode` selects self vs relation vs query source, with `relation` (`{relationId?, referenceFieldId?}`) and `querySource`.
- **`ExecuteCustomAction`** `Since 1.8` — `customActionId` (the target custom action, which must be a `CustomEvent` reaction of type **Local** or **External**), `scope` (`Self` / `Reference` / `Relation` / `Expression` / `Pointer`, shown only when the target is contextual), and `parameters` (bindings into the target's Parameters, independent of `scope`). See Execute Custom Action for the full picture.
- **`SetPermission` / `RemovePermission`** `Since 1.10` — `resource` (`entitlementType` + `entitlementId`), `principal` (`kind` + `source`) and, for Set only, `level`. See Set / Remove Permission actions.
- **`NotifyZapier`** — no options (`NotifyZapierActionOptions` is empty).

Field-compatibility mapping (Create Entry)

For `CreateDataEntry` the builder only offers source fields whose data type **and** repetition limits are compatible with the target, and the runtime resolves each source shape into the target with a fixed reduction rule. The offered set and the per-scenario resolution depend on the target field's shape:

Target is a repeated multi-value field (standalone or in a single block) — only fields matching the target's data type and repetition limits appear:

Source field type	Resolution
Single-value (standalone / single block)	Target filled directly with the source value

Source field type	Resolution
Single-value in repeated block	First block repetition's value used
Repeated (standalone / single block)	First repetition of the source used
Repeated in repeated block	First block → first field repetition used
Multi-value (standalone / single block)	First value of the source used
Multi-value in repeated block	First block → first value used
Repeated multi-value (standalone / block)	First repetition → first value used
Repeated multi-value in repeated block	First block → first repetition → first value used

Target is a repeated single-value field (standalone or in a single block) — only multi-value source fields whose minimum is ≥ the target's minimum are offered:

Source field type	Resolution
Single-value (standalone / single block)	Single repetition filled from the source value
Single-value in repeated block	First block repetition's value used for the single repetition
Repeated single-value (standalone / block)	All source values used
Repeated single-value in repeated block	First block's values used
Multi-value (standalone / block)	First value used for the single repetition
Multi-value in repeated block	First block → first value used
Repeated multi-value (standalone / block)	First value from each repetition used
Repeated multi-value in repeated block	First block → first value from each repetition used

Target is a required single-value field (standalone or in a single block) — only required source fields with an exactly matching data type are offered, and they must be standalone or in a single block (not in a repeated block whose minimum repetition is zero):

Source field type	Resolution
Required single-value (standalone / single block)	Source value copied directly to the target
Required single-value in repeated block	First block's value used
Repeated (standalone / single block)	First repetition value used
Repeated in repeated block	First block → first repetition value used
Multi-value (standalone / single block)	First value used
Multi-value in repeated block	First block → first value used
Repeated multi-value (standalone / single block)	First repetition → first value used
Repeated multi-value in repeated block	First block → first repetition → first value used

Notable actions

Beyond the raw option shapes, a few actions have builder-facing behaviour worth calling out:

- **Send Email** offers three sender modes: **System** (default sender), **Reaction Executor** (the user who ran the reaction), or **Custom** (a static user, a variable user field, or a manually entered address). **Attachments** can be pulled from file fields on the entry (variable) or uploaded as a static file.

- **Generate File** (`GenerateFileFromTemplate`) renders a file from an entity template, with a configurable **retention period** (forever, or a number of days) and an optional custom file name (placeholders or manual). On completion the user is notified and can download the file from the notification.
- **Execute Workflow Step Action** (`ExecuteWorkflowAction`) auto-executes the current step's action and chains through subsequent automatable steps; **Move Workflow to Step** jumps a workflow instance to a chosen step.
- **Change Current Step Owner** reassigns the owner of the entry's current workflow step — commonly paired with `SlaTimeExceeded` to escalate an overdue step.
- **Activate/Deactivate User** (`UserActivationToggle`) activates/deactivates user entries in bulk (for example, offboarding when a record is archived).
- **Execute Function** (`ExecuteCustomFunction`) invokes a user-authored serverless function for logic beyond the built-in catalogue; the function must be provisioned and active, and its result is captured back into the context for later actions. `Since 1.10` later actions and chained reactions can bind its `success` , `message` and any key below its `result` — see Execute Function output in chained reactions.
- **Tokens bound to a step's original owner** `Since 1.10` — an email, notification or SMS token (or a custom variable) that reads the current step's **original owner** display value — `_metaCurrentSteps[n].originalOwner._resolved.mainTitle` , or `mainPicture` — used to render as an **empty string** whenever that owner was a **group** (or a data collection), while the very same expression worked in views and cards. It now renders the group's real title, for every current-step instance rather than only the active one.

Repeatable-block write fixes `Since 1.10`

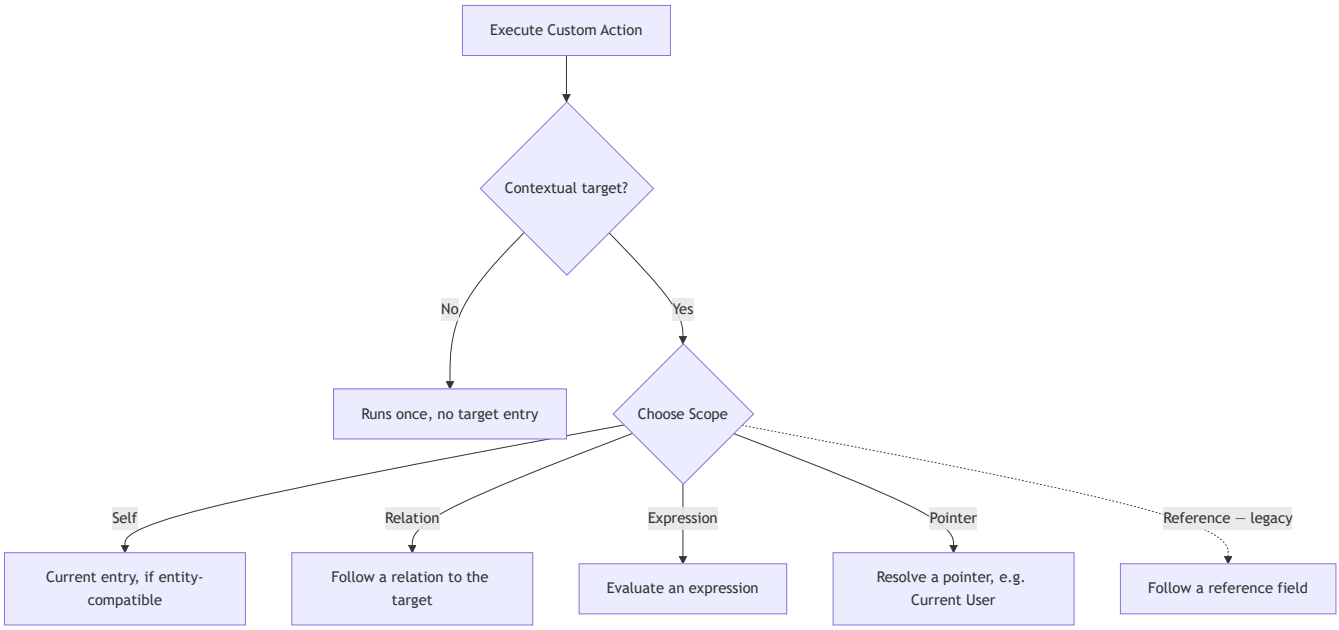
Two silent correctness bugs on repeatable-block writes were fixed this cycle — both returned a normal success response and showed up in the entry history as a completed write, so the only symptom was wrong data on the block:

- **A second Create Block Item action targeting the same block no longer overwrites the first.** A reaction with two `CreateBlockItem` actions appending to the **same** block field in one run used to leave only the second row — the first action's row was silently discarded. Both actions now append correctly, so a run that adds several rows to the same block ends up with all of them.
- **An Update Data Entry action writing a single sub-field of a repeatable block no longer collapses the block down to one row.** Writing just one sub-field across every existing row of a block used to replace the whole block with a single new row (and, if the block had a minimum row count, pad that one row back up to the minimum — so the row count could look unchanged while every row's content was wrong). The write now correctly applies to every pre-existing row, updating only the targeted sub-field on each.

Execute Custom Action `Since 1.8`

Execute Custom Action (`ExecuteCustomAction` , 106) lets one reaction run a **Local**- or **External**-level custom action on another entry (or entries, or none at all) — the automation equivalent of a user pressing a button, except it is driven by a trigger rather than a click. It can never target a **UI**-level custom action; those only run from a button (see Custom action type).

Configure it in up to three steps — the diagram shows how **Scope** is chosen when the target is contextual:



1. **Custom action** — pick one from a searchable dropdown of the callable custom actions in the current module, grouped **Local Actions** (owned by an entity in this module) and **External Actions** (owned by an entity in another module, labelled by that module's name). **UI** custom actions are never offered.
2. **Scope** — shown only when the chosen custom action is **contextual** (see Custom action type and Contextual); a non-contextual action has no target entry, so this step is skipped and the action runs exactly once. How the target entry (or entries) is found:
    - **Self** — run on the **current entry**. Offered only when the current entity matches the custom action's own entity — including when the current entity is a **Virtual Proxy** standing in for it, in which case the action runs on the entry the proxy represents.
    - **Relation** — follow a relation from the current entry to the target entry (or entries). Works both ways: an outgoing relation resolves to a single target, an incoming one to every entry that references back.
    - **Expression** — evaluate a code expression against the trigger context; every entry it resolves to must belong to a compatible entity or the action fails as a configuration error, so point it at exactly the entity you mean.
    - **Pointer** — resolve a context pointer (for example the current user) to the target entry.
    - **Reference** *(legacy)* — follow a reference field on the current entry to the target. Still fully functional for existing reactions and direct API use, but the builder no longer offers it for new configurations — use **Relation** instead.
3. **Parameters** — shown whenever the chosen custom action has **Parameters** enabled (its own Ask User toggle — see Custom action type), regardless of whether Scope is also shown. Bind each parameter the same way you map any action value: **Manual** (a fixed value), **Dynamic** (a field on the entry that fired this trigger), **Expression**, or **Pointer**. Binding is per call, so the same custom action can be invoked from two different Execute Custom Action actions with different Parameter values. An unbound parameter is not an error — it is simply passed empty.

What happens when it runs. The action resolves the target entries for the chosen scope (or none, for a non-contextual action), computes any bound Parameter values, then raises the very same custom-action event that a system button raises. The existing custom-action consumer picks it up and runs the target custom action **once per target entry, each in its own isolated context** — the target reaction sees only that entry's own data, not the calling reaction's context. The action's output records how many entries it dispatched to (`1` for a non-contextual dispatch).

Configuration failure modes to know:

- A custom action that **does not exist**, is **not of type Local or External**, or is **disabled** is rejected when you save the reaction — an Execute Custom Action can never target a **UI**-level custom action.
- **Self scope across entities is rejected** at save time — unless the current entity is a proxy for the custom action's entity, saving fails.
- **Reference/Relation scope without a target is a configuration error.** Saving is rejected if no reference field (or relation) is set.
- **Parameters are rejected outright if the target custom action doesn't have Parameters enabled** — you cannot bind values a non-parameterized action has no slot for.
- **Resolving no target entries is *not* an error.** If Relation resolves to nothing, the action succeeds and simply dispatches nothing; the reaction is not failed. If you expected work to happen, check the dispatched count in the execution history rather than looking for an error. Expression and Pointer are stricter: if what they resolve to isn't a compatible entity, the action fails outright rather than silently skipping it.

Set / Remove Permission actions `Since 1.10`

Two action types let a reaction manage **access permissions** — the same grants an administrator sets on the permission screens. **Set Permission** (`SetPermission`, 31) grants a level on a resource to users or groups; **Remove Permission** (`RemovePermission`, 32) revokes their grant. Typical uses are giving a request's assignee access to a report when a step is reached, or revoking a member's module access when their record is archived. Both write straight through data-service's permission route rather than through a command, so — unlike the data actions — they never leave a run waiting (see What to expect from the execution model).

Option	Meaning
`resource.entitlementType`	`System`, `Module`, `DataEntity`, `DataView`, `GenericView` (pages) or `Report`.
`resource.entitlementId`	The resource id — required for every type except `System`, where it must be absent.
`principal.kind`	`USER` or `GROUP`.
`principal.source`	Where the principals come from, configured like the User Activation Toggle source: a fixed reference, or a field path in the trigger context. One source may resolve to several principals.
`level`	**Set only** — the access level to grant.

A reaction may grant only these levels; `Restricted` and `None` are never grantable:

Resource	Grantable levels
`System`	Viewer, Developer, Editor, Admin
`Module`, `DataEntity`	Viewer, Developer, Editor
`DataView`, `GenericView`, `Report`	Viewer

A missing or malformed option — no resource, an unknown type, an `entitlementId` on `System` or missing elsewhere, no principal, a kind other than `USER` / `GROUP`, an invalid source, or (for Set) a missing or non-grantable level — is a save-time validation error (`missingRequired` / `invalidData`) naming the offending option path.

A reaction cannot grant more than the person who saves it. A reaction runs as the system, so saving one would otherwise be a privilege-escalation path. When a reaction is created or updated, every new or changed permission action is checked against the **saving user's own effective access** on that action's resource: Set needs at least the level it grants, Remove needs Admin. A refusal is a validation error with code `PERMISSION_ACTION_EXCEEDS_SAVER_ACCESS` and details `{entitlementType, entitlementId, requiredLevel}` (the builder shows a translated message); if the saver's

access cannot be read, the save fails closed. Unchanged permission actions are skipped only when the update is purely descriptive (name or description); any other change — triggers, criteria, other actions, re-enabling the reaction — re-checks them all, because the grant would fire in new circumstances. The check is made at save time only: an already-saved reaction keeps granting what it was saved with.

At run time, each resolved principal is handled on its own:

- **Set never lowers.** A principal who already holds a higher grantable level directly on the resource is skipped with `ALREADY_HIGHER`. **Remove** writes level `None`.
- A **user** refused on a `Module` or `DataEntity` grant because they have no system access yet (`USER_MUST_HAVE_VIEW_ACCESS_FIRST`) is first onboarded at System **Restricted** and the write is retried once. Groups are not onboarded.
- A resolved principal of the wrong kind fails with `PRINCIPAL_KIND_MISMATCH`, any other upstream error with `UPSTREAM_ERROR`; a source that resolves to nobody fails the whole action with `NO_PRINCIPALS`.
- The action output is `{applied[], failed[{principalId, code}], skipped[{principalId, code}]}`. The action counts as failed when `failed` is non-empty, the run continues, and later actions can read the output like any other.

Execute Function output in chained reactions `Since 1.10`

A function's answer can now drive what happens next. The **Execute Function** action exposes three output fields in the reaction schema — `success` (boolean), `message` (text) and `result` (an open JSON object) — so later actions in the same reaction, and reactions chained onto its success or failure, can bind them like any other action output. Because `result` has no declared shape, the builder's pickers let you bind **a key below it** (for example `result.customer.id`) instead of only the whole object, and the engine resolves that path inside the returned object when the reaction runs. Keep the function's return shape stable: the bindings are paths into it.

Action success criteria

Entry-affecting handlers extend `BaseEntryActionHandler`, which reads `options.actionSuccessCriteria` (`ActionSuccessCriteriaEnum`, default `Any=0`; `All=1`). With `Any`, the action is considered failed only when there are errors **and** no entries succeeded; with `All`, any error fails the action. On success, affected entries are recorded on the `ActionResult.effectedEntities` and the output is written back into the trigger context via `setActionOutput`, so subsequent actions can reference it.

Action options reference

Every action in a reaction's `actions[]` has the same envelope; only `options` changes with `actionType`:

```
{
  "id": "<uuid>",
  "normalizedName": "notify_requester",
  "name": {"en": "Notify requester"},
  "actionType": 100,
  "resultField": "",
  "options": { }
}
```

`resultField` is set at run time. Send it as an empty string (`""`); a `null` read back from the API is rejected on write.

One action per reaction. The API stores an array, but the save-time validation is written for a single action per reaction, and the builder edits one. Model a sequence as separate reactions chained with `ReactionSucceeded` (see

Chaining reactions).

Common building blocks

Value mappings — `EntryActionOptionsValue`

Used by `values` (create/update entry, block items), `parameters` (Execute Custom Action) and `schemaMapping` (Announce Event):

```
{"fieldName": "status", "valueType": 2, "value": {"id": "<category option id>"}}
```

`valueType` (`FieldValueTypeEnum`)	`value` holds	Example
`0` NotSet · `1` Default	nothing / the field default	—
`2` Manual	a literal in the **target** field's stored shape (a category is `{"id": "<option id>"}`, a reference is `{"dataEntity": "...", "dataEntry": "..."}`)	`true`, `"Open"`, `{"id": "…"}`
`3` Dynamic	a path in the trigger context: `value.<field>` on the entry that fired, or a full `scope.entries.<entity>.value.<field>` path	`"value.requester"`
`4` Expression	JavaScript evaluated with `value` (the **trigger** entry's value — also when the action updates another entry) and `scope`	`"(value.reopen_count || 0) + 1"`
`5` Pointer	an `@`-pointer, see below	`"@now()"`

A block sub-field is addressed by its **bare** name inside a block action (`"level"`, not `"escalation_event.level"`).

Pointers

A pointer value starts with `@`. A pointer written **with** parentheses runs against a scope that also provides `thisReference`; a pointer written **without** them runs as `scope.<name>()` against the default scope only.

Pointer	Resolves to
`@thisReference()`	`{dataEntity, dataEntry}` of the entry that fired. **The parentheses are required** — bare `@thisReference` resolves to nothing.
`@me` / `@me()` · `@current_user()`	the acting user (`SYSTEM` executor: the system user)
`@now()`	the current ISO datetime
`@today('+7d')`	a date-only string; the offset must be signed (`+` / `-`), units `d w M y h m s`

Token maps — `tokensMap` and `TokenSource`

Email, notification, SMS, Call API and custom-function actions fill `{{token}}` placeholders from `tokensMap`, an **object keyed by token name**:

```
"tokensMap": {
  "ticket_no": {"type": 1, "source": "scope.entries.tm_ticket.value.ticket_no"},
  "greeting":  {"type": 0, "source": "Hello"}
}
```

`type` (`TokenSourceType`)	`source` holds
`0` Fixed	the literal text
`1` Field	a context path — `scope.entries.<entity normalizedName>.value.<field>`; in a chained reaction `scope.entries._<triggerId>_event.value.<field>` or `scope.entries._<triggerId>__<actionId>_action.value.<field>` (hyphens in the ids replaced by `_`)
`2` Expression	JavaScript evaluated against the context

`type` (`TokenSourceType`)	`source` holds
`3` Pointer	an `@`-pointer

Rules enforced on save when `template` is **inline** (an object): the map's keys must equal the template's tokens exactly (`INVALID_TOKENS_MAP`), and every entry needs `type` (`Required parameter tokensMap.<i>.type is missing`). When `template` is a **stored template id**, nothing is checked on save: a token you do not map is sent as its literal `{{name}}`.

Recipients

```
{"sourceType": 0, "source": {"type": 1, "value": "value.requester"}}
{"sourceType": 1, "source": {"type": 0, "value": {"dataEntity": "GROUP", "dataEntry": "<group id>"}}}
{"sourceType": 4, "source": {"type": 0, "value": "ops@example.com"}}
```

Field	Values
`sourceType` — email (`EmailSourceType`)	`0` User · `1` Group · `2` DataCollection · `3` OrgUnit · `4` Manual (an address)
`sourceType` — notification (`StepOwnerType`)	`0` User · `1` Group · `2` DataCollection · `3` OrgUnit
`source.type` (`StepOwnerSourceType`)	`0` Fixed (a `ReferenceProperty`, or an address for Manual) · `1` Dynamic (a field path) · `2` PreviousOwner · `3` Expression

Email uses `toRecipients` / `ccRecipients` / `bccRecipients`; notification uses `recipients`. A wrong key name addresses nobody, silently.

Choosing the entry to write — `updateEntryMode` and `entryToUpdate`

`updateEntryMode`	`entryToUpdate`
`0` Self — the entry that fired	**omit it.** `{"mode": 0}` means *Fixed* and then requires `value` (`Required parameter options.entryToUpdate.value is missing`).
`1` Reference — an entry a reference field points at	required. `{"mode": 1, "fieldName": "<reference field>"}` (Dynamic) or `{"mode": 0, "value": {"dataEntity": "…", "dataEntry": "…"}}` (Fixed)
`2` Relation — entries reached through a relation	required, with `relationId` (and `targetEntityId` when the relation has several sources)

`actionSuccessCriteria`: `0` Any (default — fails only if no entry succeeded) · `1` All.

Options by action type

Data actions

```
// 10 CreateDataEntry — entitySource: {"entityId"} or {"fieldName"} (entity taken from a field)
{"entitySource": {"entityId": "<entity id>"},
 "values": [{"fieldName": "request", "valueType": 5, "value": "@thisReference()"},
            {"fieldName": "message", "valueType": 2, "value": "Auto-acknowledged"}]}

// 11 UpdateDataEntry — self
{"updateEntryMode": 0,
 "values": [{"fieldName": "closed_at", "valueType": 5, "value": "@now()"}]}

// 11 UpdateDataEntry — through a reference field
{"updateEntryMode": 1, "entryToUpdate": {"mode": 1, "fieldName": "parent_contract"},
 "values": [{"fieldName": "last_request", "valueType": 5, "value": "@thisReference()"}],
 "actionSuccessCriteria": 0}

// 12 ArchiveDataEntry — sourceMode 0 CurrentEntry · 1 RelatedEntries · 2 DataQuery
{"sourceMode": 1, "relation": {"relationId": "<relation id>"}, "querySource": null}

// 16 DeleteArchivedEntry — sourceMode 0 CurrentEntry · 1 Reference · 2 Relation · 3 DataQuery
{"sourceMode": 3, "relation": null, "querySource": {"dataEntity": "<entity id>", "dataEntry": "<saved query id>"}}

// 2 ArchiveDataEntity
{"entitySource": {"entityId": "<entity id>"}}
```

Repeatable-block actions

Block actions take the Update Data Entry options plus `blockField`; update and delete add a row `criteria`.

```
// 13 CreateBlockItem — sub-fields by BARE name
{"updateEntryMode": 0, "blockField": "sla_event",
 "values": [{"fieldName": "level", "valueType": 2, "value": 1},
            {"fieldName": "breached_at", "valueType": 5, "value": "@now()"}]}

// 14 UpdateBlockItem / 15 DeleteBlockItem (Delete has no "values")
{"updateEntryMode": 0, "blockField": "sla_event",
 "criteria": {"type": 3, "onFailure": {"type": 0}},
 "values": [{"fieldName": "acknowledged", "valueType": 2, "value": true}]}
```

`criteria.type`	`criteria.value`
`0` FixedIndex	positions and/or row `__id`s (a number, a string, or an array of them)
`1` Field	a field path that resolves to such a value
`2` Expression	an expression that evaluates to such a value
`3` LastRecord · `4` AllRecords	—

`onFailure.type`: `0` Skip · `1` Fail · `2` FallbackToLastOrFail · `3` FallbackToLastOrSkip.

Workflow actions

```
// 20 MoveWorkflowToStep
{"workflowId": "<workflow id>", "targetStepId": "<step id>"}

// 21 ExecuteWorkflowAction — one entry per step the reaction may automate;
//    only the entry matching the entry's CURRENT step runs, then the engine
//    keeps automating while the next step also has a matching entry.
{"actions": [{"workflowId": "<workflow id>", "stepId": "<step id>", "actionId": "<action id>"}]}

// 22 ChangeCurrentStepOwner — owner shape as a workflow step owner
{"type": 1,
 "source": {"type": 0, "value": {"dataEntity": "GROUP", "dataEntry": "<group id>"}},
 "ownerOptions": {"allowChangeOwner": true, "mode": 0, "accessPermission": 1, "selectionCriteria": 1},
 "fallbackOwner": {"allowChangeOwner": true},
 "permanentChange": false}
```

Execute Workflow Action is how a reaction "clicks" a workflow button — for example an SLA reaction that closes an entry left in *Resolved*. Listing `targetStepId` (Move) or `actions` (Execute) in the action's `overridableOptions` lets the user pick them at run time instead.

Messaging actions

```
// 100 SendNotification — channels: only 0 (InApp)
{"channels": [{"type": 0}],
 "recipients": [{"sourceType": 0, "source": {"type": 1, "value": "value.assignee"}}],
 "template": "<NOTIFICATION_TEMPLATE id>",
 "module": {"dataEntity": "DataModule", "dataEntry": "<module id>"},
 "tokensMap": {"ticket_no": {"type": 1, "source": "scope.entries.tm_ticket.value.ticket_no"}},
 "customRedirection": {"enabled": false}}

// 101 SendEmail — from.mode 0 System · 1 ReactionExecutor · 2 Custom (+ customFrom recipient)
{"toRecipients": [{"sourceType": 0, "source": {"type": 1, "value": "value.requester"}}],
 "ccRecipients": [], "bccRecipients": [],
 "template": "<EMAIL_TEMPLATE id>",
 "tokensMap": {"ticket_no": {"type": 1, "source": "scope.entries.tm_ticket.value.ticket_no"}},
 "from": {"mode": 0},
 "link": {"enabled": true, "viewId": "<view id>"},
 "attachments": [{"type": 1, "value": "value.contract_file"}]}

// 102 SendSms — recipients sourceType 0 Entity · 1 DataCollection · 2 Static; template is inline
{"recipients": [{"sourceType": 2, "source": {"type": 1, "value": "value.mobile_number"}}],
 "template": {"content": "Ticket {{ticket_no}} was resolved", "tokens": [{"name": "ticket_no", "location": ["conten
t"]}]},
 "tokensMap": {"ticket_no": {"type": 1, "source": "scope.entries.tm_ticket.value.ticket_no"}},
 "link": {"enabled": false}}

// 107 SendWhatsApp — an approved Meta template; parameters[i] fills {{i+1}}; recipients as Send SMS
{"recipients": [{"sourceType": 2, "source": {"type": 1, "value": "value.mobile_number"}}],
 "templateName": "ticket_resolved", "language": "en",
 "parameters": [{"type": 1, "source": "scope.entries.tm_ticket.value.ticket_no"}]}
```

`template` may also be inline instead of a stored template id — email `{"subject", "body", "tokens"}`, notification `{"name", "title", "content", "icon", "tokens"}`, each token `{"name", "location": [...]}` (for example `["subject", "body"]` or `["title.en"]`). Email `attachments[].type`: `0` Fixed (a stored file) · `1` Dynamic (a file field path). A Fixed attachment is a stored file the reaction now holds a reference to: if its temporary upload expired before the save, the save fails with `FILE_EXPIRED` (see Files in reaction options).

Send WhatsApp options `Since 1.10`

Option	Holds
`recipients`	the same recipient shapes as Send SMS
`templateName`	the name of an **approved** Meta template of the space's WhatsApp Business account
`language`	the template's language code, e.g. `en` or `ar`
`parameters`	token sources (Fixed, Field, Expression or Pointer, as in Token maps); item `i` fills the template body's `{{i+1}}`. May be empty.

- **Pick the template from the list.** The builder lists the space's approved templates live from `GET /api/reaction/v1/whatsapp/templates` — `[{name, language, category, bodyParameterCount}]`, never an `AUTHENTICATION`-category template, because the action always sends as category `other`. Calling it needs system View access or higher (an end user with only the `Restricted` floor is refused). It answers `409 whatsapp_not_configured` when the space has no business account id or access token, and `502 whatsapp_provider_error` with Meta's error code (never Meta's message or the token) when Meta fails. The form says which of these stopped the list from loading, flags a saved template that is no longer approved, and drops saved parameters beyond the picked template's variable count. The account settings are the `whatsapp` block of System Settings.
- **The save does not check the template against Meta.** A wrong name, language or parameter count is not caught when the reaction is saved; give `parameters` exactly `bodyParameterCount` items.
- **What is sent.** An empty value is sent as `''` (positions are kept, since Meta rejects a missing parameter); a localized value is sent in the tenant's default language, not in the template's `language`; a masked sensitive value is sent as `****`; newlines, tabs and runs of five or more spaces are collapsed to one space and the value is trimmed (Meta refuses them, error 132018). One message request goes out for all recipients; with no phone number resolved, nothing is sent and the action still succeeds.
- **Output:** `{templateName, recipientsPhoneNumbers}`, readable by later actions and chained reactions.

Integration and automation actions

```
// 90 CallApi — verb "GET" | "POST" | "PATCH" | "PUT" | "DELETE"; success = any 2xx
{"apiUrl": {"value": "https://api.example.com/tickets/{{ticket_no}}"},
 "verb": "POST",
 "headers": {"value": "{\"Content-Type\": \"application/json\"}"},
 "body": {"custom": true, "customBody": {"value": "{\"ref\": \"{{ticket_no}}\"}"}},
 "tokensMap": {"ticket_no": {"type": 1, "source": "scope.entries.tm_ticket.value.ticket_no"}}}

// 103 GenerateFileFromTemplate — retentionPeriod in days (omit = keep)
{"templateId": "<template id>", "retentionPeriod": 30,
 "fileOutputName": {"type": 2, "source": "'Contract-' + scope.entries.contract.value.ref"},
 "downloadFile": false}

// 104 ExecuteCustomFunction — resolved tokens reach the function as payload.parameters.<key>
{"functionId": "<function id>",
 "tokensMap": {"customer": {"type": 1, "source": "scope.entries.order.value.customer"}}}

// 105 AnnounceEvent
{"announcementId": "<announcement id>",
 "schemaMapping": [{"fieldName": "order_ref", "valueType": 3, "value": "value.ref"}]}

// 106 ExecuteCustomAction — scope 0 Self · 1 Reference · 2 Expression · 3 Relation · 4 Pointer
//     Reference: entryToRun {mode, fieldName | value} · Relation: entryToRun {relationId}
//     Expression: "expression" · Pointer: "pointer". Send only the chosen scope's setting,
//     and no scope at all when the custom action is not contextual.
{"customActionId": "<CustomEvent reaction id>", "scope": 3,
 "entryToRun": {"relationId": "<relation id>"},
 "parameters": [{"fieldName": "reason", "valueType": 2, "value": "Escalated"}]}

// 200 NotifyZapier
{}
```

Call API: what the platform allows `Since 1.8`

A Call API action is a server-side HTTP request built from your configuration, so the platform bounds it (security finding N-14):

Rule	What you see
HTTP(S) only	Any other scheme, or a URL that does not parse, fails the action with `CALL_API_DESTINATION_BLOCKED`.
No private or internal destinations	An address in a private, loopback, link-local (cloud metadata included), CGNAT, multicast or reserved range, IPv4 or IPv6, fails the action. Host names are checked when they resolve, so a public name pointing at a private address is refused, and every redirect (at most 5) is checked again. Proxy settings are ignored. Only an operator can open a private destination, through the `CALL_API_ALLOWED_PRIVATE_DESTINATIONS` allow-list.
30-second limit	The **whole** exchange must finish within `CALL_API_TIMEOUT_MS` (default 30 s); a slowly trickled response times out too.
10 MiB size cap	Request and response bodies, as recorded in the run's history, are capped at `CALL_API_MAX_RESPONSE_BYTES` (default 10 MiB); a larger response fails the action.
Tokens cannot add headers	The `headers` template is read as JSON first, then tokens are filled into each header name and value separately, so a token's value can never add or replace a header.
Platform headers win	`x-space-id` and `Content-Type: application/json` are set by the platform and override yours.
Masked values `Since 1.10`	A call whose URL, headers or body would carry a masked sensitive value fails instead of sending `****` (see Sensitive-data access policy).

Reserved header names `Since 1.10.` `x-shared-key`, `x-app-id`, `x-space-id` and `x-service` (any case) are the platform's own. A `headers` template that names one is refused on save with `400 RESERVED_HEADER_NAME`. A reaction

saved before this check still runs: the reserved header is dropped at run time.

Users and permissions

`source.type` (`UserActivationToggleSourceTypeEnum`): `0` Fixed (a `ReferenceProperty`) · `1` Field · `2` Expression · `3` Pointer (`value` is the path, expression or pointer).

```
// 30 UserActivationToggle — type 0 Activate · 1 Deactivate
{"source": {"type": 1, "value": "value.employee"}, "type": 1, "actionsSuccessCriteria": 0}

// 31 SetPermission (32 RemovePermission: the same without "level")
{"resource": {"entitlementType": "DataView", "entitlementId": "<view id>"},
 "principal": {"kind": "USER", "source": {"type": 1, "value": "value.assignee"}},
 "level": 0}
```

`entitlementType`: `System` (then `entitlementId` must be `null`) · `DataModule` · `DataEntity` · `DataView` · `GenericView` · `Report`. `level`: `0` View · `0.5` Developer · `1` Edit · `2` Admin, limited to the levels in Set / Remove Permission actions.

Save-time validation errors

Reaction create/update validates the options and rejects the save, naming the offending path:

Error	Status	Cause
`MUST_AUTOMATE_AT_LEAST_ONE_STEP`	400	Execute Workflow Action with an empty or missing `actions` (and no `actions` override)
`TRIGGERS_CAN_NOT_AUTOMATE_LAST_STEP`	400	an `actions[]` entry on the workflow's last step
`WORKFLOW_NOT_EXIST` · `WORKFLOW_STEP_NOT_EXIST` · `WORKFLOW_ACTION_NOT_EXIST`	404	an id in Move / Execute options that is not on the entity
`Required parameter options.entryToUpdate[.value | .fieldName | .relationId] is missing`	400	see Choosing the entry to write
`Required parameter tokensMap.<i>.type is missing` · `INVALID_TOKENS_MAP`	400	see Token maps
`Required parameter options.customRedirection.viewId is missing`	400	Send Notification with `customRedirection.enabled: true` and no `viewId`
`Required parameter options.entryToRun[.relationId | .fieldName | .value]`, `options.expression`, `options.pointer`	400	Execute Custom Action scope without its setting
`missingRequired` / `invalidData` on `options.resource`, `options.principal`, `options.level`, `options.source`	400	Set / Remove Permission and User Activation Toggle options
—	422	action type `0` or `1` (retired)
`missingRequired` / `invalidData` on `eventConfig.entityId`, `eventConfig.triggerId`, `spaceScope`	400 / 422	a `ReactionSucceeded` / `ReactionFailed` reaction that could never fire, or with a space scope other than Triggering Space. `Since dev` — see below
`RESERVED_HEADER_NAME`	400	a Call API `headers` template names a reserved platform header. `Since 1.10` — see Call API
`FILE_EXPIRED`	400	the options reference a temporary upload (an email attachment, for example) that expired before the save. `Since 1.10` — see Files in reaction options
`STORAGE_SERVICE_UNAVAILABLE`	503	file storage could not be reached while the save adds a file; retry. `Since 1.10`
`REACTION_AUTHORING_FORBIDDEN`	403	you lack Developer-or-higher access on the reaction's entity (create, update and delete). `Since 1.8` — see Who may author a reaction

Event-config shapes used with these actions

`eventType`	`eventConfig`
`1` DataEntryUpdated	`{"fields": [{"fullNormalizedName": "sla_breached"}]}` — empty matches any change
`3` WorkflowActionExecuted · `5` WorkflowStepExited	`[{"steps": [{"workflowId": "…", "stepId": "…", "actions": ["<action id>"]}]}]`
`4` WorkflowStepReached	`[{"steps": [{"workflowId": "…", "stepId": "…"}]}]`
`6` Timer	`{"timerOptions": {"repeat": {"expression": "*/5 * * * *", "options": {"timezone": "Asia/Dubai"}}}}`
`8` ReactionSucceeded · `10` ReactionFailed	`{"triggerId": "<reaction id>", "entityId": "<its entity id>"}` — **both** are required to match, and `triggerId` must be a reaction defined on `entityId`. On 1.10 a reaction without them is stored and never fires; `Since dev` the save is refused (see below). Set `spaceScope` to **Triggering Space** only, see below
`100` SlaTimeExceeded (reaction `type: 1`)	`[{"occurrencesConfig": {"startingAfter": 0, "repeatAfter": 0, "repetitions": 0}, "steps": [{"workflowId": "…", "stepId": "…"}]}]`

Give a chained reaction the Triggering Space only. A `ReactionSucceeded` / `ReactionFailed` reaction runs on the original run's stored context and entry, which belong to the space that run executed in. Give it `"spaceScope": {"rootSpace": false, "currentSpace": true, "otherSpaces": false}`.

- **On 1.10 the platform does not enforce this.** Any wider scope is expanded like for other reactions, and the chained reaction runs once per resolved space — each time with the original space's entry. With *Primary + Triggering Scopes*, a run that started in a non-root space fires the chained reaction twice; with *All Scopes* it runs in every space.
- **`Since dev` it is enforced.** The chained reaction runs once, in the space the watched run ran in, whatever scope it stored (existing reactions need no change), and saving one with any other `spaceScope` is refused with a `422` naming `spaceScope`; an absent `spaceScope` is accepted.

More chained-reaction checks at save `Since dev`:

Check	Result
`eventConfig.entityId` or `eventConfig.triggerId` missing	`400`, naming the missing one
`triggerId` is not an existing reaction, or is not defined on `entityId`	`422`, naming `eventConfig.triggerId`
The lookup of the watched reaction itself fails (a database error)	the save fails with that error, not as an invalid `triggerId`
Several reactions in one save come from the same relation	every one is validated (only the first used to be)
An Execute Workflow Action whose `actions` are asked of the user, followed by other actions or reactions in the same save	only that action skips validation (the rest of the save used to skip it too)

Also `Since dev`: when reactions on two entities watch the same reaction, each entity runs only the chained reactions **it** defines; before, each one ran twice, once with the other entity.

For `SlaTimeExceeded`, `startingAfter` and `repeatAfter` are seconds and `repetitions` counts **extra** occurrences after the first: `0` fires once; `1` with `repeatAfter: 0` fires twice at the same instant. On builds without the scheduler fix, an occurrence that is already due when scheduled (`startingAfter: 0`) is never delivered — set `startingAfter` to `120` there.

Execution pipeline

For a normal (non-timer) event the flow is:

1. **Bus event handler** (`handlers/events/**` , extending `BaseTriggerEventHandler`) receives a domain event, resolves the defining `DataEntity` from `data-service` , and loads candidate triggers: `dataReactionRepository.find({ 'definingEntity.dataEntry': entityId, disabled: false })` .
2. It computes **direct triggers** (`sourceEntity.type !== Relation`) and **related triggers** (see Related-entity triggers). Invalid triggers (`validations.isValid === false`) are skipped.
3. Affected entries are prepared once as in-memory query sources shared by every runnable trigger (a temporary collection before 1.10 — see Criteria and query filter); then, for each trigger and each entry, the handler calls `EventsHandler.handle(...)` . `spaceScope` is expanded via `SpaceService.getDataReactionSpaceScope` so the reaction can run across the configured spaces. Entries whose `store` differs from the entity's write store are skipped.
4. **`EventsHandler.handle`** validates the event type matches, runs the event matcher, merges any `query` into `criteria` , builds a `TriggerContext` , applies the criteria filter, then executes `trigger.actions` in order.
5. Each action runs through `EventsHandler.actionHandler` , which enforces loop protection, records Prometheus metrics (`osos_reaction_duration_seconds` , `osos_reaction_executed_total` — `Since 1.8`), and invokes the handler.
6. Results are aggregated into a **`TriggerHistory`** record (per-action `ActionResult` s), stored via `TriggerHistoryRepository` . The full serialized context is cached in Redis under `trigger:<triggerId>:context:<historyId>` for **600s (10 min)** — `Since 1.10` only when a reaction is listening for this reaction's result, and compressed; see Trigger-context storage — and a **`TriggerExecutedEvent`** is published on the bus.

Overall trigger status is logged as fully successful, partially successful, or failed based on the count of failed `ActionResult` s.

The trigger context

`TriggerContext` (`handlers/trigger.context.ts`) is the shared state passed to every action. It holds the firing `entity` / `entry` , all `entities` / `entries` (including AskUser and related-source entities), and a per-action `actions` map of `input` / `output` . Notable behaviours:

- **Path resolution** — `getValueByPath` / `getFieldByPath` resolve dotted paths, normalising legacy paths (`value.name` , or a bare field name) into the canonical `scope.entries.<entity>.value.<field>` form, and supporting **relation traversal** with the `.:relation.` / `.:relation_<entity>.` marker to hop across reference fields.
- **Action chaining within one trigger** — after an action succeeds, its output is stored via `setActionOutput` , so a later action in the same reaction can read the previous action's output (e.g. a created entry's id).
- **Executor identity** — `calculateIsAdminModification` sets `isAdminModification` when `executorType` is `SYSTEM` / `MANAGER` (or absent, for backward compatibility). For `SYSTEM` , the user id is forced to the system user and the delegatee is cleared, so writes run with system privileges.
- **`triggerContextId`** — on the event config (`BaseEventConfig.triggerContextId`), selects a specific context entity for the actions to operate on instead of the entity that fired. *(In 1.8, this moved from individual action options to the event config — the event owns the context, not the action.)* **Since 1.8.**

Execution scopes (spaces)

A reaction's `spaceScope` controls which spaces it executes in — expanded at dispatch by `SpaceService.getDataReactionSpaceScope`. In the builder the **Execution Spaces** options are **Root Space**, **Triggering Space**, **Root Space and Triggering Space**, **All Spaces Except Root** and **All Spaces** (`rootSpace` / `currentSpace` / `otherSpaces` combinations). This lets a single reaction fan its actions across the configured spaces instead of only the one where the event fired. A reaction chained onto another reaction's success or failure is the exception: give it the Triggering Space only — on 1.10 a wider scope runs it once per space on the original space's entry, and `Since dev` the scope is ignored at run time and refused at save (see Action options reference › Event-config shapes).

What to expect from the execution model `Since 1.10`

`Since 1.10` the engine no longer holds a worker open while it waits for `data-service` to finish an action: a run is recorded as it goes and picked up again from where it was left. None of that machinery is something you configure, but three of its consequences are yours to design around.

- **Do not rely on ordering between entries in one batch.** When a single action fans out over several entries, its commands can now be dispatched **together**, and may complete **in any order**. Ordering *between actions* in a reaction is unchanged — action 2 still runs after action 1 — but "entry A is written before entry B" was never promised and is now visibly untrue. Two commands in the same batch targeting the **same** entry are rejected before dispatch rather than raced.
- **A run survives a service restart and records its history — but it does not finish.** If the automation service restarts mid-run (a deploy, a crash), the run is recovered and its execution history row is written for the actions that had already completed, instead of the run vanishing without a trace. Recovery **writes that tail and stops**: the actions it had not reached do **not** run, because reaction actions are not safe to replay blindly. Write later actions so they tolerate an earlier one simply never having happened — the same tolerance a mid-chain failure already demands.
- **A `ReactionSucceeded` / `ReactionFailed` trigger can fire twice for one recovered run.** Recovery republishes the "this reaction finished" event, so a reaction chained onto another reaction's success or failure may be triggered more than once for the same logical run. Make the chained reaction safe to run twice — have it check for the record it would create, or make its write idempotent — rather than assuming exactly one firing.

Nothing about how you **configure** a reaction changed: the same trigger events, the same action options, the same execution-history document, and the same metric names.

Chaining reactions

Reactions compose through events rather than direct calls:

- Every completed execution publishes `TriggerExecutedEvent`. Reactions bound to `ReactionSucceeded` **(8)** / `ReactionFailed` **(10)** are matched by `ReactionSucceededEventConfig` / `ReactionFailedEventConfig` (`entityId` + `triggerId`), so you can wire "when reaction X on entity Y succeeds, do Z" (or a failure-handling branch). A follow-up reaction can read the triggering reaction's inputs and outputs in its own action settings.
- The `AnnounceEvent` **(105)** action publishes a named announcement; reactions bound to `EventAnnounced` **(11)** (matched by `entityId` + `announcementId`) run in response. `schemaMapping` carries data from the announcing context into the announcement's schema, which downstream reactions can read. This decouples the publisher from its subscribers, even across entities. **Since 1.8.**
- Data-modifying actions (create/update/archive entries) emit the ordinary `DataEntryCreated` / `Updated` / `Archived` events, which in turn can fire other reactions — the most common (and

easiest to loop) chaining path.

The triggering reaction's context is preserved for a limited time (see the Redis cache above) so a chained reaction can reuse it.

A `ReactionSucceeded` / `ReactionFailed` reaction must name **both** the watched reaction (`triggerId`) and the entity that defines it (`entityId`), and should be scoped to the **Triggering Space** only. On 1.10 neither rule is checked: a reaction missing one is stored and never fires, and a wider scope runs the chained reaction once per space. `Since dev` both are refused at save, the chained reaction always runs once in the space the watched run ran in, and when reactions on two entities watch the same reaction each entity runs only its own — see Action options reference › Event-config shapes.

A run the tenant's monthly run limit refused publishes no "this reaction finished" event, so a `ReactionFailed` reaction does **not** fire on a refusal `Since 1.10`.

Trigger-context storage `Since 1.10`

What a chained `ReactionSucceeded` / `ReactionFailed` reaction reads is the parent run's serialized trigger context — often a few hundred kilobytes — kept for 10 minutes. Since 1.10 the engine writes it **only when an enabled reaction is listening for that reaction's result**, and stores it gzip-compressed. If the engine cannot tell whether a listener exists, it stores the context as before. Two consequences for design:

- The listener must exist **and be enabled before the parent run finishes**. A `ReactionSucceeded` / `ReactionFailed` reaction created or re-enabled afterwards finds no context for runs that already finished — the same as when the 10-minute window has expired.
- A stored context that cannot be decoded is treated as *no context*: the chained reaction is skipped rather than run on missing fields.

The values history redaction masks are kept on the same condition, so a chained reaction redacts consistently with its parent. Operators can turn both behaviours off — `OSOS_TRIGGER_CONTEXT_LISTENER_GATE` (store every run's context again) and `OSOS_TRIGGER_CONTEXT_GZIP` (store it uncompressed), both on by default. A reaction service older than 1.10 cannot read the compressed form, so during a rolling upgrade compression is kept off until every reaction-service pod runs 1.10.

Loop protection

Because data actions re-emit data events, a reaction can re-trigger itself. `EventsHandler.actionHandler` guards this: it counts existing `TriggerHistory` rows for the same `traceId` + `spaceId` + active `triggerId` + `action.normalizedName`, and if the count **exceeds** `trigger.loopThreshold` it aborts that action with a "preventing a loop" `ActionResult` (`success:false`).

`loopThreshold` defaults to `0`, meaning **the action runs once per trace**. Raise it deliberately to allow a bounded number of self-re-executions. The correlation/trace id is propagated (and suffixed per-entry when a single event affects multiple entries), which is what ties an execution chain together for this count. If an expected action silently did not run, check whether the loop guard skipped it.

Workflow-step events are counted per step `Since 1.4`. For `WorkflowStepReached` and `WorkflowStepExited` reactions the count is taken per workflow step (each history row records the step): such a reaction legitimately runs once for every step a trace passes through, and a real loop has to revisit a step, so the guard still bounds it. Other event types are counted over the whole trace, as before.

Related-entity triggers

A reaction whose `sourceEntity` is set fires from a **different** entity than its `definingEntity`. `DataReactionSourceEntity` has a `type` (`TriggerSourceEntityTypeEnum`: `Self=0`, `Relation=1`, `AbsoluteEntity=2`) and a `sourceId`. `BaseTriggerEventHandler.findRelatedTriggersUsedAsSource` resolves these in two directions:

- **Incoming relations** — for the entity that fired, find relations targeting it, then find reactions whose `sourceEntity.sourceId` matches a relation id; fetch the referenced entries and run each reaction's actions against them, exposing the source entry/entity in the context.
- **Outgoing relations** — find reactions whose `sourceEntity.sourceId` regex-matches `:<entityId>` (an in-memory relation pointing at this entity), locate entries on the defining entity whose reference field points at the fired entry, and run for each. *(Example in code: a `User` entity references `OrgUnit`; when `OrgUnit` fires, the `User` reaction runs for every user in that org unit.)*

The reaction schema for the builder UI (which entities/fields are reachable in a trigger's context) is served by `SystemReactionController` (see below).

Entity isolation `Since 1.8`

A reaction runs **as its own entity**: its actions resolve field paths and write through that entity's definition. Since 1.8 the engine enforces this explicitly — **a reaction will not run against an entry that belongs to a different entity**.

- An entry whose entity differs from the reaction's own entity is **skipped**, with a warning in the service log, instead of being run through the wrong entity's field definitions.
- For related-entity triggers, the entries pulled in through a relation are filtered **per trigger** down to that trigger's own entity *before* they are fetched. This is what matters for a **reference field that accepts more than one entity**: previously every reaction sharing that relation received every referenced entry, whatever entity it actually belonged to.
- A trigger whose defining entity cannot be resolved — a deleted entity, say — is **skipped with a warning** instead of failing the whole run, so the other triggers listening to the same event still execute.

No new error is raised for any of this. A foreign entry or an unresolvable entity is a logged skip, not a failure, so nothing appears in the reaction's execution history — the reaction simply never runs for that entry. If a reaction seems not to have fired for some entries, look for skip warnings in the reaction-service log rather than for a failed action result.

REST API

All reaction endpoints require the `Auth2` authentication strategy and are scoped by the injected `spaceId`.

`DataReactionController` — CRUD

Source: `controllers/data-reaction.controller.ts`.

Method & path	Purpose
`POST /v1/reaction/{entity}`	Create a reaction on `{entity}`. Body excludes `updateIdentifier`, `creationInfo`, `updateInfo`, `validations`, `definingEntity` (server sets `definingEntity` from the path). Rejected with `ImplicitReactionCreateException` if it would create an implicit reaction.
`GET /v1/reaction/{entity}`	List reactions for an entity (LoopBack `filter`), each row also carrying its execution/failure counts — see Execution/failure counts and run history. `Since 1.10`

Method & path	Purpose
`GET /v1/reaction/{entity}/count`	Count for an entity (LoopBack `where`).
`GET /v1/reaction/{entity}/{id}`	Get one, validated to belong to `{entity}`.
`GET /v1/reaction/{entity}/{id}/history`	One page of a reaction's recorded runs, newest first, as slim rows — **system administrators only** — see Execution/failure counts and run history. `Since 1.10`
`PATCH /v1/reaction/{entity}/{id}`	Update. **Optimistic concurrency**: the body's `updateIdentifier` must equal the stored one or it throws `UpdateIdentifierConflictException`; the server then rotates `updateIdentifier` to a new UUID.
`DELETE /v1/reaction/{entity}/{id}`	Delete; blocked with `ImplicitReactionDeleteException` for implicit reactions.
`GET /v1/reactions`	List all reactions (cross-entity).
`GET /v1/reactions/module/{moduleId}`	List the reactions defined on any entity within a module, optionally narrowed by a LoopBack `filter`. `Since 1.8`
`GET /v1/reactions/custom-actions?moduleId={moduleId}`	The **callable-action discovery** endpoint behind the Execute Custom Action picker: lists the module's callable **Local**/**External** custom actions, grouped `local` / `external`. `moduleId` is required. `Since 1.8` — see Execute Custom Action.
`GET /v1/reactions/{id}`	Get any reaction by id.
`GET /v1/reactions/count`	Count all.

Non-existent entities return `NotFoundException('ENTITY_NOT_FOUND')`; a reaction that doesn't belong to the path entity returns `REACTION_NOT_FOUND`. Writes go through `ReactionWriterService` (create/update/delete), not the repository directly.

Who may author a reaction `Since 1.8`

Every reaction write (create, update, delete) needs two things: the tenant's license must include administration, and **you** must have **Developer access or higher on the reaction's entity** — the level that lets you edit the entity.

- **Refused:** `403 REACTION_AUTHORING_FORBIDDEN`, with `details: {entityId}`. If your access cannot be read, the write is refused too.
- **Imports follow the same rule.** Each reaction in an imported file is checked against the person confirming the import; one they may not author is skipped and the rest of the import still lands. An update is judged on the entity the stored reaction belongs to, not the one in the request.
- **Delegation:** a reaction created by a delegate, imports included, is judged on the delegated permissions.
- **Not checked** for the platform's own writes: service-to-service calls, implicit reactions, migrations, tenant provisioning and package installs.

Files in reaction options `Since 1.10`

A reaction that holds file ids in its options (a fixed email attachment, for example) now keeps a storage reference to them, like an entry does: a temporary upload is made permanent when the reaction is saved, and released when the reaction (or its entity) is deleted. Two save errors come with it:

Error	When	What to do
`400 FILE_EXPIRED`	The save names a temporary upload that expired before the save made it permanent. `details` lists `{fileId, path, reason}` per file — `path` such as `actions[0].options.attachments[1].value`, `reason` a localized "upload it again".	Upload the file again and save.
`503 STORAGE_SERVICE_UNAVAILABLE`	File storage could not be reached and the save **adds** a file. A save that adds no new file goes ahead.	Retry.

A save refused this way leaves nothing behind — no timer is scheduled for it.

Execution/failure counts and run history `Since 1.10`

The Reactions tab shows, per reaction, how many runs have succeeded and how many have failed **since the reaction was last modified**. For system administrators either counter opens a side panel listing the individual runs behind it; for other users the counters are plain text.

`GET /v1/reaction/{entity}` returns every ordinary `DataReaction` field plus five read-only, computed ones:

Field	Type	Meaning
`executedCount`	number	Every recorded run of this reaction, whatever its outcome — except runs the monthly run limit refused. All-time.
`succeededCount`	number	Runs in which at least one action ran and none of them failed — a partially-failed run does not count here. All-time.
`failedSinceLastChangeCount`	number	Failed runs recorded since the reaction was last modified (`statsSince`, below). A run counts once, however many of its actions failed.
`statsSince`	string (ISO-8601)	When the failure-count window opens — the reaction's last-modified timestamp (or its creation timestamp, for a reaction never modified since).
`refusedCount`	number	Runs the monthly run limit refused: recorded, but no action ran. Not part of `executedCount`. All-time. `Since 1.10`

These counts are computed fresh on every read rather than stored, so they can never drift from the underlying run history.

`GET /v1/reaction/{entity}/{id}/history` pages the runs behind a counter, newest first, and takes:

Parameter	Meaning
`outcome`	`failed`, `succeeded`, `refused` (`Since 1.10`), or `all` (default). Any other value is a validation error.
`from` / `to`	Inclusive lower / exclusive upper bound on the run timestamp (ISO-8601). Not automatically clamped to `statsSince` — the dialog pre-fills `from` with it for the failure outcome, but a caller may widen it.
`before`	Pagination cursor from a previous page's `nextBefore`.
`limit`	Page size; defaults to 10, capped at 100.

The response (`items[]`, `hasMore`, `nextBefore?`) has no total count by design — the reaction's badge already gives the total. The endpoint is restricted to **system administrators** (System Admin access with no `None` grant), and each item is deliberately slim: `{id, timestamp, status, relatedTo}`, where `status` is `succeeded`, `failed` or `none` under the same rule as the counters, or `refused` for a run the monthly run limit stopped and `relatedTo` carries the entry and its resolved title. The runs are those in the caller's current space, whereas the counts span every space. The full stored record of a run — every action's input and output — is served only by the detail route of the Reaction Logs API; this replaces the earlier design in which every row was the full record and any caller with reaction access could read it.

System-wide Reaction Logs API `Since 1.10`

`ReactionLogController` (`controllers/reaction-log.controller.ts`) serves every recorded reaction run in the caller's space — the API behind **Administration → Monitoring → Reaction Logs** and the Reactions-tab side panel. Both routes are **system-administrator only**, because the detail route returns each action's stored payload.

Method & path	Purpose
`GET /v1/reaction-logs`	One page of runs, newest first. `skip` (any depth) and `limit` (default 20, max 100).
`GET /v1/reaction-logs/{id}`	One run in full, plus its previous / next neighbours under the same filter.

Both routes take the same filters: `reactionId`; `eventName` (an `EventTypeEnum` key); `actionType` (an `AutomatedActionTypeEnum` key — a run matches when *any* of its actions has that type); `status` = `failed`, `succeeded`, `refused` or `all` (default); `from` (inclusive) / `to` (exclusive) ISO-8601 instants; `doneBy` (a user id or `SYSTEM`); `search`; and `order` = `desc` (default) or `asc`. `search` is case-insensitive in any tenant language and matches the reaction's current name, its entity or module name, the actor's full name, or an exact run or entry id; under 2 characters it is ignored. Malformed parameters are validation errors.

It reads the existing `TriggerHistory` records — there is no second store — so the Logs page, the counters and the side panel always agree on what failed and what succeeded. The total is counted under a 3-second budget and returned as `null` if that runs out; `hasMore` is always reliable. An entry with no index row comes back without `_resolved`, so a client can tell *untitled* from *deleted or unknown*.

Monthly run limit and refused runs `Since 1.10`

A tenant's plan can cap how many reaction runs it may start per **UTC calendar month** (`maxReactionRunsPerMonth`). The cap is enforced inside the automation service, so it is **not an HTTP error**: nothing the user did fails — the run simply does not happen.

- **A refused run** is still recorded, with no action results and a `refusal` of `{code: "REACTION_RUN_CAP_EXCEEDED", limit, used, period}`. No action runs, and no "this reaction finished" event is published, so a `ReactionFailed` reaction does not fire on it.
- **Where to see it.** Its status is `refused` in the run-history rows and the Reaction Logs API; filter with `outcome=refused` / `status=refused`; `GET /v1/reaction/{entity}` reports `refusedCount` apart from `executedCount`. The 1.10 web client does not yet show a refused counter or filter.
- **What counts.** Every run started by an event, a timer or a custom action. **Platform (implicit) reactions never count**, and a redelivered message is counted once. A limit of `0` refuses every run. If the counter cannot be read, the run goes ahead: the cap meters, it does not guard.

The limit is part of the tenant's plan. See Plan limits.

`SystemReactionController` — builder schemas

Used by the reaction builder to know what data is reachable in a trigger's context. Source: `controllers/system-reaction.controller.ts`.

Method & path	Purpose
`GET /v1/schema/{entity}/{trigger}`	Full reaction schema: event entity schema, per-action entity schemas, `updateContextIds`, and source entities.
`GET /v1/schema/{entity}/event/{eventType}/{newTriggerId}`	Event entity schema for a given `eventType`.

Method & path	Purpose
`GET /v1/schema/{entity}/event/{sourceType}/{sourceId}/{newTriggerId}/source`	Source entity schema for a relation source.
`GET /v1/schema/{entity}/announcement/{announcementId}/source/{sourceType}`	Announcement (+ optional relation) source schema; `newTriggerId` and `relationId` query params required when `sourceType` is `Relation`.
`GET /v1/schema/announcement/{announcementId}`	Announcement schema.

`AnnouncementController` — announcements

Announcements back the `AnnounceEvent` / `EventAnnounced` chaining pair. Source: `controllers/announcement.controller.ts`.

Method & path	Purpose
`POST /v1/announcements/{entityId}`	Create an announcement for an entity.
`PATCH /v1/announcements/{entityId}/{announcementId}`	Update.
`GET /v1/announcements`	List all.
`GET /v1/announcements/{entityId}`	List for an entity.
`GET /v1/announcements/{entityId}/{announcementId}`	Get one (scoped to entity).
`GET /v1/announcements/by-id/{announcementId}`	Get one by id.

Execution results and history

Each action produces an `ActionResult` (`models/data-entity-action/action-result.ts`) with `action`, `actionType`, `actionName`, `success`, optional `resultId` / `error` / `output`, `traceId`, `spaceId`, `input`, and `effectedEntities`. Errors are normalised by `serializeOutputError` (stack traces are only attached when `NODE_ENV === 'development'`). All results for one firing are persisted as a `TriggerHistory` document with `eventName`, `triggerId`, `triggerName`, `relatedTo`, `doneBy`, `delegatee`, `timestamp`, and the `result[]` array — this is the audit trail for automation.

Redacting secrets in reaction history `Since 1.6`

The history stores each action's `input`, `output` and `error`, which can include tokens and API credentials. These are redacted to `****` before the row is written (also delivered to the 1.6 and 1.8 lines as an update — first on 1.6):

- **Values you flag.** Each token-mapping line has a **Redact in history** checkbox (`redactInHistory` on the token source), offered on Call API, Execute Function, Send Email, Send Notification and Send SMS. The value that token evaluates to is masked wherever it appears in that run's history.
- **Values read from sensitive fields** are masked even without the flag.
- **Sensitive fields in the triggering entry's snapshot.** For Created / Updated / Archived / Restored events, sensitive fields (block sub-fields included) are masked by path in the entry value and in the value diff, and the search index is masked whole.
- **Well-known credential keys.** Any key that, ignoring case, `-` and `_`, is `authorization`, `proxy-authorization`, `cookie`, `set-cookie`, `x-api-key` or `x-app-token`, or that contains `token`, `secret`, `password` or `apikey`; plus `Bearer` / `Basic` credentials and URL user-info inside free text.

- **Learned values.** Once masked, a value is scrubbed wherever it reappears later in the run, including URL-encoded and JSON-escaped forms (up to 200 values per run; a value shorter than 4 characters is masked only where it is the whole value).

If redaction itself fails, the affected part is stored as `{redacted: "REDACTION_FAILED"}`, never raw. A chained `ReactionSucceeded` / `ReactionFailed` reaction redacts with its parent run's secrets. Redaction is independent of the sensitive-data access policy: a `Privileged` reaction's actions still receive real values — only the history is masked. Rows written before the upgrade are not rewritten.

Reaction history indexes and clean-up `Since 1.10`

Three data-service migrations make the history cheap to page and remove old leftovers. `1.10.0-a19` adds six `TriggerHistory` indexes (each filter key, then newest first) behind the Reaction Logs filters. `1.10.0-a32` adds a unique partial index on `DataEntryHistory.sourceMessageId`, so a redelivered entry-created / entry-updated message writes its entry-history row only once (rows written before it carry no message id and are outside the index). `1.10.0-a33` drops the orphaned `trigger_<10 letters>` collections that pre-1.10 criteria evaluation could leave behind — only names matching that exact pattern that no entity uses. A run that recorded no action stores `result: []`.

Troubleshooting reactions

Where a failed reaction actually surfaces — and where it does not:

- **Reaction Logs** `Since 1.10`. System administrators can browse every run in **Administration → Monitoring → Reaction Logs** (backed by the Reaction Logs API) and from the Reactions-tab counters, with each action's stored input, output and error.
- **Nothing surfaces for the user who fired it.** A reaction failure produces no error toast and no failed API response for the user whose action fired it. Outside Reaction Logs, the user-visible surface is the entry history: trigger executions (with per-action `success`) are included in `GET /v1/entities/{entity}/entries/{id}/history` and its `/v2` counterpart when the caller is a system viewer requesting without a view, **or** when the data view sets `entriesConfiguration.showTriggerHistory.enabled: true`; the per-action `error` / `input` / `output` details additionally require `showTriggerHistory.showTriggerDetails`. Both settings are off by default.
- **Every execution is persisted.** Success or failure, each firing writes a `TriggerHistory` document (see above) with the serialized per-action errors — failures live there, not on the `DataReaction` document and not on the entry's value. This is what the entry-history endpoints read.
- **Service logs.** A failed action logs `Execution action '<name>' has failed`, and an entirely failed firing logs `Execution trigger '<event>' on '<entry>' entry has failed`, in the `reaction-service` log.
- **A run that never happened** `Since 1.10`. When the tenant has used its monthly run allowance, runs are recorded as `refused` and run no action — see Monthly run limit and refused runs.
- **Failure chaining is all-or-nothing.** `TriggerExecutedEvent` is published after every execution, but a reaction bound to `ReactionFailed` fires only when **every** action in the execution failed; a partial failure fires `ReactionSucceeded` instead.
- **Metrics** `Since 1.8` — Prometheus counts every execution in `osos_reaction_executed_total` (labels `reaction_type`, `reaction_id`, `entity_id`, `space_id`, `status`) and times it in `osos_reaction_duration_seconds`.
- **What is *not* recorded:** the audit service audits reaction **definition** changes (create/update/delete) only, never executions; and a criteria mismatch is a skip logged at info level with **no** history record. Conversely, a bad

Dynamic value path in an action fails loudly *into the history* (`INVALID_ENTRY_FIELD` / `Path "…" does not exist in the context.`) — the `TriggerHistory` record is the only place that error is visible.

Runtime corrections

Behaviour corrections that change what a reaction does at run time. Those also delivered to published lines carry the earliest line.

- **Loop guard scoped to the workflow step** `Since 1.4` — see Loop protection.
- **A sub-workflow's Entry-step action can run from a later trigger** `Since 1.4`. Every sub-workflow Entry step used to be skipped, so no reaction could ever run a sub-workflow's first step. The move that *starts* the sub-workflow still does not auto-submit its Entry step, but a run fired by a later, unrelated event (On Update, a custom action, a schedule) now runs it, as a main-workflow Entry step always did.
- **Workflow metadata can be mapped** `Since 1.4`. When the trigger entity has workflows, the reaction schema exposes `_metaCurrentSteps.*`, `_metaWorkflowHistory.*` and `_metaAdhocSteps.*`, so a mapping such as *Requester ← current step owner* no longer fails with `INVALID_ENTRY_FIELD` or shows an empty picker — including in configurations saved earlier.
- **Reassigning a step to one user keeps the step's group** `Since 1.4`.
- **Block-item actions no longer succeed silently** `Since 1.4` (also delivered to the 1.6 line as an update). A Create / Update / Delete Block Item action in Reference or Relation mode that resolves no target entry now fails with `NO_ENTRIES_TO_UPDATE_FOUND` instead of reporting success with an empty output.
- **In / Not In accept a single reference or category** `Since 1.10`. A criterion such as *assignee In* `scope.me()`, or one whose value is a single reference or category object, used to match nothing; the value is now treated as a one-item list, and a null element in a reference list no longer throws.
- **SMS actions are recorded as** `SendSms` `Since 1.10`. Every SMS action used to be stored — and shown in Reaction Logs — as `SendNotification`, so an `actionType=SendSms` filter matched nothing. Runs already stored keep the old type.
- **An unresolvable saved query fails closed** `Since 1.10`. A reaction scoped by a saved query used to run with **no filter** when the query could not be loaded or had no applicable criteria — a Timer reaction then fanned out over every entry of the entity. Now a deleted query (a non-retryable 4xx), a failed response or a query with no applicable criteria **skips** the reaction (logged at error level), and any other failure (network, 5xx, 408 / 429) is retried.

Version notes

The following features carry a milestone badge:

- `AnnounceEvent` **action +** `EventAnnounced` **trigger — Since 1.8.**
- `DeleteArchivedEntry` **action — Since 1.8.**
- **Outgoing related-relation triggers — Since 1.8.**
- **Event-level** `triggerContextId` (moved from action options to the event config) — **Since 1.8.**
- `WorkflowStepOwnerChanged` **(14) trigger event** ("on owner changed"), with its owner-change-type and step filters — **Since 1.8.**
- **Custom action type and Contextual** on the custom-action trigger (replacing the *UI Event* toggle) — **Since 1.8.**
- **Custom action type values renamed** `Module` / `Internal` → `Local` / `External` (`UI` unchanged), one-way — **Since 1.8.**

- `ExecuteCustomAction` **(106) action** ("Execute Custom Action"), initially Self/Reference scope only, then widened to add Relation/Expression/Pointer scope, Parameter binding, and proxy-entity-aware resolution — **Since 1.8.**
- `GET /v1/reactions/custom-actions?moduleId=` — the callable-action discovery endpoint behind the Execute Custom Action picker — **Since 1.8.**
- **System buttons can only fire** `UI` **-level custom actions** (a `Local` / `External` binding is rejected with a 403 at execution) — **Since 1.8.**
- **Entity isolation** — a reaction never runs against an entry of another entity — **Since 1.8.**
- `GET /v1/reactions/module/{moduleId}` — **Since 1.8.**
- `ReactionFailed` **(10) and** `DataEntryRestored` **(12) trigger events** — absent from `EventTypeEnum` on the 1.4 and 1.6 lines — **Since 1.8.**
- **Reaction execution metrics** (`osos_reaction_executed_total`, `osos_reaction_duration_seconds`) — **Since 1.8.**
- `sensitiveDataAccessPolicy` **on a reaction** (Prohibited / Privileged / User-Based / Auto), with `User-Based` rejected on an automatic trigger — **Since 1.10.** See Sensitive-data access policy.
- **Update Data Entry runs again under a masking policy** — the guard is scoped to the value this write introduced instead of the whole entry — **Since 1.10.**
- **Group owner titles resolve in reaction tokens** (`_metaCurrentSteps[n].originalOwner`), where they previously rendered empty — **Since 1.10.**
- **Durable, resumable execution** — a batch's entries may complete in any order, a run interrupted by a restart writes its history and stops, and the reaction-finished event is at-least-once — **Since 1.10.** See What to expect from the execution model.
- `DataReaction.description` — an optional, localized description field on a reaction — **Since 1.10.**
- **Execution/failure counts on the reaction list, and a new run-history endpoint** (`GET /v1/reaction/{entity}/{id}/history`) — **Since 1.10.** See Execution/failure counts and run history.
- **Reactions tab redesign** — search, meta-data filter, sort, grouping, an Executed/Not Executed status filter, and an expanded-row Description/Scope — **Since 1.10.** See The Reactions tab [Build Apps › Reactions & automation].
- **Repeatable-block write fixes** — a second Create Block Item action on the same block now appends instead of overwriting, and a single-sub-field Update Data Entry write on a repeatable block no longer collapses its other rows — **Since 1.10.** See Repeatable-block write fixes.
- **Update Data Entry reaction writes can opt into the same conflict-retry protection as a direct entry write**, behind a feature flag — **Since 1.10.** See Reaction-driven writes gain the same protection [Build Apps › Create and design an entity] on Data Modeling: Entities.
- `SetPermission` **(31) /** `RemovePermission` **(32) actions**, with the save-time guard that a reaction cannot grant beyond its saver — **Since 1.10.** See Set / Remove Permission actions.
- **Execute Function output** (`success`, `message`, keys below `result`) bindable by later and chained reactions — **Since 1.10.** See Execute Function output in chained reactions.
- `CreateDataEntity` **/** `UpdateDataEntity` **retired** — refused with 422 on save, skipped at run time — **Since 1.10.**
- **Criteria evaluated over an in-memory** `$documents` **source** instead of a temporary collection — **Since 1.10.** See Criteria and query filter.
- **Trigger context stored only for a listening reaction, and compressed** — **Since 1.10.** See Trigger-context storage.
- **Run-history endpoint restricted to system administrators with slim rows, and the system-wide Reaction Logs API** (`GET /v1/reaction-logs`, `GET /v1/reaction-logs/{id}`) — **Since 1.10.** See System-wide Reaction Logs API.
- **Redacting secrets in reaction history** — **Since 1.6.** See Redacting secrets in reaction history.

- **Reaction-history indexes and temporary-collection clean-up** — **Since 1.10.** See Reaction history indexes and clean-up.
- **Runtime corrections** — loop guard per workflow step, sub-workflow Entry step, workflow-metadata mapping, reassign keeps the group, block-item actions fail instead of silently succeeding (**Since 1.4**); In / Not In single values, SMS recorded as `SendSms`, saved-query fails closed (**Since 1.10**). See Runtime corrections.
- **Authoring a reaction requires Developer access or higher on its entity** (`403 REACTION_AUTHORING_FORBIDDEN`), imports included — **Since 1.8.** See Who may author a reaction.
- **Call API limits** — private and internal destinations refused, 30-second limit on the whole exchange, 10 MiB body cap, tokens cannot add headers — **Since 1.8**; reserved platform header names (`400 RESERVED_HEADER_NAME`) — **Since 1.10.** See Action options reference › Call API: what the platform allows.
- **Send WhatsApp (107) action** with the approved-template picker — **Since 1.10.** See Action options reference › Send WhatsApp options.
- **File references in reaction options** (`400 FILE_EXPIRED`, `503 STORAGE_SERVICE_UNAVAILABLE`) — **Since 1.10.** See Files in reaction options.
- **Monthly run limit** with `refused` runs and `refusedCount` — **Since 1.10.** See Monthly run limit and refused runs.
- **A duplicated reaction can change its event**, and an absolute-entity source the new event rules out falls back to Self — **Since 1.8.** See Defining a reaction [Build Apps › Reactions & automation].
- **Chained reactions: Triggering Space enforced, impossible chains refused at save, each entity runs only its own chained reactions** — **Since dev.** See Action options reference › Event-config shapes.

The rest of the engine is **Long-standing** and predates the 1.2 milestone, so it carries no “Since” badge — including `loopThreshold`, `SlaTimeExceeded`, incoming relation-source triggers, and `NotifyZapier`.

Buttons API reference

This page is for **integrators and developers** who read, create or execute entity buttons through the API, or who need to know exactly what happens when a button is clicked. It documents the stored model, its numeric enums, validation, the endpoints and the execution flow.

> **Designing buttons in the UI?**
>
> Builders adding buttons on an entity's **Buttons** tab should start with Buttons [Build Apps] — it covers the same features in the labels the designer shows.

How a button is stored

Internally a system button is **not a bespoke collection**: it is a regular `DataEntry` whose `dataEntity` is the built-in `SYSTEM_BUTTON` entity (`StringConstants.SYSTEM_BUTTON_ENTITY_ID = 'SYSTEM_BUTTON'`, repository key `SYSTEM_BUTTON_NAME = 'system_button'`). This has two consequences:

- **Create / update / delete go through the generic data-entry CRUD** for the `system_button` entity — not through the `SystemButtonController`. The dedicated controller only exposes *read*, *status*, *options*, and *execute* endpoints.
- Buttons inherit `DataEntry` machinery: `state`, `isInvalid`, `store`, versioning, history, and space/store-based access scoping.

Source of truth: `SystemButtonController` (`controllers/system-button.controller.ts`), `SystemButtonService` (`services/system-button.service.ts`), and the models under `models/system-buttons/`.

Behaviour notes behind the designer options

- **Action types.** The designer's **Action** list groups the entity's **UI**-level custom actions under **Custom Action** (`RelevantActionTypeEnum.CustomEvent`) and offers **Navigation** (`Redirect`) and **Script** under **UI Action**. Only `CustomEvent` buttons pass through the execute handler; `Redirect` and `Script` buttons are resolved in the client. Whether the bound reaction has a user interface is read from its custom action type [Build Apps › Reactions & automation] being **UI** `Since 1.8`, rather than from a separate flag on the reaction. For deeper external logic, the reaction a button fires can in turn invoke a function.
- **Scope.** `actionTypes` is a `ButtonActionTypeEnum[]`: `SingleEntry`, `MultipleEntries` and `DataCollection`. The designer offers **Single Entry** and **Multiple Entries** (multi-select); `DataCollection` (a list) is accepted by the API and the execute handler.
- **Custom-link tokens** `Since 1.8`. Token bindings for a **Navigation** button's URL are stored as `redirection.urlTokensMap` and interpolated into `redirection.url` at read time when an `entryId` is supplied. Each token's value source is a `TokenSourceType` (`Fixed`, `Field`, `Expression`, `Pointer`). Token resolution is best-effort: a token that cannot be resolved (for example a field that no longer exists) is logged as a warning and the button falls back to the original, unresolved link text rather than blocking execution — no error reaches the end user.
- **Confirmation-dialog variables** `Since 1.8` are allowed only when `actionTypes` is exactly `[SingleEntry]` (enforced by validation).

- **Field-value mapping.** Pre-fill values live in `relevantAction.fieldsMapping`. The designer's value types map to `FieldValueTypeEnum`: **Field** = `Dynamic` (a source path on the current entry), **Manual**, **Not Set** (null), **Pointer** (evaluates a context token such as **@me** and injects the resolved value with its display title) and **Expression** (stored JavaScript). The same mapping can also be turned into a filter: a multi-value context token such as *My Groups* builds an *In* condition, so a button can both pre-fill a form and scope which entries it targets. When `relevantAction.overrideFieldsMappingLockedFields` is set, the mapped field names are treated as locked; `getSystemButtonOverridableLockedFields(id)` returns those names (else `[]`) so the override UI can lock them.
- **Availability filter.** A button may carry a saved query (`query`) and an inline filter (`criteria`); when both are present they are merged with **AND**. Saved queries are offered under the designer's entries-source picker. Behaviour on non-matching entries is `nonMatchEntriesMode`: `Disable` (default) or `Hidden`. The per-entry evaluation is described under Availability: status for entries.
- **Script buttons** `Since 1.8` store their body in `relevantAction.customScript`. The client calls it with a `context` argument exposing `entity`, `view`, `entry`, `entries` and `popupManager`; for a multi-entry button `context.entries` holds every selected row.

Data model

`SystemButtonDataEntry`

Extends `DataEntry`; its constructor pins `dataEntity = 'SYSTEM_BUTTON'`. The button-specific payload lives entirely under `value` (`SystemButtonDataEntryValue`).

`SystemButtonDataEntryValue`

Field	Type	Required	Notes
`name`	`LocalizedValue`	yes	Button label. Validated to be localized and **unique per related entity**.
`normalizedName`	`string`	no	Auto-generated programmatic identifier derived from `name`, unique per related `DataEntity`.
`description`	`NullableLocalizedValue`	no	Optional description.
`icon`	`Icon`	yes	Button icon.
`relatedTo`	`DataEntityReferenceProperty`	yes	The entity the button acts on. `relatedTo.dataEntry` is the entity id.
`relevantAction`	`SystemButtonRelevantAction`	yes	What the button does (event / redirect / script).
`actionTypes`	`ButtonActionTypeEnum[]`	yes	Which selection scopes the button supports (single / collection / multiple).
`isPublished`	`boolean`	no (default `false`)	Only published buttons are returned by the "active" endpoints and are executable.
`query`	`ReferenceProperty | null`	no	Reference to a stored data query whose `criteria` gate button availability.
`criteria`	`FilterExpression`	no	Inline filter expression that gates button availability. Merged (AND) with `query.criteria` when both present.
`nonMatchEntriesMode`	`SystemButtonNonMatchEntriesMode`	no	Behaviour when selected entries don't match the criteria: disable vs. hide.
`confirmationDialog`	`SystemButtonConfirmationDialog`	no	Optional confirmation dialog shown before execution.

`SystemButtonRelevantAction`

Field	Type	Notes
`type`	`RelevantActionTypeEnum`	`CustomEvent = 0`, `Redirect = 1`, `Script = 2`.
`triggerId`	`string`	For `CustomEvent`: the reaction/trigger id resolved against the reaction-service.
`redirection`	`SystemButtonRedirection`	Primary redirection target.
`successRedirection`	`SystemButtonRedirection`	Redirect on success.
`failureRedirection`	`SystemButtonRedirection`	Redirect on failure.
`fieldsMapping`	`EntryActionOptionsValue[]`	Field overrides passed to the action.
`overrideFieldsMappingLockedFields`	`boolean`	When set, the mapped field names are treated as locked.
`customScript`	`string`	Script body for `Script`-type buttons. `Since 1.8`

`SystemButtonRedirection`

Field	Type	Notes
`type`	`ButtonRedirectionTypeEnum`	`RelatedEntries = 0`, `SameView = 1`, `Url = 2`.
`relationId`	`string`	Relation to navigate through (for relation-based redirects).
`relationRedirectionTarget`	`ButtonRelationRedirectionTargetEnum`	`EntriesList = 0`, `ShowEntryDetails = 1`, `CreateEntry = 2`, `UpdateEntry = 3`.
`url`	`string`	Target URL for `Url` redirects.
`urlTokensMap`	`{ [key]: TokenSource }`	Token bindings interpolated into `url` at read time when an `entryId` is supplied (keys use the `str-` prefix pattern).

`SystemButtonConfirmationDialog`

`Since 1.8`

Shown before the button executes.

Field	Type	Notes
`enabled`	`boolean` (default `false`)	Master switch.
`title`	`NullableLocalizedValue`	Dialog title (supports token placeholders).
`content`	`NullableLocalizedValue`	Long-form body text (supports token placeholders).
`proceedButtonLabel`	`NullableLocalizedValue`	Confirm-button label.
`cancelButtonLabel`	`NullableLocalizedValue`	Cancel-button label.

Field	Type	Notes
`variables`	`SystemButtonDialogVariable[]`	Token bindings used to interpolate `title` / `content`. **Only allowed when `actionTypes` is exactly `[SingleEntry]`.**

`SystemButtonDialogVariable`: `{ name: string, valueType: TokenSourceType, value: any }`. `TokenSourceType` is `Fixed = 0`, `Field = 1`, `Expression = 2`, `Pointer = 3`.

`ButtonClickedHistory`

Recorded per target entry after a `CustomEvent` button executes (via `DataEntryHistoryService.createHistoryForButtonClicked`, `historyType = ButtonClickedHistory`). Stores `buttonId` and the localized `buttonName`.

Enumerations

```
RelevantActionTypeEnum          CustomEvent = 0, Redirect = 1, Script = 2
ButtonActionTypeEnum            SingleEntry = 0, DataCollection = 1, MultipleEntries = 2
SystemButtonNonMatchEntriesMode  Disable = 0, Hidden = 1
ButtonRedirectionTypeEnum       RelatedEntries = 0, SameView = 1, Url = 2
ButtonRelationRedirectionTargetEnum  EntriesList = 0, ShowEntryDetails = 1, CreateEntry = 2, UpdateEntry = 3
EventExecutorTypeEnum           SYSTEM, MANAGER, EXECUTOR   // EXECUTOR = the user who initiated the action
AskUserForEntityTypeEnum        InlineEntity = 0, View = 1
TokenSourceType                 Fixed = 0, Field = 1, Expression = 2, Pointer = 3
```

Enum values are numeric (persisted as numbers) except `EventExecutorTypeEnum`, which is a string enum.

Creating and updating a button (API)

There is **no** `POST /v1/system-buttons` create endpoint. Buttons are managed through the generic data-entry CRUD, using the `system_button` entity:

Operation	Route (from `data-entries.controller.ts`)
Create	`POST /v1/entities/{entity}/entries`
Update	`PATCH /v1/entities/{entity}/entries/{id}` (v1) / `PATCH /v2/entities/{entity}/entries/{id}`
Delete	`DELETE /v1/entities/{entity}/entries/{id}`

Here `{entity}` resolves to the `SYSTEM_BUTTON` / `system_button` entity. The request body is a `DataEntry` whose `value` is a `SystemButtonDataEntryValue` as described above.

Validation

`SystemButtonValidator` runs on the `CREATE` and `UPDATE` phases for the `system_button` entity. It enforces:

1. `value.relatedTo.dataEntry` must be present, and the related entity must exist. If the related entity is implicit and not `implicitOptions.allowEdit`, an `ImplicitEntityUpdateException` is thrown.
2. **Confirmation-dialog variables are single-entry-only.** If `confirmationDialog.variables` are present, `actionTypes` must be exactly `[SingleEntry]`; otherwise `ConfirmationDialogVariablesNotAllowedForMultiScopeException` (`SYSTEM_BUTTON_CONFIRMATION_DIALOG_VARIABLES_NOT_ALLOWED_FOR_MULTI_SCOPE`).
3. For `CustomEvent` buttons with a `triggerId`, the trigger is resolved against the reaction-service. A missing trigger throws `TriggerNotExistException`; a trigger that is invalid (`validations.isValid === false`) or

`disabled` throws `InvalidOrDisabledTriggerException`.

4. **Name uniqueness:** `value.name` must be localized (`LanguagesService.validateName`) and unique per related entity — a match on any language value against another button of the same `relatedTo.dataEntry` throws `UniqueValueException`.

Reading buttons (API)

All `SystemButtonController` routes require `Auth2` authentication and pass through the entity voter. Reads are scoped to the space's readable stores (`StoreService.getStoresForRead`) and to `state = Active`, `isInvalid = false`.

Method & Path	Purpose
`GET /v1/system-buttons`	All active buttons in the space, **enriched** per button via the reaction-service (`SystemButtonDataEntryDto[]`). N+1 across a service boundary — slow for large spaces.
`GET /v1/system-buttons/raw`	All active buttons as **raw stored entries** (`SystemButtonDataEntry[]`), no reaction enrichment; disabled-trigger buttons are **not** dropped. Intended for internal consumers (e.g. packaging import conflict detection). `Since 1.8`
`GET /v1/system-buttons/count`	Count of active buttons (`Where` filter supported).
`GET /v1/system-buttons/by-id/{id}`	Single active, valid button by id (raw entry). 404 `SystemButtonNotFound` if not found.
`GET /v1/system-buttons/{relatedToId}`	Buttons for a related entity (enriched DTOs).
`GET /v1/system-buttons/{relatedToId}/active?entryId=`	Active (published) buttons a user can see for a related entity. Optional `entryId` resolves redirection-URL tokens and confirmation-dialog variables against that record.
`POST /v1/system-buttons/{relatedToId}/status/entries`	Per-button enabled/hidden status across a set of entries (see Availability).
`GET /v1/system-buttons/{id}/options`	Reaction actions for a `CustomEvent` button (`DataReactionAction[]`).
`GET /v2/system-buttons/{id}/options`	v2 options: actions **plus** ask-user entity/view/dialog metadata (`SystemButtonOptionsDto`).
`POST /v1/system-buttons/{id}`	Execute the button (see Execution).

DTO enrichment

`getAllActiveSystemButtonsDto` / `getSystemButtonsByRelatedToDto` fetch the bound reaction from the reaction-service and fold trigger metadata into `relevantAction` as a `SystemButtonRelevantActionDto`: `triggerName`, `uiEvent`, `askUser`, `isTriggerDisabled`, `executionIndicator`, `messages`, and `executorType` (defaulting to `EventExecutorTypeEnum.EXECUTOR` when the trigger has no explicit executor). In the "active" DTO listings, buttons whose bound trigger is `disabled` are **skipped**; the `raw` endpoint keeps them.

`uiEvent` is derived, not stored `Since 1.8` — the DTO still tells the client whether the bound trigger has a user interface, but the trigger no longer carries a separate `uiEvent` flag. Both DTO builders now compute the field from the trigger's custom action type [Build Apps › Reactions & automation]: `uiEvent` is true exactly when the custom action type is **UI**. The wire shape is unchanged — still a plain boolean — so existing button consumers are unaffected; only the source of the value moved.

`askUser` is a second, independent flag `Since 1.8`. Because a button can now only ever be bound to a **UI**-level custom action (see System buttons can only fire UI-level actions below), `uiEvent` is `true` for every button — it can no longer tell a client whether clicking the button should open the "ask user" dialog first. `SystemButtonRelevantActionDto` therefore exposes a second boolean, `askUser`, computed as `true` exactly when the bound trigger's `eventConfig.askUser.enabled` is set. **Clients must gate the ask-user dialog on `askUser`, not `uiEvent`** — a button bound to a `UI` action with no ask-user configuration now executes immediately, with no empty dialog popping up first.

When `entryId` is provided to `getSystemButtonsByRelatedToDto`, the service also:

- resolves `relevantAction.redirection.urlTokensMap` against the record and interpolates them into `redirection.url`, and
- resolves `confirmationDialog.variables` into token values and interpolates them into the dialog `title` / `content` (per language).

Both resolutions are best-effort: failures are logged as warnings and the button is still returned with unresolved text.

Ask-user options (`GET /v2/system-buttons/{id}/options`)

For a `CustomEvent` button whose reaction has `eventConfig.askUser.enabled`, this returns the ask-user surface the UI must render:

- `mode = InlineEntity` → returns the reaction's inline `entity` as `askUserEntity`.
- `mode = View` → loads the configured entity + view and returns a **projected** entity via `getEntityBasedOnAskUserViewConfigurations`, which keeps only the fields listed under the view's `AllowAskUserFor` supported operation, renames the entity to `askUser`, and re-ids it as `TRASK_{entityId}_{triggerId}`.

If the trigger is missing/invalid/disabled the same `TriggerNotExistException` / `InvalidOrDisabledTriggerException` errors apply.

Fields inside a shared block are kept `Since 1.8` — the view's `AllowAskUserFor` allow-list holds `fullNormalizedName`s such as `cc.email` or `attachment`. A top-level field is kept when its `normalizedName` matches an allowed entry; a sub-field of an internal entity (a block) is kept when **any** of its block-qualified names is allowed. That "any" rule is what makes shared blocks work: when one block definition backs several top-level block fields — To / CC / BCC recipient blocks, say — the projected entity emits that block's sub-fields once per prefix (`to.email`, `cc.email`, `bcc.email`, …), all carrying the same sub-field id. A view that exposes only `cc.*` and `bcc.*` still keeps those shared sub-fields, because the match is tested against every block-qualified occurrence of the id rather than only the first one encountered. So you can now ask the user for a sub-field that lives inside a shared block.

Availability: status for entries

`POST /v1/system-buttons/{relatedToId}/status/entries` answers "for these selected records, is each button enabled or hidden?".

Request body (`GetSystemButtonsStatusForEntriesRequestDtoModel`):

```
{
  "dataEntriesIds": ["<entryId>", "..."],
  "selectAll": { "enabled": true, "...": "SelectAllDto criteria" }
}
```

You must supply **either** `dataEntriesIds` **or** `selectAll.enabled = true`; otherwise `GetSystemButtonsStatusForEntriesMustProvideIdsOrSelectAllException` (`GET_SYSTEM_BUTTONS_STATUS_FOR_ENTRIES_MUST_PROVIDE_IDS_OR_SELECT_ALL`). With `selectAll`, the entry ids are resolved from the query engine (`getEntriesIdsBySelectAllDto`).

Response is `SystemButtonStatusForEntriesResponseDtoModel[]`: `{ systemButtonId, isDisabled, isHidden }`.

Logic (`getSystemButtonsStatusForEntries` + `generateGetButtonsStatusForEntriesAggregationStages`):

- Buttons with **no** `query` and **no** `criteria` are always `isDisabled: false`.
- For buttons **with** a filter, a MongoDB aggregation runs a `$facet` — one branch per button — that counts how many of the selected entries match the button's criteria (`query.criteria` merged with inline `criteria` via `AND`). A button is **enabled only if every selected entry matches** (`matchingEntriesCount === dataEntriesIds.length`); otherwise it is disabled.
- When **no** selected entry matches at all, the non-match behaviour depends on `nonMatchEntriesMode`: default (`Disable`) → `{ isDisabled: true, isHidden: false }`; `Hidden` → `{ isDisabled: true, isHidden: true }`.
- System fields such as `@current_user` / `@myGroups` in the criteria are resolved via `getSystemFieldsStages` before the facet.

Executing a button

`POST /v1/system-buttons/{id}` with an `ExecuteSystemButtonDto` body:

Field	Type	Notes
`mode`	`ButtonActionTypeEnum`	Selection scope of this execution.
`source`	`string | string[]`	Single entry id, data-collection id, or an array of entry ids — depending on `mode`.
`askUserValue`	`IEntryValue`	The ask-user payload collected from the dialog, forwarded to the reaction.
`selectAll`	`SelectAllDto`	For `MultipleEntries` with select-all: resolves the full id set server-side.

Execution flow (only meaningful for `RelevantActionTypeEnum.CustomEvent`; `Redirect` / `Script` buttons are resolved client-side and do not go through this handler):

1. **View guard** — if the request is scoped to a `DataViews`, the button must be listed in `view.systemButtons`, else `SystemButtonNotFound`.
2. **Load button** — must be `state = Active`, `isInvalid = false`, and `value.isPublished = true`, else `SystemButtonNotFound`.
3. **Resolve target entries** from `mode` + `source`:
   - `SingleEntry` → one `ReferenceProperty` for `source`.
   - `DataCollection` → the collection's `items` (the collection must belong to the button's related entity, else `DataCollectionNotFoundException`).
   - `MultipleEntries` → the id array; if `selectAll.enabled`, ids are expanded via `getEntriesIdsBySelectAllDto`.
   - Missing `source` → `MissingPropertyException`.
4. **Re-check availability** for the resolved entries; if the button is disabled for that set, `SystemButtonIsDisabled` (`SYSTEM_BUTTON_IS_DISABLED`, 403).
5. **`UI` -level guard** `Since 1.8` — if `relevantAction.triggerId` is set, the bound trigger's custom action type is checked; a `Local` or `External` binding is rejected with `SystemButtonNotUiCustomAction` (403). See System

buttons can only fire UI-level actions below.

6. **Emit feedback + publish event** — a `JobScheduledMessage` is emitted over the socket immediately for UI feedback, then a `CustomEventTriggeredEvent` (entity id, target entries, user id, `triggerId`, timestamp, `askUserValue`) is published on the service bus, carrying `messageId`, `instanceId`, `correlationId` (trace id) and sticky attributes (`userId`, `delegateeId`, `delegationId`, `spaceId`). The reaction-service handler picks this up.
7. **History** — `createHistoryForButtonClicked` records a `ButtonClickedHistory` entry per target.

The endpoint returns the generated `messageId` (a UUID) so the client can correlate the async execution result.

System buttons can only fire UI-level actions `Since 1.8`

A button's `relevantAction.triggerId` can, structurally, point at any `CustomEvent` reaction — nothing stops it from naming a `Local`- or `External`-level custom action [Build Apps › Reactions & automation], which are meant to be invoked only by another reaction's Execute Custom Action [Build Apps › Reactions & automation] action, not by a button click. Step 5 of the execution flow above closes that gap: it loads the bound trigger's custom action type before publishing the execution event, and rejects the click with `SystemButtonNotUiCustomAction` (403, *"this button is bound to a custom action that is not a UI action"*) when it is `Local` or `External`.

The reverse direction — an Execute Custom Action action can never target a `UI`-level custom action — is enforced when the *reaction* is saved; see Execute Custom Action [Build Apps › Reactions & automation].

Execute example

```
POST /v1/system-buttons/{buttonId}
Content-Type: application/json

{
  "mode": 0,
  "source": "8f3c...entryId"
}
```

For a multi-entry execution with select-all:

```
{
  "mode": 2,
  "selectAll": { "enabled": true }
}
```

Preview mode

`GET /v1/system-buttons/{relatedToId}/active` in preview mode returns the entity's **real** buttons (real name/icon/config) but with **fake ids** (`FakeEntryGeneratorService.withFakeIds`) so none can be executed against a fake entry; `entryId` is intentionally **not** forwarded to the real query in preview mode. If the current view has an empty `systemButtons` list, the endpoint returns `[]`.

Implementation notes

- **Store/space scoping** is applied in `buildActiveSystemButtonsWhere` and in `getSystemButtonsByRelatedToDto`: buttons are filtered by `store.dataEntry ∈ readableStores`, `state = Active`, `isInvalid = false`, and (for active listings) `value.isPublished = true`.
- **N+1 caution:** the enriched DTO listings call the reaction-service once per button. `getAllActiveSystemButtonsRaw` / `GET /v1/system-buttons/raw` exists specifically to avoid this for consumers

that only need stored data.

- `getSystemButtonOverridableLockedFields(id)` returns the mapped field names when `relevantAction.overrideFieldsMappingLockedFields` is set (else `[]`) — used to lock the mapped fields in the override UI.
- The `ExecuteSystemActionOverrideValuesDto` (`{ actionId, options }`) and the commented-out `getTriggerToExecuteForSystemButton` / `validateSystemButtonTrigger` in the service indicate an older action-override path that is currently inactive; override handling now lives in the reaction-service. Do not rely on those commented blocks.

Version notes

Feature	Milestone
Custom script action (`relevantAction.customScript`)	Since 1.8
Confirmation dialog (`confirmationDialog`, dialog variables)	Since 1.8
Custom-link / confirmation-dialog tokens	Since 1.8
Raw listing endpoint (`GET /v1/system-buttons/raw`)	Since 1.8
`uiEvent` derived from the trigger's custom action type (replaces the stored flag)	Since 1.8
`askUser` on the button's relevant-action DTO (the ask-user dialog gate, independent of `uiEvent`)	Since 1.8
System buttons rejected at execution when bound to a `Local` / `External` custom action (`SystemButtonNotUiCustomAction`, 403)	Since 1.8
`normalizedName` on buttons	Since 1.6

Non-match-entries mode (`nonMatchEntriesMode`) and the v2 options endpoint predate the 1.2 milestone, so they carry no availability badge (treat as **Long-standing**).

Portal sign-in and spaces (internals)

This page is for integrators and developers who link users into the portal, build their own client against the same identity service, or need to explain exactly why a user lands where they do. It covers the portal's OIDC flow, the `/signin` deep link, the error codes behind the messages users see, and how a space is resolved and enforced.

The end-user view of the same features is Signing In & Choosing a Space [Using OSOS]. The identity service itself (providers, OTP, rate limits, audit stores) is covered for operators in Identity server, and the OIDC endpoints and grants in Authentication & OAuth.

Spaces and domains

A **space** is the tenant data boundary: it scopes the entities, collections, views and users that a request can reach. A space record carries its display data (`space_information`: name, description, icon), its address (`url`, `domain_identifier`, and the derived `full_url`), an active/inactive state and an end-user maintenance state.

The **root** space is served on the bare base domain (its `domain_identifier` is `root`). Every other space is served on a subdomain of the same base domain: with base address `example.com` and identifier `sales`, the space is reached at `sales.example.com`. The browser's host therefore decides the space; switching spaces means navigating to another host.

Two ideas are both derived from the host and should not be confused:

- **Space**: the **data** boundary, which data a request sees.
- **Domain**: the **sign-in and branding** boundary, which sign-in methods, MFA policy and external providers apply. This is what shapes the sign-in page.

Administration of spaces and domains is in Spaces & domains [Administration].

The portal's sign-in flow

The portal never collects a password. It redirects the browser to the identity service's hosted pages and receives the user back, using the OIDC **authorization-code** flow. Relative to the authority URL the portal uses:

Purpose	Endpoint
Authorization (login)	`{AUTHORITY}/auth`
Token	`{AUTHORITY}/token`
User info	`{AUTHORITY}/me`
End session (logout)	`{AUTHORITY}/session/end`
Keys (JWKS)	`{AUTHORITY}/jwks`
After-login return page	`{APP_URL}/auth-callback`

The sequence:

1. **Start-up check.** The app checks for a valid session, including a silent iframe check against the identity service. If one exists, the user goes straight in.

2. **Redirect to sign-in.** If the user is not signed in and the deployment is configured to redirect automatically, or the user selects **Sign In**, the browser goes to the identity service. The UI language is passed as the `lang` parameter.
3. **Authenticate** with a method the domain allows (see Sign-in methods).
4. **Second factor**, if the MFA context requires one (see Second factor).
5. **Return** to `{APP_URL}/auth-callback`, where the app finalizes the session and forwards the user to the destination.

If the sign-in service cannot be reached when the user selects **Sign In**, the portal shows an error message instead of a button that does nothing `Since 1.10`.

First sign-in

On a user's first sign-in the portal routes to the onboarding page at `/welcome` instead of the destination. Its start button marks the first login as complete and, when the tenant has a tour configured and enabled, starts the product tour. The page stays reachable from the profile menu's **Help** item.

The `/signin` deep link `Since 1.6`

`https://<portal>/signin?returnUrl=<path>` hands the user to sign-in and brings them back to `<path>` afterwards. The page itself renders only a loading indicator.

- **Only safe, same-site paths are honored.** External or malformed `returnUrl` values are ignored, so the sign-in flow cannot be used as an open redirect.
- **Already signed in:** the user goes straight to the target (or the home page), and **Back** does not bounce through `/signin` again.
- **Not signed in:** the user is taken to the login page. **Back** from there lands on the portal home page rather than the loader, and sign-in is not restarted, so no loop is possible.
- **`&provider=<name>` `Since 1.10`.** Starts sign-in directly with one of the domain's external providers. The portal forwards it as the authorization parameter `idp_hint`. The value is the provider's unique `name`, not its display label. A blank or unknown name is ignored and the normal login page is shown. Behaviour on the identity side: `idp_hint`.

The deep link was added to the 1.6 line and is part of 1.8 and later.

Login-page behaviours the portal relies on

- **Single-provider skip** `Since 1.10`: a domain whose only way in is one external provider skips the login page. On by default; an operator can turn it off. See Single-provider skip.
- **Cancel** `Since 1.4`: the login page's **Cancel** link (and its 10-minute idle timer) returns `access_denied` to the client; the portal treats that as "go home", landing on its home page rather than the no-access page (1.6 and later). See Cancelling the login page.

When the portal cannot start: reason codes `Since 1.10`

Before Angular starts, the portal fetches its sign-in settings for the host (see Browser bootstrap without a client secret). On failure it replaces the splash with a static bilingual page. The page shows the English message, the Arabic message, a details block (`Reason`, `Status`, `Request`, `Correlation ID`, `Time`) and a **Retry** button that reloads. The reason is also in the page's `data-reason` attribute.

Reason code	English message on the page
`TENANT_NOT_RESOLVED`	This address is not linked to an organisation.
`BOOTSTRAP_ENDPOINT_MISSING`	The sign-in service is not available on this server.
`TENANT_NOT_READY`	This organisation is still being set up. Please try again shortly.
`SERVER_ERROR`	The sign-in service returned an error.
`NETWORK`	The sign-in service could not be reached.
`TIMEOUT`	The sign-in service took too long to respond.
`BAD_CONFIG`	The sign-in configuration is not supported by this version of the app.
`DEPLOYMENT_CONFIG`	The application configuration could not be loaded.

Tenant under maintenance `Since 1.10`

While a tenant is being restored from a backup or moved to another cell, sign-in is refused with `503` `TENANT_IN_MAINTENANCE` and a `Retry-After` header. The message reads *"This tenant is being restored from a backup. Try again shortly."* (or *"…is being moved…"*). During start-up the same condition surfaces as `TENANT_NOT_READY`. Details: Errors and retries.

Sign-in methods

What the hosted login page offers depends on the domain:

Method	Shown when	Notes
Username + password	the domain's `supportLocalSignIn` is enabled	Standard local sign-in.
External provider (SSO)	one or more providers are linked to the domain	One button per allowed external OAuth2/OIDC provider.
Passkey (passwordless)	the domain's `supportPasskey` is enabled	WebAuthn, discoverable-credential flow: any passkey registered for the site can respond.

After a successful password sign-in, a user with no passkey may be offered the **Register Passkey** page (with **Skip**).

A failed local sign-in re-renders the login page with the matching message:

Code	Message
`INVALID_USERNAME_OR_PASSWORD`	Invalid username or password.
`ACCOUNT_LOCKED`	Account is locked
`ACCOUNT_DEACTIVATED`	The account is deactivated, Please contact with your system administrator

Sensitive (non-`GET`) identity requests are rate-limited per client address (10 per 15 minutes by default). See Rate limit on identity requests.

Every attempt, successful or not and by which method, is recorded for security auditing (`Since 1.4`), including failed passwords, locked or deactivated accounts, passkey failures and MFA outcomes. Users see their own history in **Logins › Attempts**. See Authentication audit trails.

Account linking and profile sync

- A provider identity already linked to a user signs straight in.

- An existing account with the same email but no link may be offered **manual linking** `Since 1.8`: the user proves ownership of the existing account by password. See Manual account linking.
- With no matching account and the provider configured for it (`supportSignup`), a user is created automatically. See Auto-provisioning.
- `syncProfileOnSignIn` `Since 1.8` refreshes the profile from the provider at each sign-in. A sync failure never blocks the sign-in.

Password pages

The identity service hosts the self-service password pages:

- **Forgot password** at `/identity/reset-password`: requests a reset link by email, then sets a new password from the emailed link. The page always shows the same success message, whether or not the email matched an account.
- **Change password** at `/identity/change-password`: for signed-in local users. External (SSO) accounts are redirected away, since their password is managed by the provider.

Reset-link lifetime and replacement are in Password reset.

Second factor

After the first factor (password or external provider), the identity service resolves the user's **MFA context** from:

- whether the user has **TOTP** (authenticator app) enrolled `Since 1.8`;
- whether the domain requires **SMS** or **email** one-time codes (`sms2FA` / `email2FA`), and whether WhatsApp is offered `Since 1.10`;
- whether the user has **passkeys** that can act as a second factor.

If a factor is required, the browser goes to `/interaction/{uid}/mfa`. There the user enters a code (or uses a passkey or a recovery code), can switch method and can request a resend. Error codes: `MFA_INVALID_CODE`, `MFA_CODE_EXPIRED`, and `MFA_ATTEMPTS_EXCEEDED` (which prompts a fresh code). If no factor applies, MFA is skipped.

- The method last chosen is stored on the **account**, not in the browser, so a reload keeps it and another user on the same computer gets their own `Since 1.8`.
- The resend limit is a fixed rolling window, not a counter for the whole sign-in attempt, so a code that timed out never leaves the user unable to request a fresh one `Since 1.8`. See OTP rate limiting.

Human verification (`Since 1.8`) can guard sign-in, MFA and password reset. When its service is unavailable, those flows refuse; there is no bypass. See Human verification [Administration › Human verification (bot protection)].

Resolving the active space

The user's spaces

The portal loads **available spaces** from `GET /api/v1/spaces/my`. Only spaces that are **active**, **not deactivated** and of which the user is a member are returned.

The active space

The portal picks the active space by matching the browser host against `` `${domainIdentifier}.${url}` `` for each space. If nothing matches, it falls back to the **root** space.

The switcher

The sidebar's space list renders only when the user has more than one space. Each entry shows the space's icon and name. Selecting one navigates to that space's `full_url`: a full page navigation to another host, not an in-page toggle, so the session check can run again on the new host. The list is hidden while the user acts under a delegation and for Restricted users.

When memberships change, the gateway's cache of hosts and memberships is refreshed automatically. A space the user was just added to that does not appear yet is usually a stale cache.

How the gateway ties a request to a space

1. **`space-resolver`** matches the request host against the cached list of space hosts and sets `x-space-id` (and the `x-space-maintenance` flag). Any such header the client sent is discarded first, so the host alone decides. An unknown host is rejected with `SPACE_ERROR` ("Space ID not found"; HTTP `404` since 1.10, `400` before).
2. **`domain-resolver`** resolves the tenant domain for the host.
3. **`user-space-checker`** confirms the signed-in user is a member of the resolved space. If not, it answers **HTTP 403** with code `SPACE_FORBIDDEN` and a list of `available_spaces`.

The portal catches `SPACE_FORBIDDEN` and routes to its no-access page (**You don't have access to this space!**), passing `available_spaces` so the user can jump to a space they may open. If the user turns out to be unauthenticated at that point, they go back through sign-in instead.

Plugin details: Space and domain resolution · Tenancy and identity.

Unavailable spaces

- **Deactivated:** the space is fully switched off. It is excluded from `spaces/my` and from host resolution, so it cannot be opened.
- **End-user maintenance:** the space still resolves, but the gateway marks the request (`x-space-maintenance`) so end users see a maintenance state.

The **root** space cannot be deactivated. Both states are set by administrators.

Query model and API reference

This page is for **integrators and developers** who create or run saved queries through the API, call list routes with quick filters and parameters, or need the exact semantics behind a condition. It documents the operators by their engine names, the request parameters, the quick-filter-group and tag rules, and the error codes.

> **Building a query in the UI?**
>
> Builders should start with Queries & filters [Build Apps] — it covers the same features with the labels the designer shows (for example the operator **is filled** is `IsNotEmpty` here, and the pointer **@myGroups** is `myGroups`). The filter-expression syntax itself is in Criteria dialects.

Queries in integrations (Grafana)

A saved query is also the unit a **Grafana** panel executes `Since 1.6`: the Apptomator data source runs saved data queries as row filters and saved statistic queries as aggregates, and any parameters a query declares become panel inputs that accept Grafana dashboard variables. See Grafana Dashboards.

`Since 1.8`, that integration is more reliable end to end. A **quick filter** built from a parameterized query now correctly substitutes its parameters when a panel applies it — previously the parameters were left unresolved and the filter matched nothing. Every request for a view's entries, including a panel's, is also capped by a server-enforced ceiling (10,000 entries by default) so a single request can never stream an unbounded collection; the Grafana data source already relies on this cap and always sends its own explicit row limit for exactly that reason. The same "never stream everything" principle also now bounds how many entity definitions an integration can retrieve at once through the API, paging results instead of returning every entity's schema in one response.

How a condition is stored

Each condition is made of three parts: a **field** (an entity structure field, or an entry-metadata field such as *Created by* / *Created at*), an **operator** (the relationship between the two sides), and a **value** (the second side of the comparison). Conditions are grouped and joined with a logical relation — **All conditions met (AND)** or **Any condition met (OR)** — and groups can be nested to build compound logic.

Operators

The operator expresses the relationship between the two sides of the condition. The available operators depend on the field type: equality operators apply to every type; text operators (contains, starts with, like, and their negations) apply to text fields; comparison operators (less/greater than, or equal) apply to dates and numbers; **In / Not in** apply to entity fields (reference, category, user); and **Is empty / Is not empty** apply to any field (see Empty and Filled require the field to exist). The **Under** operator matches entries that fall under a given ancestor in a tree (hierarchical) entity (**Since 1.2**), and **Type** matches by the stored value's type.

The full operator reference — including which field types each operator applies to and the aggregate operators — is below:

The following condition operators are supported by the query engine:

Operator	Meaning
`Equal` / `NotEqual`	Field equals / does not equal a value.
`LessThan` / `LessThanOrEqual`	Numeric or date comparison.
`GreaterThan` / `GreaterThanOrEqual`	Numeric or date comparison.
`In` / `NotIn`	Field is (is not) one of a list of values.
`Contains` / `DoesNotContain`	Text contains (does not contain) a substring — case-insensitive.
`StartsWith` / `DoesNotStartWith`	Text begins (does not begin) with a value — case-insensitive.
`Like` / `IsNotLike`	Pattern match on text — case-insensitive.
`IsEmpty` / `IsNotEmpty`	Field has no value — not present at all, null, or an empty list / field is present **and** has a value. Requiring the field to be present is `Since 1.8`.
`ElemMatch`	Matches list fields where at least one element satisfies the nested conditions.
`Under`	Hierarchy match — the record sits **under** (is a descendant of) the given record in a tree.
`Type`	Matches on the stored data type of the field.
`And` / `Or`	Combine child conditions.

For fields that hold a **list** of values, a condition can additionally use a *field operator* that first reduces the list to a single value, then compares it:

Field operator	Reduces the list to…
`Size`	The number of elements.
`Sum`	The sum of the elements.
`Avg`	The average of the elements.
`Min` / `Max`	The smallest / largest element.
`First` / `Last`	The first / last element.
`Any`	(Default) match if any element satisfies the condition.

Empty and Filled require the field to exist

`Since 1.8` — **Is empty** (`IsEmpty`) and **Is not empty** (`IsNotEmpty`) no longer test only whether the stored value is null or an empty list. The predicate now also tests **presence**: *Is not empty* matches only when the field exists on the entry *and* holds a real value, while *Is empty* matches an absent field just as it matches a null or empty-list value. Both the data query engine and the reaction query engine compile these two operators from the same shared builder, so a saved query and a reaction criterion always agree on what "empty" means.

The case this corrects is a **reference field inside a repeatable block**. When the block has no instances, the reference is stored as an empty set; the condition is expressed against the field's dotted path, and traversing that empty set used to make the "not equal to null" test succeed — so the entry wrongly passed an *Is not empty* condition even though its repeatable block had no rows at all. Requiring the field to exist excludes those entries.

A follow-on fix closed the same gap for two places that resolve a query through a different internal path: a **reaction** condition testing *Is not empty* on that same kind of field, and a **reference field** whose source query targets an entity with a repeatable block. Before the fix, on that path the predicate silently matched nothing — the reaction never fired,

and the reference field's picker showed an empty list — even though the underlying condition was correct and the fix above was already in place.

Stated limitation — a condition cannot be scoped to a single repeatable-block instance. These operators receive a field path, not a block instance, so there is no way to express "at least one instance of the block has this field filled". A block that mixes an empty instance with a filled one therefore reads as **empty** as a whole. This is **pre-existing behaviour, not introduced by the 1.8 change** — plan around it when you design a condition over a repeatable block, and use a separate marker field on the entry if you need per-instance semantics.

Value sources

The value is the second side of the condition — what is being searched for. It can be:

- **Fixed** — a specific value typed manually and left unchanged.
- **Variable (dynamic)** — taken from another field on the same entry or a related entity, so the comparison is field-to-field (for example *Assigned to* equals *metadata → modified by*).
- **Indicator (pointer)** — a context value that keeps the query relative: `@me` resolves to the current user; membership tokens keep the query relative to the current user's memberships — `myGroups` (the groups the user belongs to), `myRoles` (the user's roles), and `myOrgUnit` (the user's organizational unit); and date pointers such as `@today` compare a date field against the current date when the query runs.
- **Code expression** — a JavaScript expression, used for conditions the other value types cannot express (giving administrators extra flexibility).
- **Parameter** — a placeholder whose value is supplied later, when the query is used (see Parameterized queries).

In the builder these sources are labelled **Fixed**, **Field**, **Pointer**, **Expression** and **Parameter**, and pointers are listed with their `@` prefix (`@me`, `@myGroups`, `@myRoles`, `@myOrgUnit`, `@today`, `@now`, `@StartOfDay`, `@StartOfWeek`, `@StartOfMonth`, `@StartOfYear`, plus further group-membership pointers).

Relations, virtual fields and workflow metadata

For example, on a `Task` entity that has a `department` reference to a `Department` entity, you can filter tasks where **department → region = "West"** without leaving the `Task` query. Internally the engine follows the relation (the `:relation` path marker) and joins the related record so the condition is evaluated on it. Relations can be traversed more than one level deep — but because each level adds a join, keep relation chains reasonably shallow.

If an entity defines **virtual (computed) fields** — fields whose value comes from an expression rather than being stored — you can filter and sort by them like any other field. The engine computes the virtual field first, so the condition is applied to the computed result.

If a virtual field's expression fails to evaluate, the query still returns results (that field is simply skipped) and a warning is shown, so a broken computed field never breaks the whole list.

For entities that have a workflow, you can add conditions on **workflow metadata** — the current step, the current-step owner, previous steps, or previous-step owners, including detection of ad-hoc steps. These options appear when the view or query allows current-step / previous-step filtering.

Field (aggregate) operators

For fields that hold multiple values (multi-value fields, or fields inside a repeatable block), the builder offers **aggregate operators** that reduce the set of values *before* comparing: **Size** (count of values), **Sum**, **Average**, **Max**, **Min**, **First**, and

Last. For example, "block **Size** greater than 3" returns entries whose repeatable block has more than three rows. These operators appear in the operator reference above (in the builder: **Count of values**, **Sum**, **Average**, **Max**, **Min**, **First Value**, **Last Value**, and **Any Value** for the default).

When no field operator is chosen

A condition that does not use an aggregate operator is evaluated with the default **Any** behaviour — the condition matches when any of the field's values satisfies it. `Since 1.8` a condition saved with the aggregate operator left unset is reliably recognised as "no operator chosen" and falls through to that default, instead of being rejected as an unsupported operator. An operator name the engine genuinely does not recognise is still rejected with an error.

This also matters for **reaction criteria**, which are evaluated by the same engine: a criterion with no aggregate operator used to be treated as "not matched", so the reaction silently never fired.

Saving and naming

After building the conditions you can **save the query** for reuse in views, reports, reference fields, dashboards, and other places. `Since 1.10` the Queries tab's new / edit dialog **always saves** — its "save query" toggle and its apply-without-saving mode were removed, and the confirm button is always *Save Query*. To try conditions without saving them, build them in the entries list's **Filter** panel; saved queries are still applied from the queries list.

A saved query's multilingual name is 3–128 characters. `Since 1.10` the query's `normalizedName` is generated by the server, and a clash answers **409** `DATA_QUERY_NORMALIZED_NAME_DUPLICATE` instead of a server error.

Parameterized queries

- When creating or editing a query, choose **Parameter** as the value source and give the parameter a name.
- The same parameter name can be reused across conditions on fields of the **same** data type; it cannot be reused across fields of **different** data types.
- Saving a query that contains parameters classifies it as a **parameterized query**, shown with a dedicated *Parameterized query* tag in the query list. `Since 1.10` a shared **Parameterized** badge marks such queries everywhere an administrator picks one — the Queries tab, the reference / relation / users entries-source pickers, the view-source and summary / dashboard query selectors, a public view's saved-queries list and the admin quick-filter lists. In the Status dropdown and on quick-filter toggles it appears on the admin Entries tab only, never in the end-user list of queries to apply.
- A parameterized query can be used as a source elsewhere — for example in **reference fields** or **views**, where the parameter values are supplied in that feature's settings.

If every parameterized condition is removed (or switched to another value type), the query reverts to a standard query. Note the reverse is blocked: a standard (non-parameterized) query **cannot** be turned into a parameterized one on update.

Parameterized quick queries: per-query values `Since 1.10`

Up to 1.8 a parameterized quick filter took its values from one flat `parameters` object on the request (the Grafana path), so two parameterized quick queries applied together could not carry different values for a parameter they both named. 1.10 gives each applied quick query its own values, and gives end users a way to supply them.

- **What end users see.** Applying a parameterized query — from the Status list, a standalone quick-filter toggle, the saved-queries list or the Filter panel — opens a **Query parameters** dialog that asks for each declared

parameter. The query then runs as a **copy** whose Parameter conditions became Fixed ones; the Filter panel shows them as editable conditions, and saving that copy creates a new query rather than overwriting the parameterized one.

- **The request parameter.** `quickFilterParameters` is an object keyed by query id: `{"<queryId>": {"<parameterName>": value}}`. It is accepted by the entries, tree, summary and count routes (entity-scoped and view-scoped), by the Kanban board and Kanban column-values routes, and by both quick-export routes (Data Views & Faces [Build Apps]).
- **Resolution.** Whether a quick query is parameterized is decided by **counting the Parameter conditions in its criteria** — not by its stored `isParameterized` flag or its `parameters` array, either of which can be stale. For such a query, `quickFilterParameters[queryId]` is the **complete** answer when present (never merged with anything else); otherwise the flat `parameters` object is used, so existing Grafana panels keep working. Names match case-insensitively (`Dept` = `dept`); declared parameters are coerced to their declared type, and a name used in the criteria but never declared takes the raw value.
- **Isolation.** Each quick query's values are filled into a copy of its own criteria, so one quick filter's values can never leak into another's or into the view source's resolved parameters.
- **Missing values** fail the request with HTTP **422** `QUICK_FILTER_PARAMETER_VALUE_MISSING` (`details: {query, parameter}`); `null` counts as missing.
- **Sensitive fields.** The filled criteria is checked against the caller's sensitive-field grants like any other criteria, so a caller without the grant cannot vary a parameter on a sensitive field to probe its hidden value.
- **Never by default.** A view cannot mark a parameterized quick query as applied by default — see Quick filters, groups and ordering [Build Apps › Data Views & Faces].

A template export does not take parameter values, so it leaves parameterized quick filters out and tells the user so; the quick export applies them with their values.

Quick queries

A query saved with **Quick Filter** ticked (`useInQuickFilter`) is listed under **Quick Filter** on the Queries tab. To apply it, open the entries listing, go to the **Status** dropdown (which holds the additional filtering options), and enable the quick query: its conditions are added to the currently applied filters and combined with an **AND**. The filter conditions then appear in the custom filters list as disabled.

`Since 1.10` the quick-filter part of the form also carries:

- **Name** (the quick filter name) — the localized label end users see for this query as a quick filter. Left blank, it is filled from the query name on save; after that the two are independent.
- **Group** — the quick filter group the query belongs to (or none). You can create a group inline with **New group**.

Where a quick query appears in a gallery — inside the Status list or as a standalone toggle — and whether it is applied by default is decided **per view**, not on the query; see Quick filters, groups and ordering [Build Apps › Data Views & Faces]. A quick query is reached by end users only through a gallery that lists it (or enables its group).

Quick filter groups `Since 1.10`

A **quick filter group** classifies an entity's admin quick queries — for example *By region* over *North*, *South* and *West*. Groups are managed on the entity's **Queries** tab, in the **Quick filter groups** sub-tab: create a group, name it in each language, and **Add existing filter** or **Create quick filter** inside it. A gallery then enables the whole group at once.

- **Membership lives on the query** (`quickFilterGroup` , a group id or `null`), so a query is in **at most one** group, and deleting a query needs no group clean-up. Adding a query that is in another group moves it (the dialog warns *Moves from …*).
- Only an entity's **admin quick queries** can be grouped. A query that is not a quick query, or an end user's own saved query, is always stored ungrouped.
- **Names are unique per entity**, compared trimmed and case-insensitively per locale (`G1` , `g1` and `G1` clash) — `QUICK_FILTER_GROUP_NAME_DUPLICATE` .
- **Deleting a group keeps its quick filters**; they simply become ungrouped.

Route	Purpose
`GET` / `POST /v1/entities/{entity}/quick-filter-groups`	List / create.
`PATCH` / `DELETE /v1/entities/{entity}/quick-filter-groups/{id}`	Update / delete (delete ungroups the members, then deletes the group).
`GET /v1/quick-filter-groups[/{id}]`	Raw groups, for packaging and dependency tracking.

Create and update answer with the group plus its member query ids. A `queries` array on a write sets membership **exactly**: listed queries move in, current members not listed are ungrouped, and only the entity's admin quick queries may be listed (`QUICK_FILTER_GROUP_INVALID_QUERY`). A query response carries the read-only `quickFilterGroupName` ; a write body that includes it is rejected with 422. A group of another entity answers 404 (`QUICK_FILTER_GROUP_NOT_FOUND`). Moving a query between groups does not change the query's `updateIdentifier` . Known limitation: a group created after a package export snapshot reaches the dependency graph only on the next rebuild / sync scan.

Tags

Follow the same steps, and before saving enable the **Save with tag** toggle. Enter the tag's multilingual name (3–128 characters), choose its color, and either pick an existing **tag group** or enable **New tag group** and name it. The query is then saved with both its tag name and the *Standard query* tag. When applied from the entries listing (via the **Tag** list, as a **color frame** or **colored background**), the tag color is applied to the entries that match the query.

Changing a query's tag through the API

`Since 1.10` — `PATCH /api/v1/entities/{entity}/queries/{id}` now takes the same tag shape that creating a query already took, and answers with the updated query.

The practical gain is `tag.tagGroupName` . Sending it on an update used to be **silently dropped** — no error, no effect, because the update's request schema had no such property. It now behaves the way it does on create: naming a group that does not exist **creates the tag group on the fly** and stamps the tag with it. And because the response now returns the updated query with the tag group resolved, a client that just created a group inline **learns its id from the same response** instead of issuing a follow-up read. The status code is unchanged; what changed is that the `200` now carries a body.

The tag rules on update, in full:

What you send as `tag`	What happens
Omitted	The stored tag is left alone — the update stays genuinely partial.
`null` , or a tag with `enabled: false`	The tag is deleted outright, taking its group reference with it.
A tag carrying a new `tagGroupName`	The tag group is created on the fly and the tag is stamped with its id.

What you send as `tag`	What happens
A tag carrying a `tagGroup` id that does not exist	The request fails — an unknown group is an error, not a silent no-op.

`tagGroupName` is a request-only convenience: it is never stored on the query itself, which keeps the group's **id**. The audit record for the update is computed against the tag as it ends up, so omitting `tag` is no longer logged as a removal that never happened.

Deleting a query

Open the **Queries** tab, select the query, click the delete icon next to its name, and confirm — deletion is permanent. Some queries cannot be deleted: **implicit** (system-generated) queries, and any query currently in use by a virtual, joiner, or aggregator entity.

The Queries list `Since 1.10`

The entity's **Queries** tab has two sub-tabs, *Queries* and *Quick filter groups*. The Queries list carries the same toolbar as the Reactions tab:

- **Filters** — checkbox filters by query **Type**, **Tag** and **Field**, plus a metadata filter (created / updated by, created / updated date ranges).
- **Sort** — up to three keys among *Last modified* (the default, newest first), *Created date* and *Name*.
- **Group by** — *Type* (default), *Tag* or *None*.
- **Search** — over the query name **and its condition text**.

This is client-side list state; `GET /v1/entities/{entity}/queries` did not change. Separately, a query read now attaches a `_resolved` label to every **Fixed reference value** in its criteria, so the filter panel shows the referenced record's name instead of its id; `_resolved` is stripped again on write, so you never need to send it.

Entries-list behaviours a query relies on

On the entries listing you can type a value into the **search box** to show every entry that matches, across all searchable fields (text, number, email, and so on) with no explicit conditions. Search must be allowed on a field for it to be included — either by enabling search when creating a **view** (search is then limited to the view's added fields) or by enabling the search option on an individual field from its settings.

By default, an entity's entries are shown in a table sorted **descending by modification date**. You can change this in two ways:

- **Sort dropdown** (next to the search box): choose a **direction** and a **sort-by** field. Ascending sorts text and *Created by* / *Updated by* alphabetically (0–9, then A–Z, then أ–ي) and dates from oldest to newest; descending reverses both. You can sort by *Created by*, *Creation date*, *Updated by*, *Modification date*, or any entity field.
- **Field-header sort icon**: click the sort icon in a field's column header to toggle that field between ascending and descending.

Additional filters narrow entries by their **state** — active, archived, obsolete, or non-obsolete. They are reached from the dropdown next to the filter, available to administrators and (when allowed and enabled) to end users through views. Enable the states you want and disable the rest; you can combine states (for example *Archived* and *Obsolete* together). Entry states include **obsolete** (no longer matching the entity's current structure or workflow), **valid** (matching the current structure and workflow), **current** (present in any state), and **archived** (moved to the archive).

Statistic queries

Where a standard query returns a **list** of entries, a **statistic query** aggregates entries into a **number, chart, or list** — for example “count of requests by department”. It reuses the same filter builder, plus an **aggregation function** (Count, Sum, Min, Max, Average), a **group-by** field (dates can be bucketed by day, week, month, quarter, year, and so on; reference and classification group-by values are resolved to their display labels automatically), a **display-as** choice (Number, Chart, or List), and a **sort** direction (ascending or descending on the grouped result). Statistic queries always exclude archived and obsolete entries; very high-cardinality group-bys use a segment limit with an “Other values” bucket. They are the building blocks of dashboards and summary widgets. A statistic query is a different resource from an **entity aggregate**, which totals a column of an entries list in place rather than producing a widget — see Entity Aggregates & Entry Grouping [Build Apps › Totals & grouping].

The statistic-query model and routes are in Summary and dashboard API reference.

Relations, references and mapped fields: model and API

This page is for **integrators and developers** who read or write relation definitions, traverse related entries through the API, or need the exact rules behind reference fields, mapped fields and navigation links.

> **Linking entities in the UI?**
>
> Builders should start with Link entities [Build Apps] — it covers reference fields, the Relations tab, cascading pick-lists and mapped fields in the labels the designer shows.

This page covers the distinct — but easily confused — mechanisms OSOS uses to link data across entities, all owned by the **data-service**:

1. **Data relations** (`DataRelation`) — first-class, queryable *definitions* of how one entity relates to another. They power the "related entries" tabs, sub-lists, the entity **Joiner**, and relation-sourced reactions.
2. **Reference fields** — `Entity` -typed fields on an entity that hold the actual pointer to a referenced entry (the `source` / `sources` configuration).
3. **Mapped fields** — fields whose value is copied or synced from a referenced entry, governed by an update-mode policy.

It also documents **navigation links** (`LinksService`) — the sidebar link tree and shareable system links — because that area shares the "links & references" vocabulary.

All entity/entry-scoped endpoints below are mounted on `DataRelationController` , which requires the `Auth2` authentication strategy and runs the `entityVoter` authorization voter for every operation.

Concept map

Concept	Where it lives	What it stores	Direction
Data relation (incoming)	`DataRelation` document	Explicit definition owned by `dataEntity` , pointing at a `targetField` on `targetEntity`	Incoming (`relationType = 1`)
Data relation (outgoing)	*Derived at read time* from the entity's own reference fields	Synthetic, never persisted	Outgoing (`relationType = 0`)
Reference field	`DataEntityField` with `dataType = Entity` and a `source` / `sources`	The referenced entity + query used to pick an entry	n/a (field-level)
Mapped field	`DataEntity.dataSources[]` + field-level `mapping`	`fromFieldId → toFieldId` value copy with an `updateMode`	n/a (value sync)
Navigation link	`LinkDataEntry` (link entity)	Sidebar node pointing at a module/view/page/report	n/a

How entities link

Before the model details, it helps to see the two everyday ways an administrator connects entities in the product:

1. **Reference field** — a field of data type **Entity** on entity A that points to entries of entity B. Internally the reference is stored as a universal pointer, `ReferenceProperty`, a `{ dataEntity, dataEntry }` pair (entity id + entry id), usually accompanied by a `_resolved` block carrying the related entry's display title. Storing the pointer rather than plain text keeps the link stable even if the target entry's display title later changes.
2. **Data relation** — a first-class, named relationship between two entities (incoming/outgoing), managed from the **Relations** tab of the entity. Relations let you traverse related entries, show them as sub-lists, build **Joiner** virtual entities, and drive relation-sourced reactions.

Data relations are also what let you build **nested, hierarchical organizational structures**: an entity representing an organization can act as the parent of child entities representing its departments or projects, so data is organized according to the parent–child hierarchy between them.

The `DataRelation` model

A `DataRelation` is a document in the data-service. It extends `Identified<UUID>` and implements the shared `IDataRelation` interface.

Field	Type	Required	Notes
`id`	UUID	(server)	Primary key.
`specificationVersion`	number	yes	Defaults to `1`.
`dataEntity`	UUID	yes	The entity that **owns** this relation definition.
`name`	LocalizedValue	yes	Display name; validated by `LanguagesService.validateName`.
`description`	NullableLocalizedValue	no	
`icon`	Icon	yes	
`targetEntity`	UUID	yes	The entity on the other end.
`targetField`	string	yes	`fullNormalizedName` of the reference field that realizes the link (see below for which side it lives on).
`relationType`	`DataRelationTypeEnum`	yes	`Outgoing = 0`, `Incoming = 1`.
`normalizedName`	string	(auto)	Machine name, unique per `dataEntity`. Pattern-validated (`NormalizedNameValidation`). **Since 1.6**
`isImplicit`	boolean	no	Defaults to `false`. Implicit relations are read-only (see below).
`updateIdentifier`	string	yes	Optimistic-concurrency token (a UUID). **Since 1.6**

The `NewDataRelation` request body for create **excludes** `id` and `updateIdentifier` (both are server-assigned); the PATCH body excludes only `id`.

Incoming vs. outgoing

This is the single most important thing to understand:

- **Incoming relations are the stored `DataRelation` documents.** They describe entries in `targetEntity` that point *into* `dataEntity` via a reference field named by `targetField` (that field lives on `targetEntity`). In the product, an incoming relation is one where the current entity acts as the "parent" entity, defined and modified by an administrator from the Relations tab.

- **Outgoing relations are computed on the fly** from `dataEntity`'s own reference fields (`getEntityRelationFieldsWithoutADT()`), never persisted. Each of the entity's reference fields yields one; their synthetic `id` is a composite key (see below) and their `targetField` is the `fullNormalizedName` of a field on `dataEntity` itself. In the product, an outgoing relation is one where the current entity acts as the "child."

`transformToRelationDto` reflects this asymmetry: for an **incoming** relation it resolves the field on the *related* (target) entity; for an **outgoing** relation it resolves the field on the *owning* entity.

The V2 list/count endpoints return both buckets. For example `findV2` builds `outgoingRelations` by walking each relation field's `sources[]` (or the legacy single `source`) and emitting one synthetic `DataRelation` per source with `id = "{dataEntity.id}:{field.id}:{source.entity}"`.

Managing relations from the Relations tab

In the product, both incoming (inbound) and outgoing (outbound) relations are viewed from the entity's **Relations** tab, each shown as a card carrying the relation's name, icon, related entity, and relation field. Clicking a card opens a form with the relation's details; entering new values and clicking **Finish** saves the change. To remove a relation, click the delete icon next to it: a confirmation prompt appears, and once confirmed the relation is permanently deleted and cannot be undone.

Implicit relations

A relation with `isImplicit = true` is system-managed and locked:

- `PATCH` throws `ImplicitRelationUpdateException` (`IMPLICIT_UPDATE_RESTRICTION`).
- `DELETE` throws `ImplicitRelationDeleteException` (`IMPLICIT_DELETE_RESTRICTION`).

System-wide list/count endpoints hide implicit relations unless you opt in. `applyImplicitFilter` / `applyImplicitWhere` merge an `isImplicit: false` constraint using an explicit `{and: [...]}` conjunction (so a caller's nested `or` cannot leak implicit rows) whenever the `includeImplicit` query flag is `false` (the default). **Since 1.8**

Example: the "Themes" incoming relation on the Application Shell `Since 1.10`

The core **Application Shell** entity (`SHELL`) shows the incoming-relation pattern above in a system-managed form. Data-service migration `1.10.0-a28` creates the implicit relation `SHELL_THEMES_RELATION` on it:

Property	Value
`dataEntity`	`SHELL`
`targetEntity` / `targetField`	`THEME` / `shell`
`relationType`	`Incoming`
`normalizedName`	`themes`
`name`	"Themes" / "السمات" — "Themes that use this application shell"
`isImplicit`	`true`

The Application Shell has no outgoing field to themes: the relation simply lists the `THEME` entries whose `shell` reference field points at the shell entry. Because it is implicit it cannot be edited or deleted through the Relations API, and it is hidden from system-wide listings unless you pass `includeImplicit=true`. The migration writes the relation directly, so no `DataRelationCreatedEvent` is published for it.

An Application Shell entry is stored layout data that a theme can point at. The 1.10 web client does not render it, and no shell builder is part of 1.10.

`normalizedName` and concurrency

On **create**, if the client supplies `normalizedName` it must be unique within the owning entity (`DataRelationNormalizedNameDuplicateException` / `DATA_RELATION_NORMALIZED_NAME_DUPLICATE`); otherwise it is auto-derived from `name` via `generateUniqueNormalizedName(name, existingNames, 'relation')`. The uniqueness check runs against all sibling relations of the same `dataEntity`.

`updateIdentifier` is an optimistic-lock token. On create it is set to a fresh `v4()`. On update, if the client sends an `updateIdentifier` that differs from the stored one, the write is rejected with `UpdateIdentifierConflictException`; otherwise the server rotates it to a new UUID. **Since 1.6**

Relations API

All routes below are on `DataRelationController`. `{entity}` accepts an entity **id or normalized name** in its space (`getEntityByIdOrNormalizedNameInSpace`).

Method & path	Handler	Purpose
`POST /v1/entities/{entity}/relation`	`create`	Create a relation for `{entity}`. Publishes `DataRelationCreatedEvent`.
`GET /v1/entities/{entity}/relation`	`find`	List stored (incoming) relations of `{entity}` as `DataRelationDto[]`.
`GET /v2/entities/{entity}/relation`	`findV2`	Return `{incoming, outgoing}` (`DataRelationListDto`).
`GET /v1/entities/{entity}/relation/count`	`count`	Count stored relations (optional `where`).
`GET /v2/entities/{entity}/relation/count`	`countV2`	`{incoming, outgoing}` counts (`DataRelationCountDto`).
`GET /v1/entities/{entity}/relation/{id}`	`findById`	One relation of `{entity}` as `DataRelationDto`.
`GET /v1/relations`	`findAll`	System-wide list. `includeImplicit` (default `false`); `limit` clamped to `[1, 500]`, default `100`.
`GET /v1/relations/{id}`	`findOneById`	One relation by id (supports composite ids).
`GET /v1/relations/count`	`countAll`	System-wide count. `includeImplicit` (default `false`).
`POST /v1/entities/joiner/relations`	`findEntitiesRelations`	Given `{dataEntities: string[]}`, return the union of their incoming relations.
`PATCH /v1/entities/{entity}/relation/{id}`	`updateById`	Update (blocked for implicit). Publishes `DataRelationUpdatedEvent` with a raw diff.
`DELETE /v1/entities/{entity}/relation/{id}`	`deleteById`	Delete (blocked for implicit). Publishes `DataRelationDeletedEvent`.

Related-entries endpoints

Given an entity, an entry, and a relation, these return (or count) the entries on the *other* end. Internally each resolves the relation, loads the `targetEntity`, and delegates to `DataEntryService` factories with a `relation` context of `{targetField, entityId, entryId, relationType}`.

Method & path	Handler	Extra inputs
`GET /v1/entities/{entity}/entries/{entryId}/relations/{relationId}`	`findRelatedEntries`	`filter`, `include_virtual`
`GET /v2/.../relations/{relationId}`	`findRelatedEntriesV2`	`filterCriteria`, `search`, `include_virtual`
`GET /v3/.../relations/{relationId}`	`findRelatedEntriesV3`	`+ appliedQuickFilters`, `appliedTagsFilters` → `{entries, tags}`
`GET /v1/.../relations/{relationId}/kanban/{categoryFieldId}`	`findRelatedEntriesKanbanView`	grouped board + `categoryId`, tags/quick filters
`GET /v1/.../relations/{relationId}/count`	`findRelatedEntriesCount`	`where`, `include_virtual`
`GET /v2/.../relations/{relationId}/count`	`findRelatedEntriesCountV2`	`whereCriteria`, `search`

The V3 route accepts `appliedTagsFilters`, whose tag criteria are resolved through `TagsService.getTagsFilterCriteria(targetEntity.getId(), appliedTagsFilters)` before being applied.

Composite (field-level) relation IDs

`relationId` is polymorphic. `DataRelationRepository.findRelationById` accepts:

- **A plain UUID** — resolved with `findById`.
- **`{entityId}:{fieldId}:{sourceEntityId}`** (3 parts) — resolves the relation field `fieldId` on `entityId`, verifies `sourceEntityId` exists in the field's `sources[]` (or matches the legacy single `source`), and synthesizes an **outgoing** relation with `targetEntity = sourceEntityId` and `targetField = field.fullNormalizedName`. Unknown field or source → `RelationNotExistException` (`RELATION_NOT_EXIST`).
- **`{entityId}:{fieldId}`** (2 parts) — same, using the field's legacy `source` as `targetEntity`.

This is why the kanban and related-entries routes work for outgoing relations that have no stored `DataRelation` document.

Create example

```
POST /v1/entities/orders/relation
Content-Type: application/json

{
  "specificationVersion": 1,
  "dataEntity": "…orders-entity-id…",
  "name": { "en": "Line items", "ar": "…" },
  "icon": { … },
  "targetEntity": "…order-lines-entity-id…",
  "targetField": "order",          // field on order-lines that points back to orders
  "relationType": 1                 // Incoming
  // normalizedName optional; updateIdentifier must be omitted
}
```

Events

Relation mutations publish domain events on the service bus (`correlationId = traceId`, plus sticky attributes for `userId`, `delegateeId`, `delegationId`, `spaceId`):

- `DataRelationCreatedEvent(relation, userId)`
- `DataRelationUpdatedEvent(updatedRelation, userId, rawDiff)` — `rawDiff` computed with `just-diff` between the pre- and post-update documents.
- `DataRelationDeletedEvent(relation, userId)`

Reference fields

A **reference field** is a `DataEntityField` with `dataType = Entity` that carries a source. `getEntityRelationFieldsWithoutADT()` selects them: `dataType === Entity`, has `source` or a non-empty `sources[]`, is not a virtual field, and its source is not an ADT (audit) entity.

- `source?: UUID` — **deprecated** single source entity id. Still honored across the code (the composite-id 2-part form and the "single source (legacy)" branch in `findV2`).
- `sources?: FieldSources[]` — the current model. Each `FieldSources` entry:

Property	Meaning
`entity`	Source entity id the field references.
`entriesSource`	Optional query id used to populate the picker (nullable).
`entriesSourceParameters`	Parameters bound to a parameterized `entriesSource`.
`entriesSourceDependencies`	Full normalized names of fields the query expressions depend on (computed by the backend to know when to refresh the dropdown).
`relatedTo`	If non-null, the parent field's `fullNormalizedName` this field cascades from.
`relation`	The `DataRelation` id used for cascaded entries (only meaningful when `relatedTo` is set).

Reference field configuration

When an administrator configures a reference field, they set:

- The **source** that supplies the selectable options: a store, collection, query, or an entity/view/module.
- **Single vs. multiple** selection, and **tree vs. flat** selectors.
- **Context filter pointers** (such as *Me, My groups, My roles, My org unit*) to scope the options to the current user.

Special cases

- **Cascaded-parent filtering** — a reference field's options can be constrained by the value of another reference field on the same entry (this is what `relatedTo` and `relation` above express).
- **Circular-relation denial** — the platform prevents circular relations that could cause infinite references.
- **Resync** — a reference field can declare fields whose change recalculates it (`resyncFields`), with a **preserve value on edit** option (`preserveValueOnEdit`) to keep the recalculated value rather than overwrite it.

Display and selection behaviour `Since 1.8`

- **A collection reference always shows its resolved main title in view mode.** A reference that points at a data collection rather than an entity carries only an id plus a resolved main title, so there is no `mainTitleField` to derive from it. View mode renders the main title regardless, so the value is displayed instead of coming out blank.

- **Search-picked values survive a dropdown reload.** Reference pickers are backed by server-side search, so the option list only ever holds the current page of results. A field that holds **multiple** values keeps every id the user has already selected, even after the dropdown reloads its options for a different search term — the selection, not the visible page, is authoritative, and deselecting an id still removes it.
- **An unresolved single value normalizes to empty.** On a single-value reference field, an id that resolves to no payload settles as an explicit empty value rather than lingering as a broken reference or propagating an undefined one.

Mapped fields

Mapped fields copy values from a referenced entry into the current entity when an entry is created or updated. This is implemented by `MappingFieldService`.

The update modes appear in the field settings as **Local Update** (`Local`), **Never** (`Never`), **From Parent Only** (`ParentOnly`) and **2 way** (`TwoWay`). In the form, a mapped field in **Never** or **From Parent Only** mode is read-only; **2 way** is offered only when the mapped field's *required* setting matches its source field's, and a mapped field inside a repeatable block that differs from its reference field's block can only use **Never** or **From Parent Only**.

Configuration

Three pieces of config drive mapping:

1. **Entity-level** `dataSources: DataEntityFieldSources[]` — each source is `{ referenceEntity, referenceField, mappedFields: MappingFields[] }`. `referenceField` is the id of the reference field on *this* entity whose selected entry supplies the values.
2. **Per-mapping** `MappingFields` — `{ fromFieldId, toFieldId, updateMode }`, where `fromFieldId` is a field on the referenced entity and `toFieldId` a field on this entity.
3. **Field-level** `mapping: DataEntityFieldMapping` — `{ enabled, referenceFieldId, sourceFieldId, updateMode, searchMode }`. A target field is only written if `mapping.enabled` is true. `searchMode` (`MappedFieldSearchModeEnum`: `None`, `Full`, `StartsWith`, `Partial`) controls how the mapped value participates in search.

Update modes

`MappedFieldUpdateModeEnum` decides who wins when the source and local values disagree. Behavior is implemented in `handleMappingFields`:

Mode	Value	Effective behavior
`Local`	0	Keep the local value **unless the reference changed** (`isReferenceChanged`); on a reference change, pull the source value. Preserves local edits between reference changes.
`Never`	1	Keep the local value whenever it is defined (or when there is no source field); otherwise seed from source. Existing local values are never overwritten.
`ParentOnly`	2	Falls through to the default branch: always take the source (`fromFieldValue`). Source is authoritative.
`TwoWay`	3	If the reference changed and the caller did **not** set `childToParentHighPriority`, take the source; otherwise keep the local (child) value. Lets a child edit propagate back toward the parent. **Since 1.6**

`isReferenceChanged` compares the referenced entry id currently stored on the entry against the incoming `sourceEntryId`. In the entity-wide pass (`getValuesByEntityDataSources`), `handleMappingFields` is called with `childToParentHighPriority = true`, so under `TwoWay` the child's value takes precedence.

Resolution flow

`getValuesByEntityDataSources(spaceId, dataEntity, dataEntry, isUpdateMode)` returns a `Map<toFieldId, value>` (values may be arrays for repeatable blocks):

1. For each `dataSources` entry, load the `referenceEntity` (cached in a local map) and locate the `referenceField` by id.
2. Read the reference value at `referenceField.fullNormalizedName`. If empty and the field's `options.autoSelectFirstValue` is set, fall back to `DataEntryService.getDefaultFirstValue`. If still empty and the field is calculated or has an on-form-opened trigger, run `calculateField`.
3. If the reference field sits inside a **repeatable block** (`isParentBlockRepeatable` + `blockNormalizedName`), iterate each block instance; a block with no `dataEntry` reference pushes `null` for every mapped target. Otherwise resolve the single referenced entry.
4. For each mapped field, `handleMappingFields` loads the source entry (`state = Active`, `isInvalid = false`) and applies the update-mode rules above. Repeatable target blocks are matched by the block whose reference field's `dataEntry === sourceEntryId`.

Only fields whose `mapping.enabled` is true are populated; disabled targets are skipped.

Propagation after the source changes

Separately from the create/update pass above, an event handler pushes changes made on a **parent** entry to entries that map from it, for mapped fields in `ParentOnly` or `TwoWay` mode; for a `TwoWay` mapped field, a change made on the child is also written back to the parent entry (`UpdateMapFieldValueEntryCreatedHandler` / `UpdateMapFieldValueEntryUpdatedHandler` in data-service). `Local` and `Never` targets are never refreshed from a later parent change.

Navigation links

`LinksService` builds the authenticated sidebar tree — a different kind of "reference": a link node that points at a resource. Links are entries of the link entity (`LINK_ENTITY_NORMALIZED_NAME`).

`LinkType` values: `Module = '0'`, `View = '1'`, `Page = '2'`, `Report = '3'`, `Untyped = '4'`.

> **Reports are web-only.** A report link opens only in the web app; on mobile it resolves to a `NotSupportedLink` and cannot be navigated to.

Sidebar links (the Links entity)

Sidebar entries are managed through the **Links entity**, whose entries can be either manually owned or system-owned. System-owned links are maintained automatically: when a module, view, or report is created, updated, or deleted, the system creates, updates, or removes the corresponding Links entry. Manually added entries let an administrator place custom nodes in the sidebar, choosing type (Module, View, Page, Report, Untyped), icon, name, description, parent (for hierarchy), publish state, and permissions (Authenticated, Anonymous, Restricted, Everyone). Entries are ordered by the **Order** field, smallest to largest. Each manual entry also has a **Target** section whose sub-fields depend on the type: **Link** (the URL, used when the type is Uncategorized), **Target ID** (auto-filled for system-owned entries), and **Target View Action** (the view link, used when the type is View). Entries with **system ownership** inherit all their settings from their source — a View entry, for example, adopts its settings from the view configuration.

`getLinksForSidebar(domain, spaceId, userId, delegationId?)` assembles the tree:

- **Anonymous** (no `userId`): includes links whose `authentication.type.id` is `0` (Anonymous) or `2` (Everyone), plus `1` (Authenticated) links whose `value.type.id` is `'0'`.
- **Authenticated**: includes types `1` and `2`, plus `3` (Restricted) links where the user (or one of their groups) appears in `value.authentication.users` / `groups`.
- All links must be `published: true`, `state: 1`, match the domain (empty `domain` = all), and satisfy sharing (`ApplicationSharingLevelEnum` Public, or Private/Shared scoped to `spaceId`). Ordered by `value.order ASC`, then `value.title.en ASC`.
- When a `delegationId` is present, `filterLinksByDelegatedPermissions` drops view/report/page links whose target id is not among the delegatee's allowed resources.
- The tree is assembled by parent id; a `Module` node with no children is pruned. View links whose parent is hidden are skipped.

Related helpers: `createLinkForModule` builds a system-owned module `LinkDataEntryValue`; `archiveExistingLinks(targetId, userId, correlationId)` sends an `ArchiveDataEntriesCommand` for all active links pointing at a target (used to clean up when the underlying resource goes away).

System links (shareable links)

Besides sidebar links, OSOS provides **system links** — shareable, token-based links that resolve to a specific target: a data view, a page/generic view, a report, or a module. They carry:

- **Token** — a unique token bound to the target; the system mints a signed link and resolves the token when the link is opened, directing the visitor to the intended target.
- **Target** — the view, page, report, or module the link resolves to.
- **Authentication type** — **anonymous** (public access without signing in) or **authenticated** (requires login).
- **Validity / expiry** — a validity period and expiry date; a link can be enabled, disabled, have its validity updated, or be removed.

Anonymous links expose public views to unauthenticated visitors; to protect these views the system adds a human-verification (CAPTCHA) step, and the target can be inspected before sharing via Preview Mode.

Constraints & gotchas

- `targetField` **side depends on direction.** Incoming: `targetField` lives on `targetEntity`. Outgoing: it lives on the owning `dataEntity`. Get this wrong and `transformToRelationDto` will fail to resolve the field.
- **Outgoing relations are ephemeral.** They have composite ids and no document — you cannot PATCH/DELETE them; those routes operate on stored documents via `findById`.
- **Implicit relations are immutable** through the API.
- `updateIdentifier` **is mandatory for safe updates.** Omitting it skips the conflict check (the guard only fires when a differing value is supplied), so clients should round-trip the token.
- `findAll` **caps results** at 500 and defaults to 100 — paginate for large tenants.
- **Mapping only writes** `mapping.enabled` **targets**, and `Never` / `Local` deliberately protect local edits; a value "not updating" is often the intended update-mode behavior, not a bug.

Behaviour history

- **Adding a mapped or application-reference field required saving twice.** The entity designer's field list renders each item bound to a plain form control, but the mapped-field and application-reference-field dialogs used to insert a differently-shaped (nested) control — the new field appeared in the list, but a save did not pick up its value until something else re-wired it. Fixed by inserting the same plain-control shape the rest of the field list uses.

Users API & identity accounts

This page is for integrators and engineers who create or manage users programmatically, or who need to know exactly what the user screens in Administration do underneath. The administrator's view of the same tasks, with the screens, is Managing users [Administration]. Sign-in itself (identity server, providers, second factors) is covered for operators in Identity server & external providers.

The two-plane user model

A "user" is represented in **two** places that are kept in sync automatically:

Plane	Service	Storage	Holds
Business record	`data-service`	The `USER` data entity (data-entry documents)	Profile fields (`basic_info` , `account_information` , `phone_number`), org-unit/group assignments, `is_deactivated` flag
Identity account	`users-service`	The Mongoose `user` collection (`UserSchema`)	Credentials (`username` , hashed `password`), `email` , MFA config, login-lockout counters, `isDeactivated`

You (or an integration) manage the **business record**, the `USER` data entry, through the normal data-entry create/update/archive flows. The `users-service` listens for the resulting domain events and provisions, updates, or removes the matching **identity account**. You never create the identity account directly; it is a side effect of the data-entry lifecycle.

Only entries whose user type is **Normal** (`basic_info.user_type.id === 'NORMAL_USER'`) become sign-in accounts. **Application** users (`APPLICATION_USER`) get an API token instead (see Application users).

Provisioning: what happens when a USER entry is created or changed

Every create, update, archive or restore event for a `USER` entry triggers one **reconcile**: the identity records are brought in line with the entry *as it currently is* (re-read at handling time). Because of this, the order in which the events arrive does not matter: an update that arrives before its create is no longer lost. `Since 1.10`

For an **active Normal** entry, the reconcile upserts the identity account:

1. It looks for an existing account by the entry id, then by email (only if that account has the same id).
2. **Existing account** → copies `username` and `email` (only when non-blank, so a blank value never overwrites), and always copies `isExternal` , `externalUserId` , `isDeactivated` , `phoneNumber` and `phoneVerified` . The password is never touched.
3. **No account, but another account already owns the email** → nothing is created; the conflict is logged. This is how email de-duplication works.
4. **No account** → creates one with a **random UUID as the initial password**. This placeholder is not a usable credential.
5. For a newly inserted **internal** user (`isExternal` not true), it creates a reset token and, if the password-reset email template resolves, publishes the "set your password" email. The email is sent exactly once, by the call that inserted the account. **External** (federated or directory-imported) users get no email; they authenticate through their external provider.

An **archived** or missing entry removes the identity account; a **draft** entry is left alone.

Required identity fields. `username`, `password` and `email` are required on the identity account. If `account_information.username` or `account_information.email_address` is missing on the entry, validation fails and the account is not created (logged, not retried), so always populate both.

Uniqueness and case-insensitivity. Email and username lookups are case-insensitive, so `Jane@x.com` and `jane@x.com` resolve to the same account. Email uniqueness at creation is an application-level check, not a database constraint; only `externalUserId` has a true unique index, so two accounts cannot share the same external identity id.

Handler reliability. The provisioning handlers re-throw on failure so the event bus retries the message, with one exception: a Mongoose `validateSync` failure during account creation or restore is logged and the message is dropped instead of retried, so a malformed entry cannot get stuck in a poison-message retry loop.

Organization account and assignments. The assignment validator refuses an `ASSIGNMENT` for a user whose `account_information.organization_account.id` is not `ORGANIZATION_ACCOUNT_TRUE` ("User must be an organization account").

Editing

Editing flows through the `USER` data entry and the identity account follows automatically (see step 2 above). Editing the entry does **not** change the password; passwords change only through the reset and change flows.

The deactivation state is mirrored on every edit: the identity account's `isDeactivated` is set from the entry's `is_deactivated` whenever the `USER` entry is updated, not only through the dedicated activate/deactivate endpoints.

Archive and restore

Archiving the `USER` entry **hard-deletes** the matching identity account: its credentials and MFA configuration are removed.

Restoring an archived user re-provisions the identity account with a fresh random password and, for internal users, a new reset link, carrying over the deactivation state. Because a new password is assigned, a restored internal user must set a new password through the reset email before signing in. `Since 1.6`

Passwords

Admin-initiated reset

An admin-initiated reset requires the `SystemAdmin` role and publishes a reset event; `users-service` then creates a reset token and emails the user. The admin never sees or sets the password.

Password policy

Rule	Requirement
Minimum length	8 characters
Upper + lower case	at least one of each
Letters + digits	at least one of each
Special character	at least one symbol

Passwords are hashed before storage; the plaintext is never persisted.

Self-service flows

Users manage their own passwords through pages served by the identity service:

- **Change password**: requires an authenticated session. Not available to external accounts. Validates the old password, forbids reusing it, and enforces the password policy.
- **Request reset**: a public form. A reset email is sent only if the account exists, is **not external**, and is **not deactivated**. Lookup is case-insensitive, and the form always shows a success message whether or not a matching account was found (no account-enumeration signal).
- **Complete reset**: validates the reset token, applies the password policy, updates the password, and invalidates the token.

Reset-token lifetime and hashing are described in Identity server: password reset.

Deactivation and activation

The activate/deactivate endpoints require the `SystemAdmin` role, with two guards:

- **You cannot deactivate or activate yourself**: the request is rejected with `400 Bad Request`. (The admin screens also hide the actions on your own row.)
- **Idempotency guard**: deactivating an already-deactivated user, or activating an already-active one, returns `409 Conflict`.

Deactivation is enforced at two layers: **login is blocked** (a deactivated account cannot sign in, and an existing session cannot resolve it), and **active tokens are revoked**, so existing sessions are cut off rather than lingering until expiry. The sign-in page shows the `ACCOUNT_DEACTIVATED` message ("The account is deactivated, Please contact with your system administrator"). Reactivating clears the flag; the user signs in again with their existing credentials.

Deactivation removes every session record `Since 1.10`

The identity service keeps an index of every record that belongs to an account, and deactivation deletes exactly those: grants, sessions, access tokens and refresh tokens, so nothing lingers. Before 1.10, refresh tokens could not be found by the old scan and grant and session records were never deleted. The visible rule did not change (a deactivated account already could not sign in or refresh); the difference is that no stale record survives. Records written before the upgrade are not indexed and expire on their own lifetime.

Account lockout

Lockout is automatic and protects against brute-force sign-in:

- Each failed login increments the failed-attempt counters and stamps the last-failed-attempt time.
- When failed attempts reach **5**, the account is locked. The lockout window is set to the current time plus the number of failed attempts since the last success (in minutes), so the window grows with repeated lockouts.
- A locked account is refused login until the lockout window elapses ("Account is locked").
- A successful login clears all counters and the lock, and stamps the last-login time.

There is no admin "unlock" endpoint; a lockout clears when its window elapses or when the user next authenticates successfully.

Multi-factor authentication `Since 1.8`

Per-user MFA configuration is stored on the identity account:

- **TOTP** (authenticator app) config, with the shared secret stored encrypted and a confirmation flag once the user completes enrollment.
- A **preferred method**: one of `totp`, `sms`, `email` or `passkey`.

Enrollment, enforcement at login and management are user-facing sign-in flows rather than admin CRUD. From the admin side, MFA travels with the identity account: it is provisioned, deleted and restored with it. TOTP enrollment, passkeys and password step-up are refused for external accounts.

User profile

The current user's profile is a cached read model that assembles:

- The subject id, name and email.
- The user's groups, split into user groups, roles, org-unit groups and query groups (virtual groups are filtered out).
- The user's collections and main org unit.
- The fully resolved permission set (or the delegate's permissions when acting under a delegation).

Related read endpoints return the user's non-virtual groups, their direct manager (walked from the org-unit parents), and the members of their org unit together with its parent and child org units. See Groups, org units & membership.

The `SYSTEM` user is a special case: it short-circuits to a synthetic profile with empty groups and permissions rather than being assembled from stored data.

Group and collection membership checks

Alongside the profile, the platform exposes membership checks used across authorization and workflow to answer "does this user belong to X":

- **Static group membership**: whether the user belongs to a given `GROUP`.
- **Dynamic query-group membership**: whether the user belongs to a query group. Query-group membership is materialized into `USER_QUERY_GROUP_ASSIGNMENT` lookup entries by background handlers, so the check is a fast existence test against active assignments rather than a live re-evaluation of the query. `Since 1.8`
- **Collection membership**: whether the user belongs to a given collection.

User preferences (user state)

Per-user UI preferences are stored in the `USER_STATE` data entity. They are per-user, not admin-managed. A default state is created on first access, can be updated, and can be reset back to defaults.

Field	Meaning
`firstLogin`	`true` for a freshly created state; drives first-login UX
`language`	Preferred language
`theme`	Preferred theme
`fontSize`	UI font size

Field	Meaning
`menuMode` / `layoutMode`	Menu display and layout modes
`timeZone`	Preferred time zone
`dateFormat` / `timeFormat` / `dateTimeFormat`	Preferred date/time formatting

The default state sets `firstLogin: true` and leaves every preference empty; the client falls back to organization or system defaults until the user chooses.

Application users

Application users represent external systems. They are `USER` entries with `basic_info.user_type.id === 'APPLICATION_USER'` and an `application_information` block; they authenticate with an **API token** sent in the request header, and the platform treats each request as coming from the application user. Token mechanics, headers and revocation are in API tokens.

For an active application entry, the reconcile upserts the API token from `application_information` (`name`, `expiry_date`, `unlimited_expiry`); the token's active flag follows `is_deactivated`. A new application user's token is minted and announced only once, and only to the person who **created** the entry; if that person's identity record cannot be found, no token is created.

Regenerating a token `Since 1.8`

Before 1.8, an application's token was minted exactly once, when its entry was created; rotating it meant deleting and recreating the application and losing its identity, access rights and audit trail. A **SystemAdmin** can rotate it in place:

- The new token is **emailed to the administrator who requested it**. It is never shown on screen or returned by the API.
- The replaced token keeps authenticating for a configurable grace window, so an in-flight integration is not broken the instant the button is clicked.
- **Archived** applications cannot be regenerated (regeneration can create a token document from scratch, which would resurrect a credential for a decommissioned application); deactivated ones can.

The **Regenerate Token** action is offered wherever an application user's row menu appears: the application entry's own header action menu, the user list row menu in **Entity Manage**, and the user list row menu in **Settings**. All three share one confirmation dialog and send the same request. The full rotate-versus-self-heal mechanics and the grace-window configuration are in API tokens: regenerating a token.

Expiry date and issuer `Since 1.10`

- **Expiry date.** The application user form's **Expiry date** field (`application_information.expiry_date`) used to be seeded permanently hidden, so no application user could be given an expiry and every API token was created without one. Migration `1.10.0-a42` makes it a **Date** field that is hidden only for normal users or when unlimited expiry (`unlimited_expiry`) is on; a hide condition an administrator had already set is kept. It is not mandatory. A token validates only against a plain `yyyy-MM-dd` date: an entry whose expiry was stored earlier as a timestamp is listed by the migration (by entry id, not rewritten), and its token is treated as **expired** until the date is re-entered.
- **Issuer.** Migration `1.10.0-a34` adds a hidden, read-only `issuer` field to the same block. The marketplace flow uses it to mark the application users it creates, and those users are left to the marketplace when tokens are reconciled. Administrators neither see nor set it.

- **A new tenant's first administrator** (in a multi-tenant deployment) receives an identity account and a set-password email when the tenant becomes active. See Per-tenant identity.

Admin and read endpoints

On `data-service`'s `UserController` (all under the `Auth2` strategy):

Method & path	Auth	Notes
`POST /v1/users/{userId}/reset-password`	`SystemAdmin`	Admin-triggered password reset email
`POST /v1/users/{userId}/regenerate-app-token`	`SystemAdmin`	Rotate an application user's API token; rejects archived entries
`POST /v1/users/{userId}/deactivate`	`SystemAdmin`	400 on self; 409 if already deactivated
`POST /v1/users/{userId}/activate`	`SystemAdmin`	400 on self; 409 if already active
`GET /v1/users/profile`	Authenticated	Current user's profile
`GET /v1/users/my-groups`	Authenticated	Non-virtual groups
`GET /v1/users/direct-manager`	Authenticated	Direct manager
`GET /v1/users/my-orgunit`	Authenticated	Org-unit peers + parents + children
`GET /v1/users/state`	Authenticated	Get/create user state
`PUT /v1/users/state`	Authenticated	Update user state
`PUT /v1/users/state/reset`	Authenticated	Reset user state

Creating, editing, archiving and restoring users uses the generic data-entry endpoints on the `USER` entity. See Writing data.

Version notes

- **Restore archived user** (re-provisioning the identity account): `Since 1.6`.
- **Multi-factor authentication** (per-user MFA/TOTP config): `Since 1.8`.
- **Dynamic query-group membership** (materialized `USER_QUERY_GROUP_ASSIGNMENT` lookups): `Since 1.8`.
- **Regenerate an application's token**: `Since 1.8`.
- **Application user expiry date (settable), issuer field, order-independent identity reconcile, complete removal of session records on deactivation**: `Since 1.10`.
- Activate/deactivate, admin-initiated password reset and user preferences (user state) predate the 1.2 milestone and are **long-standing** features.

Aggregate and grouping reference

Availability

Since 1.10.

This page is for **integrators and developers** who read totals or grouped lists through the API, build a client that renders them, or need the exact rules behind the figures. It states every rule the server enforces, the reason it holds, and the routes.

Setting up totals in the UI?

Builders should start with Totals & grouping [Build Apps]. The entity tab that holds the definitions is labelled **Aggregators** in the designer; this page calls one definition an *aggregate*. They are the same thing. For what a reader sees and does with a grouped list, see Entries [Using OSOS › Create and Edit Records].

Two features work together, computed by OSOS in a single pass on the server:

- An **aggregate** is a saved "field + function" definition — *Sum of Amount*, *Count of entries* — created once on an entity and switched on for the views that should show it.
- **Grouping** splits an entries list into blocks by the value of a field. Each block carries its own subtotals, and the list as a whole carries a grand total.

Lifecycle

An aggregate takes effect **immediately**. There is no publish step, because an aggregate is a resource of its own rather than part of the entity's structure, so it never queues behind the entity's own update cycle. It still travels with the entity when you package one: exporting an entity carries its aggregates along, and deleting the entity takes them with it. Deleting a single aggregate removes it from every view that had enabled it, silently and without an error — those views keep working, they simply stop showing that figure.

Functions and the fields they accept

There are exactly five functions: **Count**, **Sum**, **Min**, **Max**, and **Average**. Note the spelling — the platform says *Average*, never "avg", and there is no median, distinct-count, first, last, or standard deviation.

Which of them you may choose depends only on the field:

The field is…	You may choose
A **Number** field	All five
Any other field type	**Count** only
An **encrypted** field	Nothing at all — the field is refused, Count included

Why Count survives on a non-numeric field: unlike the other four, Count doesn't need to read a number out of the field to add up, compare, or divide — it only checks whether the field **has a value at all**. That is what makes it meaningful over a text, date, or reference field where a sum would not be, and it is also why Count's number is not

simply a row count: an entry with no value for the counted field does not contribute to it, the same way it would not contribute to a Sum or an Average over that field. `Since 1.10` — before that release, Count quietly ignored the field it was configured on and counted every matched entry regardless, so a Count and an Average defined on the same field could report inconsistent populations; now both agree on what "no value" means.

Why an encrypted field is refused outright, rather than masked: the stored value is ciphertext, so there is no real number behind the mask — nothing to withhold, only nonsense to compute. The refusal happens when you save the definition, not quietly at read time. A field in the middle of having encryption switched *off* is refused for the same reason, because entries not yet migrated still hold ciphertext.

A **mapped** field behaves exactly like a plain field of its own type — the mapping changes nothing about which functions are allowed. A field **inside a block** can be aggregated (the block's leaf fields are offered, flattened out of the block); the block container itself cannot.

There is **no cap on how many aggregates an entity may have**, and no rule that a field gets only one: *Sum of Amount* and *Average of Amount* can both exist, and a list renders them stacked in the same cell. Give each a display name — an unnamed aggregate falls back to the field's own name, which reads badly when two of them target the same field.

Defined once, enabled per view

The definition belongs to the entity. The decision to *show* it belongs to each view, as a checkbox list on the entries-list card.

A view that enables none gets none. It does not inherit the entity's set, and it does not acquire an aggregate later because somebody added one to the entity. That intersection is a **deliberate boundary, not a convenience**: OSOS resolves the aggregates a request may compute by intersecting what the caller asked for with what the view enabled, and it does so on the server. So a client cannot widen the set by asking for more, and a published or anonymous-facing view cannot start disclosing a new total behind your back.

The cost of that guarantee is real and worth planning for: a newly created aggregate is invisible until you go and enable it on each view that should carry it. There is no "all views" switch.

Two consequences that follow from the same rule:

- **In the admin section** — browsing an entity's own entries with no view bound — every aggregate defined on the entity is computed, because there is no view to narrow by.
- **A deleted aggregate is dropped leniently.** Views that enabled it are not rewritten and do not error; the id simply stops resolving.

Grand total and per-group subtotals

Both figures are computed **on the server**, in one pass, for every platform. Neither is a client-side roll-up of what is on screen.

- The **grand total** is computed over **everything the list matches** — every entry left after filters, search, quick filters, filter-first, entry state, and store access — *not* over the rows on the current page, and not over the page of groups being displayed.
- Each **group** carries the same aggregates again as its own **subtotal**, over that group's entries.

The summary inherits the entries list's own scope rather than rebuilding it, so a total can never count a row the list itself would refuse to show.

Two things follow that surprise people:

- **Paging changes nothing.** Moving to page 4 does not change the grand total, because it was never a total of the visible rows.
- **The subtotals on screen do not add up to the grand total** in the general case. Summing group sums works; averaging group averages does not, and neither does anything computed over one page of groups.

When grouping produces exactly **one** group, the list leaves the grand-total row out — it would repeat that group's subtotal verbatim.

A blank, a dash, and `****`

Three different answers, which a builder must not conflate:

What appears in the cell	What it means
`-` (a blank cell, rendered as a dash)	**There was nothing to total.** No matching entries, or none of them holds a value for that field. An average over no rows is not zero, and a minimum over an empty set has no answer, so the platform declines to invent one.
`0`	A genuine zero — a real computed result.
`****`	**Computed, then withheld.** The field is protected and this reader may not see the value. See Protected fields.

Never read `-` as "hidden", and never read `****` as "empty". The platform keeps the two apart deliberately, and never puts a presentation string into a data field, so an integrator reading the API sees the same distinction: an absent value and a withheld value are separate flags.

Preview mode

In preview mode [Build Apps › Preview Mode], a summary response always comes back with `matchedCount: 0`, every aggregate showing `-`, and no groups — but a field that would be masked for the previewing caller is still reported as masked. A plausible fake number would be worse than an obvious blank, so preview never invents one; and because the masking verdict is still computed correctly, preview never shows an aggregate the live response would have withheld or dropped entirely.

Grouping configuration rules

- **Grouping levels** (**Add Group By**) — an ordered list of fields. **The first level is the outermost block**, the next splits inside it, and so on. A field can only be picked once, and there is no cap on how many levels you add.
- **Collapse groups by default** — every block opens collapsed the first time a reader sees the list. Off by default.
- **User change grouping** — end users may choose their own levels from the list's own **View settings** panel. Off by default. While it is off, a reader can neither add a level nor clear one; the view's own configuration is all they get.

Turning the entries-list card off only hides these settings in the designer. Nothing clears them, so a disable/re-enable round trip returns them untouched.

One rule constrains what you can save: **a grouping level must be one of the view's own listed fields.** Grouping by a field the view does not list would surface a column you deliberately hid, so that configuration is refused — including for a protected field, where collapsing the level (below) is not a license to reach a field the view never listed.

Which fields can be grouped, and why

You can group by text, localized text, number, date, category, switch, a **single-value** reference, a **mapped** field, and any field inside a **non-repeating** block.

Four kinds of field cannot be grouped. It is worth knowing that these are **impossible**, not merely unimplemented — each one breaks something the feature has to guarantee:

The field…	Why grouping cannot work
holds **several values** at once	The entry belongs in more than one block at the same time.
repeats	Same: one entry, several values, several blocks.
sits **inside a repeating block**	Same again — the entry carries one value per repetition of the block.
is a **lookup / ADT** field	Its stored value is an option id resolved against the field's own lookup list, not a pointer to an entry, so there is nothing in the value for grouping to key on.

The first three fail for one shared reason. A list can only draw an entry in **one** place, so a block boundary cannot hold an entry that belongs to three of them at once — and worse, the entry count and every subtotal would count that entry once per block it landed in, so the group counts would stop adding up to the list's own count. A **non-repeating** block is fine: it holds exactly one value per entry, so none of that arises.

The fourth fails differently. A lookup/ADT field looks like a reference — it is the same underlying type — but grouping it the way a reference is grouped would read a part of the value that does not exist on any entry, quietly dropping every entry into a single empty-keyed block with **no error at all** to notice it by. Excluding it outright is the honest outcome.

The entry's **meta-data** block and its children are excluded too — they are virtual, assembled per request rather than stored. The designer's field picker additionally leaves out **ranged** fields, which hold a from/to pair rather than one value.

Multi-value category fields `Since 1.10`

A category field can now be set to hold **several values** (an array of options rather than one). Such a field falls under the first row of the table above — an entry holding *A* and *B* would belong in two blocks — so it cannot be a grouping level. The rule is checked twice: when the view is saved, and again **when the summary is read**, because a category field can be switched to multi-value after a view that groups by it was saved; that view's grouping then stops applying rather than double-counting.

The same field is refused in the neighbouring places that need exactly one value per entry: a view's **default sort**, a dashboard statistic's **group-by**, a **Kanban** column or sub-category field, a **timeline** category, and the entity's **main / sub category** (error `MULTI_VALUE_CATEGORY_FIELD_NOT_ALLOWED`, with `details.usage` naming the place). The reverse is guarded too: switching an existing category field to multi-value while any of those uses it fails with `CATEGORY_FIELD_MULTI_VALUE_IN_USE`, so clear the usages first. See Data Views & Faces [Build Apps] for the view-side settings.

Grouping by a reference field

Grouping by a single-value reference is fully supported, and behaves a little differently from a plain field:

- The block's **key** is the related entry itself, so you can reuse the key as a filter value without knowing how a reference is stored.

- The block's **label** is that related entry's main title, resolved on the server as part of the same query — one lookup per block, not per row.
- If the related entry has been deleted or archived, or lives in a store this reader cannot open, **the block keeps its entries and simply shows no label.** That is not an error, and it is deliberate: the entries in that block are readable — that is why the block exists — so dropping the whole block to hide a title would hide data the reader is entitled to.
- **Block order** for a reference level follows the resolved label. Unlike a category or switch level, a reference has no designer-configured option order to fall back on.
- Ordering the *rows* by the reference's label needs a per-request pass over the distinct related entries, and that pass is capped at **500** distinct related entries. Past the cap the list still groups correctly and still totals correctly — only the label-based row sequence within that level is lost.

Grouping by a localized text field

A localized text field can be grouped, and it groups on **the language the reader is reading in**.

The practical consequence is worth stating plainly to whoever asks for this view: **two people reading the same list in different languages see different group keys**, and can even see a different number of blocks — when two entries share an English value but differ in Arabic, they group together for one reader and apart for the other. This is the same convention the platform already applies when sorting a localized field, which is what keeps a grouped list's blocks agreeing with the order its rows are drawn in.

Because the key *is* the text, there is no separate label to resolve. An entry with no value in the reader's current language falls into that reader's "no value" block.

Whose grouping wins, and how long it lasts

The rule is unambiguous, and the two halves are different in kind:

- **The designer's configuration is durable, and always wins on load.** It is stored on the view like any other view setting, so every reader who opens the view starts from it.
- **A reader's own grouping is theirs alone, and lasts only for the session.** Nothing about it is stored on the server, and reopening the view in a fresh context brings back the view's configured grouping — the reader's choice reverts on reopen.

What "the session" means in practice, for a reader who chose their own grouping:

Event	Their choice survives?
Reloading the page	Yes
Navigating away inside the portal and back	Yes
A new browser tab or window	No
Signing out	No
A different device, or a later visit	No
You edit the view's grouping	**No — their choice is dropped**

That last row matters when you change a view: nobody is left reading a grouping that no longer matches the view they are looking at. An earlier revision did persist a reader's grouping on the server, where it outranked every later edit you made to the view — permanently, with the reader's list and your own preview disagreeing. That behaviour is gone; the view's configuration is now the only durable source.

How group blocks are ordered

Sorting a grouped list orders the blocks as well as the rows inside them:

- **With no sort applied**, blocks follow the group key's own order — the option order you configured for a category or switch level, the resolved label for a reference level.
- **Once a reader sorts**, each block is ordered by one value taken from the sorted field: the **highest** value in the block for a descending sort, the **lowest** for an ascending one. That is the value that would sort *first inside the block* under the reader's own direction; any other choice would put a block ahead of one that visibly contains a higher-sorting row, and grouping would look as though it had discarded the sort.
- **Only the primary sort field orders the blocks.** A second sort field still orders rows inside a block, but ordering blocks by a value most of them do not differ on would order nothing.
- **The ordering value is computed only over the rows the reader can actually see** — after filters, search, quick filters, permissions, and filter-first. A row that was filtered out cannot drag its block to the top of the list, and the block headers cannot end up in a different order from the row blocks beneath them.

Protected fields

Aggregates and grouping both respect field-level protection, and each does so in the way that discloses least. See Sensitive Fields [Build Apps › Protect a field] for how a field becomes sensitive or encrypted, and how grants and reveal sessions work.

- **An aggregate over a sensitive field still appears, under its display name, with its value withheld as `****`.** Removing it would itself be a disclosure — its absence would tell the reader that the field exists and is protected. The reader learns that a figure exists and that they may not have it, which is the intended amount.
- **An aggregate over an encrypted field is not offered at all** — you cannot define one, so there is nothing to mask. There is no number behind the mask to withhold.
- **Grouping on a protected field — sensitive or encrypted — collapses into exactly one masked group**, labelled `****`, holding every matching entry, with that block's count and subtotals covering the whole matched set.
    - It is **uniform for everyone**: a reader holding the field's grant *with* an active reveal session sees the same single block as an anonymous visitor. Grouping is a shared, saved view setting rather than a per-reader one, so per-viewer groupability and shared configuration cannot both hold.
    - **Entries with no value for the field land in that same block**, deliberately, rather than in a separate empty-value block beside it. A separate block would tell every reader which records have the field filled in — and for a salary, medical, or disciplinary field, that is often the sensitive fact itself.
    - A single block also discloses nothing about the field's shape: not how many distinct values exist, not how the population spreads across them, and not their relative order — all of which a set of masked block *headers* would have leaked.
    - Such a level is **valid, saveable configuration**; OSOS no longer refuses it. It renders, it simply renders uselessly, which you can see in the designer and undo.
- **On a view an anonymous visitor can open**, an aggregate over a sensitive field is dropped whole rather than masked — the platform withholds even the definition, because its id, display name, and field name all describe a sensitive field to someone it cannot attribute anything to.

In the designer, the grouping picker offers a sensitive field (it collapses as above) but hides an encrypted field, or one mid-way through being encrypted; a level saved earlier on such a field still collapses and is flagged in the designer as no longer available.

API surface

Every response below arrives inside the usual `{success, result}` envelope — see Calling the HTTP API.

Aggregate definitions

Method	Path	Purpose
`POST`	`/v1/entities/{entity}/aggregates`	Create an aggregate on the entity
`GET`	`/v1/entities/{entity}/aggregates`	List the entity's aggregates (unpaginated)
`PATCH`	`/v1/entities/{entity}/aggregates/{id}`	Update one — a partial body is fine
`DELETE`	`/v1/entities/{entity}/aggregates/{id}`	Delete one; views that enabled it drop it leniently
`{entity}` accepts either the entity id or its normalized name.		

There is no server-side copy route `Removed 1.10`. Pre-release 1.10 builds carried `POST /v1/entities/{entity}/aggregates/{id}/copy`; it was dropped before release. The designer's *Copy* now opens the create dialog pre-filled from the original (its name suffixed " - Copy") and saves through the ordinary `POST /v1/entities/{entity}/aggregates`, so an integrator copies an aggregate the same way: read it, then create a new one with the same `field`, `aggregationFunction` and `name`.

The entries summary

The totals and the group headers come from one response, on whichever of three `GET` routes matches the context:

Route	Context
`/v1/entities/{entity}/entries/summary`	The admin section, with no view bound
`/v1/entities/{entity}/views/{id}/entries/summary`	A published view or a portal gallery
`/v1/entities/{entity}/entries/{entryId}/relations/{relationId}/summary`	A related-entries list

They share these query parameters: `aggregates`, `groupBy`, `groupSkip`, `groupLimit`, `filterCriteria`, `search`, `include_virtual`, and `appliedQuickFilters`. The view-scoped route also takes `filterFirst` and `parameters`; the entity-wide route also takes `collectionId`.

- **Which route you call is a security boundary**, not a convenience: the view-scoped route narrows the aggregates to the view's enabled set, and the entity-wide route does not.
- `aggregates` is intersected with the view's enabled set, never unioned. Omitting it means "the view's enabled set"; sending it empty means "none" — those are different answers.
- `groupBy` is honoured only when the view allows end-user grouping; otherwise the view's own levels are used. Sending it **empty does not mean "ungrouped"** — an empty list falls back to the configured levels, which is why the product's own clients suppress grouped rendering on their side instead.
- `groupSkip` / `groupLimit` page the group headers for an integrator that asks for them. **Neither the web nor the mobile client ever sends them**, so do not design a view around group pagination — plan instead for a grouping level with a manageable number of distinct values.

The same optional `groupBy` parameter exists on the `v3` entries-listing routes. There it changes nothing about *what* comes back, only the **order**, so every entry of a group arrives together — which is what a subheader-style grouped table needs in order to draw one header per block.

The end-user view load

`GET /v2/views/end-user/{id}` carries `availableAggregates` — the aggregate *definitions* (id, name, field, function) a gallery needs in order to build its own Aggregator checklist, since an aggregate toggled off has no computed value to be listed from. The older `GET /v1/views/end-user/{id}` does not carry it.

This list is already narrowed on the server — it is the view's enabled set minus anything this caller may not have. **Do not re-intersect it against the view's own `aggregates` array**: an id can legitimately be in the view's configuration and absent here, and re-filtering would only reintroduce a disagreement the server has already resolved. If resolving the list fails, it comes back empty rather than failing the view load.

Not covered by any endpoint: the entries **export** carries neither grouping nor totals — the XLSX export has no `groupBy` and no `aggregates` parameter, and a template export renders entries, not summaries. That holds for every export path, including the 1.10 gallery quick export (XLSX or CSV — see Data Views & Faces [Build Apps]): a `groupBy` sent to it only **orders** the rows so each group's entries are adjacent; it writes no group headers and no totals. If a total has to leave the platform, compute it from a summary call rather than expecting it in the file.

Version notes

Everything on this page — the aggregate definitions, enabling them per view, entry grouping, the summary routes, and `availableAggregates` — is **Since 1.10**. None of it is present in 1.8 or any earlier release. The one fix worth calling out on its own: `Since 1.10`, `Count` was corrected to count entries that have a value for the configured field, rather than every matched entry regardless of the field's value — see Functions and the fields they accept above.

Export templates API reference

This page is for **integrators and developers** who export records with a template through the API, read or write template definitions, or need to know exactly how a template is rendered.

> **Designing templates in the UI?**
>
> Builders should start with Export templates [Build Apps] — it covers writing the template file, the **Templates** tab and the view settings in the labels the designer shows.

Two template concepts

OSOS has **two distinct template concepts**, both stored inside the `DataEntity` document (not as standalone collections):

1. **Export templates** — uploaded Word/HTML/Excel files that OSOS fills with entry data to produce DOCX, PDF, or XLSX documents. These drive the `entries/export/{templateId}` endpoint.
2. **Display templates and template parameters** — a `templateId` plus a set of parameterized bindings (`TemplateParameter`) attached to an entity or a view, used to parameterize how an entry or list is rendered.

The supported outputs are **DOCX, HTML, and XLSX**, with **DOCX/HTML → PDF** conversion. Template exports never produce CSV: CSV is offered only by the two quick exports of a gallery or of an entity's entries (`format=xlsx|csv`, `Since 1.10`; see Quick export of an entity's entries [Build Apps › Create and design an entity]). Template values are filled with tokens/expressions resolved against the entry at export time, and JavaScript plus custom functions can be used inside the template.

Reports are not rendered by this engine. An earlier version of the builder page listed Reports among the places templates are used, "rendering through the same engine". In the code, a report renders through `ReportService` → `ExportService` (`data-service/src/services/report.service.ts`), not through `DataEntryTemplateExportService`; see Reports [Build Apps].

Sensitive fields in a template export `Since 1.10`

A template carries its own `sensitiveDataAccessPolicy` (`Prohibited`, `Privileged`, `User-Based`, `Auto`; absent = `Privileged`). The template read disables the caller-scoped mask (`disabledStages.hideSensitiveFields: true`) and, when the template's policy resolves to anything short of a full reveal, passes `elementSensitiveAccess` with the resolved grant scopes instead. The template's own policy applies even when a reaction's **Generate File** action renders it (a reaction's policy does not override it). An unknown executor makes a `User-Based` or `Auto` template mask (fail closed). Encrypted fields are masked unconditionally by `maskFieldsExcludedFromBuilders` before any expression is evaluated or anything is rendered. See Sensitive-data access policy [Build Apps › Protect a field].

Template file syntax

The authoring syntax for HTML (EJS), Word and Excel templates, with examples, is in Export templates — Write the template file [Build Apps › Export templates].

Concepts and data model

Export template (`DataEntityExportTemplate`)

Export templates are stored on `DataEntity.templates` (`IDataEntityExportTemplate[]`). Each template references an uploaded source file and declares how it should be rendered and delivered.

Field	Type	Required	Meaning
`id`	UUID	yes	Template identifier (used as `templateId` in the export endpoint)
`name`	`LocalizedValue`	yes	Localized template name; also the default output file name
`level`	`TemplateLevelEnum`	yes	`SingleEntry` (0) or `MultipleEntries` (1)
`type`	`TemplateTypeEnum`	yes	Output format: `Word` (0), `PDF` (1), `Excel` (2)
`file`	`FileModel`	yes	The uploaded source template file (has `id`, `mimeType`, `name`)
`exportType`	`TemplateExportTypeEnum`	yes	`UI` (0) = synchronous download, `Background` (1) = async via service bus
`additionalProperties`	`Record<string,string>`	no	Custom variables (expressions) and feature flags such as `extended`
`query`	reference or `null`	no	`Since 1.10`. Saved query an entry must meet (Single Entry only) — see Conditional export
`criteria`	`FilterExpression`	no	`Since 1.10`. Filter an entry must meet (Single Entry only)
`nonMatchEntriesMode`	enum	no	`Since 1.10`. `Disable` (0, default) or `Hidden` (1) for entries that do not meet the condition
`sensitiveDataAccessPolicy`	`SensitiveDataAccessPolicyEnum`	no	`Since 1.10`. What the template may disclose from sensitive fields; absent = `Privileged`. See Sensitive fields in a template export

Notes verified from code:

- `type` (the requested **output** format) is independent of the **source** file's MIME type. For `type === PDF`, the service branches on `file.mimeType`: a DOCX source is rendered then converted DOCX→PDF; an HTML source is rendered then converted HTML→PDF. Any other source MIME type for a PDF template yields no file.
- `level` controls per-entry helpers: `getEntryComments` and `getEntryEmails` return `[]` unless `level === SingleEntry` (see Custom functions).

Template engines by type

The output format determines which templating engine runs:

`type`	Source file	Engine	Delimiters
`Word`	DOCX	`docx-templates` `createReport`	`{{ … }}` (`cmdDelimiter`), `fixSmartQuotes: true`
`PDF` (from DOCX)	DOCX	`docx-templates`, then DOCX→PDF conversion	`{{ … }}`
`PDF` (from HTML)	HTML	`ejs.render` (async), then HTML→PDF conversion	`openDelimiter: '{{'`, `closeDelimiter: '}}'`
`Excel`	XLSX	`xlsx-template` (`substitute(1, data)`); a streamable template is written by ExcelJS's streaming writer instead (`Since 1.4`, see Large exports)	library default (`${...}` placeholders)

The data object passed to every engine is `{ entries, entity }` (plus extra collections for extended export — see below). For Excel, entries are first flattened by `mapEntriesForExcelTemplate`: resolved reference arrays are collapsed to comma-joined `mainTitle` strings per available language, and vertical-tab characters are replaced with spaces.

Template parameters (`TemplateParameter`)

Since 1.4.

Display/parameterized templates use `TemplateParameter`, defined in `models/template-parameter.model.ts` and `@osos/shared`.

Field	Type	Required	Meaning
`name`	string	yes	Parameter name; must match the parameter name in the template
`valueType`	`TemplateParameterValueTypeEnum`	yes	`Fixed` (0), `Expression` (1), `Field` (2)
`value`	any	no	Fixed value, or an entry expression such as `value.my_block[scope.index].my_field` when `Expression`
`component`	string	no	Optional component reference
`source`	string	no	Optional source reference

`valueType` semantics from the model JSDoc: **Fixed** — `value` is used verbatim; **Expression** — `value` is an entry expression evaluated to produce the value; **Field** — value type `2` (the enum member exists; its resolution is performed by consumers).

Parameters are grouped by render context in `TemplateParameterTypes`:

```
interface TemplateParameterTypes {
  listEntry?: TemplateParameter[];   // list rendering
  viewEntry?: TemplateParameter[];   // read/view rendering
  editEntry?: TemplateParameter[];   // edit rendering
}
```

This bundle is attached in two places, both `{ templateId, parameters }`:

- `DataEntity.displayTemplate` → `EntityDisplayTemplate`
- A data view → `ViewDisplayTemplate`

API endpoints

Export entries with a template

`GET /v1/entities/{entity}/entries/export/{templateId}`

Controller: `DataEntryTemplateExportController.exportEntriesWithTemplateV2`. Authenticated (`Auth2` strategy). Produces a file download (`application/vnd.openxmlformats-officedocument.spreadsheetml.sheet`, `...wordprocessingml.document`, or `application/pdf`) or a JSON acknowledgement.

Path parameters:

Name	In	Meaning
`entity`	path	Data entity id

Name	In	Meaning
`templateId`	path	Export template id (the `id` on `DataEntity.templates[*]`)

Query parameters:

Name	Type	Meaning
`mode`	number (`ExportWithTemplateSourceMode`)	`SelectedItems` (0) or `source` (1)
`itemsToExport`	string[]	Entry ids to export; **required** when `mode === SelectedItems`
`filterCriteria`	object	Filter criteria applied when `mode === source`
`search`	string	Free-text search (ignored/forced `undefined` in `SelectedItems` mode)
`sourceCollection`	string	Collection id; its `items` are added as an `_id IN (...)` filter in `source` mode
`appliedQuickFilters`	string[]	Quick-filter ids

Behavior (verified):

1. Loads the entity, then `getEntityTemplate(dataEntity, templateId)` — throws `TemplateNotExists` (HTTP 404, code `TEMPLATE_NOT_EXIST`) if no template with that id exists on the entity.
2. **Extended templates** — if `additionalProperties.extended === 'true'`, delegates to `DataEntryExtendedTemplateExportService.exportFromTemplate` (see Extended export). Always synchronous.
3. **Background export** — else if `exportType === Background`, sends an `ExportTemplateInBackgroundCommand` on the service bus and immediately returns `200 { "success": true }`. The file is generated asynchronously by `ExportTemplateInBackgroundHandler` and delivered via a `FileGeneratedEvent`.
4. **UI export** — else renders synchronously via `exportEntriesWithTemplate`, uploads the result to storage, publishes `FileGeneratedEvent`, sets `Content-Type`/attachment headers, and streams the file body back.

In cases 2 and 4 the generated file is also uploaded to the storage service and a `FileGeneratedEvent` is published (carrying the file, template id, `exportType`, entity id, and the list of exported entry ids).

Error cases:

Condition	Exception	HTTP
No template with `templateId` on the entity	`TemplateNotExists`	404
`mode === SelectedItems` with empty/missing `itemsToExport`	`MustProvideITemsToExportProperty`	400
The entry does not meet the template's condition, or a conditioned template is called in `source` mode (`Since 1.10`)	`TEMPLATE_CONDITIONS_NOT_MET`	422

Check whether a template is in use

`GET /v1/templates/{id}/used`

Controller: `TemplatesController.isTemplateUsed`. Returns a boolean. *Long-standing.*

This operates on templates stored as data **entries** in the internal `templates` entity (`StringConstants.TEMPLATES_ENTITY_NAME = 'templates'`), i.e. message/email templates referenced by entity triggers — not on the `DataEntityExportTemplate` export templates above.

Logic (verified):

1. Look up the template entry by `id`; return `false` if it does not exist.
2. If the template's `value.template_type.id === 'EMAIL_PART_TEMPLATE'`, find the main templates that reference this part as their header or footer (`value.template_details.template_header.dataEntry` / `template_footer.dataEntry` equal to `id`); collect those ids. Otherwise the candidate set is just this template's own id.
3. Return `true` if any `DataEntity` has a trigger action wired to one of those template ids (`triggers.actions.options.template IN (...)`); otherwise `false`.

Rendering pipeline

`exportEntriesWithTemplate` is the core renderer used by the UI export path, the background handler, and the internal `GenerateFileFromTemplateHandler`. Steps:

1. **Fetch the source file** from storage as an `arraybuffer`.
2. **Resolve the output file name** — the passed `selectedFileName`, else the localized template `name`.
3. **Gather export data** (`getTemplateExportData`):
   - `mode === source`: builds filters from the source collection's `items` (`_id IN`) and any `filterCriteria.criteria`.
   - `mode === SelectedItems`: requires `itemsToExport`, forces `search = undefined`, and filters `_id IN itemsToExport`.
   - **A view-triggered export reads through the view's own columns.** The view's configured `fields` are its display columns, and `buildScope` uses the same list both to narrow which relation/ADT fields it resolves and to build the read's `$project`, so a template referencing a field the triggering view does not display renders that field blank (known limitation, still open). An earlier version of this page said the export widened to every non-sensitive field `Since 1.8`; that widening was reverted, because a `$project` naming both a block and its sub-field (`value.address` and `value.address.street`) is a MongoDB path collision that produced a file with no records at all (`data-entry-template-export.service.ts`, `getTemplateExportData` JSDoc). The view's row-level scoping (obsolete/invalid/archived and search rules, its query data source, its default sorting) still applies.
   - Runs `dataEntryService.findWithResolvesFactory` with `disabledStages.hideSensitiveFields: true` — this *disables* the caller-scoped mask, so the template's own sensitive-data access policy decides (`elementSensitiveAccess`); Encrypted fields are then masked unconditionally.
   - If the template declares `additionalProperties`, evaluates them per entry (see Custom variables).
4. **Render** with the engine matching `type` (see the table above), injecting the shared custom functions.
5. **Return** `{ fileName, fileContent, mimeType, exportEntriesIds }`.

Custom variables

Long-standing.

`additionalProperties` is a `Record<string, string>` where each value is an expression wrapped in `{{%= … %}}`. During data gathering, `parseAdditionalProperties` strips the `{{%=` / `%}}` wrappers, evaluates the inner expression per entry via `executeExpression`, and merges the results into each `entry.value` under the property key — making them available to the template as normal fields.

The reserved key `extended` (`additionalProperties.extended === 'true'`) is a **feature flag**, not a variable — it routes the export to the extended service.

Custom functions in templates

Every render (`docx`, `html`/`ejs`, and expression evaluation) receives a shared function set from `getSharedCustomFunctions(entity, templateConfig)`. Verified functions:

Function	Purpose
`today(manipulateAmount?)`	Current date, timezone-adjusted, with optional date-math manipulation
`date(value, formatString?, tzOffset?)`	Format a date (default `Constants.DATE_FORMAT`), timezone-adjusted
`loc(localizedValue)`	Resolve a `LocalizedValue` to the user's language
`formula(text)`	Excel only: the one way to emit a formula cell (a plain string starting with `=` stays text)
`mdToText(value)`	Strip Markdown to plain text (also normalizes single-tilde strikethrough and line breaks)
`fetchEntry(ref)`	Resolve a single referenced entry from `{ dataEntity, dataEntry }`
`fetchEntriesById(entityId, entriesIds)`	Resolve multiple entries by id
`fetchEntries(entityId, filter)`	Resolve entries by an arbitrary filter
`getStepName(entry, stepId)`	Workflow step name (falls back to ad-hoc steps)
`getCurrentStepOwner(entry)`	Owner of the active current step (if it has a resolved `mainTitle`)
`getStepOwner(entry, stepId)`	Owner of a step from workflow history
`getStepReceivedAt(entry, stepId?)`	Received-at timestamp for a step (handles `recievedAt` / `receivedAt`)
`getStepActions(entry, stepId)`	Actions defined on a step
`getStepActionName(entry, stepId, actionId)`	Name of a specific step action
`getStepHistory(entry, stepId)`	All history records for a step in the active workflow
`getStepLastHistory(entry, stepId)`	Most recent history record for a step
`getEntryComments(entry, lang?)`	Entry comments (with mentions resolved); `[]` unless `level === SingleEntry`
`getEntryEmails(entry)`	Emails associated with the entry; `[]` unless `level === SingleEntry`

DOCX and HTML renders additionally get:

- **DOCX** — `getImage(image, width, height, entryId?, alt?)` returns image bytes for embedding (supported extensions: `png`, `gif`, `jpg`, `jpeg`, `svg`; unsupported extensions are logged and skipped) and `mdToHtml(value)` renders Markdown to HTML wrapped in `<body>…</body>` (with single- and double-tilde strikethrough support).
- **HTML/EJS** — `getImageSrc(image, entryId?)` returns a base64 `data:` URI for the image, and `mdToHtml(value)` (configured with `breaks: true`).

Background export

When `exportType === Background`, the endpoint enqueues an `ExportTemplateInBackgroundCommand` (carrying the entity id, the full `entityTemplate`, `mode`, `itemsToExport`, `sourceCollection`, `filterCriteria`, `search`, `view`, and `appliedQuickFilters`) with sticky attributes for user, space, and delegation, then returns `{ success: true }`.

`ExportTemplateInBackgroundHandler` re-loads the entity, calls the same `exportEntriesWithTemplate`, uploads the result, and publishes a `FileGeneratedEvent` (with `exportType = Background`) plus a `CommandExecutedMessage` over the socket. Errors are logged and swallowed (no failure event is published on error in this handler).

Internal: generate file from template

`GenerateFileFromTemplateHandler` handles `GenerateFileFromTemplateCommand` (used by workflow/automation actions, not an HTTP route). It optionally loads a view by `viewId`, resolves the entity and template, calls `exportEntriesWithTemplate` with `command.fileName`, optionally uploads to storage when `uploadToStorageService` is set, and returns the file (or the uploaded file record) in a `CommandExecutedMessage`. On error it emits a `CommandExecutedMessage` with `success: false` and the error.

Extended export

Since 1.6.

`DataEntryExtendedTemplateExportService.exportFromTemplate` is used when `additionalProperties.extended === 'true'`. It runs the same gather-and-render pipeline but always in `source` mode and enriches the template `data` object with entity-model context beyond `{ entries, entity }`:

```
{ entries, entity, module, reactions, notifications, workflows,
  lists, queries, views, relations }
```

These are resolved from the data entity (`getModule`, `getReactions`, `getNotifications`, `getWorkflows`, `getLists`, `getQueries`, `getViews`, `getRelations`), making it suitable for generating documentation/specification exports of an entity's configuration itself. It shares the same engine selection and custom-function set as the base service. This path is always synchronous (no background branch).

Streamed Excel exports `Since 1.4`

An Excel template export of many entries is written **page by page** (500 entries at a time, each page written and released before the next is read) when the template has a simple enough shape; any other shape is rendered in memory as before. The difference matters at scale: in the measurement that came with the change, 25,000 entries took 12.8 s instead of 323 s, with about a quarter of the memory, and the file was 3 MB instead of 24 MB. This was also delivered to the 1.4, 1.6 and 1.8 lines as an update.

A template is streamed only when **all** of these hold:

- It has exactly one sheet.
- Every placeholder fills a whole cell, is a plain `${table:entries.<key>}` (no sub-type), and all of them sit on the same row; that row holds nothing else.
- Nothing below that row has a value, and no merged range reaches the table row or below it.
- The sheet has no formulas, images, Excel tables, conditional formatting, data validation, auto-filter or defined names.

When a template does not qualify, data-service logs the reason at INFO (`template <id> of entity <id> is rendered in memory: it …`), which tells you what to remove. Operators can force every template back to in-memory rendering with `TEMPLATE_EXPORT_XLSX_STREAMING_DISABLED=true`. On both paths a user value that starts with `=` stays text; only a value produced by the `formula()` expression helper is written as a formula.

Templates on virtual entities `Since 1.6`

The Templates tab is also available on **virtual entities** such as Groups: templates can be managed on a virtual entity when it is not implicit, or when it is implicit but allows editing (`implicitOptions.allowEdit`); otherwise the entity's normal edit rule applies. A template on a proxy, joiner or aggregate virtual entity is saved through the virtual-entity endpoint (`PUT /v1/entities/virtual/{id}`). A template export over a virtual entity includes the rows the user selected. Both were also delivered to the 1.6 and 1.8 lines as an update (the selected-rows fix also to the 1.4 line). For the other admin operations on virtual entities, see Virtual entities: admin operations [Build Apps › Virtual entities].

Conditional export `Since 1.10`

A **Single Entry** export template can carry a condition, so that only entries that meet it can be exported with that template. In the template form, the **Filter** section sets the condition and **For other entries** decides what users see on an entry that does not meet it: **Disable export option** (the default) or **Hide export option**. A **Multiple Entries** template cannot have a condition (its filter controls are disabled). A template without a condition behaves exactly as before.

The condition is stored on the template:

Property	Meaning
`query`	A saved query the entry must meet (`null` clears it).
`criteria`	A filter expression the entry must meet.
`nonMatchEntriesMode`	`Disable` (0, the default when absent) or `Hidden` (1) — the same values system buttons use.

A template "has a condition" when it is Single Entry and has a saved query **or** at least one criteria rule; an empty rule group is not a condition. A saved query that cannot be loaded, or criteria that cannot be parsed, matches no entry (and is logged).

- **Status route.** `POST /v1/entities/{entity}/templates/status/entries` with `{dataEntriesIds: string[]}` (1 to 100 ids) returns `[{templateId, isDisabled, isHidden}]` for each *conditioned* template. Through a view it reports only the templates that view exposes, and evaluates conditions only when the view can read every requested entry; outside a view it is open to system viewers and above, like the export route. Row and entry-details menus use it to show each template enabled, disabled (with the reason on hover) or hidden, and call it only when a presented template has a condition. Views that expose a template have the route in their execution plan (migration `1.10.0-a22` backfilled existing views).
- **Export route.** `GET /v1/entities/{entity}/entries/export/{templateId}` refuses an entry that does not match, and any `mode` other than selected items for a conditioned template, with `422 TEMPLATE_CONDITIONS_NOT_MET`. The check runs before anything is rendered or queued.
- **Not gated.** The **Generate File From Template** reaction action does not apply template conditions.

Version notes

Feature	Since	Evidence
Template parameters + display template models (`TemplateParameter` , `EntityDisplayTemplate` , `ViewDisplayTemplate`)	1.4	introduced in `1.4.0-alpha.01`
Extended template export (`exportFromTemplate` / `extended` flag)	1.6	first tag `1.6.0`
Background export (`TemplateExportTypeEnum` , `ExportTemplateInBackgroundCommand` , `FileGeneratedEvent`)	Long-standing	pre-1.2

Feature	Since	Evidence
Custom variables (`additionalProperties` / `parseAdditionalProperties`)	Long-standing	pre-1.2
`GET /v1/templates/{id}/used`	Long-standing	pre-1.2
Streamed Excel template exports (500 entries per page)	1.4	first published on the 1.4 line; also delivered to the 1.6 and 1.8 lines
Selected virtual rows included in a template export	1.4	delivered to the 1.4, 1.6 and 1.8 lines
Templates tab on virtual entities	1.6	frontend, 1.6 and 1.8 lines
Conditional export (`query` / `criteria` / `nonMatchEntriesMode` , status route, `TEMPLATE_CONDITIONS_NOT_MET`)	1.10	develop only (migration `1.10.0-a22`)
Per-template sensitive-data access policy (`sensitiveDataAccessPolicy`)	1.10	form control shown only when the entity has a sensitive field

Groups, org units & membership

This page is for integrators and engineers who read or write group membership and the org chart programmatically, or who need to know exactly how OSOS resolves a group or a manager to a person. The administrator's view of the same model is in Managing groups [Administration] and Organization units [Administration]. The permission side of groups is in Permissions API.

Groups

What a group is

`GROUP` is not a physical collection of its own. It is a **virtual view that unions several concrete identity source entities**. When you pick a "group" in the interface, you pick a row that came from one of these entities. A group's members are therefore defined **directly** (a Users Group populated by hand or by directory sync) or **derived** (an org unit's staff, a role's holders, or the result of a saved user query). Each kind is managed in the screen for its source entity.

Because `GROUP` is virtual, it is a **core identity entity** that exists in every environment, and groups are **excluded from package export/import** as first-class resources (see Groups and packaging).

Group types

Type (`group_type.en`)	Description	Source entity	Entity ID
Users Group	A regular group, created and managed manually by adding members.	Users Group	`USERS_GROUP`
Role Group	Created automatically behind every role; membership derived from the role's assignments.	Role	`ROLE`
Org Unit Group	Created automatically behind every org unit; membership derived from the org chart.	Org Unit	`ORGUNIT`
Query Group	A **dynamic** group: every user who satisfies the query joins, and any user who no longer does leaves, with no manual assignment.	User Query Group	`USER_QUERY_GROUP`
Virtual Group	A computed group managed by the system for internal purposes; never created or edited manually.	—	—

Seeded field labels: `USERS_GROUP` has **Group Name** (`name`), **Group Description** (`description`), **Group Icon** (`icon`); `USER_QUERY_GROUP` has **Group Name**, **Description**, **Icon** and **query** (a data-query selector over users).

How membership is stored

Membership is recorded as assignment rows, not as an array embedded on the group:

Entity name	Entity ID	Purpose
`group_assignments`	`GROUP_ASSIGNMENTS`	Virtual: links a **group** (any type) to a **user**.
`user_group_assignments`	`USER_GROUP_ASSIGNMENTS`	Links a **Users Group** specifically to a **user** (`value.users_group`, `value.user`).
`user_query_group_assignment`	`USER_QUERY_GROUP_ASSIGNMENT`	Materialized query-group membership, written by background handlers. `Since 1.8`

A `group_assignments` row carries `value.group.dataEntry` (the group id), `value.user.dataEntry` (the member) and `value._metaOrder.value` (an ordering key used to sort members deterministically). Each row also has a top-level `state` and an `isInvalid` flag. Only rows with `state = Active` and `isInvalid != true` count as live membership.

Uniqueness. `user_group_assignments`, `user_query_group_assignment` and `space_users` each carry a compound **unique** index over the active `(user, group)` (or `(user, space)`) pair, so a second active assignment for the same pair is rejected. `Since 1.8` — migration `1.8.0-a16` removed existing duplicates and `1.8.0-a17` re-asserted the indexes, which had been silently dropped in environments where the entity was edited.

The admin screens expose this as a **Users** tab on a Users Group record and a **Groups** tab on a user record (the seeded relations onto `USER_GROUP_ASSIGNMENTS`).

Directory synchronization

For directory-backed groups, membership is populated by the directory importers: each OSOS Users Group marked external is matched to a directory group by `value.externalGroupId`, and each directory member to a OSOS user by `account_information.externalUserId`; a `user_group_assignments` row is created for every match. Groups that do not exist in OSOS, and members that do not map to a user, are skipped and logged. Full detail, including the fields a Users Group needs for this, is in Directory sync operations.

Listing the members of a group

The data service exposes one read endpoint for group membership. It resolves a group of any type to its **distinct, active member users**:

```
GET /v1/group/{groupId}/users?includeResolved={true|false}
```

- **Authentication:** required (OSOS auth strategy).
- **Authorization:** guarded by the standard entity voter.
- **Path parameter** `groupId`: the group's id.
- **Query parameter** `includeResolved` (boolean): when `true`, each returned user reference is enriched with a `_resolved` block containing the user's indexed display metadata; when `false` or omitted, only the bare reference is returned.

The controller always filters the underlying users to `state = 1` (Active), so inactive (archived or draft) user accounts are never returned. A user's `is_deactivated` flag is a value field, not a state, so **deactivated** users are still returned as members.

Response shape

Each element is a user reference:

Field	Type	Present when	Meaning
`dataEntity`	string	always	The entity id of the reference (a user).
`dataEntry`	string	always	The member user's id.
`_resolved`	object	`includeResolved=true`	Resolved display metadata from the entry index.

Membership resolution rules

`getUsersInsideGroup` builds a MongoDB aggregation over `group_assignments` that enforces, in order:

1. **Match** rows where `value.group.dataEntry` is in the requested group id(s), `state = Active` and `isInvalid != true`. Several group ids may be passed at once; they are treated as a union.

2. **Optional user pre-filter**: if a users filter is supplied (the HTTP endpoint supplies `state = 1`), the service first resolves the matching user ids from the `USER` entity and restricts assignments to those users. This is how inactive users are excluded.
3. **De-duplicate** by `value.user.dataEntry`, keeping the first row: a user who belongs to a group through more than one assignment appears once.
4. **Sort** by `value._metaOrder.value` ascending, so member order is stable.
5. **Optional resolve**: when `includeResolved` is requested, member metadata is fetched from the data-entity index in a single batch and attached as `_resolved`.

There is also an internal `activeOnly` mode (not exposed on the HTTP endpoint): together with `includeResolved`, members whose index entry is not `Active` are dropped entirely rather than returned without resolved metadata. If no assignments match, the service returns an empty list.

The same resolution is used wherever a group is an audience or recipient (notifications, email/SMS triggers, comments), so membership configured once affects owner assignment, notification delivery and comment targeting consistently.

Groups as workflow-step owners

Reference fields that accept an owner declare their accepted types as `USER,GROUP`, for example the workflow step's **Original Owner** field. This is what lets a builder assign a group as a step owner.

When a step whose owner is a **group** is executed, the engine expands the group to a concrete user in `OwnerTypeProcessor`, depending on the step's owner options:

Owner option	Effect when the owner is a group
Default (single owner)	Resolves the group and assigns its **first member** (`groupUsers[0]`) as the owner.
`selectionCriteria = Manager`	Resolves the **org-unit manager** of the group via `getOrgUnitManager`; if none is found, the owner is cleared.
`selectionCriteria = Manual`	The auto-resolved owner is cleared so a person picks the owner manually.
`isParallel = true`	The single owner is cleared; the step is handled as a parallel, multi-owner assignment.
Array of group owners (dynamic multiple)	Expands every group to its members via the dynamic-multiple-owner handler; if all resolved entries are users, the first is picked as the active owner (unless Manual or parallel).

`getOrgUnitManager` first looks for an active assignment on the org unit itself where `value.isManager = true` and the referenced user is active. If none exists, it walks **up the org-unit parent chain** (nearest ancestor first) and returns the first manager it finds.

Because default owner assignment takes the *first* member, the `_metaOrder` ordering of a group's members is not cosmetic: it decides who owns a group-owned step.

Groups and packaging

`GROUP`, and the identity entities behind it (`USER`, `ORGUNIT`, `WORKING_HOURS`), exist independently in every environment, so they are **core entities that are never exported as part of a package**. When an imported package references a group that does not yet exist in the target environment, auto-create does **not** create a `GROUP` row directly. It creates an entry in the concrete source entity implied by the group's `group_type` (Users Group, Role, Org Unit or Query Group), falling back to `USERS_GROUP` when the type is unknown.

When you move a workflow or view that assigns a group between environments, the group definition and its membership do not travel with the package. The equivalent group must exist in the target environment, or the reference must be re-pointed after import. See Packaging.

Operational notes

- **Active-only everywhere.** Both member users and their assignment rows must be Active to count. Draft (`0`) and Archived (`2`) states are excluded.
- **Duplicates are collapsed** at read time, and since 1.8 prevented at write time by the unique indexes above.

Org units

Concept overview

An org unit is an ordinary data entry of the built-in `ORGUNIT` entity (normalized name `org_unit`). Org units reference each other through a self-referencing `parent` field, which turns the collection into a tree. Users are attached to org units through **Assignment** entries (`ASSIGNMENT` entity), each of which links a user, a role and an org unit and carries an assignment level (Main / Additional), an assignment type (Original / Delegation) and an *is manager* flag.

> **Two "organization" concepts.** `ORGUNIT` forms the hierarchy. The singleton **Organization** entity (`ORGANIZATION` , `OrganizationDataEntry` / `OrganizationDataEntryValue`) only holds tenant branding and identity: `name` (localized, required), `description` , `logo` , `contact_information` (phone/email block), `organization_header` , `organization_footer` , `address` and `motto` . Initial setup seeds one "Default Organization" entry. A separate **OrgConfig** document in the users-service holds sign-in page branding (name, logo, favicon, custom CSS, optional sign-up URL). None of these carry a parent/hierarchy relationship.

The hierarchy powers three capabilities:

Capability	Driven by	Entry point
Scoping ("my org unit users")	the user's Main assignment org unit + its parent + all descendants	`GET /v1/users/my-orgunit`
Direct-manager resolution	walking the `parent` chain upward for the nearest manager assignment	`GET /v1/users/direct-manager`
Workflow step owner (relative)	the same upward walk, driven by a step's *relative options*	workflow engine (owner resolution)

The Org Unit entity

`ORGUNIT` is seeded as an implicit (system) entity during initial setup. It is **tree-structured**: its definition carries `treeStructure.supported = true` with the tree field set to `parent` .

Fields (`orgunit.json` , mirrored by the `OrgUnitDataEntryValue` model):

Field (normalized-Name)	Label	Type	Notes
`name`	Name	Localized text	Required, **unique**, shown as the entry title
`description`	Description	Localized text	Optional
`icon`	Icon	Icon	Optional, used as the entry picture
`parent`	Parent	Relation → `ORGUNIT`	The hierarchy link. `sourceOptions.denyCircularRelation: true` , so a unit cannot become its own ancestor

Field (normalized-Name)	Label	Type	Notes
`org_unit_level`	Org unit level	Relation → `ORGANIZATION_UNIT_LEVELS` collection	The unit's level
`isMainAssignment`	Level Assignment is main	Boolean	Default `false`. When `true`, the unit accepts only Main assignments (`OrgUnitAcceptJustMainLevelException`)
`manager`	Manager	Relation → `USER`	**Optional**. Denormalized, auto-synced from assignments

The entity is **restricted** (`restrictedEntityIds` includes `ORG_UNIT_ENTITY_ID`): readable by system viewers and restricted-with-grants users, editable by admins only. `implicitOptions.allowEdit = true`, so builders can add fields.

Org unit levels

`org_unit_level` points at the `ORGANIZATION_UNIT_LEVELS` string ADT collection (label **Org Unit levels**). Out of the box it holds a single level:

Level id	Label (en)	`level`	`index`
`5cf7d2cc-…-461e636f`	First level	0	0

Levels are descriptive labels; the actual hierarchy is defined by `parent`, not by the level value.

The seeded root

Initial setup creates one root org unit, **"Executive Management"** (`org-root.json`), with `parent: ""`, the seeded first level, and a `home` icon.

Assignments

Each **Assignment** (`ASSIGNMENT` entity, `assignment.json`) ties one user to one org unit with a role:

Field	Label	Type	Meaning
`role`	Role	Relation → `ROLE`	The role held
`assign_level`	Assign Level	Single-select	`MAIN` or `ADDITIONAL` (`AssignmentLevelTypes`)
`assign_type`	Assign type	Single-select	`ORIGINAL` or `DELEGATION` (`AssignTypes`)
`user`	User	Relation → `USER`	The assigned user
`org_unit`	Org Unit	Relation → `ORGUNIT`	The org unit
`isManager`	Is Manager	Boolean	Whether this user manages the org unit

The assignment validator enforces, on save:

- the unit's `isMainAssignment` (only Main allowed there);
- a role with **Unique** assignment type can be assigned once per org unit; a role **linked with one org unit** can exist in only one org unit;
- a **Main** assignment needs a role of type *job position* ("role must be of type job position if the the assignment level is main"), and a user can have only one non-archived Main assignment ("each person can have only main assignment");
- the user must be an organization account ("User must be an organization account");
- a role *not linked with org units* has its `org_unit` cleared.

Roles relate to the structure via `RoleLinkWIthOrgUnitTypes`: *Not linked with org unit*, *Linked with one org unit*, *Linked with many org units*.

A user's org unit is their Main assignment. `getUserOrgUnitId()` returns the `org_unit` of the single active assignment where `assign_level.id === MAIN`. When an assignment is created, a handler copies a Main assignment's role and org unit onto the user's `main_assignment_role` and `org_unit` fields, and an Additional assignment's role onto `additional_assignment_role`. `direct_manager` on the user is a calculated field derived by walking the parent chain.

The auto-synced `manager` field

The org unit's `manager` field is a denormalized mirror kept up to date by event handlers. When an assignment is **created, updated or archived**, a handler (`update-orgunit-manager-on-assignment-entry-*`) checks whether it is a **Main** assignment with `isManager` true and, if so, patches the org unit's `manager` to that user. (Long-standing.)

Manager *resolution* for workflows and notifications does **not** read this convenience field; it queries assignments while walking the parent chain (next section).

Seeding the `manager` field `Since 1.8`

`manager` is guaranteed to be present on `ORGUNIT` in every environment since 1.8: a fresh environment seeds it with the entity definition, and a migration adds it to environments provisioned earlier. On an environment provisioned before that migration the field was absent, even though the assignment handlers had always written to it.

The cause: the initial-setup migration reads **every** JSON file in the `entity/` folder and skips any entity id already inserted. Two files used to define `ORGUNIT`: `org-unit.json` (without `manager`) and `orgunit.json` (with `manager` and the full `treeStructure`). The directory listing yielded the dash-named file first, so the incomplete definition took the id and `orgunit.json` was skipped. Two changes put this right:

- The stale `org-unit.json` duplicate was removed from the migration sources, so a fresh environment seeds `ORGUNIT` from `orgunit.json`, now the entity's sole seed.
- Migration `1.8.0-a31-add-manager-field-to-org-unit-entity` backfills `manager` onto `ORGUNIT` on existing environments. It is idempotent (it appends the field only where `manager` is missing), and its `down()` is deliberately a no-op, because removing the field could destroy data already captured on org-unit entries.

Manager resolution (walking up the tree)

`getUserDirectManager(userId, relationGrade = 1)` resolves a user's manager by climbing the hierarchy:

1. Find the user's **Main** assignment and its `org_unit`.
2. `$graphLookup` up the `parent` chain from that unit, ordered by depth (nearest first, including the user's own unit at depth 0).
3. Walk that list looking for an assignment on each unit with `isManager: true` and a **different** user than the subject.
4. Return the **N-th** manager found, where **N =** `relationGrade` ("how many manager levels up", starting at 1 for the immediate manager).
5. If no manager is found at the requested grade, return the org unit only (no `directManger`).

This is exposed to the portal as `GET /v1/users/direct-manager`. A user's direct manager also drives the **manager-permissions model**, the manager-scoped grants that let a manager act on records of the users they manage (see Permissions API: manager permissions).

Relative step owners in workflows

Workflow steps can define **relative owner options** (`StepOwnerRelativeOptions`): `enabled`, `relationType` and `relationGrade` ("the number of levels to go up in the hierarchy starting from 1"). During owner resolution, when a

step's `source.relativeOptions.enabled` is true (and it is not the final step), the engine replaces the resolved owner with `getUserDirectManager(owner, relationGrade)`. If no manager is found at that grade, the step is left with **no owner** (logged as a warning) rather than falling back silently.

Two workflow **recipient source** tokens are tied to the hierarchy, used by notifications and email triggers:

Token	Constant	Meaning
`$OWNER.DIRECT_MANAGER`	`CURRENT_STEP_OWNER_DIRECT_MANAGER_RECIPIENT_SOURCE_VALUE`	The step owner's direct manager
`$OWNER.ORG_UNIT_MANAGER`	`CURRENT_STEP_OWNER_ORGUNIT_MANAGER_RECIPIENT_SOURCE_VALUE`	The manager of the step owner's org unit

Scoping ("related org unit users")

`getUserRelatedOrgUnitsUsers(userId)` returns the users considered "related" to a user through the structure. Starting from the user's Main assignment org unit, an aggregation collects the active users assigned to the user's **own** unit, its **direct parent**, and **all descendant** units (via `$graphLookup` down the `parent` chain). Only Active assignments and users are included; results are de-duplicated. This is `GET /v1/users/my-orgunit`, used for example by the user-picker fetcher to scope selectable people.

Administering the hierarchy over the API

There is **no** dedicated org-unit REST controller: org units and assignments go through the generic data-entry CRUD for the `org_unit` and `assignment` entities (see Writing data), plus the read helpers above. Build the tree with `parent` (leave the root's `parent` empty), set a manager with a Main assignment that has `isManager: true`, give each user exactly one Main assignment, and extend `ORGANIZATION_UNIT_LEVELS` to describe depth.

Materialized ancestors

Because org units are a tree, OSOS materializes each node's ancestor list to avoid repeated recursive lookups. Any entity whose definition has `treeStructure.supported = true` participates.

- **`DataEntryTreeService`** computes ancestors with a MongoDB `$graphLookup` over `value.<treeField>.dataEntry → _id`, sorted by depth, and stores the ordered ancestor ids at `_meta.ancestors.value.<treeField>` (for org units, `_meta.ancestors.value.parent`). It exposes `calculateAncestors`, `findDescendants`, `updateNodeAncestors`, `updateNodeAndDescendantsAncestors`, `initializeAllAncestors` and `rebuildAncestors`. `Since 1.2` (with the `1.2.0-a02-calculate-ancestors` backfill migration).
- **Sync handlers** (`sync-entry-ancestors-data-entry-created` / `-updated`) fire on entry create/update, check `treeStructure.supported`, and recompute ancestors for each tree field. Moving a subtree recomputes the node and all its descendants.
- Writes use raw `$set` on `_meta.ancestors...` (never `updateById`), consistent with the migration-safety rule for partial updates.

Data model summary

- `ORGUNIT`: normalized name `org_unit`, entity type `11`, implicit, restricted, editable (`implicitOptions.allowEdit = true`). Value model `OrgUnitDataEntryValue`.
- `ASSIGNMENT`: normalized name `assignment`. Links user and org unit with role, level, type and `isManager`.
- `ORGANIZATION_UNIT_LEVELS`: a string ADT collection of level labels.
- Related constants in `libs/constants.ts`: `ORG_UNIT_ENTITY_ID='ORGUNIT'`, `ORG_UNIT_ENTITY_NAME='org_unit'`, `ASSIGNMENT_ENTITY_ID/NAME`, `AssignmentLevelTypes`, `AssignTypes`, `RoleLinkWIthOrgUnitTypes`, and the two

`$OWNER.*` recipient tokens.

Portal shell: home page, sidebar and share links (API)

This page is for integrators and developers who build their own navigation on top of the platform, or need to explain why a user's sidebar shows what it shows. It covers home-page resolution, the visibility rules, the single side-bar endpoint and its payload, the Restricted-user menu, and share (system) links.

The end-user view is Portal Layout & Navigation [Using OSOS]. The generated endpoint reference is in side-bar and system-link.

Home page

Opening `/` (or selecting the organization logo) shows the user's home page. Which page that is is resolved **per user** from configuration (`DefaultPageResolverService`), so users can land on different pages. The home page renders a configurable generic view (a page built from widgets) and can run **redirection rules** that forward the user to another target on arrival.

Sidebar sections

The sidebar is assembled top to bottom from independent sections, each shown only when it has content the user may see. A thin divider separates them; the module/view block scrolls on its own when taller than the viewport.

Order	Section	Shown when
1	Administration menu	The user has at least View-level admin access, or a Restricted grant (see below)
2	Space list	The user has no active delegation and is not Restricted (and has more than one space)
3	Services shortcuts	Services navigation is enabled and the user is not Restricted
4	Modules and views	The user has links (modules, pages, views, reports)

The Administration entries are filtered to the user's admin permission level.

Services shortcuts

Shortcut (English label)	Route	Live count
Services Catalog	`/s`	—
Outbox	`/my`	Ongoing requests
Inbox	`/inbox`	Ongoing inbox items
Watching	`/rm`	Ongoing watched requests

Route keys and older pages still say "My Requests" for Outbox and "Request Monitoring" for Watching. **Watching** is dropped from the menu entirely, rather than shown with zero, when the user watches nothing. Request access rules are in Requests & services.

Modules and views

A **module** is a container only: selecting it expands or collapses it. A module is omitted when it has no visible child. Inside it are **views**, **pages** and **reports**. Entries carry an icon and a localized title and are ordered by the builder's order, then alphabetically by title. Under a view the sidebar also lists the user's **saved queries**, **own collections** and **collections shared with them** (through their groups) for that view.

Reports are web-only: on mobile a report link maps to an unsupported-link placeholder and is filtered out of the sidebar.

On small screens the mobile layout's sidebar has a search field that searches recursively across link titles, saved queries and collections.

Visibility rules

A link appears only when all of these pass:

- **Published and active**: not unpublished, not archived.
- **Authentication type**: signed-in users see links marked *Authenticated*, *Everyone*, or *Restricted* that target them (directly or through a group). Anonymous visitors see only *Anonymous* or *Everyone* links.
- **Domain**: the link applies to all domains or to the current domain.
- **Sharing**: the link is Public, or Private/Shared to the current space.
- **Delegation**: under a delegation, links to views, reports or pages that were not delegated are removed.

The Restricted floor `Since 1.8`

Some accounts are provisioned at a system-level **Restricted** floor. For them the portal strips the shell:

- the space list and the Services shortcuts are hidden;
- the Administration menu is replaced with a minimal one: **Modules** (if granted module access) and/or **Data Entities** (if granted module or entity access). Module access implies entity access, so a module grant shows both.

A Restricted user with no module or entity grant sees neither block, only the modules and views their grants allow. Permission model: Permissions [Administration › Permissions & access].

`GET /v2/side-bar`

The portal fetches the whole sidebar from one endpoint.

Name	In	Meaning
`x-domain-id`	header	Current domain
`x-space-id`	header	Current space
`include`	query	Which parts to compute

`include`	Name	Returns
`0`	All (default)	Links **and** service-view status
`1`	Links	Links only (`serviceViewsStatus` is `null`)
`2`	OnlyService	Service-view status only

The response (`SidebarDtoV2`):

- `links` : the module/page/view/report tree.
- `serviceViewsStatus` : `hasServiceCatalog` , `hasRequests` , `hasInbox` , `hasObserverRequests` , and the counts `onGoingRequests` , `onGoingInbox` , `onGoingObserverRequests` .
- `isRestricted` : whether the user holds the Restricted floor.
- `restrictedAccess` : `{ hasModuleAccess, hasEntityAccess }` , used to build the minimal Restricted admin menu. Each is true when the user holds at least View access to any module or entity.

Each link is a `GenericLink` with `id` , `order` , localized `title` , `description` , `icon` , `type` , `url` , `target` and `children` . The `url` is derived from the type:

Link type	Value	URL
Module	`0`	none (container only)
View	`1`	`/v/<targetId>`
Page	`2`	`/<targetId>`
Report	`3`	`/reports/<targetId>` (web only)
Untyped	`4`	the link's own external `target.url`

View links also embed `userQueries` , `userCollections` and `sharedCollections` , each a lightweight link with its own `url` (for a collection, `/v/<viewId>/c/<collectionId>/entries`). The mobile app reads the same tree; how it maps each type is in Mobile Architecture.

Share links (system links)

A **system link** opens a view through a short shareable URL carrying a generated token instead of a sign-in. In the portal the route is `/vt/:token` . The builder generates it; its validity defaults to one year.

Endpoint	Auth	Purpose
`GET /v1/system-link/generate`	Authenticated	Generate a new unique token when creating a link
`GET /v1/system-link/resolve/{token}`	Anonymous allowed	Resolve a token to the data view or generic view it points to

Top bar

- The bell's unread badge shows the exact number up to 99, then `99+` .
- **Search** opens with **Ctrl + F** or **F3**; selecting outside the panel closes it.
- The **OSOS AI** button appears only when AI is enabled for the tenant. It opens a side panel that embeds an external app in an iframe, opened in the current UI language, so it depends on the AI service being reachable; it is not part of the portal shell itself. See OSOS AI assistant.
- **Counter widgets** (configurable live counters) can render next to the bell.
- On record pages the breadcrumb carries a quick-entry paginator for stepping between entries.

Profile routes

- `/user/profile` : the **User Preferences** page (basic, organization and account information, security, logins). `/user/settings` sits alongside it.

- `/edit-profile`: a lightweight profile record editor that warns before navigating away with unsaved changes.
- `/welcome`: the onboarding page, reachable from the profile menu's **Help**.

The settings panel (the gear button) saves per account (written to the user's preferences, following them across devices) or per session (stored in the browser session; the page reloads). **Reset** clears the session settings and restores the account defaults. The panel shows the running web and backend version numbers.

Virtual entities: model, API and internals

This page is for **integrators and developers** who create or read virtual entities through the API, write through a proxy entity, or need to know exactly how virtual entries are built and kept in sync.

> **Building a virtual entity in the UI?**
>
> Builders should start with Virtual entities [Build Apps] — it covers when to use each kind and how to create it in the labels the designer shows.

Kinds and storage

The three virtual entity kinds are, distinguished by `DataEntity.type` (`DataEntityTypeEnum`):

Kind	`DataEntity.type`	Enum value	What it does
Proxy	`VirtualProxy`	`50`	1:1 proxy of a single source entity. Each qualifying source entry is **materialised** as a proxy entry in a backing collection. `Since 1.10`, write-capable when bound to a governing Proxy view that permits it.
Joiner	`JoinerProxy`	`51`	Proxy that also exposes related entities joined through relations (`scope`). Read-only projection.
Aggregator	`VirtualAggregator`	`52`	Entity assembled by aggregating/joining several sources with explicit field mappings. Read-only projection.

All three share the same `DataEntity` document shape: the derivation rules live in `DataEntity.typeOptions`, whose concrete type depends on the entity `type` (`VirtualProxyEntityOptions` for proxy and joiner, `AggregatorVirtualEntityOptions` for the aggregator).

> **Note**
>
> This page documents the *proxy / virtual entity* machinery (`type` 50/51/52 with `typeOptions`). It is distinct from the per-field `DataEntity.virtual` option (`DataEntityVirtualOptions`: `enabled`, `source`, `pipeline`, `dependsOn`), which is a separate computed-field pipeline feature and is not covered here.

How virtual entities stay in sync

Every virtual entity records which source entries each of its entries was built from. When a source entry is created, updated, archived, or deleted — or when a query that feeds a virtual entity changes — OSOS refreshes the affected virtual entries. Aggregator entities additionally recompute when their sources change. This synchronization runs in the background, so a virtual entity always reflects the current state of its sources. The mechanics for proxy entities are detailed under Lifecycle.

> **Eventual consistency**
>
> Refresh is **event-driven**: the affected virtual entries are rebuilt in response to bus events published *after* the source change commits. Expect a short propagation delay rather than an instant, synchronous update — a virtual entity converges on the current state of its sources, it is not updated in the same transaction as the source. A write made **through** a write-capable Proxy entity is different: the write to the host is synchronous, and the caller gets a proxy-shaped preview of the row back immediately — but the row you read back afterward is still the one this same background refresh writes. See Write-through.

Type options data model

Proxy and joiner: `VirtualProxyEntityOptions`

`typeOptions` for `VirtualProxy` and `JoinerProxy` entities is a **flat list of sources**. The first/primary source is the root; every other source references its parent through `scope`.

```
interface VirtualProxyEntityOptions {
  sources: VirtualProxyEntityOptionsSource[];
}

interface VirtualProxyEntityOptionsSource {
  alias: string | null;                 // unique; used as field-name prefix "alias$fieldNormalizedName"
  type: VirtualProxyEntitySourceTypeEnum; // Primary = 0 | Additional = 1
  sourceEntityNormalizedName: string;
  sourceQuery?: string | null;          // legacy; VirtualProxy pre-migration fallback only — see below
  sourceViewId?: string | null;         // VirtualProxy only, Since 1.10 — the bound Proxy view; see below
  scope: { alias: string | null; relation: string } | null; // join to a parent source via a relation
  level: number;                        // Primary = 1; child = parent.level + 1
  isMandatory?: boolean;                // default false
}
```

Verified field semantics (from the model decorators and JSDoc in `virtual-proxy-entity-options-source.model.ts` and the `@osos/shared` interface):

- **`alias`** — for a `Primary` source the alias equals the source entity's normalized name; for `Additional` sources it is the relation's normalized name plus a disambiguating suffix. It is the prefix for joined field normalized names, in the form `alias + "$" + fieldNormalizedName`.
- **`type`** — `VirtualProxyEntitySourceTypeEnum.Primary` (`0`) or `Additional` (`1`). Exactly one `Primary` source is expected; the handlers locate it with `sources.find(x => x.type === Primary)`.
- **`sourceViewId`** (`VirtualProxy` primary source only, `Since 1.10`) — the id of the bound `kind: Proxy` view that governs this entity's capability ceiling and, when the view is query-sourced, its materialisation scope. See Proxy view governance. Required on every `VirtualProxy` primary source created or rebound going forward.
- **`sourceQuery`** (`VirtualProxy` primary source, **legacy**) — a `DataQuery` id. `Since 1.10`, a new or rebound `VirtualProxy` primary source that still carries a `sourceQuery` is **refused** — see Binding. It survives only as a **pre-migration fallback**: an entity that predates 1.10 and has not yet been bound to a `sourceViewId` still resolves its materialisation scope from this field. `JoinerProxy` additional/scoped sources still use `sourceQuery` as before — the retirement is specific to a `VirtualProxy` **primary** source.
- **`scope`** — `{ alias, relation }` links an `Additional` source to a parent source through a named relation (`relation` = the relation's normalized name). `null` for the primary source. Modelled by `VirtualProxyEntityOptionsSourceScope`.

- **`level`** — depth in the source tree; `Primary` is `1`, each child is `parent.level + 1`.

Aggregator: `AggregatorVirtualEntityOptions`

```
interface AggregatorVirtualEntityOptions {
  sources: AggregatorVirtualEntityOptionsSource[]; // [0] is the primary source
}

interface AggregatorVirtualEntityOptionsSource {
  alias: string | null;
  type: VirtualProxyEntitySourceTypeEnum;   // Primary | Additional
  sourceEntityNormalizedName: string;
  sourceQuery: string | null;
  includedFields: string[];                 // default []
  fieldMappings: AggregatorEntityFieldMapping[]; // required
}

interface AggregatorEntityFieldMapping {
  normalizedName: string;                   // field in THIS (aggregator) entity
  type: AggregatorEntityFieldMappingType;   // Fixed=0 | Dynamic=1 | NotSet=2 | Expression=3
  value: string | null | unknown;           // source field name, fixed value, expression, or null
  optionsMappings?: OptionsMapping[];        // category option-id remapping
}

interface OptionsMapping {
  sourceOptionId: string;                   // option id in the SOURCE category field
  targetOptionId: string | null;            // corresponding option id in THIS entity
}
```

Field-mapping `type` values (`AggregatorEntityFieldMappingType`) drive how `value` is interpreted:

`type`	Value	Meaning of `value`
`Fixed`	`0`	A fixed literal written into every aggregated entry.
`Dynamic`	`1`	The normalized name of a field in the source entity to copy from.
`NotSet`	`2`	The field is left unset.
`Expression`	`3`	An expression evaluated to produce the value.

`optionsMappings` *(Since 1.6)* lets an aggregator map category-field option ids when the source and target entities use different option ids for the same semantic value. Each `OptionsMapping` maps a `sourceOptionId` to a `targetOptionId` (nullable). It is only required when aggregating category fields whose option ids differ across the source and target.

The aggregator sources are a *positional* array — `sources[0]` is the primary entity where aggregation starts; the remaining entries are additional sources joined into it.

Proxy view governance: capability ceilings `Since 1.10`

`Since 1.10`, a `VirtualProxy` entity's write capability — whether it can be created, edited, archived, restored, or saved as a draft at all, and through which fields — is governed entirely by a **Proxy view** bound to it, not by anything stored on the entity itself. The governing rule: **the ceiling is derived fresh on every read, never cached** — a proxy entity carries no stored record of what it is currently allowed to do, so revoking a capability on the governing view takes effect immediately everywhere the proxy is used.

The `kind: Proxy` view and its restricted editor surface

A data view gains a third `kind`: `Implicit` (system-built), `UserDefined` (an ordinary view), and `Proxy` — a view built specifically to govern a `VirtualProxy` entity. A view's `kind` is fixed once set; it cannot be changed later, even by a payload that simply omits it.

A `kind: Proxy` view's admin editor is deliberately narrower than an ordinary view's. Every save is projected onto an allow-list — identity (name, icon, description), the source entity/query, the field list, the Create/Edit/Archive operation toggles (each with its own field list), and the create-as-draft option — and **every other property is silently reset to its default** on save, server-side, not merely hidden in the UI. In particular a Proxy view has **no publish state, no anonymous/authenticated access settings, and no permissions of its own** — its sharing always matches the host entity's — because access to the entity it governs is controlled through the entity's own permissions, not through the governing view.

When a `VirtualProxy` entity is created without naming an existing governing view, OSOS **auto-creates a default one**: read-only (no operations enabled), scoped to every field the source entity currently has, presented as a Table. An admin enables Create/Edit/Archive on it afterward to make the proxy write-capable.

Binding a proxy entity to its source view `Since 1.10`

Binding (or rebinding) a `VirtualProxy` entity to a governing view is validated on every create and every update:

- The named view must actually be `kind: Proxy`, and must govern the **same** source entity the proxy entity is on — a proxy cannot bind to a view built for a different entity.
- The entity's source cannot still carry a legacy `sourceQuery` — see Type options above.
- The binding cannot form a **cycle**, cannot chain more than **10 views deep**, and must ultimately resolve to a real, non-proxy host — a Proxy view is allowed to itself govern another `VirtualProxy` entity (a proxy bound to a proxy), as long as the chain eventually bottoms out on genuine source data.

The field ceiling is enforced from **both directions**: narrowing a governing view's field list is refused if a bound proxy entity still needs one of the dropped fields, and narrowing a proxy entity's own field selection is refused if it would exceed what its governing view still covers.

Once bound, a `VirtualProxy` entity cannot be unbound — an update that omits the governing view on an already-bound entity is refused outright; the only way to change it is to rebind to a different `kind: Proxy` view.

Deriving capabilities

At read time, OSOS resolves the proxy's **capability ceiling** by walking its binding chain (through every intermediate proxy-bound-to-proxy hop) and **intersecting** the capabilities each bound view grants — a longer chain can only ever narrow what is permitted, never widen it. Any failure along the way — an unresolvable entity, a chain that never reaches a real host, or any other error — resolves to a **fully closed** ceiling: nothing granted, no fields exposed. There is no "unrestricted by default" fallback.

The resolved ceiling covers:

Capability	Meaning
`bound`	`false` only when the binding chain could not be resolved at all.
`fields`	The intersected set of field paths the proxy may expose.
Screens (entries list / entry details / dashboard)	Which screens the chain permits at all.

Capability	Meaning
`allowCreateEntryAsDraft`	Whether a draft can be started on the proxy.
Create / Edit / Archive	Whether each operation is granted — an operation absent from every view in the chain grants nothing; there is no separate override.
Save-invalid on create / edit	Whether a write that fails validation is still accepted, per operation.
Show invalid entries, and its two filters	Gated together — the two filters can only ever be on when the invalid-entries view itself is.

Any view actually bound to the proxy entity — the one an end user opens to work with its entries — has this same ceiling clamped onto it before it is ever served: its field list is filtered down to the ceiling, its operations are filtered to only what the ceiling grants, and each surviving operation's own field list is intersected with the ceiling too, so a view that declares "all fields" for an operation still cannot exceed what the proxy is actually allowed. This is what makes the ceiling **enforced**, not just advisory — every read path that resolves a view over a proxy entity applies it, not only the admin editor.

`Since 1.10`, reading a `VirtualProxy` entity's details (single-entity read only — not the entries list) also returns this resolved ceiling as a `proxyCapabilities` field, so a caller can tell up front what it is allowed to do without probing. The field is **absent** on a non-proxy entity, and is present with `bound: false` on a proxy whose binding could not be resolved — the two cases are deliberately distinguishable. See Calling the HTTP API for the endpoint.

> **Note**
>
> `proxyCapabilities` is not yet reflected in the published OpenAPI specification for the entities endpoint. Treat this guide as authoritative on its shape until the specification catches up.

Field-ceiling-driven picker cascade

The same ceiling also narrows what an admin sees while **building** a Proxy entity's own field list: any field the governing view's ceiling withholds is dropped from the pickers used to configure the proxy entity itself (and from anything built on top of that field list, such as table columns or an operation's own field list). A display-only field at the top level is left alone — it never actually reads or writes data — but every field nested inside a Block is checked regardless, because the save-time guard checks the same way; a field silently exempted only in the picker would otherwise fail to save for a reason the picker never showed. See Data Views & Faces [Build Apps] for how the view designer itself surfaces this narrowing while building the governing view.

API

All endpoints are on the data-service `DataEntitiesController` (`/v1/entities/...`). The bodies below are the verified DTO shapes.

Create a virtual entity

`POST /v1/entities/virtual`

Request body (`CreateVirtualDataEntityDtoModel`):

Field	Type	Required	Notes
`name`	`LocalizedValue`	yes	

Field	Type	Required	Notes
`description`	`NullableLocalizedValue`	no	
`normalizedName`	`string`	yes	unique; validated against `NormalizedNameValidation` (length + pattern). Restricted names are rejected.
`permissions`	`Permission[]`	no	stored into access rights, not the entity doc.
`type`	`DataEntityTypeEnum`	yes	must be one of `VirtualProxy` (50), `JoinerProxy` (51), `VirtualAggregator` (52).
`typeOptions`	see below	yes	
`icon`	`Icon`	yes	
`module`	`UUID`	yes	must **not** be the core module id (`ImplicitModuleException` otherwise).
`entryNames`	`DataEntityEntryNames`	yes	
`displayOption`	`DataEntityDisplayOptionEnum`	no	defaults to `KeyValue`.

`typeOptions` accepts two shapes (handled in `createVirtual`):

1. **Flat / interactive** — `{ sourceEntityNormalizedName, sourceQuery?, sourceViewId? }` (`sourceViewId` added `Since 1.10`, `VirtualProxy` only — see Proxy binding, at create time below). The controller wraps this into a single `Primary` `VirtualProxyEntityOptionsSource` (`alias: null`, `type: Primary`, `scope: null`).
2. **Persisted / passthrough** — `{ sources: [...] }`. If `typeOptions.sources` is an array it is passed through **untouched**. This path is used by packaging import to replay a previously exported entity (proxy/joiner sources with `scope`, or aggregator sources with `fieldMappings`); rebuilding it as a `VirtualProxyEntityOptions` would strip aggregator-specific fields.

OSOS creates the entity directly and returns it.

Minimal example (interactive proxy of a `tickets` entity, with no governing view named — OSOS auto-creates one):

```
POST /v1/entities/virtual
{
  "name": { "en": "Open Tickets" },
  "normalizedName": "open_tickets",
  "type": 50,
  "typeOptions": {
    "sourceEntityNormalizedName": "tickets"
  },
  "icon": { "...": "..." },
  "module": "…module-uuid…",
  "entryNames": { "...": "..." }
}
```

Proxy binding, at create time `Since 1.10`

For a `VirtualProxy` create specifically, OSOS resolves the governing view **before** the entity itself is created, from the primary source's `sourceViewId` / `sourceQuery`:

1. **You name a `sourceViewId`.** The named view must exist and pass the binding checks above. No default view is created.
2. **You name neither a `sourceViewId` nor a `sourceQuery`.** A **default read-only Proxy view is auto-created**, scoped to every field the source entity currently has, and bound to the new entity. If the entity create that follows then fails, the auto-created view is deleted again — unless the entity actually persisted despite the error, in which case the view is deliberately kept rather than leaving a persisted entity with a dangling binding.

3. **You name a legacy inline `sourceQuery` with no `sourceViewId`.** Refused outright for a **new** entity. The one exception is a **packaging import** replaying a package built before 1.9, which may still carry `sourceQuery` inline — that import is accepted and left for a migration to relocate onto a bound view, rather than failing.

`sourceViewId` is rejected outright on a `JoinerProxy` / `VirtualAggregator` create — binding is `VirtualProxy`-only.

Update a virtual entity

`PUT /v1/entities/virtual/{id}`

Request body (`UpdateVirtualDataEntityDtoModel`): `name`, `description`, `permissions?`, `typeOptions?` (`VirtualProxyEntityOptions | AggregatorVirtualEntityOptions`), `icon`, `entryNames`, `displayOption?`, `fields[]`, `internalEntities[]`, `internalCollections[]`, `templates?`, and a required `updateIdentifier`.

Concurrency / safety checks performed before the update command is sent:

- Implicit entities cannot be edited (`ImplicitEntityUpdateException`).
- `updateIdentifier` must equal the stored one, else `UpdateIdentifierConflictException` (optimistic concurrency).
- The entity must not be locked by another user (`LockedObjectException`).

Emits `UpdateVirtualDataEntityCommand` and returns the entity.

`Since 1.10`, if `typeOptions` on a `VirtualProxy` update names a new `sourceViewId`, the same binding checks run again against the new view — including a check that the entity's own (possibly also-updated) field selection still fits the new view's ceiling. Omitting `sourceViewId` on an update to an **already-bound** entity is refused: an already-bound proxy cannot be unbound through an update.

Read entity details

`GET` on the entity-details endpoint returns a `DataEntityDto` whose `typeDetails` is resolved from `typeOptions`:

- **`VirtualAggregator`** → `DataEntityAggregatorDetailsDto` with `primarySource`, `additionalSources[]`, and `mappedSources` (computed by `DataEntityService.getVirtualEntityMappedSources`).
- **`JoinerProxy` / `VirtualProxy`** → `DataEntityJoinerDetailsDto` with `primarySource` and, for each source that has a `scope.relation`, a resolved `sources[]` entry carrying the relation DTO.

If an entity has no `typeOptions.sources`, a warning is logged and details are returned with empty `typeDetails`.

`Since 1.10`, a `VirtualProxy` entity's details additionally carry `proxyCapabilities` — see Deriving capabilities above.

Delete

`DELETE /v1/entities/{id}` archives the entity (shared delete endpoint, not virtual-specific). `Since 1.10` a system administrator can also remove a virtual entity for good with `DELETE /v1/entities/{id}/permanent` — see Admin operations.

Scoping: which source entries are proxied

`Since 1.10`, the effective scope for a `VirtualProxy` entity is resolved through its **bound Proxy view**, not through a query attached to the entity directly:

- **Bound (`sourceViewId` set — the normal case since 1.10)** — the bound view is loaded. If the view is **Query-sourced**, that view's own query is the effective scope. If the view is **Store-sourced**, there is no filter — every source entry is proxied. A Query-sourced view that names no query, or a binding that no longer resolves at all, is treated as **"refresh/write nothing"** rather than "no filter" — a broken binding fails closed; it never silently widens to mirror the whole source.
- **Not yet bound (pre-1.10 entity only)** — falls back to the legacy `sourceQuery` field directly (`null` meaning "all entries"), exactly as before this release.

Once the effective query id (or `null`, or the "nothing" sentinel) is resolved, OSOS builds the fetch criteria the same way as before:

- If a specific source entry id is given, it filters to that entry.
- If a scoping query is in effect, the query's criteria is loaded and AND-ed in.
- If neither is given, **all** source entries are returned.

Source entries are fetched without resolving relations or categories, selecting only the fields the proxy actually needs: id, entity, value, lifecycle status, state, state history, store, invalid flag/reason, and creation/update info.

The same scope also gates every **write** made through the proxy — see Write-through.

How a proxy entry is built

For `VirtualProxy` entities, `ProxyDataEntryService.refreshEntry` turns each qualifying source entry into a proxy `DataEntry`:

- **Deterministic id** — `hashId(normalizedName, sourceId)` = `Utils.hashString("${dataEntity.normalizedName}_${sourceEntry.id}")`. This makes the proxy id stable and idempotent per (proxy-entity, source-entry) pair.
- **`entryType`** = `EntryTypeEnum.Virtual` (`1`).
- **`basedOn`** = a single `ReferenceProperty { dataEntity, dataEntry }` pointing back at the source entry (used later to find proxy entries for a given source entry via the entity index `basedOn.dataEntry`).
- **`value`** — copied field-by-field from the source entry using `copyValuesByPaths` over `dataEntity.getAllFieldsPaths()`. The copier walks dotted paths and recurses into arrays, so nested and array-valued fields are reconstructed on the target.
- **Mirrored from source** — `isInvalid`, `invalidReason`, `state`, `stateHistory`, `store`, and `creationInfo` / `updateInfo` actor info (`doneBy` / `delegatee`, falling back to `creationInfo` when `updateInfo` is absent). `entityVersion` is set to the proxy entity's `version`.
- When an `originalEntry` is passed, that instance is refreshed in place instead of creating a new one (so id, `basedOn`, and `creationInfo` are preserved on update).

Write-through: entry writes on a proxy entity `Since 1.10`

`Since 1.10`, when a `VirtualProxy` entity's capability ceiling grants it, writes made directly against the proxy are **redirected onto the host (source) entity** rather than refused. Nothing below is available on an unbound proxy, or one whose binding could not be resolved.

What routes through to the host

Operation	Routes through?
Single-entry create (a plain entry create, or through a data view; also reached from a reaction's Create Data Entry action)	Yes
Batch create	Yes for the main row; a batch create's related rows are refused outright — they would need to reference the proxy row before the host write's own materialisation event has produced it.
Full update (`PUT`) and patch (`PATCH` , JSON-Patch ops)	Yes — the incoming value is diffed/translated and applied as a write on the host.
Bulk patch	Yes, row by row, inside the same transaction as the rest of the batch.
Archive, and restore — single and bulk	Yes — the proxy row is resolved back to its host entry, and that host entry is archived/restored.
Draft create/update	Yes, gated separately on the ceiling's own draft flag. A new draft's store and workflow are seeded from the host entity, since a proxy has neither of its own.
`JoinerProxy` / `VirtualAggregator` create, archive, restore, batch-create	**No** — write-through is `VirtualProxy` -only; the other two virtual kinds remain read-only projections. Patch is the one long-standing exception: an Aggregator entry has always supported patch, translated through its field mappings rather than through a ceiling.

How a redirected write is scoped

Every write-through path follows the same shape:

1. The relevant capability (create/edit/archive) must be granted by the ceiling, or the write is refused outright.
2. The write is narrowed to the allowed field set — the ceiling's fields, further narrowed by whatever view the caller actually used to make the write.
3. Anything **outside** that field set is refused, never silently dropped. (One narrow, deliberate exception: the handful of workflow/ordering metadata fields an ordinary edit form always re-submits are exempt from the ceiling check — but only when they are restored verbatim from the stored entry, and only on the ordinary edit path; a caller supplying its own override of that metadata gets no exemption.)
4. The write is checked against the proxy's effective scope — evaluated against the value the write would actually produce, before anything is persisted. A write that would land, or move an entry, outside the governing view's scope is refused rather than silently written out of bounds or silently dropped.
5. The write lands on the host entity under the host's own id space — the proxy's governing view has no bearing on the host write's own view-level rules.

No new sync logic is introduced by write-through: the host's own create/update/archive/restore events are what refresh the proxy row afterward, exactly as for any other source change — see Lifecycle. The one exception is the HTTP response to a create, which includes a proxy-shaped preview of the just-written row immediately, without waiting for that refresh to land — the real, persisted proxy row is still the one the refresh handler writes once the event arrives.

Field sync: keeping a proxy's cloned field definitions in step `Since 1.10`

A `VirtualProxy` entity's field definitions are **cloned** from its source at the time the proxy entity itself is created or its field set is updated — they are not a live reference. `Since 1.10` , OSOS also keeps that clone in step when the **source** field definition changes later: before this, a field turned required on the host stayed optional on every proxy cloned from it, and a write through the proxy could be refused by a requirement its own form never showed.

Only a deliberately narrow set of properties is re-synced from the source — the data type, validations, render options, and (for the `unique` flag specifically) the one part of a field's display options that is a correctness rule rather than a display choice. Everything else — every display setting, and any workflow requirement — stays the proxy's own; a **requirement** in particular is never inherited, because a proxy entity has no workflow of its own to enforce it against.

A one-time repair pass applies this same rule in bulk to every `VirtualProxy` entity that had already drifted before this fix, without generating change notifications of its own — it only catches up stored field definitions that ordinary indexing already picks up on the entity's next update.

Source-workflow lock state on a Proxy entry `Since 1.10`

A `VirtualProxy` entity carries no workflow of its own — like every virtual entity — so a proxy row shows no workflow state at all unless OSOS derives one. `Since 1.10`, it does: a proxy row's edit lock now follows its **source** entry's active workflow, the same way an Aggregator entry's Edit visibility already follows its source workflow.

- If the source's workflow has reached a terminal state, every non-Block field on the proxy is locked.
- If an active workflow step exists but the current user isn't its owner, every field is locked too.
- Otherwise, the source's own step requirements and the governing view's own required-fields rules apply, and any proxy field whose source counterpart is hidden or locked is locked to match — this only ever **adds** locks on top of the governing view's own, never removes one.
- A row whose source cannot be resolved, or whose source has no workflow at all, is left at the governing view's own lock state; nothing is derived.

Lifecycle: keeping proxy entries in sync

`VirtualProxy` entries are **materialised** and kept in sync by event handlers on the service bus. Each handler discovers affected proxy entities with a query of the form:

```
where: {
  type: DataEntityTypeEnum.VirtualProxy,
  'typeOptions.sources.sourceEntityNormalizedName': <sourceEntity.normalizedName>,
}
```

> **Note**
>
> These refresh handlers explicitly filter `type: VirtualProxy` — they are specific to proxy entries. `JoinerProxy` and `VirtualAggregator` entries are **not** materialised by this service, but they are not left uncomputed either: each has its own materialisation service running its own aggregation pipeline. See How joiner entries are materialized below for the joiner mechanics, and How aggregator entries are materialized for the aggregator side.

Trigger event	Handler	Behaviour
Source **entity** created (type `VirtualProxy`)	`RefreshProxyEntriesOnDataEntityCreatedHandler`	Resolves the primary source entity, fetches all matching source entries, `refreshEntry` s each and `createAll` s them into the proxy collection; publishes `DataEntryCreatedEvent` per entry. Fails the message if the source entity is missing.
Source **entry** created	`RefreshProxyEntryOnDataEntryCreated`	For every proxy entity on that source: if the proxy's effective scope doesn't match the new entry, skip; otherwise create the proxy entry and publish `DataEntryCreatedEvent`.

Trigger event	Handler	Behaviour
Source **entry** updated	`RefreshProxyEntryOnDataEntryUpdated`	Uses the entity index (`basedOn.dataEntry`) to find the existing proxy entry, then reconciles: create if newly-matching, delete if it no longer matches the effective scope, else refresh in place. Publishes Created/Updated/Deleted accordingly.
Source **entry** permanently deleted	`DeleteProxyEntryOnDataEntryPermanentlyDeleted`	Finds proxy entries via `basedOn.dataEntry` and cascade-deletes them plus their index rows.
Source **query** updated (criteria changed)	`RefreshProxyEntriesOnQueryUpdatedHandler`	Only runs when `criteriaChanged`. For each proxy entity bound to that query it **rebuilds the whole set**: deletes all proxy entries, recreates from the current query result, and publishes Created/Updated/Deleted events by diffing ids.

Additional entry handlers exist for archive / delete / invalidate transitions (`refresh-proxy-entry-on-data-entry-archived`, `-deleted`, `-invalidated`, `-data-entries-archived`) that propagate those state changes to the proxy copies.

Error-handling convention across handlers: `ENTITY_NOT_FOUND` is treated as non-fatal (logged and skipped / message failed); other errors are logged and rethrown so the bus can retry.

How joiner entries are materialized

While the proxy refresh handlers above filter `type: VirtualProxy` only, `JoinerProxy` entries are materialized by a separate service (`JoinerDataEntryService`). Its `buildJoinerStages` compiles a MongoDB aggregation pipeline that is executed with a trailing `$out` stage (`DataEntryRepository.refreshVirtualEntity` appends `{$out: <normalizedName>}`), overwriting the joiner's backing collection with the freshly joined rows.

The pipeline is built from the joiner's flat `sources` list:

- **`$lookup` per additional source** — the primary source is joined to each additional source through its `scope` relation. For a **level-0** (non-array) link the lookup uses `localField` / `foreignField`; for an **array-valued** link it uses a `let` + `$expr` pipeline instead.
- **`$unwind` per array level** — one unwind is emitted for each array level along the joined path (block-, field-, and multi-value–aware). `destructTargetField` is applied when the target field is repeatable but **not** when it is multi-value.
- **Inner vs left join** — an `isMandatory` additional source (or an additional-id filter) sets `preserveNullAndEmptyArrays: false`, i.e. only entries with a match on that source are kept (**inner join**). Otherwise it is `true`, so entries are kept even without a match (**left join**). This is the read-side realisation of the mandatory/optional choice made when creating a joiner [Build Apps › Virtual entities].
- **Provenance** — each joined row records the source entries it was built from in `basedOn[]`, using `includeArrayIndex` so the generated ids are deterministic.

Like proxy entries, joiner entries are refreshed on source entity/entry changes and on **saved-query updates** (a joiner can be driven by a query), and removed when a source entry is permanently deleted — via bus handlers under `joiner-entity/*`, `joiner-entries/*`, and `refresh-joiner-entries-on-query-updated`.

How aggregator entries are materialized

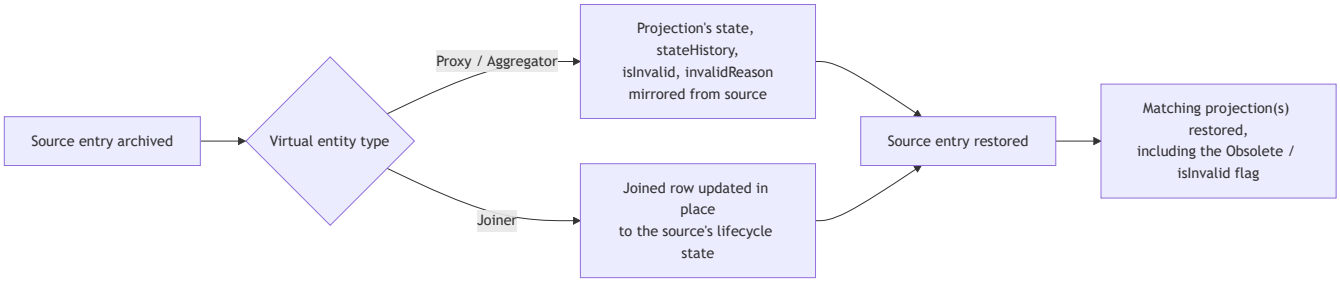
Aggregator entries are materialized the same way as joiner entries: a dedicated service (`AggregatorDataEntryService`) runs its own MongoDB aggregation pipeline — built from the aggregator's primary and additional sources and their

field mappings — and refreshes the aggregator's backing collection from it, in parallel to `ProxyDataEntryService` for `VirtualProxy` and `JoinerDataEntryService` for `JoinerProxy`.

Like a joiner's, an aggregator's rebuild now matches source entries in state **Active** or **Archived** (still excluding `Draft`), not `Active` only — see Archive/restore state propagation for why that matters.

Archive/restore state propagation, for every virtual entity type `Since 1.8`

Since 1.8, archiving and restoring a source entry correctly propagates its lifecycle state to projections of **all three** virtual entity types — Proxy, Joiner, and Aggregator — not just Proxy. This is distinct from population/materialisation (each type has its own materialisation path, as above); it is specifically about keeping an already-materialised (or joined/aggregated) row's archived/restored state faithful to its source.



- **Proxy and aggregator archive handling now mirrors the full lifecycle, not just `state`.** Before 1.8, archiving a proxy's or an aggregator's source entry only flipped `state: Archived` on the projection. It now writes `state`, `stateHistory`, `isInvalid`, and `invalidReason` straight from the source entry — matching exactly what the restore path writes back — so an entry that had become obsolete before being archived, for example, carries that through correctly.
- **Joiner rows are updated in place instead of being deleted.** Archiving a source used to hard-delete the corresponding joiner row (and its index entry), which meant a restore had nothing left to bring back. The joiner row is now updated in place to the source's lifecycle state instead.
- **A restore handler now exists for every type.** Proxy, Joiner, and Aggregator projections are each restored — including the obsolete/invalid flag — when their source entry is restored.
- **Rebuilds no longer drop archived rows.** A joiner's or aggregator's rebuild (triggered by editing any field on it, not only a source remap) now matches source entries in state Active or Archived, not Active only. Before this, rebuilding a joiner or aggregator entity permanently dropped every row whose source was archived.
- **A joiner row is only "live" when every one of its participating sources is.** A joiner row can combine several sources. If any participating source is archived, the row's own state is now downgraded to Archived (adopting that source's state history) even if the primary source is still active — the first archived participant wins deterministically when more than one source qualifies.

Aggregate-entry Edit visibility follows the source workflow `Since 1.8`

An aggregator entry has no workflow of its own to inspect: the aggregator entity carries no `workflows`, and a projected/aggregated entry carries no current-step metadata. The workflow that actually governs editability runs on the **source** entry — which tracks with the fact that editing an aggregate row only ever redirects to the source entry's own edit page.

Since 1.8, the platform resolves that source workflow's terminal state and uses it to hide the Edit action once the source workflow has completed — both for a single opened entry and per-row in list views (the admin Entity Manage list and the public view portal's data view) — matching what the source entry's own view/list already does.

Admin operations on virtual entities `Since 1.10`

The entity-level admin features of 1.10 are documented once, in Data Modeling: Entities [Build Apps › Create and design an entity]. This table records only how they treat the three virtual kinds — proxy, joiner and aggregator:

Feature	On a virtual entity
Move to another module (single and bulk)	**Supported.** The move changes nothing but the module, keeps the installed-package and archive checks, and publishes the usual module-change event. Before 1.10 moving any virtual entity failed with a `500`.
Permanent delete	**Supported** for all three kinds, with the same "unlink first" dependency check as any other entity.
Custom indexes (Indexes tab)	**Not supported** — a virtual entity has no collection of its own to index. The tab is not offered, `index-paths` answers `[]`, and sending custom indexes is refused with `ENTITY_INDEXES_NOT_SUPPORTED`. For the same reason the 1.10 collated list-read indexes are not added to virtual entities.
Export templates [Build Apps]	`Since 1.6`. Templates can be managed on a virtual entity that is not implicit (or whose `implicitOptions.allowEdit` is true), and are saved through the virtual-entity update. A template export includes the rows the user selected.

Two repairs in the same release concern virtual entities that had silently stopped refreshing; both run once as upgrade migrations and need no action:

- `group_assignments`. The pipeline behind this built-in virtual entity could produce duplicate ids for a user with two main assignments under one ancestor, which froze the materialised collection (and every grant derived from it) at its last good state. The pipeline is replaced and every virtual entity rebuilt.
- **Aggregator sources without field mappings.** Aggregators created interactively could store a source with no `fieldMappings`, and such an aggregate was skipped by every refresh. Those sources get an empty mapping list and the aggregates are rebuilt.

Editability of virtual entries

Once created, every virtual entity behaves like any other entity for **reading**: you can build views on it, add it as the source of a reference field, filter and search it, and use it in queries, statistics, and reports.

For **writing**, the three kinds differ:

- **Joiner** and **Aggregator** entries stay computed read models — to change what they show, change their source data or their configuration, not the entry itself.
- A **Proxy** entry can be created, edited, archived, and restored directly, exactly like an ordinary entry — **when its governing view's capability ceiling allows it** (see Proxy view governance). A freshly created Proxy entity is read-only by default; an admin must explicitly turn on Create/Edit/Archive on its governing view before end users can write through it.

> **Note**
>
> Virtual entries are computed projections at their core, but editability is not assumed to be uniform: for **Aggregator** entities the entry renderer checks each field against the backend via `GET .../editable/{fieldId}` before deciding whether it can be edited, rather than treating the whole entry as blanket read-only; for a **Proxy** entity, what is editable is whatever its governing view's ceiling currently grants.

Version notes

- The proxy/aggregator source model (`VirtualProxyEntitySourceTypeEnum`, `scope`, `isMandatory`, `level`) and `ProxyDataEntryService` are **foundational** and **Long-standing** — they predate the 1.2 milestone scheme, so no milestone badge is assigned.
- The `JoinerProxy` (51) and `VirtualAggregator` (52) entity types and `AggregatorEntityFieldMappingType` are likewise **Long-standing** (predate 1.2).
- **Since 1.6** — `OptionsMapping` (category option-id remapping on aggregator field mappings) was introduced.
- **Since 1.8** — cascade permanent-delete of proxy entries (`DeleteProxyEntryOnDataEntryPermanentlyDeleted`).
- **Since 1.8** — archive/restore lifecycle-state propagation (`state`, `stateHistory`, `isInvalid`, `invalidReason`) extended from Proxy alone to Joiner and Aggregator as well; see Archive/restore state propagation.
- **Since 1.8** — the aggregate-entry Edit action is hidden once the source workflow completes, both for a single entry and per-row in list views; see Aggregate-entry Edit visibility.
- **Since 1.10** — **Proxy view governance and entry write-through**: the `kind: Proxy` view, its restricted editor surface, capability-ceiling derivation and enforcement, binding/cycle guards, the `proxyCapabilities` entity-response field, and create/update/patch/archive/restore/batch-create write-through for Proxy entries. See Proxy view governance and Write-through.
- **Since 1.10** — move (single and bulk) and permanent delete work on all three virtual kinds; custom indexes are not supported; see Admin operations.
- **Since 1.10** — field sync between a proxy's cloned fields and their source; see Field sync.
- **Since 1.10** — source-workflow-derived lock state on a Proxy entry; see Source-workflow lock state.
- **Since 1.10** — `sourceQuery` on a `VirtualProxy` primary source is **retired** in favor of view-based scoping, surviving only as a pre-migration fallback; see Type options and Scoping.

Behaviour history

- **Virtual entity creation is synchronous.** Earlier versions of this documentation described creation as emitting a `CreateVirtualDataEntityCommand` over the service bus. That indirection has been retired in favour of a direct, synchronous call — there is no bus round trip on the create path.

Delegation API & enforcement

This page is for integrators, operators and engineers who need to act under a delegation programmatically, or to reason about exactly how a delegation is enforced. The administrator's view (creating, scoping, forwarding and ending a delegation) is Delegation [Administration]. The gateway plugin is also listed with the other plugins in Gateway operations, and the delegate's permission resolution in the engine is in Engine: Permissions.

The core delegation entity and the active-delegations endpoint predate the 1.2 milestone. Scope-based permission filtering and administration inheritance were added later, but are also long-standing (pre-1.2).

The delegation record

A delegation is a normal data entry of the `delegation` entity (system id `DELEGATION`). Its `value` holds:

Field	Label	Type	Required	Meaning
`delegator`	Delegator	reference (user)	yes	The user whose duties and permissions are delegated.
`delegated_to`	Delegated To	reference (user)	yes	The delegate.
`from_date`	From Date	date-time	yes	When the delegation becomes active.
`to_date`	To Date	date-time	no (nullable)	When it ends. `null`/absent means open-ended.
`is_active`	Is Active	category (`DelegationTypeEnum`)	yes	`ACTIVE` or `INACTIVE`: the manual on/off switch.
`delegation_scope`	Delegation Scope	object	yes	What is delegated (see below).

Delegation scope

Field	Label	Type	Meaning
`base`	Base	category (`DelegationScopeEnum`)	The default posture: `ALLOW_ALL` or `DENY_ALL`.
`exception`	Exception	array of references	System objects the base posture does *not* apply to. Defaults to `[]`.
`allow_administration`	Administration	array of category (`DelegationAdministrationEnum`), optional	When it contains `ALLOWED`, the delegate also inherits the delegator's administration (system) permissions.
`forward_notifications`	Forward Notifications	boolean, default off	Deliver a copy of the delegator's in-app notifications to the delegate. `Since 1.10`
`forward_emails`	Forward Emails	boolean, default off	Copy the delegate on the delegator's emails. `Since 1.10`

Enum values: `DelegationTypeEnum` = `ACTIVE`, `INACTIVE`; `DelegationScopeEnum` = `ALLOW_ALL`, `DENY_ALL`; `DelegationAdministrationEnum` = `ALLOWED`.

The `USER` entity carries an implicit relation **Delegation** ("delegations given by the user", `USER_GIVEN_DELEGATIONS_RELATION`) onto `DELEGATION` through the `delegator` field. `Since 1.6`

What the exception picker offers `Since 1.8`

The options offered in **Exception** are scoped to what the administrator creating the entry can reach:

- A caller with a **system-level** grant (system viewer, editor or admin) is offered the **entire published application tree**: modules, views, pages, reports and service actions.
- Any other caller sees only the views, pages and reports they hold an access right on, plus the service actions they may submit or observe, plus the ancestor nodes needed to keep those items attached to the tree.
- An anonymous caller is offered the anonymous-enabled views and pages; a caller with no permissions at all is offered nothing.

This governs only what the picker *offers*. `delegation_scope.exception` is applied at request time as a filter over the **delegator's own** permissions (see Scope enforcement), so a wider option list can never widen a delegate's actual access.

When is a delegation active?

A delegation is honoured only when **all** of these hold:

1. `is_active` is `ACTIVE`.
2. `from_date <= now`.
3. `to_date` is `null` **or** `to_date >= now`.
4. The entry `state` is `Active` (not archived or deleted).

The active-delegations query returns the delegations where the given user is the **delegate** (`delegated_to` matches). If the same delegator appears more than once, results are de-duplicated so the delegate sees one delegation per delegator. Each returned delegation has its `delegator` reference resolved with display metadata.

How OSOS enforces delegation

Delegation is applied **per request** via the `X-DELEGATION-ID` request header. A client that wants to act under a delegation sends the delegation entry's id in that header; there is no server-side "delegation mode" state, and the header's presence on each request is what activates delegation for that request. Enforcement is split between the API gateway (delegate binding, time window and identity swap) and the data service (`is_active`, entry state and scope).

On mobile, selecting a delegation persists the choice and reloads the app; a client-side interceptor (`DelegationInterceptor`) then attaches the delegation header to every request. On the web, signing out ends the active delegation. `Since 1.6`

Gateway enforcement

Enforcement in the API gateway (Kong) runs through the custom `user-delegation` plugin (packaged as `kong-plugin-user-delegation-0.1.0-1.rockspec`, depending on `lua-resty-redis-connector 0.11.0`). It runs at **priority 1000**: after token introspection and app-token auth (which establish `x-user-id`), and before `user-space-checker` (999) and `osos-access` (997). The flow:

1. **Trigger.** Without an `X-DELEGATION-ID` header the plugin passes the request through untouched.

2. **Require an authenticated user.** If `x-user-id` is missing, it returns `400 MISSING_USER_ID`.

3. **Load delegation details.** It looks the delegation up in Redis under a **tenant-scoped** key, `osos:delegation:v2:<delegation_id>`, composed with the `envPrefix` and the tenant id (a request whose tenant is unresolved is refused with `500 APP_ERROR` before any lookup). On a miss it calls `GET <data_service_url>/api/v1/delegations/<id>` with internal service headers (the gateway's service identity and the shared key, plus the tenant), then caches the result for `cache_ttl` (default **600 s**). Failure modes: a non-200 is forwarded as `DELEGATION_API_ERROR`; a request that fails outright returns `500 DELEGATION_API_REQUEST_FAILED`; `success: false` returns `500 DELEGATION_API_LOGIC_ERROR`; an undecodable body returns `500 DELEGATION_API_DECODE_ERROR`; a Redis connection failure returns `500 REDIS_CONNECTION_ERROR`. Infrastructure details (internal hosts, Redis errors) are logged at the gateway, not returned to the caller.

4. **Bind the delegation to the caller.** `Since 1.6` The plugin compares `delegated_to` with `x-user-id`. A delegation granted to someone else is refused with **`403 DELEGATION_FORBIDDEN`** ("Delegation does not belong to the current user"); a delegation with no `delegated_to` is treated as incomplete (`500 DELEGATION_DATA_ERROR`). See Delegatee binding.

5. **Validate the time window.** `from_date` and `to_date` are parsed from ISO 8601 date-time strings (with or without milliseconds, ending in `Z` or a `±HH:MM` offset, which is applied) into UTC timestamps; an impossible calendar date is rejected rather than rolled over. If now is before `from_date`, or after a present `to_date`, the plugin returns **`403 DELEGATION_EXPIRED`**. A null or absent `to_date` means no upper bound. Unparseable dates return `500 DELEGATION_DATE_PARSE_ERROR`.

6. **Swap identity.** On success the plugin rewrites the upstream headers so downstream services see the request as coming from the delegator:

Header	Value
`x-user-id`	the **delegator**'s id
`x-delegatee-id`	the original caller (the delegate)
`x-is-delegation`	`"true"`

Delegatee binding `Since 1.6`

The `X-DELEGATION-ID` header is supplied by the client, and the gateway fetches the delegation with an internal service call. The plugin now requires the caller to be the delegation's `delegated_to`, and answers `403 DELEGATION_FORBIDDEN` otherwise.

The check needs both halves: the data service returns `delegated_to` on `GET /v1/delegations/{id}` (also delivered to the 1.4, 1.6 and 1.8 lines as an update), and the gateway caches under the new `v2` key, so details cached before the change, which lack `delegated_to`, are never read back and rejected. Upgrade the data service before the gateway.

`DELEGATION_EXPIRED` does not disclose the delegation. The `403` body carries only `details.error = "Delegation period is not current."`, not the delegator, the period or the gateway's clock. The refusal is logged at info level with the delegation id, the caller and the trace id, which is how an operator finds the delegator and the period. The web client reads only the code, and clears the active delegation.

Access checks under delegation

The downstream `osos-access` plugin (priority 997) reads `X-DELEGATION-ID` and, when present, fetches a delegation-scoped user profile from the data service, cached in Redis under `shared:user-profile-<userId>_<delegationId>`.

With a delegation id present, view access is decided directly by whether the delegator holds the view's access permission, rather than by the normal ownership-based check. A fixed set of view ids bypasses the delegation path and always uses the normal check:

```
SERVICE_CATALOG, ONGOING_INBOX, INBOX_REQUESTS,
ONGOING_MY_REQUESTS, COMPLETED_MY_REQUESTS,
ONGOING_OBSERVER, COMPLETED_OBSERVER
```

Scope enforcement in the data service

The data service resolves the delegate's effective permissions in `getAllDelegateePermissions(userId, delegationId)`; this is where `delegation_scope` takes effect:

1. Load the active delegation (same active rules as above). If not found, the delegate gets empty permissions.
2. Read `delegation_scope.base` and collect `delegation_scope.exception` as the exception set.
3. Start from the delegator's own full permissions and attach a `delegation` block (`{ delegationId, scope, exceptions }`) to the resolved permissions.
4. **Administration**: all system permissions are inherited when `allow_administration` contains `ALLOWED`. Otherwise `system` is emptied, except `Since 1.10` for an `ALLOW_ALL` delegation with no exceptions, which keeps the System grants below Admin and drops only System Admin. Module and entity permissions are always kept.
5. **Views, pages and reports** are filtered against the exceptions:
   - `DENY_ALL` with no exceptions: no views, pages or reports.
   - `ALLOW_ALL` with no exceptions: all of the delegator's views, pages and reports.
   - With exceptions, each item is matched against the exception patterns (by item id or by its parent module, plus the general patterns `so:all`, `so:app`, `so:app:module`). In `ALLOW_ALL` mode a match **excludes** the item; in `DENY_ALL` mode a match **includes** it.

Request-time list filtering

For the request-style inbox, observer and service-catalog list endpoints, the delegation request filter narrows the query when a delegation is active. It matches specific view URLs and injects a `where` filter on a per-view field:

View	Action type	Filter field
`SERVICE_CATALOG`	Submit	`value.relatedEntityId`
Ongoing/Completed My Requests	MyRequest	`value.reference.dataEntity`
Ongoing/Completed Inbox	Inbox	`value.reference.dataEntity`
Ongoing/Completed Observer	Observe	`value.reference.dataEntity`

The operator depends on scope: `ALLOW_ALL` uses `nin` (exclude the exception services); `DENY_ALL` uses `inq` (include only the exception services). The general exceptions `so:all` and `so:service` short-circuit this: under `ALLOW_ALL` they mean "no filter", under `DENY_ALL` they mean "exclude all". The filter applies to `/entries` and `/count`.

Forwarding to the delegate `Since 1.10`

With `forward_notifications` and/or `forward_emails` on, the delegate receives a copy of what is addressed to the delegator while the delegation is active, and only for work the delegation covers. Forwarded sources are the **Send Notification** and **Send Email** reaction actions and, for email, scheduled report deliveries. Each message is attributed to the service, module, report or view it concerns, and the scope is applied to that attribution (Allow All: sent unless

excluded; Deny All: sent only when an exception grants that service, module or report, or one of the entity's views; a message tied to one specific view needs that view granted). Skipped: a missing or deactivated delegate, the delegator themselves, the same delegator–delegate pair counted twice, and, for notifications, a delegate who is already a direct recipient. A copy is never forwarded again. If the delegates cannot be worked out, forwarding is skipped and logged, and the original delivery always goes ahead. Notification-side detail: Notification configuration [Administration › Notifications & email templates].

Endpoints

Both public endpoints require an authenticated user (OAuth2 introspection) and live in the data service's delegations controller.

`GET /v1/delegations/{id}`

Returns a single delegation projected to `MyDelegationsDto`, used by the gateway's `user-delegation` plugin:

Field	Type	Notes
`id`	string	Delegation entry id.
`delegator`	reference	The delegating user.
`from_date`	date-time	Start of the window.
`to_date`	date-time | null	End of the window (nullable).
`delegated_to`	reference	The delegate. `Since 1.4`, needed for delegatee binding.

The projection omits `is_active` and `delegation_scope`.

`GET /v1/delegations/my`

Returns the current user's **active** delegations (where they are the delegate), as an array of `MyDelegationsDto`. Only delegations that pass the full active check are returned, de-duplicated per delegator.

Configuring the gateway plugin

The `user-delegation` plugin is configured globally in `config/kong.yml`:

Config	Type	Required	Default	Purpose
`data_service_url`	string	yes	—	Base URL of the data service used to fetch delegation details.
`shared_key`	string	yes	—	Internal shared key sent as `X-SHARED-KEY`.
`cache_ttl`	number	no	`600`	Redis TTL (seconds) for cached delegation details.
`redis.url`	string	no	`redis://localhost:6379`	Redis connection URL.
`redis.ssl`	boolean	no	`false`	Enable TLS to Redis.
`redis.ssl_verify`	boolean	no	`false`	Verify the Redis server certificate.
`redis.useSentinel`	boolean	no	`false`	Use Redis Sentinel instead of a standalone server.
`redis.sentinel.sentinels[]`	array of `{host, port}`	—	—	Sentinel endpoints.

Config	Type	Required	Default	Purpose
`redis.sentinel.username` / `password` / `masterName`	string	—	—	Sentinel auth and master name.
`envPrefix`	string	no	—	Prefix prepended to Redis cache keys (per-environment namespacing).

Action log and audit

Every action performed under a delegation is completed **in the name of the delegator**, while OSOS records that the real actor is the **delegate**, so the audit trail can always tell them apart. Delegation identifiers are attached to every workflow and comment request; the delegate's identifier (`x-delegatee-id` , with `x-is-delegation: true`) is passed to the services for auditing. See Monitoring, logs & audit [Administration › Monitoring: logs & audit] and, for the record level, Timeline [Build Apps › Timeline & History].

Button-click history is attributed to the delegate `Since 1.8` . Before this fix, clicking a system button while acting under a delegation stamped the resulting `ButtonClickedHistory` record with an empty actor: `delegatee` was dropped because the controller handling the click never passed it to the history-creation call, although the gateway had already resolved and forwarded it. Value changes and owner reassignments made under a delegation were never affected.

Operational notes and edge cases

- **Caching.** The gateway caches delegation details for `cache_ttl` (600 s by default) and refetches only on a miss. When a delegation entry is **edited or archived**, the data service deletes the gateway's cached copy, so the change applies at the next request. `Since 1.8` it deletes both key generations, the current `osos:delegation:v2:` key and the older `osos:delegation:` key, so a cell running gateways of both generations during a rolling upgrade is covered; before, an edited or archived delegation could stay usable at a newer gateway for up to `cache_ttl` .
- **Delegated permissions follow permission changes** `Since 1.4` . A delegate's cached delegated permissions (kept up to 300 s) are cleared together with the access-right, view and report changes they embed, so a permission revoked from the delegator no longer lingers for the delegate.
- **Split enforcement.** The gateway validates the delegate binding and the time window and performs the identity swap; `is_active` , `state` and `delegation_scope` are enforced by the data service.
- **Open-ended delegations.** A null `to_date` never expires on its own; end it by setting `is_active` to `INACTIVE` , archiving the entry, or setting a `to_date` .
- **Identity downstream.** Under an active delegation, downstream services see `x-user-id` = the delegator, with `x-delegatee-id` and `x-is-delegation: true` carrying the real caller and the flag.
- **Guest (anonymous) users** have no delegations.
- **API tokens** are a separate mechanism for external systems, not people. See API tokens.

History and timeline API reference

This page is for **integrators and developers** who read a record's history or a timeline feed through the API, pin or publish posts from another system, or need to know exactly how history is recorded and filtered.

> **Setting this up in the designer?**
>
> Builders choosing what a view shows of a record's history, or adding a timeline to an entity, should start with Timeline & History [Build Apps] — it covers the same features in the labels the designers show.

OSOS exposes two distinct but related concepts that both answer "what happened to this record, and when":

1. **Entry History** — the per-record audit/activity feed for a single data entry (field changes, workflow actions, owner reassignments, SLA breaches, trigger runs, button clicks, create/archive). Served by the **data-service**.
2. **Timeline** — a curated public/internal *feed of posts* generated from data entries via a per-entity "timeline contribution" mapping. Posts can be pinned and published/unpublished (manually, on a schedule, or by workflow) and carry a social layer (reactions and comments). Served by the **timeline-service**.

This page documents both from a builder/developer perspective: the HTTP endpoints, the shape of what you get back, how records are recorded, and the constraints that govern visibility.

> **Scope note:** this page focuses on the read APIs, the record model, and the recording pipeline. The workflow engine that *produces* `_metaWorkflowHistory` and the trigger engine that *produces* `TriggerHistory` documents live outside these two services; only their contribution to the history feed is described here.

Part 1 — Entry History (data-service)

Concept

Every data entry accumulates a history feed. Unlike a single collection, the feed returned by the history endpoint is **assembled at read time** from several sources, each mapped into a common `DataEntryHistory` shape:

Source	Origin	History type(s) produced
Field-change records	`data_entry_history` collection (persisted on create/update)	`Update`
Owner reassignment records	`data_entry_history` collection	`ReassignOwner`
Button-click records	`data_entry_history` collection	`ButtonClickedHistory`
Workflow step actions	entry's `value._metaWorkflowHistory` (computed)	`WorkflowAction`
Create / archive	entry's `stateHistory` (computed)	`Create`, `Archive`
SLA overdue events	SLA service	`SlaOverdue`
Trigger runs	`trigger_history` collection (computed)	`TriggerHistory`

Persisted history records live in a dedicated MongoDB collection via `DataEntryHistoryRepository`; the workflow, state, and SLA slices are derived on the fly from the entry document and other services and merged in by the controller.

The `DataEntryHistory` record

Source: `models/data-entry/data-entry-history.model.ts`.

Field	Type	Notes
`id`	string	generated
`specificationVersion`	number	currently `1`
`relatedTo`	`ReferenceProperty`	`{dataEntity, dataEntry}` the record belongs to
`historyType`	`HistoryTypeEnum` (number)	see enum below
`actionInfo`	`ActionInfo`	who/when — `{doneAt, doneBy, delegatee?}`
`additionalData`	polymorphic	shape depends on `historyType` (see below)
`isValueResolved`	boolean	true once references were resolved & denormalized
`isInvalid`	boolean | null	entry was in an invalid state at the time
`invalidReason`	`DataEntryInvalidReasonEnum` | null	why it was invalid

`ActionInfo` (`action-info.model.ts`): `doneAt` (ISO string, required), `doneBy` (`ReferenceProperty`, required), `delegatee` (nullable reference — present when the action was performed on behalf of another user).

`HistoryTypeEnum`

Source: `packages/osos-shared/src/models/enums.ts`. **These are numeric enum values** — the API returns the number, not the name:

Value	Name	Meaning
0	`Update`	one or more field values changed
1	`WorkflowAction`	a workflow step was executed
2	`StateChange`	(defined; not emitted by the history assembler)
3	`ReassignOwner`	a workflow step owner was changed
4	`ButtonClickedHistory`	a system button was clicked
5	`TriggerHistory`	a trigger/automation ran
6	`Create`	the entry became Active
7	`Archive`	the entry was archived
8	`SlaOverdue`	an SLA due date was breached
9	`PermanentDelete`	(defined)
10	`Restore`	(defined)

> The `state` slice only ever emits `Create` and `Archive` today (see `getDataEntryStateHistory` — it maps `newState === Active` → `Create`, otherwise → `Archive`). `PermanentDelete` / `Restore` are defined in the enum but not produced by the current assembler.

`additionalData` shapes by type

- **`Update`** → `UpdateAdditionalData[]`. Each element is a JSON-diff operation:

  ```
  {
    "op": "replace",          // just-diff Operation: add | replace | remove
    "path": ["field", "sub"], // path segments into the entry value
    "value": { "old": <any>, "new": <any> }
  }
  ```

  On first write a field typically has only `new` (no `old`).

- **`ReassignOwner`** → `ReassignOwnerChanges`: `{workflowId, stepId, oldOwner, newOwner, originalOwner?, note?, permanentChange?}`.
- **`ButtonClickedHistory`** → `{buttonId, buttonName}`.
- **`TriggerHistory`** → `TriggerHistoryDto` (see Trigger history).
- **`SlaOverdue`** → `{stepId, dueDate, receivedDate, status, stepName}`.
- **`WorkflowAction`** → the stored `StepHistory` spread plus `stepName` and a `workflowContext` object `{workflowId, instanceId, parentInstanceId, isSubWorkflow}`.

Multilingual field changes in the history feed `Since 1.8`

When a field is multilingual, an edit that touches only one of its languages now appears in the entry's history as a per-language line — `label: old → new` — for each language that actually changed.

Previously both sides of the diff were resolved to whichever language the UI was set to before being displayed. Editing only the Arabic value and then reading the history with the interface in English collapsed both sides to the same, unchanged English text, so the change looked empty even though the edit had been recorded. The stored history record always carried the full per-language before/after values (see the `Update` shape above) — only the rendering was wrong.

What the history card shows now:

- **Only the languages that actually changed** are listed. A language whose value is identical on both sides is skipped, and blank counts as the same as missing.
- **Languages follow the space's configured language order**, with any language outside that list appended at the end.
- **The language's label is shown in the current UI language, the values are not** — each value is displayed in its own language, untranslated. So the same edit reads identically whatever language you view the history in.
- **A language added during the edit renders as a clean addition** — the new value only, with no arrow and no "old" side — and a language cleared renders as a removal. This follows each language's own before/after state rather than the field's overall change type, so adding one language while editing another is rendered correctly on both lines.

How history is recorded

History is written **reactively, off the service bus** — controllers do not write history inline. Handlers under `handlers/events/data-entries/history/` subscribe to domain events:

Event	Handler	Action
`DataEntryCreatedEvent`	`AddHistoryOnDataEntryCreatedHandler`	`createHistoryForValueChanges(...)` using `creationInfo.doneAt`
`DataEntryUpdatedEvent`	`AddHistoryOnDataEntryUpdatedHandler`	`createHistoryForValueChanges(...)` using `updateInfo.doneAt`
`WorkflowStepOwnerChangedEvent`	`AddHistoryOnWorkflowStepOwnerChangedEventHandler`	`createHistoryForOwnerChanges(...)`
`DataEntryDeletedEvent`	`DeleteHistoryOnDataEntryDeletedHandler`	`deleteHistoryForDataEntry(entryId)`
(entries permanently deleted)	`delete-history-on-data-entries-permanently-deleted.handler.ts`	bulk `deleteHistoryForDataEntry`

Button clicks are written from `system-button.controller.ts` via `createHistoryForButtonClicked(...)` (using `Utils.dateTime()` as `doneAt`).

> **Correction — Since 1.8.** A button clicked during an active delegation [Administration › Delegation] is now correctly attributed to the delegate who actually clicked it, not the delegator. The controller already resolved the real actor for the bus event it publishes, but was not passing that on to the history write, so every `ButtonClickedHistory` record stamped `actionInfo.delegatee` empty — the history showed only the delegating user, with no way to tell who had actually clicked the button. Button-click history now stamps `delegatee` the same way value-change and owner-reassignment history already do.

Key recording behaviors (`data-entry-history.service.ts`):

- **Update / Create records are skipped when the diff is empty** — both the handler and `createHistoryForValueChanges` early-return if `valueDiff` has no length, so a no-op save produces no history row.
- **The `delegatee` is threaded through** from the message's `stickyAttributes.delegateeId`, so history captures "X acting for Y".
- **`isInvalid` / `invalidReason`** are copied from the entry onto the Update record, so the feed reflects whether the entry was valid at that revision.
- **References are resolved at write time.** `createHistoryForValueChanges`, `...ForOwnerChanges`, and `...ForButtonClicked` all call `resolveHistoryValues(...)` and persist with `isValueResolved: true`, so reference labels (user names, collection item titles) are denormalized into the stored record and don't need re-resolution on read.
- **Duplicate protection.** Inserts that hit a MongoDB duplicate-key error (`code === 11000`) are logged and swallowed rather than thrown, so a re-delivered bus message doesn't crash the handler.

`resolveHistoryValues` walks `actionInfo.doneBy` / `delegatee` and the `additionalData` values, collecting `ReferenceProperty` targets and UUID collection-item ids, then batch-loads them (`DataEntityIndexRepository` for references, `DataCollectionsRepository` + the entity's `internalCollections` for collection items) and rewrites each into a `ReferencePropertyWithResolved` carrying a `_resolved` label. Already-resolved values are left untouched.

> **Recording is asynchronous.** Entry history is produced by bus handlers reacting to `DataEntry*Event`s, not synchronously in the write request — expect eventual consistency between the write response and the history read.

Reading entry history (v1)

`GET /v1/entities/{entity}/entries/{id}/history`

Returns the **complete** assembled feed for one entry (no pagination). The controller (`findEntryHistoryById`) merges, in order:

1. Persisted history (`getEntryHistory(id)`).
2. Workflow history (`getEntryWorkflowHistory`).
3. Trigger history — **only** when the caller is a system viewer with no view, or the active view has `entriesConfiguration.showTriggerHistory.enabled`.
4. State (create/archive) history — only when allowed to see state history.
5. SLA overdue history.

Then it **sorts ascending by `actionInfo.doneAt`**, resolves any unresolved records, and masks sensitive fields before returning.

Reading entry history (v2, paginated)

`GET /v2/entities/{entity}/entries/{id}/history?lastTimeStamp=<ISO>`

Long-standing (`findEntryHistoryByIdV2`). The v2 endpoint adds keyset pagination and is the one to build against for large feeds:

- **Page size is fixed at** `limit = 12` (server constant; not client-settable).
- `lastTimeStamp` is the cursor: pass the `actionInfo.doneAt` of the last item you received to get the next page. Every slice filters `doneAt < lastTimeStamp`.
- Results are **sorted descending** by `doneAt` (newest first).
- **Timestamp-tie handling:** after slicing to `limit`, the loop *extends* the limit while the record just past the boundary shares the exact `doneAt` of the last kept record — so records with identical timestamps are never split across a page boundary.
- **Preview mode:** when the request is in preview mode the endpoint returns fake history (real entity/workflow ids, fake content) via the fake generator.

Both v1 and v2 apply the same view-based filtering, resolution, and sensitive masking described below. Prefer v2 (`lastTimeStamp` cursor, fixed page size 12, `doneAt DESC`) for large entry histories.

Visibility and view filtering

The view designer's history switches map to these keys: **Entry history** = `entriesConfiguration.history.enabled`, **Changes** = `showChanges`, **Changer** = `showChanger`, **Reaction history** = `showTriggerHistory.enabled`, **Reaction history details** = `showTriggerHistory.showTriggerDetails`, **Workflow history** = `showWorkflowHistory.enabled`, **Display step owner** = `showStepOwners` (**All** = `AllOwners`, **Owned by User** = owner-scoped).

When the request carries a data **view** (end-user context), history is filtered by `clearEntryHistoryAgainstView(...)` and related helpers:

- **History off entirely.** If `view.entriesConfiguration.history` is missing or `history.enabled === false`, the endpoint returns `[]`.
- **Field-level filtering.** `Update` records are reduced to only the fields the view exposes (`entriesConfiguration.fields`) *and* that are not hidden. Hidden fields are evaluated per record, honoring conditional `hidden.expression` (evaluated via `ExpressionService`). `_metaOrder.value` is always allowed. An Update record whose diffs are all filtered out is dropped.
- **`showChanges` / `showChanger`.** With `history.enabled` true but `showChanges === false`, Update records keep the event but drop `additionalData` (you see *that* something changed, not *what*). With `showChanger === false`, `actionInfo` is reduced to just `doneAt` (the actor is hidden).
- **Workflow history gating.** `showWorkflowHistory.enabled` controls whether `ReassignOwner` / `WorkflowAction` records appear at all. Owner visibility is further gated by `showStepOwners`:
  - `AllOwners` → everyone's steps visible.
  - Otherwise a record is only shown/attributed if the current user (or one of their groups) is among the step's `executedBy` / `owner` / `originalOwner` (for `WorkflowAction`) or `newOwner` / `oldOwner` (for `ReassignOwner`). When not allowed, `executedBy`, `owner`, `originalOwner`, and `actionInfo.doneBy` are stripped from the record rather than the record being removed.
- **State history gating.** Create/archive history is included only when there is no view, or the view allows archived entries, or "view as admin" is enabled.

- **Trigger history gating.** As above — requires a system viewer (no view) or `showTriggerHistory.enabled`; `showTriggerDetails` controls whether the trigger input/output/error payload is included.

Sensitive-field masking

Since 1.8 — `maskSensitiveHistoryFields(...)` (`AUDIT_SENSITIVE_MASK`, `buildSensitiveFieldScope`).

This runs on **every** history read (v1 and v2), after resolution, as the last step before returning. Behavior:

- Builds the set of `fullNormalizedName`s for fields with `options.sensitive === true` on the entity. If none, it's a no-op fast path.
- For each `Update` record's diff whose `path` matches a sensitive field, the `old` and `new` values are replaced with `AUDIT_SENSITIVE_MASK` — **unless** the requesting user holds a matching sensitive-field grant (`SensitiveFieldsService.canAccessSensitiveField` against the field's module/entity/path scope).
- Masking is applied per-property: a first-write diff with only `new` does not gain a spurious masked `old`.
- Non-`Update` records are returned unchanged.

The code treats this as **unconditional and permanent for the audit trail**: once a value is written to history it is shown masked to anyone without the grant, so historical sensitive values cannot leak through the history API.

Trigger history

Source: `trigger-history.service.ts`, model `trigger-history.model.ts`. Trigger history is **Long-standing** (`TriggerHistoryService`).

Trigger runs are persisted as `TriggerHistory` documents (`trigger_history` collection) with `{triggerId, spaceId, eventName, relatedTo, triggerName, timestamp, doneBy, delegatee?, result: ActionResult[]}`.

`getTriggersHistory(entityId, entryId, showDetails?, limit?, lastTimeStamp?)` queries with an OR condition so a run appears on an entry when **either**:

- `relatedTo.dataEntry === entryId` (the entry that triggered it), **or**
- `result.effectedEntities.entry.dataEntry === entryId` (an entry the trigger *affected*).

Each stored run may contain multiple `result` action results; the service `flattenDeep`s them into one `DataEntryHistory` per action result with `historyType = TriggerHistory` and an `additionalData` of `TriggerHistoryDto`:

`TriggerHistoryDto` field	Notes
`triggerId`, `triggerName`	which trigger
`eventType`	the event that fired it (`eventName`)
`actionType`	the action kind that ran
`success`	per-action success flag
`details.actionName`	always present
`details.{eventName, entity, entry, startTimeStamp, input, output, error, doneBy}`	**only when `showDetails` is true**

`showDetails` is driven by the view's `showTriggerHistory.showTriggerDetails` (system viewers with no view get details = true). `limit` + `lastTimeStamp` mirror the v2 pagination contract (`timestamp DESC`, `timestamp < lastTimeStamp`).

Derived slices: state, workflow, SLA

These are **computed at read time from the entry document / other services**, not stored in the history collection:

- **State** (`getDataEntryStateHistory`) — maps `entry.stateHistory` entries with `newState > 0` into `Create` (Active) or `Archive` records, using `changedAt` / `changedBy` / `delegatee` .
- **Workflow** (`getEntryWorkflowHistory`) — iterates **all** of `entry.value._metaWorkflowHistory` (main workflow **and** every sub-workflow), resolving each `stepHistory` into a `WorkflowAction` record. Each carries a `workflowContext` `{workflowId, instanceId, parentInstanceId, isSubWorkflow}` . **Since 1.4** the assembler processes all workflow histories rather than only the active one, so sub-workflow steps appear in the feed. Sorting is by timestamp descending, with a tie-break that places automatic sub-workflow transitions (`StepType.SubWorkflowStep`) before user actions at the same instant.
- **SLA** (`getEntryOverdueSlaHistory`) — pulls overdue records from `SlaService` for the entry, attaches the step name from the active workflow, and emits `SlaOverdue` records attributed to the system user. **Long-standing**.

All three accept optional `limit` / `lastTimeStamp` for the v2 path. Deleting an entry deletes its persisted history (`deleteHistoryForDataEntry` on delete / permanent-delete events); these derived slices disappear with the underlying entry/records.

Part 2 — Timeline (timeline-service)

Concept

A **timeline** is an activity feed of **posts**. A post is itself stored as a data entry under a dedicated Posts entity (`Constants.POSTS_ENTITY_ID`), and is produced automatically from a source entry whenever that entry's entity has a **timeline contribution** configured. You therefore don't create posts by hand — you define the timeline settings on the entity, and the system keeps posts in sync as an **event-sourced read model**. The mapping (which field becomes the title, the description, the attributes, the category/icon, publish & pin config) is the "timeline contribution options" for the entity.

Recording is event-driven, symmetric with entry history:

Source event	Timeline handler	Effect
`DataEntryCreatedEvent`	`PostDataEntryCreatedHandler` → `TimelineService.submitPost`	build & create a Post
`DataEntryUpdatedEvent`	update handler → `TimelineService.updatePost`	rebuild the Post from the entry
entry archived/deleted	archive/delete handlers → `archivePost` / `deletePost`	remove/retire the Post
workflow step reached	`trigger-workflow-step-reached-event.handler.ts`	drive workflow-based publish state

`submitPost` / `updatePost` build the Post via `buildPost(...)` and dispatch a `CreateDataEntryCommand` / `UpdateDataEntryCommand` on the bus (with `skipValidation = true`), so posts flow through the same data-service write path as any entry — which is also why **a post gets its own entry history** (Part 1) *and* participates in the timeline feed. Do not confuse the entry-history API (data-service) with the timeline post API (timeline-service).

Activating a timeline (designer)

A timeline is attached to an existing entity. Open the entity, choose **Edit Entity** (the arrow next to **Add Entry**), and pick **Timeline** from the top bar (Details – Interactions – Structure – Workflow – Timeline). Add the timeline from the middle of the screen; it is seeded with a default name and icon taken from the entity, both of which you can change.

Once you save the timeline settings and enable them on the entity, the system begins syncing posts with the entity's entries automatically: posts are created for new entries, updated when an entry changes, and archived or deleted when the entry is archived or deleted. Disabling the timeline on an entity removes its associated settings.

Post templates

Each mapped field is bound to its value source through one of the following `MappingTypeEnum` mapping types: **Fixed** (one literal value for all posts), **Variable / entry-mapped** (`EntryMapping` — takes the value from a path in the source entry value), or **Hidden** (`Hidden` — does not appear in the post). Mandatory fields accept only Fixed or Variable — there is no Hidden option, because the user is required to supply a value.

In the timeline builder you choose the template under **Post Configuration**: open **Post Template** and select **General Post** or **Calender Post** (the product's spelling of the seeded category title). The set of fields exposed on the post changes with the selected template. The mapping **Type** options in the designer are **Field** (= `EntryMapping`), **Fixed** and **Hidden**.

Every tenant has two post categories — **General** and **Calendar (Events)** (see Seeded post categories for how they are created):

- **General Posts** expose **Title**, **Description**, and **Attachments**.
- **Calendar Posts** additionally expose **Start Date & Time**, **End Date & Time**, a **Cover** image, and a **Location** (a physical place or a link for a virtual event).

> **Note on templates:** currently only the **General Post** template is fully rendered in the web UI; the Calendar post category is defined in the model but does not yet have a dedicated display template.

Seeded post categories `Since 1.10`

The timeline's implicit **Posts** and **Post Categories** entities and the two default categories are now created per tenant by data-service's **migration chain** (1.10.0-a21-seed-timeline-post-categories) , which also runs for a tenant while it is being onboarded. Before 1.10 they came from a boot-time seed in timeline-service that called data-service over HTTP; an onboarding tenant refused those calls, so a tenant onboarded onto a multi-tenant cell could end up with **no post categories at all**.

What this means for integrators:

- The seed creates **only what is missing** (matched on the entities' normalized names and the categories' fixed ids), so re-running it on a seeded tenant writes nothing.
- The category ids are **fixed and identical in every tenant** — reference them freely, but do not create categories with those ids yourself.
- timeline-service no longer seeds anything at boot. A future change to these code-owned definitions ships as a data-service migration rather than as a restart.

Reading the feed

All timeline endpoints require authentication (`Auth2AuthenticationStrategyName`). Source: `timeline.controller.ts` .

Method & path	Purpose	Authorization voters
`GET /v1/posts`	list posts in the feed	`canShowPostVoter` , `systemViewerVoter`
`GET /v1/posts/{id}`	single post	`canShowPostVoter` , `systemViewerVoter`
`GET /v1/posts/count`	count posts	`canShowPostVoter` , `systemViewerVoter`

Method & path	Purpose	Authorization voters
`GET /v1/posts/entities/{entityId}`	posts for a specific source entity	`denyEndUserVoter`, `canViewPostEntityVoter`
`GET /v1/posts/entities/{entityId}/count`	count for a source entity	`denyEndUserVoter`, `canViewPostEntityVoter`
`PATCH /v1/posts/{id}/pin`	pin / unpin	`canPinPostVoter`, `canEditEntityVoter`
`PATCH /v1/posts/{id}/publish`	publish / unpublish / schedule	`canPublishPostVoter`, `canEditEntityVoter`

`GET /v1/posts` accepts a standard LoopBack `Filter<Post>` (query param `filter`). The controller merges in **audience filters** automatically via `TimelineService.additionalFilters()`:

- always `state === Active`, plus an OR over:
  - `value.anonymous.enabled === true` (public posts), or
  - authenticated + not restricted, or
  - authenticated + restricted **and** the current user (or one of their groups) is listed in `value.authenticated.entitledEntries`.

So the feed is inherently audience-scoped: an anonymous caller sees only anonymous posts; an authenticated caller additionally sees unrestricted and entitled-restricted posts. When a **view** is present, `EndUserService.postFilter` is merged in as well, and if the view's timeline display does not `allowPublishPost`, the `publishState.workflow.currentStep(Name)` fields are stripped from each returned post.

`GET /v1/posts/entities/{entityId}` is an admin/builder view (end users are denied): it filters by `value.reference.dataEntity === entityId`.

Preview mode: every feed/count/by-entity endpoint short-circuits to fake posts (`FakePostGeneratorService`) when `previewMode` is set, and the count endpoints always agree with the corresponding list endpoint by counting the same fake feed.

The `Post` model

Source: `post.model.ts`. A Post mirrors a data entry (`id`, `specificationVersion`, `dataEntity`, `value`, `creationInfo`, `updateInfo`, `state`). The interesting content is in `value` (`PostDataEntryValue`), notably:

- `title`, `description`, `attributes` — mapped from the source entry per the contribution's `MappingTypeEnum` (`Fixed` = literal value, `EntryMapping` = path into the source entry value, `Hidden` = null).
- `reference` `{dataEntity, dataEntry}` — back-pointer to the source entry.
- `module`, `timelineConfig` `{category, title, icon, publishConfig, pinnedConfig}`.
- `anonymous`, `authenticated` — audience config (drives `additionalFilters`).
- `pinned` (bool) + `pinnedState`, `published` (bool) + `publishState`.
- `publishInfo` / `unpublishInfo`, `pinnedInfo` / `unpinnedInfo` — `ActionInfo` stamps of the last publish/pin action.

Pinning

```
PATCH /v1/posts/{id}/pin
Body: { "pinned": true | false }
```

Constraints enforced by the controller:

- `pinned` is required (`missingRequired('pinned')` otherwise).

- If the contribution defines an automatic `pinnedConfig.pinnedDate`, you cannot manually pin (`UnablePinnedPostManuallyException`); likewise a configured `unpinnedDate` blocks manual unpin (`UnableUnPinnedManuallyPostException`).
- Setting `pinned` to the value it already has throws `DuplicatePinnedStatusPostException`.

On success it calls `TimelineService.handlePinnedPost(spaceId, entry, manual, {userId, isPinned})`, which recomputes `pinnedState`, sets `pinned`, stamps `pinnedInfo` / `unpinnedInfo`, and persists via an `UpdateDataEntryCommand`.

Publishing and scheduling

In the designer's **Publish** section, **Steps to publish at** sets the workflow publish steps (`publishStepIds`); leaving it empty keeps `publishConfig.manual` on. **Publish Date** / **Unpublish Date** map date fields of the source entry. The designer's hints state the combined rule: with manual publishing "the post will be published when you click publish and the mapped publish date is elapsed"; with workflow publishing "when post reaches publish step and the mapped publish date is elapsed" (`calculatePublishedState`). Automatic pin/unpin dates (`pinnedConfig`) have no control in the designer.

```
PATCH /v1/posts/{id}/publish
Body (one of):
  { "publish": true | false }                // manual publish/unpublish
  { "publishDate": "<ISO>" }                  // schedule publish
  { "unpublishDate": "<ISO>" }                // schedule unpublish
```

A post moves through several statuses: **draft** → **published** → **unpublished**. The system supports three ways to control publishing, tracked in `publishState` and resolved by `getPostPublishState`: **manual**, **scheduled** `{publishDate, unpublishDate, published}`, and **workflow** `{currentStep, currentStepName, published}` — for example, publishing a post automatically when a workflow reaches a manager-approval step.

Rules (from `publishPost` + `manualPublishValidations`):

- `publishDate` and `unpublishDate` are **mutually exclusive** in one request (`TimelinePublishDateConflictException`).
- If not scheduled, `publish` is required.
- Manual publish/unpublish requires `timelineConfig.publishConfig.manual` on the contribution, else `PublishNotAllowedException` / `UnpublishNotAllowedException`.
- You cannot schedule a publish/unpublish for a field that is already driven by a configured `publishDate` / `unpublishDate` mapping (`UnableScheduleManualPublishUnpublishPostException`).
- Scheduled dates must be **in the future**; if both are given, `publishDate < unpublishDate`.
- Re-publishing an already-published post (or re-unpublishing an unpublished one) is rejected.

Scheduling dispatches a `ScheduleJobCommand` (`JobName.ScheduleJob`) with id `publish-<postId>` / `unpublish-<postId>` for the future date; the effective `published` boolean is derived by `calculatePublishedState(...)`. Pin scheduling works analogously (`pinned-<postId>` / `unpinned-<postId>`), though the manual-pin branch of `getPostPinnedState` is currently commented out — pin scheduling is date-driven. Workflow-driven publishing is set from the workflow-step-reached handler, which maps a reached step into `handlePublishPost(..., workflow, {currentStep, currentStepName, isPublish})`.

Reactions and comments

Posts carry a social layer on top of the feed:

- **Reactions** — emoji reactions on a post, with a reaction count and the current user's own reaction; one reaction is stored per user. The web UI offers the full emoji palette, whereas the mobile app is currently limited to a single reaction (`like`).
- **Comments** — posts can be commented on, and a comment count is shown with the post. Comments support **@mentions** (a search box picks a person, and mentioned users are notified) and per-comment audience scoping. Comment attachment types are **configurable by an administrator per view** (with a default size limit of roughly 1MB) rather than a fixed list of file types.

> **Important distinction:** timeline reactions here are **social** (emoji) and are entirely different from "reactions" in the automation sense (event-driven triggers and actions).

Preview mode

When designing an entity or a view, you can inspect the timeline in preview mode, where the system shows a deterministic fabricated feed (sample posts with placeholder content) instead of real data, so you can preview the feed before publishing. This mirrors the preview behavior of the entry-history endpoints, which return fake-but-shaped history under the same flag.

Cross-cutting notes for developers

- **Numeric enums on the wire.** `historyType` and entry `state` are serialized as numbers. Map them client-side (`HistoryTypeEnum`, `DataEntityStateEnum`).
- **Two different "history" for a post.** A Post is a data entry, so it has its own `.../entries/{id}/history` feed *and* it participates in the timeline feed. Keep the two APIs distinct.
- **Recording is asynchronous.** Entry history and timeline posts are produced by bus handlers reacting to `DataEntry*Event`s, not synchronously in the write request — expect eventual consistency between the write response and the history/timeline read.
- **Deletion.** Deleting an entry deletes its persisted history; derived slices (workflow/state/SLA/trigger) disappear with the underlying entry/records.
- **Pagination contract.** Prefer v2 (`lastTimeStamp` cursor, fixed page size 12, `doneAt DESC`) for entry history; timeline feed endpoints use the standard LoopBack `Filter` / `Where`.

Notes and limitations

- Only the **General Post** template currently renders in full; the Calendar post category does not yet have a dedicated display template.
- The timeline is an **event-sourced read model**; if a bus event is missed, the feed may appear briefly stale until the next update.
- The timeline service depends on the core data-service; if it is unreachable, the feed may fail to load.
- On mobile, reactions are currently limited to a single `like`, while the full palette is available on web.

Notifications and delivery (API and internals)

This page is for integrators and developers who read a user's notifications from their own client, register devices for push, or need to explain why a notification did or did not reach someone. It covers the stored notification, the user-facing notification API, push registration and delivery, live delivery over the real-time connection, and the email and SMS paths.

The end-user view is Receiving Notifications [Using OSOS]. How an organization configures templates, triggers and channels is in Notification configuration [Administration › Notifications & email templates]. The generated endpoint reference is under notification-api.

Channels

Channel	Where the user sees it	Produced by
In-app	The portal's bell panel and **All Notifications** page; the mobile app's Notifications tab	A notification entry created for the user
Push	The user's registered **mobile** devices, even with the app closed	The in-app notification, mirrored to the user's device tokens through Firebase Cloud Messaging
Email	The user's mailbox	Email events: automation emails, password reset, API-token confirmation
SMS	The user's phone	SMS events from automations (delivery only; there is no SMS inbox)

A stored notification carries a `channels` list, but the only channel value defined is **In-app** (`NotificationChannelTypeEnum.InApp = 0`). Email and SMS are not stored on the notification record; they are separate events.

The web portal does not receive push. The frontend ships a generated client for the push-token endpoints but never registers a token, and has no browser push or service-worker messaging. Web users get notifications in the bell panel, live while the portal is open (see Live delivery). Push is a mobile-app channel only.

How notifications are produced

Notifications are event-driven. When something happens (an entry is created, a workflow step is reached, a password reset is requested, a file finishes generating), the notification service decides the channel(s), resolves recipients and content, and dispatches. Content is template-driven: templates contain tokens resolved at send time, so one template produces a personalized message per recipient. See Templates [Administration › Notifications & email templates].

Each in-app notification is stored as a **data entry**, so it is administered like any other entry. It carries:

- a **title** and **content**, both localized (for example English and Arabic);
- an **icon** (for example `file` for a finished export);
- an optional **attachment** (for example the generated file);
- a **reference** to the item it is about, plus the **module**, so the client can link straight to it;
- an optional in-app **link** `Since 1.10` (see Links);
- `readAt`, empty until the user opens it.

Scoping

Every notification belongs to one user and one space. The API returns only the caller's own notifications, only in the current space, and never archived ones. This is enforced on the server (`NotificationController.additionalFilters` filters by user id and current space and excludes archived entries), not just in the UI.

The user-facing API

All calls are made as the signed-in user, in the space of the request host.

List: `GET /v1/my`

Returns the caller's notification records in the current space. Standard filtering applies (paging, or only unread). The portal applies it for the user.

Count: `GET /v1/my/count`

```
{
  "count": 42,
  "unreadCount": 5
}
```

- `count` : notifications matching the current view.
- `unreadCount` : those whose `readAt` does not exist or is `null` .

This drives the bell's badge (shown as `99+` above 99).

`Since 1.10` A dedicated index backs both calls (user and space), where every call used to scan the full notification collection, and a case-sensitivity mismatch in the count query was fixed, so `unreadCount` agrees with what the list returns.

Mark as read: `PATCH /v1/my/{id}/mark-as-read`

Stamps `readAt` with the current time and flags the notification as read; it then stops counting toward `unreadCount` . Only the caller's own notification in the current space can be marked; a request matching nothing is rejected.

There is **no** "mark all as read" and **no** delete endpoint. Marking is one notification at a time, and notifications leave the list only when archived elsewhere. (The portal's translation files contain a "Mark All As Read" string, but no screen uses it.)

On mobile, tapping a notification marks it read and decrements the unread count **optimistically**, before the server confirms. If `mark-as-read` is rejected, the app rolls the change back so the counter stays accurate.

Links `Since 1.10`

A notification may carry a `link` to a page inside the application. Clicking it, in the bell panel or on the notifications page, marks it read and navigates there. Only in-application addresses are followed; anything else is ignored and the notification opens as before (the record or view it refers to). The marketplace publishing flow uses this to send submitters and reviewers to the right package page. Configuring it: In-app link on a notification [Administration › Notifications & email templates].

The **All Notifications** page always opens on its first page `Since 1.6` . (Earlier, returning to it could show the entries of the page you left while the page selector said page 1.)

Forwarded notifications `Since 1.10`

When a delegation has forwarding switched on, the delegatee receives a copy of the delegator's notifications (and emails) for the delegated scope while the delegation is active, never a duplicate of something already addressed to them. The copy carries the delegation, the delegator and the original target. Clients classify it as:

State	Meaning	Client behaviour
Current	The session already acts for that delegator	Opens the original target directly
Switchable	Opening needs a switch into the delegation	Confirms the switch, then opens the target
Unavailable	The delegation has ended, or another delegation is active	Rendered disabled

The mobile app marks a forwarded notification read only when the open goes ahead, and never downloads the delegator's attachment. Forwarded emails are a copy with the delegatee in CC (or BCC, if the delegator was BCC). Configuration: Delegatee copies [Administration › Notifications & email templates]. Mobile state handling: Delegation-forwarded notifications.

Live delivery

While a client is connected to the real-time service, a new notification is also pushed **live** over that connection, so the list and badge update without a reload. The web portal shows a toast with the notification's title and content (or a generic "New Notification" message) and increments the unread count.

`Since 1.10` The web portal ignores a live notification that belongs to a **different space** from the current one: no toast, no badge change. The live connection still carries all of the user's notifications; the portal compares each notification's space with the current space in the browser. A notification that does not name its space is shown rather than dropped. Before this, an automation whose execution covered several spaces could deliver the same notification into every open space session: one event, two identical pop-ups, and a badge counted twice. Reloading the list (`GET /v1/my`) was never affected, because the server scopes it to the current space.

Socket details: Real-time.

Push

Registering a device: `POST /v1/push-tokens`

```
POST /v1/push-tokens
Body: { "token": "<device push token>" }
```

Behaviour (`PushNotificationsTokensController.create`):

- A **new** token is stored and linked to the caller, with a `lastUsedDate` .
- A token that **already exists** but belongs to another user (a shared device) is reassigned to the caller and its `lastUsedDate` refreshed.
- The same token is never duplicated.

The mobile app obtains the device token from the push provider and registers it when the user signs in and grants notification permission. Revoking the permission on the device stops the alerts; the stored token simply stops receiving. The endpoint reference, including the `Since 1.10` delete and invalidate operations, is in PushNotificationsTokens.

Delivery

When an in-app notification is created, the notification service (`CreatedNotificationHandler`) mirrors it to the user's devices if:

1. the notification has the **In-app** channel;
2. the user has at least one registered push token; and
3. the notification has a usable title.

It sends the title and body to every registered token through **Firebase Cloud Messaging** (the `fcm-sender` service), which fans out to all devices at once (`sendEachForMulticast`) and plays the default alert sound on Apple devices. The push also carries data for deep-linking: the referenced item, module, icon, any attachment and the space's base URL.

Delivery is **best-effort**: a stale token or an FCM rejection is logged and counted, but never stops the notification from appearing in the in-app list.

`Since 1.10` Each device gets a push **once**, even if its token was registered more than once, and registrations are tidied: a token FCM reports as invalid is removed after the send, and a token not refreshed for **90 days** (`PUSH_TOKEN_STALE_AFTER_DAYS`) is removed by the maintenance job. Signing in on the device again registers it afresh. Operator view: Push delivery and device-token upkeep [Administration › Notifications & email templates].

Language of a push

Titles and bodies are stored per language, and the push sender picks the recipient's language, falling back to the first available language rather than sending nothing.

Known limitation. User-language resolution in the notification service is a placeholder: `getUserLanguage` always returns `en` (`services/notification-service/src/services/user.service.ts`). In practice pushes go out in English when an English version exists, otherwise in the first available language. The in-app list returns all stored languages, so the clients display the user's language.

Email

Email events are sent by the `email-sender` service over SMTP. Typical emails:

- **Automation (trigger) emails**: subject, HTML body, attachments, and a deep link to the item.
- **Password reset**: a secure reset link (`UserPasswordResetHandler`).
- **API token generated**: a confirmation to the associated address containing the token and a reminder to store it safely (`AppTokenGeneratedHandler`).

A link into the platform is built against the space's own host and is **shortened** before it is embedded. If shortening fails, the full link is used. Reset and API tokens are kept out of the email log; see Notification configuration [Administration › Notifications & email templates].

SMS

SMS events (typically from automations) are sent by the `sms-sender` service, which supports several regional carriers and gateways, for example EAND, EAND Bulk, SDG and Smart Dubai. Links are shortened first. Each send produces a delivery-status record (success or failure per recipient) tracked internally. Channel setup: SMS channel [Administration › Notifications & email templates].

File-ready notifications

For a file produced in the background (for example an XLSX or PDF export), the `FileGeneratedEventHandler` creates an in-app notification titled **"Your file is ready to download"** with the file attached, mirrored to push.

`Since 1.10` If a background list export fails part-way, no file is produced and **no** file-ready notification is sent, so a user is never handed a broken or half-written workbook.

Preferences

There is **no per-user notification preference or subscription API**. The notification API exposes listing, counting, marking as read and device-token registration only; nothing turns channels on or off, mutes categories or sets a digest frequency.

- In-app and push follow from whether a notification is created for the user and whether a device is registered with OS permission granted.
- Email and SMS follow from how the triggering automation is configured and the contact details on the user's profile.

Quick reference

User action	What the client does
See notifications	`GET /v1/my` (own, current space, not archived)
See unread count	`GET /v1/my/count` → `count` / `unreadCount`
Open or clear one	`PATCH /v1/my/{id}/mark-as-read` → stamps `readAt`
Get pop-ups on a phone	Mobile app registers the device via `POST /v1/push-tokens` after permission is granted
Stop pop-ups	Revoke notification permission on the device
Receive an export	File-ready notification with the file attached
Open a forwarded notification	Opens directly, or confirms a switch into the delegation first

Report model and API reference

This page is for **integrators and developers** who create, run or export reports through the API, or need to know exactly how a rendition is produced. It documents the report model, the validation rules and error codes, the endpoints, the export pipeline and scheduled delivery.

> **Building a report in the UI?**
>
> Builders should start with Reports [Build Apps] — it covers the same features in the labels the designer shows.

A **report** is a saved, tabular projection over a data entity. It binds a **data query** (the row source) to an ordered list of **fields** (the columns), plus presentation, scheduling, notification, and access options. Once defined, a report can be **executed** to return live rows, **counted**, and **exported** to PDF or XLSX renditions.

- **Web-only.** Reports are available in the web application only. A report link opens only on the web; on mobile it resolves to a `NotSupportedLink` and cannot be navigated to.

Concepts and data model

A report is stored as a `DataEntry` of the core **Report** entity. The concrete model is `ReportEntry`, whose `value` holds a `ReportEntryValue`:

```
ReportEntry (extends DataEntry)
  value: ReportEntryValue
```

`ReportEntry`'s constructor force-assigns `dataEntity = REPORT_ENTITY_ID`, so every report row is tagged to the Report core entity regardless of the payload.

ReportEntryValue fields

From `report-entry-value.model.ts`:

Field	Type	Required	Notes
`name`	`LocalizedValue`	yes	Report display name (per-language). Must be unique per related data entity — see validation.
`normalizedName`	`string`	no	Auto-generated programmatic identifier derived from `name`; unique per related `DataEntity`.
`description`	`NullableLocalizedValue`	no	Localized description; used in the PDF header text.
`icon`	`Icon`	(`require: true`)	Report icon.
`dataSource`	`DataEntrySource`	(`require: true`)	The row source. Only the **Query** type is supported (see below).
`fields`	`string[]`	(`require: true`, `minItems: 1`)	Ordered list of entity field paths to project as columns. Order is preserved.
`allowChangeFields`	`boolean`	no	If true, export callers may narrow the exported columns via `selectedColumns`.
`relatedTo`	`DataEntityReferenceProperty`	(`require: true`)	The entity the report runs against (`{dataEntity, dataEntry}`).

Field	Type	Required	Notes
`authenticated`	`AuthenticatedViewOption`	no	Access/authentication config. Defaults to `{isRestricted: false, enabled: true}`.
`isPublished`	`boolean`	no	Default `false`. Publishing gates the permission requirement and schedule validation.
`schedule`	`ReportScheduleModel`	no	Optional scheduled delivery timer.
`inAppNotification`	`SendReportNotificationActionOptions`	no	In-app notification delivery config.
`emailNotification`	`SendReportEmailActionOptions`	no	Email delivery config.
`displayOrder`	`number`	no	Default `0`. Ordering hint.
`design`	`ReportDesignOptions`	no	PDF layout / header / footer options.
`domain`	`ReferenceProperty`	no	Optional domain reference.
`sharing`	`ApplicationSharing`	yes	Application-sharing metadata.
`help`	`ReportHelp`	no	Contextual help text. **Since 1.2**

> **Note on `require` vs `required`.** Several properties in the model use the misspelled decorator key `require: true` (e.g. `icon`, `dataSource`, `fields`, `relatedTo`). LoopBack's property validation reads `required` (not `require`), so these are **not** enforced as required by the framework schema — the business rules in `ReportService` (below) are what actually enforce `dataSource` type, `fields` content, and `relatedTo`.

Data source (the row source)

`dataSource` is a `DataEntrySource`:

Field	Type	Notes
`type`	`DataCollectionTypeEnum`	Must be `Query` (enum value `2`). `Store` and `Collection` are rejected.
`store`	`string`	Store name (pattern-constrained).
`collection`	`UUID`	Collection id.
`query`	`UUID`	The data query id that produces the rows.

`DataCollectionTypeEnum` = `{ Store = 0, Collection = 1, Query = 2 }`. Report validation requires `type === Query`, so every report resolves its rows through a data query [Build Apps › Queries & filters].

The `report-reference-property.model.ts` constrains a report reference to always point at the `DataEntity` core entity (`dataEntity` is a JSON-schema `const` of `CoreEntities.DataEntity`) with a `dataEntry` UUID. `ReportEntryValue.relatedTo` is typed as the more general `DataEntityReferenceProperty`, but the same `{dataEntity, dataEntry}` shape applies.

Access and authentication

`authenticated` is an `AuthenticatedViewOption`:

Field	Type	Notes
`enabled`	`boolean`	Whether authenticated access is enabled (schema default `false`).

Field	Type	Notes
`defaultView`	`boolean`	Whether this is the default view (schema default `false`).
`isRestricted`	`boolean`	Whether the report has explicit permissions attached. **Calculated only** — it is not accepted from create/update payloads directly for this purpose; it drives the publish permission check.

Design options

The designer's **Report Design** choices map to `ReportDesignOptions`, whose defaults are `pageHeader = true`, `organizationHeader = true`, `pageFooter = true`, `organizationFooter = false`.

Permissions

Permissions are not stored on the report document; they live in access-rights (see Internals and invariants). Publishing a **restricted** report with no permissions selected is rejected.

Validation rules

All create/update paths run through `ReportService.validateReportValue` and `validateNameUniqueness`.

`validateReportValue(value)`

1. `value.dataSource.type` **must** be `DataCollectionTypeEnum.Query`, else `ReportDataSourceSupportedException` (error `REPORT_DATA_SOURCE_NOT_SUPPORTED`).
2. When `relatedTo.dataEntity === CoreEntities.DataEntity`:
   - The query (`dataSource.query`) must exist, else `DataQueryNotFound`.
   - The related entity (`relatedTo.dataEntry`) must exist, else `EntryNotExistException`.
   - The entity must not be archived, else `EntityArchivedException`.
   - The query's `dataEntity` must equal `relatedTo.dataEntry`, else `DataQueryNotBelongToEntity`.
   - **Field validation** (`validateReportFields`): `fields` must be non-empty (`ReportFieldsLengthException`, `REPORT_FIELDS_LENGTH`) and every entry must exist in the entity's `getAllFieldsPaths()` — unknown fields raise `ReportFieldsNotExistException` (`REPORT_FIELDS_NOT_EXIST`), listing the offending paths.
3. If the report `isPublished` and has a `schedule.enabled` one-time send whose `oneTime` timestamp is in the past, `ReportSendingTimeCanNotBeInThePastException` (`REPORT_SENDING_TIME`).

`validateNameUniqueness(report, selfId?)` — for each configured language, if `name[lang]` is set, the service looks for an existing report with the same `value.relatedTo.dataEntry` and the same `value.name.<lang>`. On PATCH, `selfId` excludes the current report (`id neq selfId`). A collision raises `ReportExistException` (`REPORT_EXISTS`). Uniqueness is therefore **scoped per related data entity, per language**.

Endpoints

All routes are on `ReportEntryController`, authenticated with the OSOS OAuth2 strategy and authorized per-route. `id` is the report entry id.

Method	Path	Purpose
`POST`	`/v1/reports`	Create a report.
`GET`	`/v1/reports`	List reports (LoopBack `filter`).

Method	Path	Purpose
`GET`	`/v1/reports/count`	Count reports (LoopBack `where`).
`GET`	`/v1/reports/{id}`	Get one report (joins permissions).
`PATCH`	`/v1/reports/{id}`	Update a report.
`PATCH`	`/v1/reports/{id}/permissions`	Update only the report's access/permissions.
`DELETE`	`/v1/reports/{id}`	Delete a report.
`GET`	`/v1/reports/{id}/entries`	Execute the report to rows (v1 filter).
`GET`	`/v2/reports/{id}/entries`	Execute the report to rows (criteria + search).
`GET`	`/v1/reports/{id}/entries/count`	Row count (v1 where).
`GET`	`/v2/reports/{id}/entries/count`	Row count (criteria + search).
`GET`	`/v2/reports/{id}/export/{format}`	Export rows to a file (`pdf` or `xlsx`).

Create (`POST /v1/reports`) — body is a `ReportEntryDto` without `id` (a `NewReportEntry`): the `ReportEntryValue` plus optional `permissions: Permission[]`. The flow runs `validateNameUniqueness` then `validateReportValue`, strips `permissions`, sends a `CreateDataEntryCommand` **on the service bus** (async, CQRS-style) for the Report entity, awaits `invokerService.registerToken(messageId)` to get the created result, then stores permissions separately in access-rights. The row is created by the command handler, not inline.

Read (`GET /v1/reports/{id}`) — returns the stored report plus a `permissions` array **joined at read time** from access-rights (`convertAccessRightToPermissions(Report, id)`); permissions are not stored on the report document.

Update (`PATCH /v1/reports/{id}`) — body is a partial `ReportEntryDto`. After validation, a **publish permission guard** applies: if `value.isPublished` **and** `authenticated.enabled` **and** `authenticated.isRestricted` **and** the supplied `permissions` are empty **and** no permissions already exist, it throws `ReportPermissionsNotSelectedException` (HTTP 422, `REPORT_PERMISSION_NOT_SELECTED`). The report is then persisted via `updateById`; access-rights are reconciled **before publishing** (order matters — dependency-sync re-fetches via `GET /v1/reports/{id}`): if `isRestricted === false`, stored permissions are cleared; otherwise a non-empty supplied set is stored. Finally a `DataEntryUpdatedEvent` is published, carrying a changed-field marker for `isPublished` when it changed.

Update permissions (`PATCH /v1/reports/{id}/permissions`) — authorized by `systemAdminVoterFactory()` (system-admin only). Body is an `UpdateResourcePermissionsDto` (`permissions` + `authenticated`). It applies the same publish guard, writes the new `authenticated` block, publishes a `DataEntryUpdatedEvent`, then reconciles access-rights.

Delete (`DELETE /v1/reports/{id}`) — sends a `DeleteDataEntryCommand` on the bus, awaits its token, then publishes a `ReportDeletedEvent`. Deletion is command-driven like creation.

Executing a report

v1 (`/v1/reports/{id}/entries`, `/v1/reports/{id}/entries/count`) accept a LoopBack `filter` / `where`. The controller resolves the related entity from `report.value.relatedTo.dataEntry`, then forces `state = Active`, `virtual != true` (unless `include_virtual=true`), and `dataEntity = <entity id>`, and calls `dataSourceService.find(entity, report.value.fields, filter, count?, report.value.dataSource)`.

v2 (`/v2/reports/{id}/entries`, `/v2/reports/{id}/entries/count`) accept a structured `filterCriteria` / `whereCriteria` (a `FilterExpression` tree) plus a free-text `search` param. The controller builds an `AND` expression that always pins `state = Active` (and `virtual != true` unless `include_virtual`), optionally AND-ed

with the caller's criteria, and delegates to `dataSourceService.findV2` / `countV2`. The entries endpoint passes `revealSensitiveFields = true`.

Common query params:

Param	Type	Meaning
`include_virtual`	boolean	Include rows flagged `virtual` (default excludes them).
`search`	string (v2)	Free-text search across the projected fields.
`filterCriteria` / `whereCriteria`	object (v2)	Structured filter expression.

Export renditions

Triggering an export from the UI

End users start an export from the report's actions menu in the web application — **Export to Excel** or **Export to PDF**. Both options call the export endpoint below.

Export endpoint

`GET /v2/reports/{id}/export/{format}` — `format` is an `ExportFormatEnum`: `pdf` or `xlsx` (there is no CSV export). Query params: `filterCriteria`, `search`, `include_virtual`, and `selectedColumns` (a string array). The response is a file download — content type and `Content-Disposition: attachment; filename=...` come from the exporter result. The declared content types are `application/vnd.openxmlformats-officedocument.spreadsheetml.sheet` (XLSX) and `application/pdf`. **Since 1.2** the export URL carries `{format}` as a path segment.

Format scope: reports never export CSV

The report route's `format` is typed `ExportFormatEnum` (`pdf` | `xlsx`), the declared content types are XLSX and PDF only, and the XLSX and PDF exporters are the only two registered. The CSV that 1.10 introduced belongs to the **quick export** of a gallery and its admin twin on the entity page — `GET /v3/entities/{entity}/views/{id}/entries/export` and `GET /v3/entities/{entity}/entries/export`, both `format=xlsx|csv` — documented in Data Views & Faces [Build Apps]. Nothing in that change reaches reports, which remain XLSX / PDF and web-only (the mobile app has no reports).

`ReportService.exportReportV2` loads the report, derives the file name from the localized report `name` in the caller's language, builds the same `state = Active` (+ optional `virtual != true`) `AND` filter as v2 execute (AND-ed with any `filterCriteria`), selects columns, fetches rows via `dataSourceService.findV2(..., revealSensitiveFields = true)`, and calls `ExportService.export`.

Column selection and value formatting

By default the exported columns are `report.value.fields`. If `selectedColumns` is provided **and non-empty and** `report.value.allowChangeFields` is true, the exported columns become `selectedColumns` intersected with the report's own `fields` (a report can never export a column it does not define). If `allowChangeFields` is false, `selectedColumns` is ignored. **Since 1.2** (`selectedColumns` + `allowChangeFields`).

`ExportService.export` resolves the concrete exporter through `ExporterFactoryService` (an unknown format yields `InvalidExportFormatException`). Before handing rows over, `convertDataEntryToExportableData` flattens each `DataEntry.value` into display-ready strings per field type:

Field kind	Export rendering
ADT string / date / datetime / time	Resolved collection value; dates `dd/MM/yyyy`, datetime `dd/MM/yyyy hh:mm`, time `hh:mm` (user time-zone or UTC).
Reference / User	`_resolved.mainTitle` (localized).
Localized text	Localized value for the user's language.
DateTime	`dd/MM/yyyy hh:mm`.
Date	`dd/MM/yyyy`.
Date / DateTime range	Two endpoints joined by `-`.
Time	`hh:mm`.
Month-Year	`MM/yyyy`.
Category	`_resolved.label` (localized).
Blob / file	File `name`.
Map	`JSON.stringify(geometry.coordinates)`.
Switch	`_resolved.mainTitle` (localized).
Password	Masked as `****`.
Other (icon, phone, ...)	Raw value.

Repeatable / multi-value fields render each element with the same rule and join with `, ` (ranges join their two dates with `-`). Report execution and export share this converter, so exported values match what the entries endpoint returns.

Sensitive fields. Execution and export request sensitive-field values (`revealSensitiveFields = true`), but each sensitive field is still masked (`****`) for anyone without sensitive-field [Build Apps › Protect a field] access to it, and password-typed fields are always masked. Masking applies identically to the exported file, the on-screen report, and the emailed attachment.

XLSX export

`ExportExcel` (format `xlsx`, ExcelJS) builds one worksheet named after the report/entity:

- Columns come from the entity's fields filtered to the report's field set (`getAllFields(true)`), each 20 units wide, header = localized field name.
- **Block fields** produce a merged sub-header row spanning their child columns. **Repeatable blocks** expand horizontally — each block instance repeats the child columns (`<field>.<index>`), sized to the maximum instance count across the exported rows.
- If a default **organization header** image exists, it is placed in a merged `A1:D1` banner and the data starts on row 2.
- Header cells are bold, centered, thin-bordered; sub-header cells bold with a bottom border.
- Returns `{ fileName: "<name>.xlsx", file: <ArrayBuffer>, mimeType: spreadsheetml.sheet }`.

XLSX has no column cap.

PDF export

`ExportPdf` (format `pdf`) uses the `@osos/fluentreports` engine:

- **Column cap.** At most `MAX_COLUMNS_COUNT` (default **7**) columns are rendered; `fields` beyond the 7th are dropped (`rptFields.slice(0, 7)`). This is a hard PDF-only limit.

- Column header labels use the localized field name (columns are internally keyed `<normalizedName>#<localizedName>`; block fields render `block.field`).
- **Layout** follows `design.layout` (`ReportLayoutEnum`: `Portrait = 0`, `Landscape = 1`) — landscape when `layout === Landscape`.
- **Header/footer sections** are driven by `ReportDesignOptions`: `organizationHeader` / `organizationFooter` pull the default organization's `organization_header` / `organization_footer` images; `pageHeader` prints the report name/description text (default `true`); `pageFooter` prints `Page {0} of {1}` / `{1} من {0} صفحة` page numbers (default `true`, and printed even when there is no `design`); `reportHeader` / `reportFooter` carry a `ReportHeaderFooterValue` whose `type` is `Text` (`0`) or `Image` (`1`).
- **RTL/Arabic** support uses right-aligned images and the Tajawal font (registered from `FONTS_PUBLIC_DIR`, default `public/fonts`).
- Section text is truncated to `SECTION_MAX_CHAR_COUNT` (default 171) across `SECTION_MAX_LINE_COUNT` (default 3) lines.
- On any render error, the exporter logs and throws `DataExportException`.
- Returns `{ file: <Buffer>, fileName: "<name>.pdf", mimeType: "application/pdf" }`.

The `PDFExporterOptions` handed to the exporter set `pdfDesign = report.design` and `pdfHeaderText = "<report name> \n <description>"`, plus an empty `pdfDesignConfigs` — so the exporter falls back to its built-in defaults.

Scheduling and notifications

A report may carry delivery configuration; these are stored on the report but executed by other subsystems (timer/notification services), not by the endpoints above.

- **`schedule`** (`ReportScheduleModel` extends `TimerOptionsEventConfig`): `enabled: boolean`, plus `oneTime?: string` and `repeat?` (a `DataEntityTriggerRepeat` with a cron/human-readable `expression` and optional `timezone` / `startDate` / `endDate`). A published report with an `enabled` one-time schedule in the past is rejected at save time (see validation).
- **`emailNotification`** (`SendReportEmailActionOptions`): `to` / `cc` / `bcc` recipient lists, a `template` (UUID or inline `EmailTemplateValue`), a `tokensMap`, optional `link`, `attachment` / `attachments`, and an optional `from` config.
- **`inAppNotification`** (`SendReportNotificationActionOptions`): `channels`, `recipients`, optional `module` / `notificationType`, a `template` (UUID or inline `NotificationTemplateValue`), and a `tokensMap`.

Scheduled delivery and the attachment — correction `Since 1.8`

When a `schedule`d report's timer job fires, `data-service`'s `ScheduledReportJobTriggeredHandler` loads the report and sends the configured email only when it is both `isPublished` and `schedule.enabled`. `emailNotification.attachment.enabled` and `emailNotification.link.enabled` are independent toggles — a common configuration is link-only, with the attachment turned off.

Before this fix, the handler unconditionally read the attachment upload's result even when `attachment.enabled` was `false`, so the read threw and the email was silently skipped — the surrounding try/catch rethrew into the message-bus consumer with no user-facing surface, and the bus retried and then gave up. Exporting/uploading the attachment is now **best-effort**: it runs inside its own try/catch, and a storage outage, a render failure, or a disabled attachment all degrade to sending the email **without** an attachment instead of dropping the notification. The failure is still logged as a structured warning (`{reportId, err}`) so an attachment failure stays diagnosable per report.

Error reference

Exception	Code	HTTP	Cause
`ReportDataSourceSupportedException`	`REPORT_DATA_SOURCE_NOT_SUPPORTED`	4xx (validation)	`dataSource.type` is not `Query`.
`ReportFieldsLengthException`	`REPORT_FIELDS_LENGTH`	4xx (validation)	Empty `fields`.
`ReportFieldsNotExistException`	`REPORT_FIELDS_NOT_EXIST`	4xx (validation)	One or more `fields` are not on the entity.
`ReportSendingTimeCanNotBeInThePastException`	`REPORT_SENDING_TIME`	4xx (validation)	Published report scheduled for a past one-time send.
`ReportExistException`	`REPORT_EXISTS`	4xx (validation)	Name collides with another report on the same entity/language.
`ReportPermissionsNotSelectedException`	`REPORT_PERMISSION_NOT_SELECTED`	422	Publishing a restricted report with no permissions selected.
`DataQueryNotFound`	—	4xx	`dataSource.query` does not exist.
`DataQueryNotBelongToEntity`	—	4xx	Query's entity ≠ `relatedTo.dataEntry`.
`EntryNotExistException`	—	4xx	`relatedTo.dataEntry` entity missing.
`EntityArchivedException`	—	4xx	Related entity is archived.
`InvalidExportFormatException`	—	4xx	Export `format` has no bound exporter.
`DataExportException`	—	5xx	XLSX/PDF render failure.

Internals and invariants

- **Only Query sources.** Report execution always resolves rows through a data query; store/collection sources are rejected. `ReportDataSourceSupportedException` is the single guard.
- **Fields are the column contract.** `fields` (order-significant) drive both execution projection and export columns. PDF silently truncates to 7 columns; XLSX has no such cap.
- **Permissions live in access-rights, not on the doc.** Read joins them in; write reconciles them separately and deliberately *before* publishing so the dependency-sync re-fetch sees the new set.
- **CQRS for create/delete.** Create and delete go through service-bus commands (`CreateDataEntryCommand` / `DeleteDataEntryCommand`) and await an invoker token; update writes the repository directly and emits a `DataEntryUpdatedEvent`.
- **Sensitive fields revealed on execute/export, then masked per access.** v2 execute and export pass `revealSensitiveFields = true`; each sensitive field is still masked (`****`) for anyone lacking access to it, and password-typed fields are always masked.
- **Legacy `virtual` handling.** Filters use `virtual != true` (not `virtual == false`) so pre-existing rows without the field are still returned.

Auto-increment sequences and counter widgets: internals and API

This page is for **integrators and platform engineers** who need to know exactly how sequence numbers are allocated and stored, what the server enforces when an entity with a sequence field is saved, and what the counters endpoint returns.

> **Setting up numbering or a counter in the UI?**
>
> Builders should start with Counters and record numbers [Build Apps].

Counter widgets vs. sequences

The two features are easy to confuse because they share a name, but they are unrelated in the code and serve different purposes:

	Counter widget	Auto-increment sequence
Purpose	Show a live *summary* number (how many entries match)	Assign a unique *running* number to each new entry
Built on	A view, optionally filtered by a saved query	The `Counters` / `CounterSequence` data model
Where it lives	The interface (top bar, dashboards, pages)	A data-entity field's `autoIncrementOptions`
Changes over time	Recomputed as data changes	Allocated once, on entry create

A counter widget counts the entries in a bound **data view**, optionally narrowed by a saved **query** (a `DataQuery`) — its counting criteria is a query reference, not the dashboard's separate, aggregation-capable `StatisticQuery` model used by Number/summary cards.

`Since 1.4` the end-user list of counter widgets reads the candidate widgets first and checks only their views, instead of loading every view up front, so it no longer slows down as a tenant accumulates views (also delivered to the 1.4, 1.6 and 1.8 lines as an update). See Dashboards, Summaries & Counter Widgets [Build Apps › Summaries, dashboards & counters] for the listing rules.

How auto-increment sequences work

The auto-increment feature is built on two pieces:

- An **auto-increment field option** on a data-entity field (`options.autoIncrementOptions`, model `FieldOptionsAutoIncrement`), which declares that the field is fed by a counter and names the counter to use.
- A **counter** (`Counters`) plus one or more **counter sequences** (`CounterSequence`) stored in the data-service database, which hold the configuration and the current running value(s).

The number produced is a plain integer. There is no prefix/suffix/padding mask stored on the counter itself — formatting (zero-padding, prefixes) is not part of the counter model.

End to end:

1. A data-entity field carries `options.autoIncrementOptions`. When `enabled` is `true`, the field is an auto-increment field.
2. When the entity is **created or updated**, `DataEntityService` reconciles a backing `Counters` document for each enabled field (creating, migrating, or updating it — see Counter lifecycle). The per-counter settings (`startAt`, `incrementBy`, `sharingLevel`) are **copied onto the counter and stripped from the field options**.
3. When a **data entry is created**, `DataEntryWriterService.handleAutoIncrementFields` finds every enabled auto-increment field and allocates the next value for each (in parallel), writing it into `entry.value.<fullNormalizedName>`.
4. Allocation goes through `SequenceService.getNextValue`, which resolves the counter, evaluates any conditional keys, then atomically bumps the matching `CounterSequence` via `CounterSequenceRepository.getNextValue`.

Auto-increment values are allocated **on create only** — there is no re-allocation on update or patch. (`handleAutoIncrementFields` is called from the create path and from batch create; it is not called from `updateDataEntry` or `patchDataEntry`.)

Failure behaviour

`handleAutoIncrementFields` wraps its work in a `try/catch` that **logs and swallows** any error. If counter allocation fails, the entry is still created — the field is simply left without a generated number. Do not rely on entry creation failing when a counter is misconfigured.

Inside the repository, if the atomic update throws, it is rethrown as a `GeneratingCounterSequenceException` (`GENERATING_COUNTER_SEQUENCE_ERROR`), but that still surfaces to the swallowing catch above.

Sequence data model

`Counters` (collection `_counters`)

One document per counter. The document `id` **is the counter key** — the same string referenced by a field's `autoIncrementOptions.key`. Stored in MongoDB collection `_counters`.

Field	Type	Default	Notes
`id`	string	—	Primary key; not generated. This is the counter *key*.
`lastValue`	number	—	**Deprecated**. The live value now lives on `CounterSequence`, not here.
`startAt`	number	`1`	First value handed out when a sequence has not been used yet.
`incrementBy`	number	`1`	Amount added on each subsequent allocation.
`sharingLevel`	number (`AutoIncrementSharingLevelEnum`)	`0` (None)	See Sharing levels.
`referencedBy`	`CountersReferencedBy[]`	—	Entity fields that point at this counter (config link).
`usedBy`	`CountersUsedBy[]`	—	Entity fields that have actually consumed a value. Gates config changes.
`conditionalIncrement`	boolean	`false`	When `true`, the counter runs multiple independent streams keyed by `keys`.
`keys`	`CountersKey[]`	`[]`	Key definitions used to pick/segment the sequence stream.

`CountersReferencedBy` (sub-model): `module`, `entity`, `field` — all UUIDs identifying the field that references this counter.

`CountersUsedBy` (sub-model): a single string `entityField`, set the first time the counter is actually used to produce a value for a field. Its format is `<entityId>_<field.normalizedName>` (e.g. `xyz-uuid_requestNumber`). Once set it is not changed. Presence of `usedBy` entries is what locks certain configuration changes (see lifecycle rules).

`CounterSequence` (collection `_counters_sequence`)

Holds the actual running value(s) for a counter. There can be **many** sequence documents per counter when `conditionalIncrement` is on — one per distinct combination of key values. Stored in MongoDB collection `_counters_sequence`.

Field	Type	Notes
`id`	string	Random UUID (v4), generated on first upsert.
`counterId`	string	The owning counter's `id`.
`lastValue`	number	The most recently issued value for this stream.
`[key: string]: any`	—	Dynamic key columns. When conditional, one property per `CountersKey.keyName`, holding that key's evaluated value, used to match the correct stream.

`CountersKey`

Defines one key that segments a conditional counter into independent streams.

Field	Type	Notes
`keyName`	string	Unique; validated against `NormalizedNameValidation` (min/max length + normalized-name pattern). Becomes a property name on `CounterSequence`.
`calculationMethod`	number (`TokenSourceType`)	How `value` is interpreted. Schema default `Expression`. **Only `Expression` is accepted.**
`value`	string	The expression to evaluate (for the `Expression` method).

`TokenSourceType` values: `Fixed = 0`, `Field = 1`, `Expression = 2`, `Pointer = 3`. **Counter keys must all be `Expression`.** Any key with a non-`Expression` `calculationMethod` causes `UnsupportedConditionalIncrementCalculationMethod` (`UNSUPPORTED_CONDITIONAL_INCREMENT_CALCULATION_METHOD`) to be thrown when the entity is saved.

`FieldOptionsAutoIncrement` (field option)

Lives on a data-entity field at `options.autoIncrementOptions`.

Field	Type	Default	Notes
`enabled`	boolean	— (required)	Marks the field as auto-increment. `FieldService.isAutoIncrementField` returns `!!options.autoIncrementOptions?.enabled`.
`key`	string	— (required)	The counter key (== `Counters.id`) to draw from.
`sharingLevel`	number	`0`	Copied to the counter, then removed from the field options at save time.
`startAt`	number	`1`	Copied to the counter, then removed from the field options.
`incrementBy`	number	`1`	Copied to the counter, then removed from the field options.

Field	Type	Default	Notes
`conditionalIncrement`	boolean	`false`	Copied to the counter and kept on the field.
`keys`	`CountersKey[]`	`[]`	Copied to the counter and kept on the field.

After the entity is saved, the field's `autoIncrementOptions` is rebuilt to keep **only** `enabled`, `key`, `conditionalIncrement`, and `keys` — the numeric tuning (`sharingLevel`, `startAt`, `incrementBy`) is authoritative on the counter, not the field.

Sharing levels

`AutoIncrementSharingLevelEnum`:

Value	Name	Meaning
`0`	`None`	Not shared.
`1`	`Module`	Shared at module level.
`2`	`System`	Shared at system level.

`sharingLevel` is persisted on the counter and defaults to `None`. Note: the code that would share one counter across multiple fields is not fully implemented — counter creation is keyed strictly by `key`, and an in-code `TODO` notes that creating a counter whose key already exists currently throws and that sharing "should be handled later". Treat cross-field sharing as partial/unfinished.

Allocating the next value

`SequenceService.getNextValue(field, dataEntry)`:

1. Reads `key = field.options.autoIncrementOptions?.key`. If absent, logs an error and returns `undefined`.
2. Loads the counter with `countersRepository.findById(key)`. If not found, logs and returns `undefined`.
3. Calls `evaluateConditionalKeys(counter.keys, dataEntry)` — for each key whose `calculationMethod === Expression`, it replaces `key.value` with the result of `ExpressionService.evaluateExpression(key.value, null, dataEntry)`. This turns the stored expression into the concrete value for *this* entry.
4. Calls `counterSequenceRepository.getNextValue(counter)` to atomically obtain the next number.
5. Records usage: an `$addToSet` on the counter adds `{ entityField: '<dataEntry.dataEntity>_<field.normalizedName>' }` to `usedBy`, so the counter is now marked as consumed by this field.
6. Returns the numeric value.

`CounterSequenceRepository.getNextValue(counter)` performs the atomic bump:

- If `counter.conditionalIncrement` and `counter.keys` are present, it builds a `keys` object mapping each `keyName -> key.value` (the already-evaluated value). Otherwise `keys` is empty.
- It runs a MongoDB `findOneAndUpdate` on `_counters_sequence` with:
  - **filter**: `{ counterId: counter.id, ...keys }` — matches the stream for this counter and this exact key combination.
  - **update** (aggregation-pipeline `$set`):
    - `_id`: existing `_id` or a new UUID v4 (upsert case).

        - `counterId`: existing or `counter.id`.
        - the `keys` fields.
        - `lastValue`: if the current `lastValue` is null → `counter.startAt`; otherwise `lastValue + counter.incrementBy`.
    - **options**: `{ upsert: true, returnDocument: 'after' }`.
- Returns `result.value.lastValue`.

Because the read-modify-write is a single atomic `findOneAndUpdate` scoped to `(counterId + key values)`, concurrent entry creation cannot hand out the same number twice for the same stream. The first allocation for a stream yields exactly `startAt`; each subsequent one adds `incrementBy`.

On any error the repository logs it and throws `GeneratingCounterSequenceException`.

Worked example

Counter `requestNumber` with `startAt = 100`, `incrementBy = 5`, non-conditional:

Create #	`_counters_sequence.lastValue`	Value written to entry
1st	(none) → `100`	`100`
2nd	`100 + 5` → `105`	`105`
3rd	`105 + 5` → `110`	`110`

With `conditionalIncrement` and a key `year` (expression evaluating to the entry's year), each distinct year gets its own sequence document and its own independent run starting at `startAt`.

Conditional (multi-stream) sequencing

Turning on `conditionalIncrement` with one or more `keys` makes a single counter maintain **independent numbering streams**. Each stream is identified by the combination of evaluated key values, and each has its own `CounterSequence` document. Common use: reset/segment numbering per year, per department, per branch, etc., where the key expression derives the segment from the entry being created.

Rules enforced at entity save time (`validateConditionalIncrement`):

- Every `CountersKey.calculationMethod` **must be** `Expression`; otherwise `UnsupportedConditionalIncrementCalculationMethod` is thrown.
- If the counter has already been used (`usedBy` non-empty), you **cannot** change `conditionalIncrement` or change the `keys` set — `CanNotChangeConditionalIncrementConfigurations` (`CAN_NOT_CHANGE_CONDITIONAL_INCREMENT_CONFIGURATIONS`) is thrown. Conditional configuration is effectively frozen once numbers have been issued.

Interaction with uniqueness: a normal auto-increment field is forced `options.unique = true`. When `conditionalIncrement` is on (on the incoming options or on the existing counter), the field is set `options.unique = false` — because the same number can legitimately recur across different streams.

Counter lifecycle at entity save-time

`DataEntityService.handleAutoIncrementField(field, dataEntity)` runs for every field with `autoIncrementOptions.enabled` when an entity is created or updated. It looks up the existing counter by

`referencedBy.field === field.id` and then:

- **No existing counter** → create a new `Counters` with `id = key`, the `incrementBy` / `startAt` / `sharingLevel` (defaulting `1` / `1` / `0`), `conditionalIncrement`, `keys`, and a single `referencedBy` entry for this field. (Creation is wrapped in a `try/catch` that swallows errors; per the in-code `TODO`, a duplicate key currently throws and sharing is not yet handled.)
- **Existing counter, key changed** →
  - If this entity-field has **not** consumed the old counter, create a fresh counter for the new key, and if the old counter was referenced only by this field (`referencedBy.length === 1`, "junk"), delete it.
  - If this entity-field **has** consumed the old counter, throw `FailToAddFieldException` ("can not update auto increment key after it is used"). **The counter key is immutable once used.**
- **Existing counter, same key** → if the counter has **not** been used yet, update its `sharingLevel` / `startAt` / `incrementBy` (only when provided), `conditionalIncrement`, and `keys`, and clear `usedBy`. If it has already been used, tuning is left as-is (the block is skipped).

In short: `startAt`, `incrementBy`, and `sharingLevel` are freely editable until the first value is issued; after that the counter's numbering is effectively locked, and the key can never change.

Sequences API

There is exactly one HTTP endpoint for counters, and it is **read-only**. Counters are created and mutated as a side effect of saving data entities, not through a dedicated write API.

`GET /v1/auto-increment-counters`

Lists auto-increment counters. Defined in `CountersController.find`.

- **Auth**: requires authentication via the `Auth2` strategy (`@authenticate`), and authorization through `entityVoterFactory()` (`@authorize`).
- **Query**: accepts a standard LoopBack `filter` for `Counters` (`@param.filter(Counters)`), so you can filter/limit/select/order over the counter fields documented above.
- **Returns**: `Counters[]` (the raw counter documents).

There is **no** create/update/delete counter endpoint in this controller, and **no** endpoint that exposes `_counters_sequence` values directly.

```
GET /v1/auto-increment-counters?filter={"where":{"conditionalIncrement":true}}
Authorization: Bearer <token>
```

Notes for engineers

- **Two collections, split responsibility.** `_counters` holds configuration; `_counters_sequence` holds the live running value(s). `Counters.lastValue` is deprecated — do not read or write it as the source of truth.
- **Atomicity is at the sequence level.** The `findOneAndUpdate` upsert on `_counters_sequence` scoped by `counterId` + evaluated key values is the concurrency guarantee. Any change here must preserve that single-round-trip atomic increment.
- **Keys are evaluated per entry.** `evaluateConditionalKeys` mutates the in-memory `counter.keys[].value` to the evaluated result before building the stream filter; the persisted counter keeps the *expression*, while the sequence document stores the *evaluated* value.

- **Usage tracking gates edits.** `usedBy` (on the counter) and the `referencedBy.length === 1` "junk" check drive whether config edits are allowed and whether an orphaned counter is deleted. The `usedBy` string format is `<entityId>_<field.normalizedName>`.
- **Value is written by full normalized name.** `incrementField` does `set(dataEntry, 'value.<field.fullNormalizedName>', nextValue)`. There is an in-code `TODO` noting the value is not coerced to string for string-typed fields.

Preview mode internals

This page is for **integrators and platform engineers** who need to know how preview mode works under the designer: the request header, what the backend does with it, how sample data is generated, and the contract between the page designer and the embedded mobile client.

> **Previewing in the designer?**
>
> Builders should start with Preview Mode [Build Apps] — it covers both previews in the labels the designers show.

Two mechanisms

Two mechanisms sit under the name. The first is a **request-level mode**: a request is tagged as preview, and the services answering it return generated data shaped exactly like the real thing instead of reading the store. The second, in the page builder, is a **live mobile preview**: the real mobile client, embedded in the builder and driven by the page you are editing.

Sample content is generated for the main data surfaces: entries (list, kanban board, tree, entry details), collections, saved queries and quick filters, system buttons, entry history and email logs, workflow status, dashboards and their statistics, timeline (post) feeds, and comment threads.

Which previews request sample data

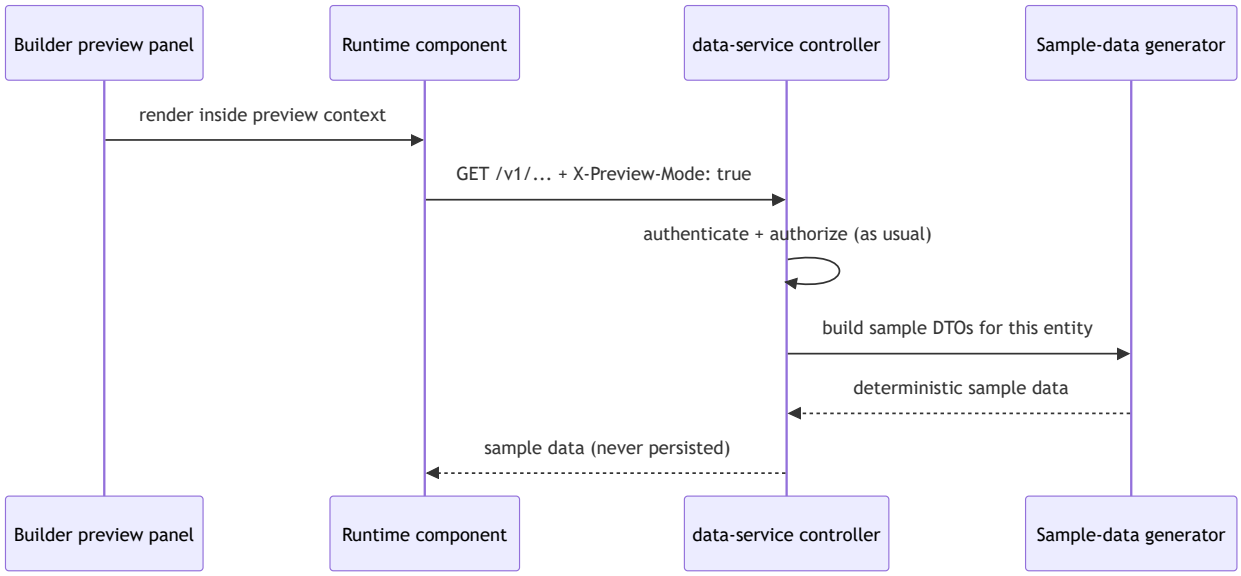
Only the **view designer's preview panel** requests sample data. On the web, the panel provides the `PREVIEW_MODE` injection token and only components rendered inside it flag their reads (`view-preview-panel.component.ts`, `dashboard-card.component.ts`, `entities-list.service.ts`); the view designer's mobile tab asks the embedded client to do the same. The **page designer** (pages / `GenericView`) does not: its canvas and its live mobile preview show the real, already-saved records. In the mobile client, `PreviewModeService.isActive` marks a preview build (write guards, realtime opt-outs) while the narrower `fakeDataHeaderEnabled` — true only for the DataView preview — decides whether `PreviewModeInterceptor` adds `X-Preview-Mode: true` (`mobile-app/lib/core/services/preview/preview_mode_service.dart`).

Marking a request as preview

The `X-Preview-Mode` header

Preview is **request-scoped**, not a stored property of a view or page. A request opts in by carrying the header `X-Preview-Mode: true` — nothing else about the request changes: same route, same authentication, same query parameters. Only the exact value `true` (case-insensitive) counts; any other value, or an absent header, means a normal request.

On the web side you never set the header by hand. The builder's preview panel marks itself as a preview context, every runtime component rendered inside that panel flags its own reads, and a single HTTP interceptor adds the header to those flagged requests only. Ordinary portal traffic is untouched.



What preview relaxes — and what it does not

Preview mode is **not** a permission bypass. The request is still authenticated, and the same authorization voters that guard the real endpoints still run. Access levels, restricted views, and sensitive-field rules are all still in force.

There is exactly one relaxation, and it is narrow. Some routes authorize by loading the target resource by id (a view, a saved query, a collection). Sample data is addressed by **fabricated ids** that deliberately do not exist in the store, and a resource that cannot be found causes the request to be denied before the controller ever runs. So, in preview mode only, a **genuine not-found** on that lookup is allowed through, letting the controller answer with sample data. Any other failure — a database error, for example — is not masked; it still fails.

The practical consequence: preview is a rendering aid, not an inspection tool. It cannot show you data you are not entitled to, and it is not a reliable way to judge what a restricted end user will see. Verify that against a normal, non-preview view once the design is published; see Permissions [Administration › Permissions & access].

One thing is deliberately never faked: **relation definitions**. The name and target entity of a relation are schema metadata, so they always come back real. Only the entries rendered under a relation tab are sample data.

The generated sample data

Sample values are produced by a deterministic generator seeded from a stable key — the sample entry's id, the entity and field, the widget id, and so on. Same key, same values, every time. That is what keeps a list row and its own detail page in agreement, and what keeps a preview from reshuffling itself after an unrelated edit.

Where the entity has real configuration to draw on, the generator prefers it over inventing something: **category options**, **tags**, and **kanban columns** come from the entity's real configured values (only *which* sample entry carries them is fabricated), and **system buttons** and **saved queries** are the view's real ones with only their ids replaced — so a real button can never be fired against a sample entry, and selecting a sample saved query does nothing.

Values by field type

Field type	Sample value
Number	A random integer between 1 and 9999.
Date	A past date.
Time	A recent time of day.

Field type	Sample value
Date & time	A recent timestamp.
Yes/no	A random boolean.
Localized text	A sentence per configured language.
Geolocation (GeoJson)	A real GeoJSON point feature with randomized longitude/latitude, so map fields render instead of breaking. `Since 1.8`
Object / JSON	An empty object — the only shape that stays compatible with a renderer that writes nested properties onto the value. `Since 1.8`
File	Empty (no file).
Text (and any other type)	Content-aware text: an email address, phone number, URL, or person name when the field's name suggests one, otherwise a sentence of filler text.

Two structural rules matter as much as the per-type values:

- **Blocks are faked field by field**, recursing into the block's own sub-fields. A repeatable block is filled to its configured minimum number of repetitions, and each repetition is seeded separately so the repetitions do not come out identical. `Since 1.8`
- **Workflow status** is built from the entity's *real* first workflow and its real step ids and ordering; only the execution state is fabricated, and the step owner is deliberately left empty so nothing in the preview looks executable.

Reference-field pickers

Opening a relation or user-reference picker inside a preview returns a short list of **sample entries** to choose from. `Since 1.8`

This used to fail: the picker resolved the entry being edited against the real store, and the entry being edited in a preview is a fabricated, unsaved one — so the request came back with "entity not found" and the picker stayed empty. The picker now short-circuits to sample entries after its normal field-validation and source-authorization checks, before any real lookup.

Dashboards, statistics, and timeline feeds

A brand-new entity has an auto-created dashboard with **zero** widgets and no timeline posts, so previewing either would show a permanently empty screen. Both now fall back to sample content. `Since 1.8`

- **Dashboard** — when the real dashboard resolved for the request has no widgets at all, the preview substitutes a sample dashboard with two demo widgets. A dashboard that *does* have widgets configured keeps its real layout; only the aggregated numbers are sample values.
- **Statistics** — a sample widget resolves to a fabricated widget definition (name, chart type, aggregation) plus sample category rows and a total. A real, already-configured statistic query keeps its real resolved source criteria and view scoping; only its numbers are faked.
- **Timeline** — a preview always renders a small sample feed, whether or not real posts exist, with the publish and pin state populated so the timeline widget renders its controls.
- **Comments** — a short sample thread, including one sample attachment, anchored to the entry being previewed. The mention/autocomplete list comes back empty.

Mobile Preview

The builder can also show you how your design renders on a phone, without leaving the browser and without a device. In the view builder this is a **mobile preview** tab beside the web preview; in the page builder it is a live panel next to the canvas.

> **Note:** The mobile preview renders the real mobile client inside the builder, but it is still a preview: it is a single embedded screen, not a full app session. Some surfaces are web-only — **reports**, for example, are not available on the mobile client — so preview those on the web.

Live mobile preview in the page builder `Since 1.8`

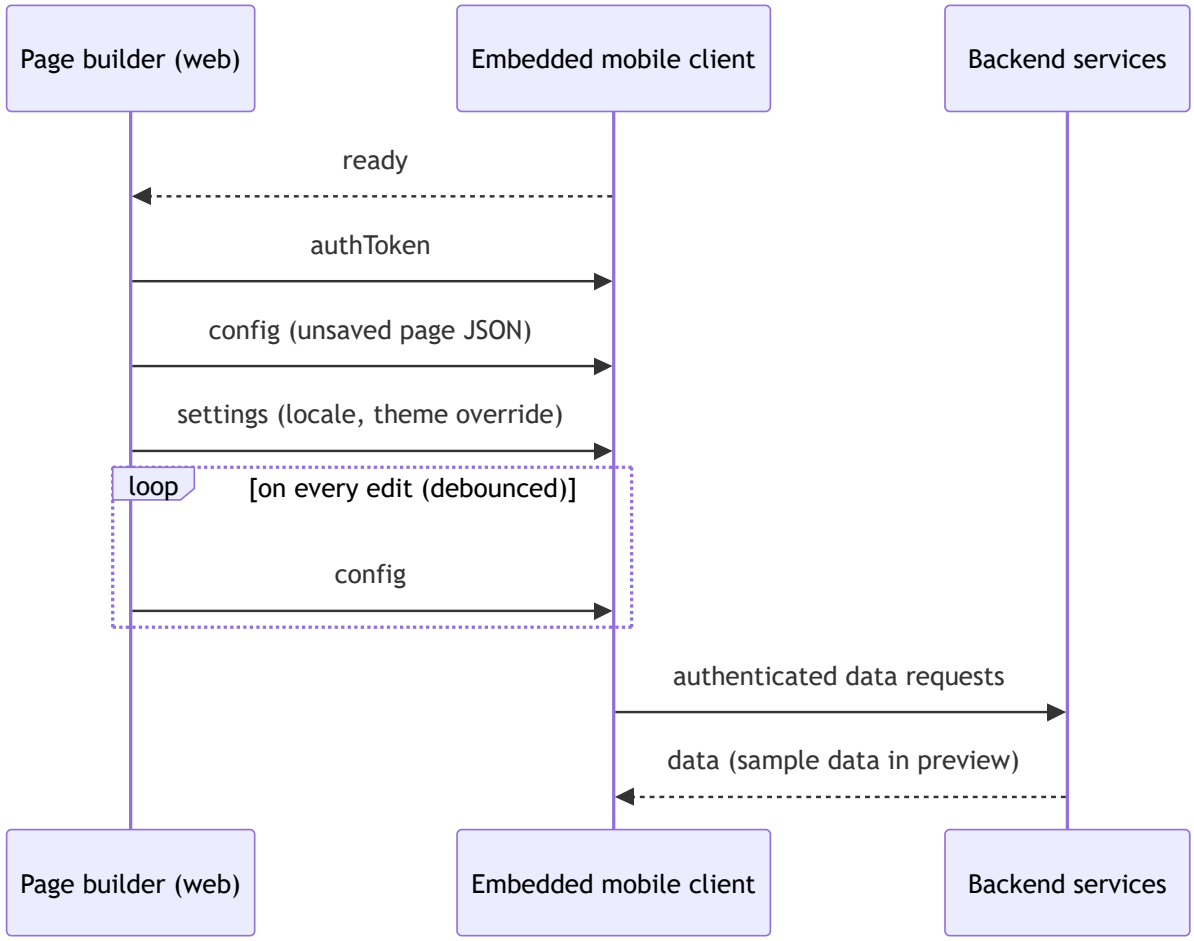
The page builder's **Content** tab hosts a live mobile preview: the actual mobile client, embedded in the builder, rendering the page **you are editing right now** — including unsaved changes. Every edit is pushed to it, so the phone follows the canvas without a save or a reload.

Because it is the real client, what you see is the real mobile rendering of your widgets, not a browser approximation of it: the same layout engine, the same widget implementations, the same theme. The embedded client makes its own authenticated calls for widget data. For a page these calls do **not** carry `X-Preview-Mode`, so the widgets show real, already-saved records (see Which previews request sample data); only the view designer's mobile tab requests sample data.

The builder-to-app bridge

The builder and the embedded client talk over a small, versioned message contract:

- **Handshake** — the embedded client announces that it is ready; the builder replies with the session's access token, the current page configuration, and the display settings.
- **Live configuration push** — after every edit (debounced, so typing does not flood it) the builder re-sends the page configuration. The client re-applies it in place rather than remounting, which is why your scroll position and any local form state survive an edit.
- **Display settings** — locale plus a theme override carrying the portal's live primary colour, because the preview build fetches no theme of its own; an invalid or missing colour simply leaves the mobile theme alone.
- **Device frame** — a fixed logical phone frame, scaled uniformly to fit the space available and never scaled *up*, so the frame never blurs.
- **Origin isolation** — messages are posted only to the preview host's own origin, never a wildcard, and the embedded client validates the sender's origin before accepting a message.



What the mobile preview renders

The preview mounts the **production page screen** in preview mode, so widget rendering is identical to the real app, with three deliberate differences: the app bar is drawn (for visual parity) but its navigation is inert, there is no help button, and **redirection rules never fire** — a redirect would navigate the embedded client away from the very page you are previewing.

Configuration parsing is tolerant per component rather than all-or-nothing, because the builder pushes half-finished states constantly:

- A malformed widget is **dropped** and the rest of the page still renders.
- Missing or partial display options are backfilled with defaults (full width, no forced new row, stretch alignment).
- If a pushed configuration cannot be parsed at all, it is ignored and the **last good configuration stays on screen** — the preview never blanks on a bad intermediate edit.

Sort rows during a live edit

Sort rows get one specific rule of their own: a row is kept only if it carries **both** a field and a direction. A row that has one but not the other — the state you are in for a moment after adding a sort row and picking the field but not yet ascending/descending — is **dropped**, and only that row. `Since 1.8`

Without the rule, that half-finished row made the whole configuration update unparseable, so the push was discarded and the preview silently froze at its previous sort while you kept editing.

Staying inside the preview

The preview is a single embedded screen with no surrounding app shell, so a real navigation would strand it. Two guards keep it in place: navigation calls are swallowed, and any route that is not a preview route is redirected back to the preview host.

Both guards carry the same exemption for **entry-extension routes** — the discussion, workflow, history, and information tabs on an entry's details screen. `Since 1.8`

Those tabs are meant to render *in place*, nested inside the previewed page. Before the exemption was applied to both guards, opening one of them inside a previewed entry tripped the redirect, bounced the route back to the preview host, and blanked the whole preview instead of showing the tab.

Preview fidelity fixes

A few 1.8 fixes make preview mode more reliably show real data — accuracy corrections rather than new capabilities. `Since 1.8`

In the page builder's live mobile preview, the **Summary**, **Timeline**, and **Timeline hero-slider** widgets used to go on rendering whatever they had fetched the first time they mounted, rather than following later edits. Because the mobile preview keeps the same widget tree mounted across live edits (see What the mobile preview renders), a widget whose fetch ran only once at mount time never noticed when the page was pointed at a different view, entity, or element — so it could silently keep showing data tied to a source you had already replaced. All three widgets now detect that kind of change and force a refetch.

Separately, the page builder's live mobile preview could show outright fabricated data, not just stale data: it was inadvertently inheriting the view builder's fake-sample-data behavior, so the backend generated synthetic rows for it exactly as it does for an unsaved view. The mobile client now keeps that decision separate: only the view builder's mobile preview — which is meant to show synthetic rows for an unsaved, in-progress view — requests fake data; the page builder's live preview always reflects the real, already-saved content the canvas shows.

The page builder's preview panel also picked up two smaller fixes: it now includes the page's real id in the configuration it pushes to the mobile preview, so page-scoped widget requests (such as the Timeline widget's feed) resolve against the actual page instead of falling back to an unscoped request; and it gained the same expand-to-dialog icon the mobile preview already had in the view builder.

Availability

Capability	Availability
Request-level preview mode (`X-Preview-Mode`)	Since 1.8
Deterministic sample entries, collections, queries, buttons, history	Since 1.8
Geolocation and object-field sample values; block / repeatable-block faking	Since 1.8
Reference-field pickers return sample entries	Since 1.8
Dashboard, statistics, and timeline sample data with zero real widgets/posts	Since 1.8
Live mobile preview in the page builder	Since 1.8
Sort-row coercion during a live edit	Since 1.8
Redirect guard honouring entry-extension routes	Since 1.8

Data stores: model, routing and API

This page is for **integrators and platform engineers** who create or read stores through the API, or need to know exactly how the platform decides where an entry is written and which spaces can read it.

> **Sharing records between spaces in the UI?**
>
> Builders should start with Share an entity's records across spaces [Build Apps].

Overview

A **data store** is the physical partition that a data entity's *entries* are written into. Every data entry carries a single `store` reference, and that reference decides which partition the entry lives in and, therefore, which spaces can read it. Stores are **not** a general-purpose key–value cache for builder code; they are the storage-routing layer that sits underneath a data entity's entries and is driven by per-entity **store rules**. Conceptually, stores are dedicated units categorized by type to define the scope of sharing and permissions for an entity's entries.

OSOS supports **multi-store routing**: an entry is **written to a single store** (the highest-priority store available to the current space) but **reads aggregate across all stores** available to that space.

> **Availability:** Long-standing. The store-rules model and the Stores API predate the 1.2 milestone, so treat this as a foundational, always-present capability rather than a recent feature.

Store types

There are three kinds of store, defined by `DataStoreTypeEnum` (`backend/packages/osos-shared/src/models/enums.ts`):

Type	Numeric value	Scope	Persisted as a `DataStore` document?
`Public`	`0`	One global store shared by every space	No — virtual/default
`Private`	`1`	One store per space	No — virtual/default
`Shared`	`2`	A store spanning two or more explicitly selected spaces	Yes

- **Public store** — created automatically with the entity; holds data accessible from all spaces. Only its **weight** can be adjusted.
- **Private store** — created automatically with the entity; holds data accessible only within the owning space. Only its **weight** can be adjusted.
- **Shared store** — defined by an administrator when needed; enables sharing data between specific spaces chosen at creation. The administrator selects the participating spaces and sets a **weight**.

Only **Shared** stores are real documents in the `DataStore` collection and are the only type you create, read, or update through the Stores API. `Public` and `Private` are synthesised at runtime from fixed identifiers and never stored as `DataStore` rows.

Fixed identifiers

From `backend/packages/osos-loopback/src/store/types.ts` :

Constant	Value
`PUBLIC_STORE_ID`	`"PUBLIC"`
`PRIVATE_STORE_ID`	`"PRIVATE"`
`PUBLIC_STORE`	`ReferenceProperty { dataEntity: "DataStore", dataEntry: "PUBLIC" }`
`PRIVATE_STORE(spaceId)`	`ReferenceProperty { dataEntity: "DataStore", dataEntry: "PRIVATE_<spaceId>" }`

The public store has one fixed `dataEntry` (`PUBLIC`). Each space gets its own private store whose `dataEntry` is `PRIVATE_<spaceId>`.

Store rules

A data entity owns a list of **store rules** (`dataEntity.storeRules: StoreRule[]`). A store rule (`store-rule.model.ts`) has three fields:

Field	Type	Notes
`type`	`DataStoreTypeEnum` (number)	`Public` / `Private` / `Shared`
`storeId`	`string | null`	Set **only** when `type === Shared`; the `DataStore` id
`priority`	`number` (required)	Higher priority wins when routing a write

When a data entity is created, it is seeded with two default rules (`StoreService.defaultStoreRule`):

Rule	Type	`storeId`	`priority`
Public default	`Public`	`PUBLIC_STORE_ID` (`"PUBLIC"`)	`100`
Private default	`Private`	`PRIVATE_STORE_ID` (`"PRIVATE"`)	`50`

These defaults are attached in the create-data-entity and create-virtual-data-entity command handlers. Adding a **Shared** store to an entity appends a third rule (with the store's `id` and the requested weight).

In the API surface the store rule's `priority` is exposed as `weight`. They are the same value; the DTOs use `weight`, the persisted model uses `priority`.

How a store rule routes a write

`StoreService.getStoreForWrite(storeRules, spaceId)` decides where a new entry is written:

1. Drop any **Shared** rule whose store does not include the current `spaceId` in its `spaces` list. `Public` and `Private` rules are always kept.
2. From the remaining rules, pick the one with the **highest** `priority`.
3. Resolve it to a `ReferenceProperty`:
   - `Shared` → `{ dataEntity: "DataStore", dataEntry: <storeId> }`
   - `Private` → `PRIVATE_STORE(spaceId)`
   - anything else → `PUBLIC_STORE`

Because a freshly created Shared store is given a `weight` / `priority`, its value relative to the Public (100) and Private (50) defaults determines whether new entries in the participating spaces land in the shared store or fall back to the public/private defaults.

How store rules gate a read

`StoreService.getStoresForRead(spaceId)` returns the set of stores visible from a space: the Public store, that space's Private store, and every Shared store whose `spaces` include the space. The entry-read/validation paths (for example in `data-entry-writer.service.ts`) check that an entry's `store.dataEntry` is in this set; an entry whose store is not visible from the current space is treated as *not found* in that space. This is the mechanism by which stores scope entry visibility across spaces.

Data model: `DataStore`

`data-store.model.ts` — persisted only for **Shared** stores. Extends `Identified<UUID>` (so `id` is a UUID).

Field	Type	Required	Notes
`name`	`LocalizedValue`	Yes	Localised display name
`type`	`DataStoreTypeEnum` (number)	Yes	In practice always `Shared` (`2`) for persisted stores
`dataEntity`	`string`	Yes	Id of the owning data entity
`spaces`	`ReferenceProperty[]`	Yes	The spaces the store spans
`description`	`NullableLocalizedValue`	No	
`creationInfo`	`ActionInfo`	No	Who/when created (with optional delegatee)
`updateInfo`	`ActionInfo`	No	Who/when last updated

Persistence is a plain LoopBack `DefaultCrudRepository` (`DataStoreRepository`) over the Mongo datasource.

API reference

All endpoints live in `StoresController` and require authentication via the OSOS `Auth2` LoopBack strategy. They are all scoped to a data entity by `entityId` in the path. Before doing anything, each handler loads the entity and runs `Helper.validateDataEntityExistsActive` (the entity must exist and be active).

Method	Path	Purpose
`POST`	`/v1/entities/{entityId}/stores`	Create a Shared store for the entity
`GET`	`/v1/entities/{entityId}/stores`	List all stores for the entity (defaults + shared)
`GET`	`/v1/entities/{entityId}/stores/{storeId}`	Get one store rule / store
`PUT`	`/v1/entities/{entityId}/stores/{storeId}`	Update a store

Response shape: `StoresResponseDto`

Every endpoint returns this shape (or an array of it):

Field	Type	Notes
`storeId`	`string`	Store id, or `"PUBLIC"` / `"PRIVATE"` for the defaults
`storeName`	`LocalizedValue`	Only populated for Shared stores
`type`	`DataStoreTypeEnum` (number)	Required
`description`	`NullableLocalizedValue`	Optional; Shared only
`spaces`	`ReferenceProperty[]`	Optional; Shared only
`weight`	`number`	The store rule priority

For Public/Private entries the response carries only `storeId`, `type`, and `weight` — `storeName`, `description`, and `spaces` are left unset because those stores have no `DataStore` document.

Create a store — `POST /v1/entities/{entityId}/stores`

Request body (`CreateDataStoreRequestDto`, which extends `DataStore` and adds `weight`):

Field	Type	Required	Notes
`name`	`LocalizedValue`	Yes	
`type`	`DataStoreTypeEnum` (number)	Yes	**Must be `2` (Shared)** — see validation
`spaces`	`ReferenceProperty[]`	Yes	At least two, no duplicates
`description`	`NullableLocalizedValue`	No	
`weight`	`number`	Yes	Store rule priority; must be unique within the entity
`dataEntity`	`string`	—	Ignored in the body; the controller overwrites it with `entityId` from the path

Server-side flow:

1. Reject if any existing store rule on the entity already has `priority === weight` → `StoreWithSameWeightExistsException`.
2. `storeDto.dataEntity` is forced to the path `entityId`.
3. `StoreService.createStore` validates (see below) and persists the `DataStore`, stamping `creationInfo` / `updateInfo` with the caller (and `delegatee` when acting on behalf of another user).
4. A new `StoreRule { storeId, priority: weight, type }` is appended to the entity's `storeRules`.
5. An `UpdateDataEntityCommand` is published on the service bus and a token is registered via `InvokerService.registerToken`. **The rule attachment to the entity is therefore asynchronous / eventually consistent** — the HTTP response returns before the entity update is confirmed.

Response: `StoresResponseDto` for the created store (`storeName`, `description`, `spaces`, `type`, `weight`, `storeId`).

Example request:

```
{
  "name": { "en": "Regional shared store" },
  "type": 2,
  "spaces": [
    { "dataEntity": "Space", "dataEntry": "space-a" },
    { "dataEntity": "Space", "dataEntry": "space-b" }
  ],
  "description": { "en": "Entries visible to both regional spaces" },
  "weight": 75
}
```

Create-time validation (`validateCreateStore`)

Condition	Exception	Error code
Fewer than 2 spaces	`NotEnoughStoreSpacesSelectedException`	`NOT_ENOUGH_STORE_SPACES_SELECTED`
Duplicate space ids	`StoreSelectedSpacesDuplicationNotAllowedException`	`SELECTED_SPACES_SHOULD_NOT_HAVE_DUPLICATION`

Condition	Exception	Error code
`type !== Shared`	`OperationNotSupportedForThisType`	`OPERATION_NOT_SUPPORTED_FOR_THIS_STORE_TYPE`
Weight collides with an existing rule	`StoreWithSameWeightExistsException`	`STORE_WITH_SAME_WEIGHT_ALREADY_EXIST`

A source comment confirms the intent: *"At the moment of implementation only supports type: shared."* Creating Public or Private stores through the API is not supported.

List stores — `GET /v1/entities/{entityId}/stores`

Optional query parameter `filterCriteria` (a `FilterCriteria<DataStore>`) is merged with a forced `where.dataEntity = entityId` before querying, so filters only ever match this entity's Shared stores.

The response concatenates:

1. **Default rules** — every store rule whose `type !== Shared` (i.e. the Public and Private defaults), each returned as `{ storeId, type, weight }`.
2. **Shared stores** — fetched from the `DataStore` collection for this entity, each returned with full detail and its `weight` looked up from the matching store rule's `priority`.

For Shared stores the service also resolves each space reference: it loads the matching `DataEntryIndex` rows and attaches the space's index `meta` as a `_resolved` field on each space reference (used to render the space's display metadata without a second call).

Get one store — `GET /v1/entities/{entityId}/stores/{storeId}`

Looks up the store rule on the entity by `storeId`.

- If no rule matches → `DataSourceStoreNotExist`.
- If the rule's `type !== Shared` (a Public/Private default) → returns `{ type, weight, storeId }` only, without hitting the `DataStore` collection.
- If the rule is Shared → loads the `DataStore` (via `getStoreById`, which throws `DataSourceStoreNotExist` on `ENTITY_NOT_FOUND`) and returns full detail plus the rule's `weight`.

Update a store — `PUT /v1/entities/{entityId}/stores/{storeId}`

Request body (`UpdateDataStoreRequestDto`) — all fields optional, treated as a partial update:

Field	Type	Notes
`name`	`LocalizedValue`	Shared stores only
`description`	`NullableLocalizedValue`	Shared stores only
`weight`	`number`	New store-rule priority
`spaces`	`ReferenceProperty[]`	Shared stores only; **must have at least 2 entries**

Server-side flow:

1. The store rule must exist on the entity → else `DataSourceStoreNotExist`.
2. If `weight` is provided and collides with **another** rule's priority → `StoreWithSameWeightExistsException`.

3. If the rule is **Shared**, `StoreService.updateStore` updates the `DataStore` document. It rejects a `spaces` array with fewer than 2 entries (`NotEnoughStoreSpacesSelectedException`), then merges each supplied field over the existing values (`name`, `description`, `spaces`) and refreshes `updateInfo`. `type`, `dataEntity`, and `creationInfo` are preserved. For Public/Private rules the `DataStore` collection is not touched.
4. The store rule's `priority` is set to the new `weight` (or left unchanged), the entity's `storeRules` array is rewritten, and — as with create — an `UpdateDataEntityCommand` is published and a token registered (asynchronous / eventually consistent).

> **Note:** `updateStore` uses the repository's `updateById`, which rebuilds the model before writing. `DataStore`'s constructor does not force-seed array fields, so the omitted-field merge is done explicitly in `updateStore`. If you extend the model, keep this in mind (see the repository's migration guidance on partial updates).

Collated store indexes for list reads `Since 1.10`

Every entity list read starts by matching the entry's `store` (then its `state` and `virtual` flag), and it runs with the entity collation `{locale: 'en', strength: 2}`. MongoDB only uses an index whose collation matches the operation's, and none of the implicit indexes had one, so on a large entity that leading store match was a collection scan.

From 1.10 each non-virtual entity also carries:

- collated twins of the implicit `store` and `sharingLevel` indexes, and
- a store-and-order index named `IX_<entity>_store_metaOrder_state_ci` (keys `store.dataEntry`, `value._metaOrder.value`, `_id`, then the `state` / `virtual` filter), so a list page sorts and skips on index keys instead of in memory.

New entities get them when they are created; existing entities receive them through data-service migrations `1.10.0-a38`, `a41` and `a44`. The indexes are built asynchronously by the indexing service, so a freshly upgraded tenant serves list reads from the new indexes only once its index queue has built them. There is nothing to configure, and store rules and the Stores API are unchanged. Virtual entities are skipped, because a MongoDB view cannot carry an index.

Error reference

Error code	Meaning
`NOT_ENOUGH_STORE_SPACES_SELECTED`	A store needs at least two spaces
`SELECTED_SPACES_SHOULD_NOT_HAVE_DUPLICATION`	The selected spaces contain duplicates
`OPERATION_NOT_SUPPORTED_FOR_THIS_STORE_TYPE`	Attempted a create with a non-Shared type
`STORE_WITH_SAME_WEIGHT_ALREADY_EXIST`	Another store rule already uses this weight
`DATA_SOURCE_STORE_...` (`DataSourceStoreNotExist`)	The store / store rule does not exist for this entity

Building notes and constraints

- **Weight/priority is per-entity unique.** You cannot have two store rules on the same entity with the same weight; both create and update enforce this.
- **Weight is the routing knob.** New entries in a space are written to the highest-priority store rule that applies to that space. With the default weights the Public rule (`100`) outranks the Private rule (`50`), so new entries go to the public store unless the weights are changed. Set a Shared store's weight above `100` to make it win for the

spaces it spans; a weight between `50` and `100` only outranks the private default, which the public default already does, so it has no effect unless the Public weight has been lowered below it.

- **Shared stores must span ≥ 2 spaces with no duplicates**, on both create and (when `spaces` is supplied) update.
- **Only Shared stores are manageable via the API.** Public and Private stores exist implicitly on every entity and cannot be created or edited.
- **Rule attachment is eventually consistent.** Create/update return before the `UpdateDataEntityCommand` is processed, so an immediately following read of the entity's store rules may not yet reflect the change.
- **Cross-space visibility is enforced on read.** An entry is only visible in a space whose read set (Public + that space's Private + Shared stores covering it) contains the entry's store.

Summary and dashboard API reference

This page is for **integrators and developers** who create or execute summaries through the API, read dashboards or counter widgets, or need the exact server behaviour behind a card. In the API a summary is a **statistic query**.

> **Building dashboards in the UI?**
>
> Builders should start with Summaries, dashboards & counters [Build Apps] — it covers the same features in the labels the designer shows.

Everything on this page is OSOS's **own** dashboarding. `Since 1.6` the same statistic queries can also feed **Grafana** panels through the Apptomator data source, which is the right tool when you need Grafana's visualization and transformation range, or a single dashboard spanning OSOS and other systems — see Grafana Dashboards. Choose the in-product dashboard when the audience already lives in the portal; choose Grafana when the audience lives in Grafana.

Overview

Three related building blocks in the OSOS **data-service** let you present aggregated numbers over your entities:

- **Statistic queries (summaries)** — the reusable, server-side aggregation definitions (count / sum / min / max / average, optionally grouped) that produce a single number, a list, or a chart series. In the builder these surface as **summary cards**.
- **Dashboards** — ordered collections of statistic-query elements (optionally arranged into named groups) attached to the system, a module, or a data entity.
- **Counter widgets** — lightweight "count of entries matching a view (and an optional query)" tiles surfaced to end users.

All three live in `backend/services/data-service`. Every endpoint requires the `Auth2` authentication strategy (`@authenticate` at the controller class level).

Statistic queries (summaries)

A statistic query is the unit of aggregation. It is defined **per data entity** and stored in its own collection (`StatisticQueryRepository`). Each dashboard card is built on one statistic query.

In the designer, **Display in overview tab** sets `showInEntityOverview`: the summary appears in a dedicated overview tab in the gallery, which the gallery/view settings must enable.

Data model

Source: `models/statistics-queries/statistic-query.model.ts`.

Field	Type	Required	Notes
`id`	UUID	—	Assigned on create
`specificationVersion`	number	yes	Defaults to `1`

Field	Type	Required	Notes
`name`	LocalizedValue	yes	Localized display name
`normalizedName`	string	no	Auto-generated from `name` if omitted; unique per entity
`description`	NullableLocalizedValue	no	
`dataEntity`	UUID	yes	Pinned by the server from the route entity — clients cannot set/override it
`icon`	Icon	yes	
`source`	DataEntrySource	yes	Where to read from: all entries (Store), a collection, or a query
`sourceOptions`	StatisticQuerySourceOptions	no	Parameter values for a parameterized query source
`aggregation`	StatisticsQueryAggregation	yes	The grouping + function (see below)
`orderBy`	number (`SortDirEnum`)	yes	`0` = Asc, `1` = Desc — applied to the `category` of the result
`showAs`	number (`StatisticsQueryDisplayEnum`)	yes	`0` = Number, `1` = List, `2` = Chart
`chartType`	number (`ChartTypeEnum`)	no	`0` Column, `1` Bar, `2` Line, `3` Pie, `4` Donut
`options`	StatisticQueryOptions	no	`{ colors?: string[] }`
`showInEntityOverview`	boolean	yes	Whether it appears in the entity overview
`filterField`	string	no	The one field an execution caller is allowed to filter on at run time

The `aggregation` object (`statistics-query-aggregation.model.ts`):

Field	Type	Notes
`field`	string	The field the aggregation function operates on
`groupByField`	string[]	Fields to group by; the first is the main category, the rest become `subCategory`
`groupByOperator`	number | null (`GroupByOperatorEnum`)	Date bucketing operator (year, quarter, month, week, day, hour, …)
`aggregationFunction`	number (`AggregationFunctionEnum`)	`0` Count, `1` Sum, `2` Min, `3` Max, `4` Average

Validation rules

`StatisticQueryController.validateQuery` enforces (throws `StatisticQueryValidationExcpetion` otherwise):

- `chartType` **must be null/undefined** unless `showAs === Chart` .
- If there is a `groupByField` **and** `showAs === Chart` , `chartType` **must** be provided.
- If there is **no** `groupByField` , `showAs` **must** be `Number` .
- If there **is** a `groupByField` , `showAs` **must not** be `Number` .
- `name` is validated for all configured languages and checked for duplicates within the same `dataEntity` .
- `source` is validated via `DataSourceService.validateDataSource` .

`normalizedName` uniqueness is enforced per entity on both create and update. On create, if omitted, it is generated with the `statistic_query` prefix via `generateUniqueNormalizedName` . **Since 1.6** — the `normalizedName` field and its generation/uniqueness handling were added in the 1.6 line.

API authoring pitfalls: source.store and field paths

The builder fills `source` and `aggregation` for you; when creating statistic queries directly over the HTTP API, two conventions are easy to get wrong, and neither produces a helpful error.

`source.store` must be the entity's normalized name. For a `Store` source (`type: 0`), `DataSourceService.validateDataSourceStore` resolves the entity by `normalizedName` **only** — not by entity id and not by display name. Any other value fails at create/update with `DataSourceEntityNotExist` — message `Entity <store> not found`, error code `ENTITY_NOT_FOUND` — which is misleading when the value is a real entity id or display name: the entity exists, it just isn't registered under that string as a normalized name. Additionally, the `store` property carries the schema pattern `(^[a-zA-Z_s][a-zA-Z0-9_]{2,127}$|^$)`, which rejects UUIDs (they contain hyphens) and display names with spaces at the schema level. At execution time there is **no existence check at all** — the store string is used verbatim as the MongoDB collection name (`DataEntryRepositoryFactoryService.getDataEntryRepository` sets `settings.mongodb.collection` to it), and the platform's entity-to-collection mapping is exactly `entity.normalizedName`. For a `Query` source (`type: 2`) you do not set the store yourself — it is taken from the referenced query's `dataSource.store`.

Aggregation field paths need the `value.` prefix. Entry field data is stored under the `value` property of every `DataEntry`, so `aggregation.field` and each `aggregation.groupByField` entry must be written as `value.<fieldName>` (entry metadata paths such as `creationInfo.doneAt` are used as-is; this is the convention the frontend always sends). The API does not enforce it: `StatisticQueryService.getFieldParts` strips `value.` only to look the field up in the entity metadata, swallows any lookup failure (logged, never thrown), and interpolates the **raw** path verbatim into the `$group` stage — the category from `$<groupByField>` and the aggregated value from `$<fullLookupPath>`. A bare path such as `amount` therefore builds a pipeline over `$amount`, a key absent from every document, and **fails silently**: MongoDB evaluates the missing path as `null`, so all rows collapse into a single ungrouped `null`-category bucket, and `Sum` / `Average` over the missing path yields `0` / `null` — with no error anywhere.

A minimal correct create body (`POST /v1/entities/{entity}/statistic-queries`), using the numeric enum values from the quick-reference table at the end of this page:

```
{
  "name": { "en": "Total amount by status" },
  "icon": { /* … */ },
  "source": { "type": 0, "store": "purchase_order" },  // 0 = Store; the entity's normalizedName
  "aggregation": {
    "field": "value.amount",                           // value. prefix required
    "groupByField": ["value.status"],                  // value. prefix required
    "aggregationFunction": 1                           // 1 = Sum
  },
  "orderBy": 0,                                        // 0 = Asc
  "showAs": 2,                                         // 2 = Chart (groupByField present)
  "chartType": 0,                                      // 0 = Column — required when showAs is Chart
  "showInEntityOverview": false
}
```

Display types and chart formats

The `showAs` setting controls how a summary card renders, in one of three forms:

1. **Number** — the result as a single KPI value.
2. **List** — the top values as a list.
3. **Chart** — one of the following chart types (`chartType`):

Chart type	Notes
Column	Vertical bars; supports stacked multi-group.
Bar	Horizontal bars.
Line	Trends over a group key.
Pie	Proportions.
Doughnut	Proportions with a centre label.

Charts use theme-aware colours, an external HTML legend, and a **segment limit with an "Other values"** bucket for high-cardinality group-bys.

> Note: two chart capabilities are currently **disabled in the code** — gradient fill and brightness-based segment sorting — so charts render with flat theme colours.

Summary colors: copy and paste `Since 1.10`

A summary's **Select Colors** field (stored as `options.colors`) holds at most **20** colors. Each color chip now has **copy** and **remove** buttons (shown on hover, keyboard focus and touch), and a **copy all** action copies the whole list as `#a, #b, …` in list order. **Paste** appends every token that is a `#RGB` or `#RRGGBB` color — split on commas or whitespace, case kept, short forms not expanded, duplicates kept — until the 20-color limit is reached; anything else is ignored. So you can carry a palette from one summary to another, or from a design tool, in one step. A long list collapses to two rows with a **+N more** / **Show less** toggle. This is a designer convenience only; the stored shape of the summary did not change.

Card interactions

Each summary card also supports:

- **Drill-down (click-to-browse):** clicking a value or chart segment opens the matching entries.
- **Enlarge:** open the card in a larger dialog.
- **Auto-refresh:** the card refreshes its data automatically (change detected via the result `hash` diff).
- **Additional time filters:** narrow the card to *this week*, *this month*, or a *custom date range*.

> In preview mode (see below), dashboards and summaries are fed deterministic **fake data**, so builder previews will not match production numbers.

Endpoints

All entity-scoped routes are authorized by `entityVoterFactory()` at the class level.

Method	Path	Purpose
`POST`	`/v1/entities/{entity}/statistic-queries`	Create a query for an entity
`GET`	`/v1/entities/{entity}/statistic-queries`	List queries for an entity
`GET`	`/v1/entities/{entity}/statistic-queries/count`	Count queries for an entity
`GET`	`/v1/entities/{entity}/statistic-queries/{id}`	Get one (view-aware, see below)
`PATCH`	`/v1/entities/{entity}/statistic-queries/{id}`	Update one
`DELETE`	`/v1/entities/{entity}/statistic-queries/{id}`	Delete one
`GET`	`/v1/statistic-queries`	List all queries in the system
`GET`	`/v1/statistic-queries/count`	Count all queries in the system

Method	Path	Purpose
`GET`	`/v1/statistic-queries/{id}`	Get one as `StatisticQueryDTO`, without an entity in the path
`GET`	`/v1/statistic-queries/by-ids`	`Since 1.10` — the batch form of the route above; see Batch load by id
`GET`	`/v2/entities/{entity}/statistic-queries/{id}/execute`	Execute and return the aggregated result

`{entity}` is resolved by id **or** normalized name within the current space (`getEntityByIdOrNormalizedNameInSpace`).

`GET /v1/statistic-queries/{id}` is the route dashboards use to resolve their elements. Its authorization is against the **query's own** `dataEntity` (voter `voteByLoadedEntityFactory('repositories.StatisticQueryRepository', 'dataEntity', 0)`), because the class-level `entityVoterFactory` would otherwise misread the query id as an entity id and deny a Restricted user. It returns a `StatisticQueryDTO` which additionally carries `relatedData` (`{ dataEntity?, dataModule? }`) with the lite entity/module the query belongs to.

Create and update publish service-bus events (`StatisticQueryCreatedEvent`, `StatisticQueryUpdatedEvent`, `StatisticQueryDeletedEvent`). On create, the query is also auto-attached to relevant dashboards via `DashboardRepository.addStatisticQueryToDashboard`, and each touched dashboard emits a dashboard-updated event.

Batch load by id `Since 1.10`

A dashboard used to resolve each card's definition with its own `GET /v1/statistic-queries/{id}` — one request per card, an N+1 on a busy module portal. `GET /v1/statistic-queries/by-ids?ids=…&ids=…` returns the same `StatisticQueryDTO` (with `relatedData`) for several ids in one request, and the portal now batches every id requested in the same tick into one call per request scope (plain, public view, preview).

- **At most 100 ids per request**, duplicates included; more answers HTTP 400.
- **Authorization is per query, not per request.** The route itself only requires a signed-in role (`SystemViewer` or `Restricted`); each query found is then checked against **its own** `dataEntity` with the same rule as the single-id route. A query the caller cannot read — or an id that does not exist — is **left out of the array** rather than failing the batch. Do not rely on the response order or length matching the request; match by `id`.
- Through a **data view**, a query is kept only if it belongs to the view's entity and passes the same view checks as the entity-scoped read and execute routes (see View-scoped execution); the view's existing allow-list for `/v1/statistic-queries/{id}` already covers the path. In preview mode, fake ids still return their fabricated definitions.
- In the portal, an id the backend leaves out fails **only that card**, with the same error state a 403 / 404 on the single route produced; the rest of the dashboard renders.

Executing a statistic query

`GET /v2/entities/{entity}/statistic-queries/{id}/execute` returns:

```
{
  "result": [ /* StatisticQueryResult[] */ ],
  "count":  123,          // total matching rows
  "query":  { /* FilterExpression | undefined */ },
  "hash":   "…"           // hash of result, for change detection
}
```

- `where` from the request is **narrowed to the query's** `filterField` only (`Utils.extractAllowedFilters`) — any other filter keys are dropped.

- The `query` field is the resolved criteria of the stored query referenced by the source (or `undefined` when the source is not a query).
- `hash` is `Utils.hashString(JSON.stringify(result))`.

Each `StatisticQueryResult` row (`statistic-query-result.model.ts`) has: `category`, `originalCategory`, optional `subCategory[]` (`{ field, result, originalCategory? }`), and `result` (the aggregated value).

Under the hood (`StatisticQueryService.execute`) the service builds a MongoDB aggregation pipeline: it always filters to `state = Active` and excludes `invalidReason = Obsolete`, applies the source-query criteria and the caller `where`, unwinds array/relation fields, `$group`s by the resolved category/ sub-categories, and maps `aggregationFunction` to the Mongo operator (`Count → {$sum: 1}`, `Sum → $sum`, `Min → $min`, `Max → $max`, `Average → $avg`). Category and switch/collection values are resolved to their labels afterwards, numeric categories/results are rounded to two decimals, and rows are sorted by `category` honouring `orderBy` (localized values compared by the user's language).

Date group-by operators map to `$dateToString` formats or arithmetic (`GroupByOperatorEnum`): year, biannual, quarter, month, ISO week (01–53), day of week/month/year, and hour/minute/second buckets. `BiannualOfYear` and `QuarterOfYear` additionally carry the `$year` so results are prefixed as `year.bucket`.

View-scoped execution

When the request is made through a **data view** middleware (a `view` is bound), `checkAgainstDataView` and the view's `dashboardConfiguration` change behaviour:

- The view's `dashboardConfiguration.enabled` must be `true`, otherwise `ViewDashboardConfigurationNotEnabledException`.
- If `statisticQueriesMatchingSource` is set and `overrideSourceForStatisticQuery` is not, the query's `source` must match the view's `dataSource`, otherwise `StatisticQuerySourceNotSameAsViewException`.
- If `overrideSourceForStatisticQuery` is true and the query source is `Store` (all entries), the source is **replaced** by the view's `dataSource` before execution.
- If the view's `dataSource.type === Query` and `mergeViewSourceQueryWithStatisticQuery` is true, the view's source query is merged (AND-ed) into the aggregation as an additional criteria. **Since 1.6.**

The "N Entries" caption counts with the view query `Since 1.4`

When a view merges its own source query into its summaries (`mergeViewSourceQueryWithStatisticQuery`, above), the count behind a card's **"N Entries"** caption now applies that same merged query, exactly as the aggregation and the *Click to browse* list do — so the caption equals the number of entries the user sees after clicking through. Before the fix the caption counted without the view query and could be larger. A **parameterized** view query takes its values from the view's own source options; the summary's own parameter resolution remains the fallback for any name the view does not resolve. This correction was also delivered to the 1.4, 1.6 and 1.8 lines as an update.

Preview mode

Since 1.8 — when the `PREVIEW_MODE_BINDING` is set, the execute route still performs the real entity / statistic-query / view lookups and view-scope enforcement, but the aggregated `result` / `count` are produced by `FakeStatisticQueryGeneratorService` instead of the real aggregation. The `query` field is still resolved from the real referenced query.

Statistic query vs. entity aggregate `Since 1.10`

A **summary** (a `StatisticQuery` — everything on this page) and an **entity aggregate** are two different resources. They are easy to confuse because both end up computing a Count, Sum, Min, Max or Average over an entity's entries, and both draw on the same set of five functions. The model for aggregates — the field-type rules, how a view enables one, how grouping works, and how protected fields behave — lives on Entity Aggregates & Entry Grouping [Build Apps › Totals & grouping] and is not restated here. This section only draws the boundary:

	Summary (statistic query)	Entity aggregate
Where it is authored	On a dashboard, or the entity's dashboard tab	On the entity's own **Aggregators** tab
How it is computed	Its own execute endpoint, one query at a time	The entries-list summary endpoint, alongside every other aggregate that view enabled
What it produces	A standalone card — a number, a list, or a chart, with its own category breakdown	Figures on an entries list: a grand total, and a subtotal per group when the list is grouped
How it is grouped	Its own category / date-bucket grouping, independent of any view	The view's own entries-list grouping, shared by every aggregate on that list
Lifecycle	Owned by the dashboard that carries it; removing the element can remove the query	Owned by the entity; deleted on its own, and dropped leniently from any view that had enabled it

Choose a summary when you want a card that stands on its own and can sit on a dashboard. Choose an aggregate when the figure belongs to an entries list the reader is already looking at.

Sensitive-data access policy on a summary `Since 1.10`

A summary can now carry a **sensitive-data access policy**, the same setting reactions and export templates carry: it decides whether this summary may read protected values at all, always, or only according to the person running it. The control appears on the summary's own form, and only when the entity actually declares a sensitive field. See Sensitive Fields [Build Apps › Protect a field] for what each policy means.

Three things are specific to summaries:

- **An unset policy behaves exactly as before.** Every summary authored before this setting existed keeps its previous behaviour; nothing was migrated or re-evaluated.
- **The "Auto" option is not offered.** A summary has no scheduled or automatic run that is distinct from someone opening it, so there is nothing for that policy to distinguish; saving it is rejected.

Duplicating a summary `Since 1.10`

Every summary card in the dashboard builder carries a **Duplicate** icon next to Enlarge, alongside the duplicate affordance the reactions and views lists already offer. Duplicate opens the summary editor in **create** mode against the original's id — it is never an update to the original, and the original is untouched whether you save or cancel.

- **Everything is carried over except the name** — source, aggregation, grouping, display type, colors, `showInEntityOverview`, description, and the sensitive-data access policy above.
- **The suggested name** is the original's name with a localized "Copy" suffix appended, in the language you are currently working in — if the original has no text in that language, the suffix is applied to whichever language it does have, rather than proposing an empty name. You can still edit the suggestion before saving, and the usual name-uniqueness check applies.
- **Duplicate follows the same permission gate as Edit** — if you cannot edit a summary, you cannot duplicate it either.

Dashboards

A dashboard is a `DataEntry` stored in the `dashboard` collection. Its `value` (`DashboardEntryValue`) holds the ordered `elements` array and describes what the dashboard is attached to. It presents a set of cards (summaries), organized into groups that can be rearranged by drag-and-drop.

Data model

`DashboardEntryValue` (`dashboard-entry-value.model.ts`):

Field	Type	Notes
`id`	UUID	
`name`	LocalizedValue	Dashboard name
`normalizedName`	string	Auto-generated from name; unique per `relatedTo` scope
`elements`	(DashboardElement | DashboardGroup)[]	The ordered contents
`relatedTo`	ReferenceProperty	What the dashboard is attached to (see below)
`sharing`	ApplicationSharing	Sharing settings

`relatedTo` determines the dashboard's scope:

- **undefined** → a **system** dashboard.
- `{ dataEntity: 'DataModule', dataEntry: '<module-uuid>' }` → a **module** dashboard.
- `{ dataEntity: 'DataEntity', dataEntry: '<entity-uuid>' }` → an **entity** dashboard.

`checkSupportedRelatedTo` rejects any other `relatedTo.dataEntity` with a `DataNotSupportedException` ("Only related to DataModule or DataEntity is supported").

Fixed — dashboard fields no longer vanish on save `Since 1.8.` The `DASHBOARD` entity now declares `name`, `elements` / `relatedTo`, and `sharing` as real fields on its schema, not just `normalizedName`. Before this fix, a fresh install's system dashboard carried no field definitions beyond `normalizedName`, so the generic write path silently stripped every other key — including `name` — the moment the dashboard was first saved. Because a client always round-trips whatever `value` it last fetched, the very first edit to a brand-new dashboard (adding a summary, adding a group) failed with a validation error demanding the `name` that had just been stripped away. A migration repairs existing installs, restoring the fields on any dashboard that lost them without touching dashboards that still have a name.

Each element is one of two shapes (`elements` is stored as loose objects and discriminated by `elementType`):

`DashboardElement` (`elementType: 'Element'`) — `dashboard-element.model.ts`:

Field	Type	Notes
`id`	UUID	Unique within the dashboard
`elementType`	`'Element'`	Fixed
`element`	ReferenceProperty	`{ dataEntity: 'StatisticQuery', dataEntry: '<query-uuid>' }`
`group`	UUID	The `DashboardGroup` this element belongs to (optional)
`index`	number	Position in the dashboard

`DashboardGroup` (`elementType: 'DashboardGroup'`) — a named block that other elements can be nested under: `id`, `name` (localized, required), `description`, `icon`, `index`, optional `group` (a group may nest under another group). In the builder, related summaries are dragged into a group and the group is given a distinctive name so it renders as a single block.

Dashboard levels

A dashboard can be built at three scopes:

- **Entity level** — the entity's **Dashboard** tab holds its summaries; you can add, modify or delete a summary or group there, and it can also be shown from the gallery/view settings.
- **Module level** — the module's **Dashboard** tab aggregates summaries from the entities of the module; when adding a summary you pick the entity to pull a pre-built summary from.
- **System level** — the system dashboard (`relatedTo` undefined) is edited at **Administration › Applications › Dashboards**. A page can show a dashboard's summaries through its **Summary** widget.

Default dashboard provisioning

Dashboards are never shipped as seed or fixture JSON — no seed files create system entities. Instead they are provisioned **reactively, event-driven** via node-ts service-bus handlers (`data-service/src/handlers/events/`):

- **Per new data entity** — `DefaultDashboardDataEntityCreatedHandler` creates one default dashboard for each newly created `DataEntity`: `relatedTo` the entity, an empty `elements` array, sharing level **Private**, `spaces = ROOT_SPACE_ID`, and state `Active` (`name.en = 'dashboard'`).
- **Per new module** — `CreateDefaultDashboardOnModuleCreatedHandler` creates a default dashboard `relatedTo` the module on `DataModuleCreatedEvent` (skipped if one already exists), alongside the default module page/generic view.

Both handlers run `CoreEntryNormalizedNameService.ensureForCreate` and publish a `DashboardCreatedEvent`. This is why `DashboardController` exposes no create endpoint (see below) — the initial dashboard already exists by the time a user opens the Dashboard tab.

Endpoints

Method	Path	Purpose
`GET`	`/v1/dashboards`	List dashboards (access-right filtered, or view-scoped)
`GET`	`/v1/dashboards/{id}`	Get one dashboard
`PUT`	`/v1/dashboards/{id}`	Replace a dashboard's contents

There is **no create or delete** endpoint on `DashboardController` — dashboards are provisioned/managed elsewhere; this controller only reads and updates. `PUT` is authorized by `voteByRelatedToFactory('value.relatedTo', 1)`.

`GET /v1/dashboards` behaves differently depending on context:

- **No view** → `DashboardRepository.findWithAccessRight(userId, filter)` returns the dashboards the logged-in user can access (English collation).
- **Through a data view** → the view must be a `DataViews` instance (`InvalidDataViewException` otherwise) with `dashboardConfiguration.enabled === true` (`DashboardConfigurationNotEnabledException` otherwise), then `DashboardService.getDashboardForEndUsers` returns only the view's allowed dashboards, with elements optionally filtered by matching source (see below).

`GET /v1/dashboards/{id}` validates the id is a UUID (`InvalidDashboardId`) and returns the single access-right-filtered dashboard.

Update rules (PUT)

`updateById` performs a full replace of `value.elements` and applies extensive validation:

- **id validity** — the path id must be a UUID.
- `relatedTo` is **preserved** from the existing dashboard (clients cannot re-target a dashboard), and `sharing` is forced to `Public` with no spaces.
- **`normalizedName` is also fully server-owned** `Since 1.8` — alongside `relatedTo` and `sharing`, saving a dashboard now takes `normalizedName` from the persisted record unconditionally; a client has no business rewriting the identifier its uniqueness is built on. Previously a save silently erased it, because the client's model never carries that field and the update replaces the dashboard's contents outright. `name` stays something you supply, but it now falls back to the previously saved value when a save omits it, so a client that doesn't round-trip the name can no longer erase it.
- **Unique element ids** (`checkElementsMostHaveUniqueId`) — every element/group must have a valid UUID `id`, with no duplicates (`InvalidElementID`).
- **No duplicate statistic query** (`checkElementsMostHaveUniqueElement`) — the same `element.dataEntry` cannot appear in two elements (`DashboardElementUsedMoreThanOneException`).
- **Group integrity** — every element's `group` must reference a group present in the dashboard (`InvalidDashboardGroup`); group names must be non-empty for all languages and unique across languages (`EmptyLocalizedException` / `DashboardElementDuplicateException`).
- **Statistic-query scope** — for module/entity dashboards, every referenced statistic query's entity must belong to the dashboard's module/entity, else `DashboardStatisticQueryNotInModule` (collected and thrown as `DashboardValidationException`).

Permission checks (`checkUserPermission`):

Dashboard scope	Who may edit
System (`relatedTo` undefined)	System admin or system developer only (`SystemDashboardEditNotAllowedException`)
Module	System admin/developer, or a module editor (`EditModuleDashboardNotAllowedException`)
Entity	System admin/developer, a module editor, or the entity editor (`EntityDashboardEditNotAllowedException`)

A user with no system access at all (`isSystemNone`) is rejected with `DataNotSupportedException` .

Cleanup of orphaned statistic queries — for **entity** dashboards, when elements are removed on update, `handleUnusedStatisticQueries` deletes the statistic queries backing the removed elements, but only if that query is not still referenced by another element in the dashboard. This runs only for interactive users; **service-to-service calls** (e.g. packaging import) skip this cleanup, because the import manages statistic queries independently and element ids may still reference source-environment ids.

View source filtering of elements

`DashboardService.getAllowedDashboardSameSourceElements` (only when `dashboardConfiguration.statisticQueriesMatchingSource` is set) filters each dashboard's elements down to statistic queries whose `source` matches the view's `dataSource` — dashboard groups are always retained, and if the view's data

source type is `Store` (all entries) every query is allowed. Access to a dashboard through a view is gated by `getDashboardQueryFilter` against the view's `dashboardConfiguration.dashboards` allow-list, throwing `DashboardAccessNotAllowedException` (or a plain `Error`) for a disallowed id.

View dashboard configuration reference

`DashboardConfiguration` (on a `DataViews`) controls all of the above:

In the view designer: `enabled` is **Dashboard**, `clickToBrowse` is **Click to browse**, `statisticQueriesMatchingSource` is **Summaries to show** (**Show all summaries** or matching-source only), `overrideSourceForStatisticQuery` is **Apply view's source query to summaries that have all entries as a source**, `mergeViewSourceQueryWithStatisticQuery` is **Append view's source query to summaries that have different query as a source**. `isDefaultLandingPage` is deprecated and has no switch in the view designer: the screen a view opens on is set under **Behavior** › **Default landing screen**. A view exposes a counter through its **Counter view** display (`MiniCard` in `availableDisplays`).

Property	Default	Meaning
`enabled`	—	Dashboards enabled for this view (requires `viewConfiguration.enabled`)
`clickToBrowse`	false	Whether clicking a statistic query drills into entries
`dashboards`	—	Allow-list of dashboards exposed through the view
`statisticQueriesMatchingSource`	—	Filter dashboard elements to queries whose source matches the view source
`overrideSourceForStatisticQuery`	false	Replace a `Store` query source with the view source at execution
`mergeViewSourceQueryWithStatisticQuery`	false	AND the view's source query into the statistic query (used when matching-source is off) — **Since 1.6**
`showEntityOverview`	—	Show entity overview in the entries list
`isDefaultLandingPage`	false	**Deprecated** — dashboard as the view's landing page

Counter widgets

A counter widget is a small "count of entries" tile bound to a **data view** and, optionally, a **query**. It is a `DataEntry` stored under the entity name `counter_widget` (`StringConstants.COUNTER_WIDGET_ENTITY_NAME`). **Long-standing** — the counter-widget entity predates the 1.2 milestone.

Data model

`CounterWidgetDataEntryValue` (`counter-widget-data-entry-value.model.ts`):

Field	Type	Notes
`name`	LocalizedValue	
`normalizedName`	string	Auto-generated, unique per related entity
`description`	NullableLocalizedValue	
`icon`	Icon	
`moduleId`	string	The owning module
`dataEntity`	string	The entity whose entries are counted
`view`	ReferenceProperty	`{ dataEntity: 'DataViews', dataEntry: '<view-uuid>' }`
`displayOrder`	number	Sort order among widgets
`query`	ReferenceProperty	Optional query used to filter/mark entries

Creation and validation

In the UI, counters are managed at **Administration › Applications › Counters** (**New Counter**: **Name**, **Icon**, **View**, **Counting Criteria**, **Description**). Over the API, counter widgets are **created and updated through the generic data-entry CRUD** for the `counter_widget` entity — `CounterWidgetController` itself only exposes read endpoints. `CounterWidgetValidator` runs on the `CREATE` and `UPDATE` phases and enforces:

- A view must exist matching `view.dataEntry` **and** the widget's `dataEntity` **and** `moduleId` (`DataViewNotExistException` otherwise).
- The view's `viewConfiguration.enabled` must be true (`EntriesConfigurationMustBeEnabled`).
- The view must expose a **MiniCard** display in `availableDisplays` (`CounterViewMissedMiniCardException`).
- If a `query` is set, it must exist for the same `dataEntity` (`DataQueryNotFound`).

Endpoints

Method	Path	Auth	Purpose
`GET`	`v1/counter-widgets/end-user`	any authenticated user (`autho-rize: skip`)	List the widgets visible to the user in the space
`GET`	`v1/counter-widgets/{id}/count`	view middleware voter	The current count for a widget
`GET`	`v1/counter-widgets/{id}/entries`	view middleware voter	The entries behind a widget (with match flags)
`GET`	`v1/counter-widgets`	system admin voter	Admin listing as `CounterWidgetDataEntryDto` (resolved view/module names)

> Note: the counter-widget routes are registered **without** a leading slash (`v1/counter-widgets/...`), unlike the dashboard and statistic-query routes.

The `{id}/count` and `{id}/entries` routes require the request to come through a view middleware (`onlyViewMiddlewareVoter`) and additionally check that the widget's `value.view.dataEntry` equals the current middleware view id, throwing `CounterViewNotMatchedWithMiddlewareException` on mismatch.

End-user listing

`getCounterWidgets` (behind `/end-user`) returns widgets that are:

- in a module of the current space,
- bound to a view that is **both** allowed for the user **and** present in the space (`intersection` of user-allowed and space views),
- `state === Active`,

ordered by `value.displayOrder DESC`.

`Since 1.4` — the listing no longer loads every view of the tenant to compute that intersection. It fetches the candidate widgets first (the space's modules × the user's entitled views, `Active`, ordered by `displayOrder`) and then checks only **their** view ids for space visibility; a user with no entitled views gets `[]` immediately. The result is unchanged, but the cost no longer grows with the tenant's view count (the fix measured the view lookup falling from about 1.9 s to about 2 ms on 3,000 views). Also delivered to the 1.4, 1.6 and 1.8 lines as an update.

Counting and entries

`getCounterCount` loads the widget's view, verifies entry access (`checkCanAccessEntries`), resolves the entity in the space, and — if the widget has a `query` — wraps that query's `criteria` in an `AND` `FilterExpression` before delegating to `DataSourceService.countV2`. Returns `{ count }`.

`getCounterEntries` similarly loads view + entity, forces `order = ['value.updateInfo.doneAt DESC']`, fetches entries via `DataSourceService.findV2`, and returns `CounterWidgetDataEntryEndUserDto[]`. Each DTO adds a boolean **`isIncludedInQueryResult`**: when the widget has a query, the service spins up a temporary collection, inserts the fetched entries, runs the parsed query criteria as a `$match`, and flags which entries matched — then drops the temp collection. Entries that do not match the query are still returned but flagged `false`. **Long-standing** — the `isIncludedInQueryResult` flag predates the 1.2 milestone.

Admin listing DTO

`GET v1/counter-widgets` (system admin only) runs a MongoDB aggregation directly: it applies text search / filter parsing, a default `_id` sort (plus any requested order), skip/offset/limit, resolves `creationInfo.doneBy` and `updateInfo.doneBy` from the entry index, and joins the view and module names, returning `CounterWidgetDataEntryDto` objects (widget + `viewName` + `moduleName`).

Quick reference: enums

Source: `packages/osos-shared/src/models/enums.ts`.

Enum	Values (index)
`DataCollectionTypeEnum`	0 Store, 1 Collection, 2 Query
`AggregationFunctionEnum`	0 Count, 1 Sum, 2 Min, 3 Max, 4 Average
`StatisticsQueryDisplayEnum`	0 Number, 1 List, 2 Chart
`ChartTypeEnum`	0 Column, 1 Bar, 2 Line, 3 Pie, 4 Donut
`SortDirEnum`	0 Asc, 1 Desc
`GroupByOperatorEnum`	0 FullDate, 1 Year, 2 Biannual, 3 BiannualOfYear, 4 Quarterly, 5 QuarterOfYear, 6 Month, 7 MonthOfYear, 8 Week, 9 WeekWithMonth, 10 WeekWithYear, 11 WeekWithMonthAndYear, 12 DayOfWeek, 13 DayOfMonth, 14 DayOfYear, 15 Hours, 16 Minutes, 17 MinutesAndHours, 18 Seconds, 19 SecondsAndMinutes, 20 SecondsAndHours

Workflow participation reference

This page is for integrators and developers who drive workflow steps on behalf of users, or who need to explain exactly why a participant's action was accepted or refused. It is the technical layer behind the end-user guide Acting on Workflow Steps [Using OSOS]. The workflow definition model, the action and owner configuration, step-up verification, external steps and the concurrency guard are documented in Workflow model and API reference; this page does not repeat them. Error codes and retry advice for every service are in Errors and retries.

The status panel

The portal reads the entry's workflow status and renders it in the entry's **WorkFlow** side tab. It never asks the user to call the backend.

Step states (`MainFlowStatusStepTypeEnum`)

Each step in the status carries a type that places it relative to "now":

State	Meaning
`Executed`	The step was completed; it carries who acted, when, and any notes.
`CurrentMain`	The active step, on the workflow's main path.
`CurrentAlternative`	The active step, reached through a branch off the main path.
`Pending`	A future step that has not been reached yet.
`Skipped`	A step that was jumped over, for example by a Move to Step.
`ReExecuted`	A step that was reached and acted on more than once.

Per step, when the page permits it: **owner** (populated once the step is reached), **executed by / executors** with timestamps and notes (several people for a group-owned step), **received at / executed at**, and **notes**.

Owner visibility (`ShowStepOwnerModeEnum`)

A view can hide step owners entirely, show **all** owners (`AllOwners`), or show only the steps **owned by the current user** (`OwnedByMe`). Anonymous viewers never receive owner or executor identities.

Main and execution modes

The status is requested in one of two modes (the `mode` query parameter):

- `main` — the linear happy path: executed steps, the current step, then the predicted upcoming steps. This is what the WorkFlow tab shows by default.
- `execution` `Since 1.4` — the full execution tree, including how many times steps were revisited and how sub-workflows nested inside the parent. The WorkFlow tab's **Show History** switch requests it.

`Since 1.8` the view configuration chooses which mode the tab opens on (**Workflow path** or **Workflow history** in the view designer); the switch is only offered when the view enables the execution tree. With no configured default, or with the execution tree turned off, the tab opens on the path. The mobile app follows the same view setting.

The status also reports whether the remaining path is **determined**. If the current step's main action branches on a condition evaluated only at run time, the upcoming steps cannot be predicted and the status says so.

Group-owned steps list every member

For a step owned by a group or a list of people, the status and the timeline list every member with their own state: executed and timestamped for whoever acted, not executed for the rest. The platform recovers the owning group from the step's configuration even after the step has narrowed to the single executor. This is the behaviour on every supported line (1.4, 1.6 and 1.8), not a new capability. Model detail: Group ownership on a group-owned step.

Execution history after a reset `Since 1.8`

After a **Reset workflow**, the execution history contains only the current run. Before 1.8 only the current-step display honoured the reset, so a reset entry could show the same step twice in the execution view: once as `Executed` from the earlier run and again as `CurrentMain`. A step can no longer be both.

What the engine checks when a participant acts

An action is processed by the workflow engine, which enforces these rules:

Rule	Refusal
The caller must own the active step. Group-owned steps accept any member. The final Exit step is exempt.	`InvalidWorkflowOwner`
On a parallel step (several owners must each act), an owner may act once.	`ParallelStepOwnerAlreadyExecutedStepException`
If the action's confirmation dialog is on and its comment mode is `Required`, a note must be supplied. Comment modes are `Optional`, `Required` and `Disabled` (shown to builders as **Hidden**).	`WorkFlowActionCommentRequired`
An action that is conditionally hidden for this entry cannot be executed. `Since 1.8` the refusal carries its reason instead of an unexplained server error; it answers 422.	`WorkFlowActionExecutionNotAllowed` (`WORKFLOW_ACTION_EXECUTION_NOT_ALLOWED`)
The entry is validated against the step's requirements before the transition is committed. When required fields are empty, the portal routes the user to edit the entry first.	validation errors

On success the engine records the step in the history (who, when, notes), advances the entry to the target step, resolves the next owner and notifies the portal. If the target is a sub-workflow, the parent step pauses until the sub-workflow completes.

Bulk execution

The bulk route applies one action to several entries. All selected entries must be on the same step, otherwise the whole call is refused (`WorkflowStepsNotSameException`, long-standing). Each entry is then processed independently and the result is reported per entry; the portal asks the user to re-take a failed one from its detail page.

`Since 1.10` running a step action on entries selected in a list no longer fails as an access error. Before 1.10 it was refused on every page, whatever the action. A gated action across a selection takes one verification for the whole selection; before 1.10 a selection with more than one gated entry was refused with a warning to act on one entry at a time. Route and body: Step-up verification before executing an action.

Lost races and stale saves

`Since 1.10` every workflow write is conditional on the state it was validated against; the loser of a race answers `409 UPDATE_CONFLICT`, nothing is recorded, and the web client opens its **This record has changed** dialog. Before 1.10, an Approve racing a Reject could both report success while the automations behind them ran twice. Field edits made in between outside the workflow are kept, and a step-up proof survives a lost race. Archiving an entry someone already

archived fails with `ENTRY_ARCHIVED` (*The data entry is already archived*) instead of an unexplained error. Full contract: Concurrency guard on workflow writes.

`Since 1.4` an automation that moves an entry to another step is no longer silently undone. Before, saving the entry from a copy loaded before the automation ran could put the entry back on its old step while the automation reported success. A save now leaves the workflow position alone, and the automation moves the entry from the step it is on now.

Ownership

Reassign (change owner)

A participant reassigns a step to a specific person through **Reassign Owner** (owner-initiated) or **Change Owner** (an administrative grant on the view). Allowed callers:

- anyone with **edit** permission on the entry, or
- the **current owner** when the step's design allows changing owner (`allowChangeOwner`), or a member of the step's original owner group.

Refusal	Code
The target already owns the step and nothing else changed	`UserAlreadyOwnException` (shown as *This user already own this step.*)
Caller is not permitted to change this step's owner	`ActionNotAllowed` / `CanNotChangeOwnerException`

A reassignment can be **permanent** (`permanentChange`): the new owner sticks if execution returns to this step later; otherwise the change applies to the current visit only (long-standing). In the portal the permanent option is offered in the administrative **Change Owner** dialog (**Change Type**: **Temporary** / **Permanent**), according to which of the two change-owner grants the view enables; the owner's own **Reassign Owner** dialog makes a temporary change.

`Since 1.10` the note left with a reassignment or owner change is shown in the entry's discussion. See Comments and mentions reference.

Eligible owners

The owner picker offers a searchable, paged list of eligible owners. A candidate is dropped when it is archived (not in the Active state — users, groups and org units), a deactivated user account, or an obsolete or invalid entry.

Take ownership

A step owned by a **group** or **list** (a single-user selection whose selection type is *Take Ownership*) with **no individual owner yet** can be claimed with **Take Ownership**. The claim requires:

- the step is not already owned by an individual, else `CURRENT_STEP_ALREADY_OWNED` (answered **422** `Since 1.8`), and
- the caller belongs to the step's original owner group or collection, else `CanNotChangeOwnerException` .

`Since 1.8` corrections to the portal's **Take Ownership** button:

- A step that already has an owner never offers the button, including a step owned by a group that hands the work to its members one at a time, where the first member already holds it.
- The check follows the active step. Inside a sub-workflow it looks at the active step rather than the outermost one, so membership of the group owning a parent step no longer offers a claim on an unrelated sub-workflow

step.

- Membership of a **query group** counts the same as membership of a regular group, and the step's actions appear accordingly.
- Pressing the button no longer comes back with a bare server error with no message: it succeeds or names the reason.

Unowned group-owned steps `Since 1.8`

A step handed to a group with **single user** / **manual** selection can sit unowned on purpose: after someone reassigned it back to the group, or because a change-owner automation resolved to "the group is eligible, but no single person" — the rule now leaves the step unowned instead of failing. **Take Ownership** is offered when the step has no owner, its recorded original owner is a group, query group or data collection (not a single person), its owner configuration is single user with manual selection, and the caller is a member. The check is re-evaluated against the step's current configuration, so an automated owner change can no longer hide the action from every member. Model: Unowned group-owned steps.

Acting without taking ownership `Since 1.10`

On a view whose admin actions enable **Execute action on behalf step owner** (`executeAsAdmin`), a member of the group owning a *Take ownership* step does not press **Take Ownership** first:

- The entry header shows the step's own action buttons. Pressing one claims the step silently, then runs the action; if the claim fails, the action is not run.
- On a view where that option is off, **Take Ownership** is a split button: every other header action the user can see (Back, Edit, Duplicate, custom and template buttons) sits in its dropdown, with the view's admin actions as their own submenu when it enables any.
- If someone else takes the step first (*step already owned*), the entry is reloaded; on an edit page the unsaved values are kept.

Related corrections in the same release: for a workflow entry the header shows the step's owner, not the viewer; a system administrator again sees **Archive** in the entries list for a workflow entry whose current step they do not own; and the header no longer stays loading when no main action applies (for example an archived entry the user can neither restore nor delete).

Acting on behalf of an owner

A manager can act on behalf of the assigned step owner without reassigning the step first. The step is recorded against the manager who acted, so a later step that resolves its owner from this one *by execution* follows the manager, not the originally assigned owner.

Delegation

When a delegate acts, the delegation travels with the command: `delegateeId` and `delegationId` are carried as sticky attributes on every workflow command, so each action is stamped with who acted as delegate and under which delegation. The entry UI resolves the delegatee display values, and the current-step details and history show the delegate as the actor (the **On behalf of** badge).

Ad-hoc steps

A step owner can insert an out-of-band step without changing the design, when the step allows it (`allowAddAdHoc`). Two types exist (`AdhocStepTypeEnum`):

Type	Purpose
`AdditionalApproval`	Route the entry to another person for an extra sign-off.
`AdditionalInformation`	Send it to someone to gather more information.

The portal dialog asks for the type, a **User** or **Group** owner (eligible-owner picker with search, `Since 1.8`) and an optional **Note**. There is no fallback owner. The ad-hoc step is added as a sub-step of the current step; the workflow pauses there until its owner acts, then returns to where it left off. Only the current step owner can open the picker from the end-user portal. `Since 1.8` the note is saved on the created step as its description and shown later as a tooltip next to it. An ad-hoc insertion computes from its snapshot and fails with a conflict rather than reverting a concurrent change.

SLA status

A step can carry a time-to-execute target. Each step's SLA is tracked with a status (`SLAStatus`):

Status	Meaning
`Pending`	Within its allowed time.
`Achieved`	Completed within its target.
`Overdue`	The target elapsed before completion.

Overdue tracking drives escalation and reminders. The overdue check runs once a minute, so detection can lag by up to one minute (long-standing). The portal marks a breached step **Exceeded Time** in the entry's History tab. Configuring targets: Workflows (Design) [Build Apps › Workflows & approvals].

Change-step (admin) actions

Change-step actions are offered only when the view enables them, each gated by its own flag. If the flag is off, the call is refused with `ForbiddenWorkflowAdminActionException` . Modes (`ChangeStepModeEnum`):

Portal label	Mode	View flag
Return to previous step	`ReturnToStep`	`returnToStep`
Move to Step	`MoveToStep`	`moveToStep`
Reset Workflow	`ResetWorkflow`	`resetWorkflow`
Close workflow	`EndWorkflow`	`closeWorkflow`

Return to step sends the entry back to an earlier, executed step; *Move to step* jumps it forward and marks the skipped steps `Skipped` ; *Reset workflow* clears executed steps and starts over (not offered while the entry is on its initial step: *Can not reset workflow if current step is initial step*); *End workflow* closes it. Each accepts an optional note. The engine's movement rules still apply: forward-only manual moves, return only to executed steps, no direct move onto the Exit step, and movement only within the entry's open workflows.

Stronger owner capabilities a view can grant: **change owner as admin** (temporary), **change owner as admin permanently**, and **execute as admin** (act on a step the caller does not own). Without them the caller is held to the

normal owner rules. An administrator executing as admin is not asked for step-up verification.

`Since 1.8` every change-step action and **Change Owner** is shown but disabled while the current step is an External Step, so it cannot skip the external system's mandatory report-back; the backend rejects manual completion of an External Step regardless. See External step.

The admin-view override applies only to non-participants `Since 1.6`

A view whose admin actions enable **Edit entry as admin** or **View entry as admin** lifts the step's field rules (Hidden, Locked, Mandatory). Because the grant covers the whole view, it used to lift them for the person working the step on that view too. It now applies only to someone who takes no part in the current step — not its owner, not a member of the owning group, collection or organisation unit, and not one of an unclaimed step's original owners. For a participant, the step's rules apply everywhere on that view (details, edit form, editable fields, history, the list's hidden cells) and on save; a new entry created there follows the first step's rules. Also delivered to the 1.6 and 1.8 lines as an update. Model: The admin-view override applies only to non-participants.

Aggregate entries after the source workflow finishes `Since 1.8`

Aggregate (virtual) entries are projected from a source entry. Once the source entry's workflow reaches a step where editing is no longer allowed (its terminal step), the **Edit** action on the aggregated row disappears both in the entry header and as the row action in an aggregate list. Before 1.8 the action was still offered and only redirected to the source entry's already read-only edit page. Individual fields were already locked; only the action had not caught up. Design side: Virtual entities [Build Apps].

Messages and codes

What the participant sees	Code
Not the owner	`InvalidWorkflowOwner`
Already acted on a parallel step	`ParallelStepOwnerAlreadyExecutedStepException`
Comment required	`WorkFlowActionCommentRequired`
Action not allowed for this entry	`WorkFlowActionExecutionNotAllowed`
Step already owned	`CURRENT_STEP_ALREADY_OWNED`
This user already own this step.	`UserAlreadyOwnException`
Not authorized to change the owner	`ActionNotAllowed` / `CanNotChangeOwnerException`
Admin action forbidden by the view	`ForbiddenWorkflowAdminActionException`
Manual completion not allowed (External Step) `Since 1.8`	see External step
Return link invalid or expired (External Step) `Since 1.8`	see The return contract
This action requires additional verification of your identity before it can be executed. `Since 1.10`	step-up required
This record has changed `Since 1.10`	`409 UPDATE_CONFLICT`
Selected entries are on different steps	`WorkflowStepsNotSameException`

Removed `Since 1.10`: the "not available for several entries at once" warning for a gated bulk selection. A single verification now covers the selection; an oversized selection shows *This action requires additional verification, which can cover at most N entries at a time. Select fewer entries and try again.*

Comments and mentions reference

This page is for integrators and developers who read or write comments and reactions through the API, or who need to explain exactly what a user sees in a record's **Discussion** tab. It is the technical layer behind the end-user guide Comments & Collaboration [Using OSOS]. The generated endpoint reference is in Comment — comments-service and Reaction — comments-service; the call token service is documented in Real-time (sockets and RTC).

Storage and ordering

A comment is always attached to one record: it stores a `relatedTo` reference made of the record's entity (`dataEntity`) and the record's id (`dataEntry`). Threads are loaded per record, so a comment never appears on another record. Comments are returned **oldest first**, ordered by the time each was made. Each carries the text, the author (`commenter`), attachments, the audience if any, and the time.

Two places use the service:

- **The record's Discussion tab**, when the view has discussions enabled. This is the general case for any entity.
- **Timeline posts.** For the built-in `POSTS` entity, comments and reactions go through the same service, gated by whether the timeline display is enabled in the view.

Alongside the thread a client can request an **attachment list** (only the files attached across the record's comments) and a **comment count** (for badges and counters).

`Since 1.8` the author, the directed audience and, for comments posted on someone's behalf, the delegate are resolved by the service before the thread is returned, so names and pictures render without an extra lookup.

`Since 1.10` a comment posted by a visitor who was not signed in (on a public view) has no author account. It is shown as **Anonymous** and nobody can change its audience. Such a comment displays alongside the rest of the public record page and the signed-out visitor's workflow-action dialog on that page.

Workflow notes in the thread

When the view's **Workflow comments** setting (`showWorkflowComments`) is on, each workflow step note (who executed it and when) is merged into the thread as a read-only comment labelled with the step it came from (long-standing).

Owner-change notes `Since 1.10`

A note left when **reassigning** a step or **changing its owner** is folded into the same thread under the same setting; before 1.10 it was not. An owner change is not a step execution, so it was never part of the step history the thread is synthesized from. The platform reads it from a narrowly-scoped source, `GET …/entries/{id}/owner-change-notes` , which returns only owner-change records that carry a note, and renders it like a step note.

Because the note is assembled on read rather than stored as a comment, posting it does **not** fire the real-time "comments updated" signal that a normal comment post does. A Discussion tab already open in another window picks it up only when that view re-fetches the thread (reopening the entry, switching tabs and back, or the surface's own refresh).

Writing and editing

A new comment accepts three things:

What the client provides	Field	Required
The message text	`commentText`	No (may be empty or null)
Files to attach	`attachments` (each with `id`, `name`, `mimeType`, `size`, optional `thumbnail`)	No
People or groups to direct it to	`audience` (references; split internally into users and groups)	No

The commenter and the time are stamped by the service; the client never sets them. The comment is linked to the record in the route.

Editing is limited to the comment's own author; any other caller is refused. Only the **audience** can change; the text is not changed by the edit operation. An empty audience re-opens the comment to everyone (subject to the private-comment rule below).

Deleting. There is no end-user delete in the comments API. Comments are managed through the platform's data lifecycle: a removed comment is in an archived state and is filtered out of threads. Reactions, by contrast, can be removed directly.

Attachments

Attachments are validated against the view's attachment settings. There is no fixed list of accepted types; the allowed types are configured per view.

- **Size.** Each file must be within the configured maximum. If the view sets none, the default cap is **1 MB (1,048,576 bytes)** per file.
- **Types.** Files are checked against the view's allowed-extensions list. If the view enables attachments but configures **no** allowed extensions, files are rejected. An entry can be a file extension (`.pdf`), an exact MIME type (`image/png`) or a wildcard MIME type (`image/*`).
- If the view does not enable attachments, any file is rejected.

A failing file rejects the whole post, with an error naming the attachment by position. The portal shows *Failed to upload the file. Please ensure it meets the upload requirements and try again.*

`Since 1.10` an attached file is held in temporary storage and kept only once the comment is posted; a file attached to a comment that is never posted is not kept. See Files: storage rules.

Mentions

When a view allows mentions, the portal's comment box opens an autocomplete on **@**. The mentions autocomplete searches **active** users and groups by group name and person full name (in each configured language) and by a person's email address or username. Results are capped at a handful of suggestions.

Who can be suggested depends on the view's **mentions-list mode**, set by the builder:

Mode	Who can be mentioned
Everyone	Any active user or group.

Mode	Who can be mentioned
Based on view permission	Only users and groups with access to the view (used when the view is restricted).
Custom — Fixed	A fixed list chosen by the builder.
Custom — Dynamic	Users and groups taken from fields on the record, for example an assignees field.

Anonymous (not signed-in) visitors cannot mention anyone, even where mentions are enabled.

`Since 1.8` builders and administrators adding a comment from the back office (an entry's sidebar in Administration, not the portal) use the same mentions and audience lookup. Before this fix a back-office search always returned a "you don't have access permission" error, because only the portal's view-based check existed and it does not apply outside a view. A back-office search now requires the caller to hold at least **view** access somewhere in the back office, through their user or a group.

How a mention is stored

On post, the service scans the text for mention markers of the form `@<uuid>` (a person or group id wrapped in `@<…>`), resolves each to a real user or group, and stores three lists on the comment: the combined **mentions**, the **mentioned users** and the **mentioned groups**. The last two drive notifications.

`Since 1.10` a mention that can no longer be resolved (an account or group since deleted or deactivated) renders as `@Unknown` (Arabic "غير معروف") instead of breaking the page. Before, such a mention blanked the whole entry view until a reload. Mention names are also resolved the moment the comment is posted, so a name just mentioned appears immediately.

The mention notification

The platform ships a built-in trigger on comments that runs **Send Notification** to the mentioned users and groups on the in-app channel, titled **Mention**, with the message **{{person}} has mentioned you in a comment**, where `{{person}}` is the author's name. Mentions, the autocomplete and the list modes are long-standing.

`Since 1.10` on an installation that hosts several organizations, a mention could previously notify **nobody**: the comment entities and the built-in mention trigger were never set up per organization. They are now created for every organization, existing ones and any added later. Nothing changed in how users write or mention.

Audience (directed comments)

A comment can be directed to an **audience**: users and/or groups, stored split into users and groups. A directed comment is visible only to its author, the targeted users and the members of the targeted groups. A comment with no audience is visible to everyone who can see the thread. The portal's comment box offers the audience picker only when the view allows it.

Private comments (limited to the current view)

A view can keep its comments private (`limitCommentsToCurrentView`, shown to builders as limiting posted messages to the view). Comments posted there are flagged private (`commentIsPrivate`) and tagged with the view they were created in. Reading a thread then returns the comments made privately in the same view plus every non-private comment, from any view; the audience rules still apply on top. One record can therefore carry separate, view-scoped discussions (long-standing).

Reactions

On timeline posts and other records where reactions are enabled, the only reaction type defined is **Like**.

- Adding or changing a reaction replaces the caller's previous one: at most one reaction per user per record. Identity and time are stamped by the service.
- Removing a reaction deletes only the caller's own.
- A client can list a record's reactions and get a reaction count.

Reacting to a timeline post also registers an impression on the post, and requires the post to be published and the timeline to be active in the view.

Calls on a record

The portal's **Join** action (shown in view mode on non-draft entries whose entity has real-time enabled) opens a call whose channel is the entry's id, so everyone joining from the same entry shares the call. The portal requests a token for the channel on the user's behalf from the RTC service, which issues Agora tokens. A token request names the **channel**, the **role** — **publisher** (can send audio and video) or **audience** (receive only) — and an **expiry** that defaults to **3,600 seconds (1 hour)**. The token is bound to the user id and the channel; a client refreshes it before it expires. Requests without a channel, without an identified user or with an unrecognised role are rejected. Calls are web-only. Service detail: Real-time (sockets and RTC).

Entry emails

A record accumulates the emails the platform sent about it. The entry's **E-Mail** tab (when the view shows it) lists them per record and opens each with its content and attachments. It is a read-only history; nothing is composed from it.

Feature and access gates

View configuration (set by the builder) decides availability:

- **Discussions enabled** — required to see or add comments at all for non-timeline records. On the `POSTS` timeline, the timeline display must be enabled instead.
- **Allow add comments** — required to write text comments.
- **Allow add attachments** — required to attach files, with its own size and type limits.
- **Allow mentions** — required to mention, and the user must be signed in.

The caller's access to the record decides reading and writing:

- Reading a record's comments needs at least **view** access to the record, granted at system, module or entity level.
- Adding a comment or a reaction needs **edit** access, at the same levels.

`Since 1.8` reading comments honours **module-level and entity-level** grants, not only system-wide viewer access, so a restricted user with a module or entity grant can see the discussion without a blanket system role.

Preview mode `Since 1.8`

When a builder previews a view that has no real data, comment threads show placeholder content: a realistic thread over the real record with sample commenters and text, so the layout can be judged. It never affects real comments.

Endpoints

All comment and reaction endpoints require authentication and the view and access checks above. Comment and reaction routes are served by the comments service under the comments API base (for example `/api/comments`); the token route by the RTC service.

Purpose	Method and path
List a record's comments	`GET /v1/{entity}/{relatedToId}`
List comment attachments	`GET /v1/{entity}/{relatedToId}/attachments`
Count a record's comments	`GET /v1/{entity}/{relatedToId}/count`
Add a comment	`POST /v1/{entity}/{relatedToId}`
Edit a comment's audience	`PATCH /v1/{entity}/{relatedToId}/{commentId}`
Mentions autocomplete	`GET /v1/comments/{entityId}/{entryId}/mentions-list?keyword=…`
Audience autocomplete	`GET /v1/comments/audience-list?keyword=…`
List reactions on a record	`GET /v1/reaction/{entity}/{relatedToId}`
Count reactions	`GET /v1/reaction/{entity}/{relatedToId}/count`
Add or change the caller's reaction	`PUT /v1/reaction/{entity}/{relatedToId}`
Remove the caller's reaction	`DELETE /v1/reaction/{entity}/{relatedToId}`
Get a call (Agora) token	`GET /api/rtc/{channel}/{role}?expiry=…`

Sensitive and encrypted fields: model, API and enforcement

This page is for **integrators, security reviewers and platform engineers**. It documents how sensitive and encrypted fields are stored, granted, masked, revealed and enforced, the endpoints involved, and the exact refusals a client must handle.

> **Protecting a field in the UI?**
>
> Builders should start with Protect a field [Build Apps]. Tenant administrators grant access in Sensitive Permissions [Administration]. Operators configure encryption keys in Field encryption keys and providers.

Since 1.8 — the sensitive-fields feature (marking, masking, and per-user access control of individual field values) landed as one change in the 1.8 release line. **Since 1.10** — Encrypted fields, Decrypt, the three-purpose step-up mechanism and the sensitive-data access policy.

Some fields hold information that only a few people should ever see — salaries, national IDs, bank accounts, and the like. OSOS lets a data-entity designer mark such a field as **sensitive**. Once a field is sensitive, OSOS **masks** its value for every user who has not been explicitly granted access to it — both on live entries and in audit history — and only users who hold an explicit **sensitive-field grant** can read the real value.

This is a **separate dimension** from ordinary permissions. A user can have full access to an entity and its records and still see a masked value in a sensitive field until they are granted access to that specific field. Access is controlled by per-field, per-user/group grants stored in the `access_right` collection — not by the entity's own view/row permissions, and **not** bypassed by administrator roles.

`Since 1.10` A field can instead be marked **Encrypted**, which is a different mechanism resting on the same grant model: the value is stored as ciphertext in the database rather than masked on the way out, and it is revealed one field on one entry at a time through a dedicated Decrypt call — which, `Since 1.10`, additionally requires its own single-use step-up verification alongside the grant. A field is Sensitive **or** Encrypted — never both. See Encrypted fields.

Concept and terminology

- **Sensitive field** — a `DataEntityField` whose `options.sensitive === true`. Only scalar-type fields qualify; reference (Entity) and Block container fields are rejected (see Constraints).
- **Scope** — the canonical string identifying one sensitive field across the space, built by `buildSensitiveFieldScope(moduleNn, entityNn, fieldFullNn)`. See Sensitive field scopes.
- **Grant** — an `AccessRight` record of permission type `SENSITIVE_FIELD` (`PermissionTypeEnum.SensitiveField = 'SENSITIVE_FIELD'`) whose `value.scope` equals a field scope, entitling one user or group to read that field.
- **Direct grant vs. inherited grant** — a grant on the field's own exact scope is *direct*; a grant on a wildcard scope that covers it is *inherited*. See Direct and inherited grants.
- **Mask** — the value substituted for a value the requesting user may not read. A live read substitutes an unguessable 26-character sentinel (`SENSITIVE_DISPLAY_MASK`), not the plain `****` an earlier version of this page described — see Masks for why that distinction matters. Audit payloads use a separate, simpler `***` sentinel.

- **Unmask session** — a short-lived, re-authenticated ("step-up") session that additionally gates whether a *granted* value is actually revealed on read.
- **Encrypted field** `Since 1.10` — a `DataEntityField` whose `options.encrypt === true`. Its value is stored as ciphertext (AES-256-GCM), not merely masked on read, and it is mutually exclusive with `sensitive` on the same field. See Encrypted fields.
- **Decrypt** `Since 1.10` — the single reveal path for an Encrypted field: one call, one field, one entry, gated on the field grant **and** a single-use step-up proof (`Since 1.10`). See Decrypting a value.
- `StepUpPurpose` `Since 1.10` — the three-way discriminator (`Unmask` / `Decrypt` / `WorkflowAction`) fixed at step-up challenge time; the three purposes share endpoints but a proof or session minted for one can never satisfy another. See Step-up unmask, Decrypt, and WorkflowAction.
- **Sensitive-data access policy** `Since 1.10` — a per-element setting on a reaction, an export template, or a summary that decides what *that element* may disclose when it runs, independently of who reads its output. See Sensitive-data access policy.

Sensitive field scopes

Every sensitive field is identified across the whole space by a single canonical **scope** string. A scope is built from normalized (machine) names, separated by `/`:

```
<module>/<entity>/<field>
```

For a field that sits inside a **block**, the last segment is the dotted `block.subfield` normalized name — for example `hr/employee/payroll.iban`.

A grant carries a scope and a field carries a scope, and the two are compared **segment by segment**. Both must have the same number of `/`-separated segments to match at all. Any segment may be the wildcard `*`, which matches whatever sits in that position:

Grant scope	What it covers
`hr/employee/salary`	Exactly that one field — a **direct** grant.
`hr/employee/payroll.iban`	The `iban` sub-field of the `payroll` block on `employee`.
`hr/employee/*`	Every sensitive field on the `employee` entity — an **entity-wildcard** grant.
`hr/*/*`	Every sensitive field anywhere in the `hr` module — a **module-wildcard** grant.
`hr/*/salary`	The `salary` field on every entity in the `hr` module.

A grant whose scope names a field exactly is a **direct** grant on that field; a grant that reaches the field through a wildcard segment is an **inherited** grant. Both confer exactly the same read access — the difference is *where* the grant lives, and therefore where you go to change or revoke it.

Scopes are not typed by hand. The scope picker presents the space as a **module** → **entity** → **field** tree: selecting a leaf field authors the concrete three-segment scope, while selecting an entity or a module node authors the corresponding wildcard scope covering everything beneath it.

Segment-wise matching is implemented by `matchesSensitiveScope` (`libs/sensitive-field.ts`), which every runtime check goes through.

Marking a field: field-definition endpoints

There is no dedicated endpoint for *toggling* a field sensitive — marking is done through the normal field/entity definition editors and APIs.

Note: Sensitive fields are supported for top-level fields and for sub-fields inside repeatable blocks. The Block *container* field itself cannot be marked sensitive (see Constraints).

On the field controller (`data-entity-fields.controller.ts`), the definition endpoints carry the sensitive logic:

Method	Path	Behavior for sensitive fields
`POST`	`/v1/entities/{entity}/fields`	`validateAndNormalizeSensitive` ; if `options.sensitive` , `syncSensitiveFieldAccess(scope, users, groups)` else `removeSensitiveFieldAccess(scope)` ; then `delete field.sensitiveAccess` before persist.
`PATCH`	`/v1/entities/{entity}/fields/{fieldId}`	Same sync/clear + strip logic.
`DELETE`	`/v1/entities/{entity}/fields/{fieldId}`	`removeSensitiveFieldAccess(scope)` for the deleted field.
`GET`	`/v1/entities/{entity}/fields` and `/fields/{fieldId}`	`attachSensitiveFieldsAccess` repopulates each sensitive field's `sensitiveAccess` from `access_right` before returning.

Granting access to a sensitive field

Sensitive-field access is granted **per scope**, independently of the ordinary permission levels. It is a distinct grant type, so giving a user Edit on the entity does **not** give them access to its sensitive fields.

There are two write paths onto the same `AccessRight` records:

1. **The field editor** — open the entity, and in the field's access/options panel select the users or groups that should be able to read the field's real value. Under the hood, send the field with `options.sensitive = true` and the desired `sensitiveAccess.users` / `sensitiveAccess.groups` reference arrays. The set is **declarative**: `syncSensitiveFieldAccess` diffs the desired principals against existing Active grants for the scope, deletes stale ones, and creates missing ones, then publishes an `AccessRightChangedEvent` . Sending an empty selection revokes all grants for that field, and grants created this way carry no `expiry` .
2. **The grants-management API** — `/v1/sensitive-field-grants*` , added in 1.8, which backs the admin **Sensitive Permissions** page and the per-user / per-group tabs. It can author wildcard scopes and validity windows, which the field editor cannot. See Sensitive-field grants API, and Sensitive Permissions [Administration] for the administrator-facing procedure.

The declarative sync also runs at the whole-entity level via `DataEntityService.syncSensitiveFieldsAccess` (used by the create/update data-entity command handlers), which additionally tears down grants for fields that stopped being sensitive or were removed entirely (using the previous entity snapshot), so no orphaned grants are left behind.

Direct and inherited grants `Since 1.8`

A grant whose scope is the field's exact `module/entity/field` path is **direct**. A grant scoped to `module/entity/*` or `module/*/*` is **inherited**: it covers every sensitive field under that entity or module, including fields created after the grant was written.

This is a different axis from *how a user acquired* the grant. `GET /v1/sensitive-field-grants/by-user/{userId}` reports the acquisition route separately as `assignmentType` (`Direct` or `group`), so a single grant can be direct-scoped and group-acquired at the same time. The two axes answer different questions:

Axis	Question	Where it is reported
Scope shape (direct vs. inherited)	*How wide is this grant?*	The scope string itself; `inherited-holders` lists wildcard holders for one field.
Assignment type (`Direct` vs. `group`)	*Whose grant is it?*	`UserSensitiveFieldGrantDto.assignmentType` / `acquiredThrough`.

Both are flattened to plain scope strings by the time enforcement runs — `canAccessSensitiveField` sees only `permissions.sensitiveFields` and cannot tell a direct grant from an inherited one.

Sensitive-field grants API `Since 1.8`

`SensitiveFieldGrantsController` (`controllers/sensitive-field-grants.controller.ts`) is a thin delegation layer over `AccessRightsService`. It introduces **no new storage** — `AccessRight` remains the single source of truth, and these routes are new read/write shapes over the same records the field-editor sync path writes.

Auth (every route): `@authorize({voters: [systemAdminVoterFactory()]})`, which votes by HTTP verb — `GET` requires system **Viewer** access or above; `POST` / `PATCH` / `DELETE` require `isSystemDeveloper()` (system Developer, Editor, or Admin). The admin page itself is gated tighter on the frontend, to system Admin only.

Method	Path	Returns
`GET`	`/v1/sensitive-field-grants`	`SensitiveFieldGrantDto[]` — every Active grant, space-wide.
`GET`	`/v1/sensitive-field-grants/by-user/{userId}`	`UserSensitiveFieldGrantDto[]` — a user's direct + group-inherited grants.
`GET`	`/v1/sensitive-field-grants/by-group/{groupId}`	`SensitiveFieldGrantDto[]` — a group's own (direct) grants.
`GET`	`/v1/sensitive-field-grants/inherited-holders`	`InheritedSensitiveFieldHoldersDto` — principals holding a wildcard grant covering one field.
`POST`	`/v1/sensitive-field-grants`	`CreateSensitiveFieldGrantsResultDto` — created grants + skipped duplicates.
`PATCH`	`/v1/sensitive-field-grants/{id}`	`SensitiveFieldGrantDto` — expiry-only update.
`DELETE`	`/v1/sensitive-field-grants/{id}`	`204`, hard delete.
`PATCH`	`/v1/sensitive-field-grants/{id}/archive`	`204`, soft archive.

`GET /v1/sensitive-field-grants` lists every `Active` (non-archived) grant across the space. A single aggregation matches `SENSITIVE_FIELD` grants, `$lookup`s the entitled user/group to resolve a display label, and enriches each row with localized `scopeLabels`. It is deliberately **not** expiry-window-filtered — a future-start grant must stay visible so a manager can see and edit it. `SensitiveFieldGrantScopeLabelsDto` resolves each scope segment independently: a `*` segment, or a normalized name absent from the catalogue (a since-deleted field, say), resolves to `null` for that segment rather than failing the whole list.

`GET .../by-user/{userId}` returns every active grant that effectively covers the user — direct grants plus those inherited from every group they belong to. There is **no de-duplication**: a scope held both ways yields two rows, distinguished by `assignmentType` (`UserPermissionAssignmentTypeEnum.Direct = 0` / `.group = 1`). For an indirect row,

`acquiredThrough` (resolved group reference) and `acquiredThroughGroupType` (`GroupTypeEnum`) identify the acquiring group; both are `null` for a direct grant.

`GET .../by-group/{groupId}` returns a group's own direct grants only — a group cannot inherit from another group — and so reuses the plain `SensitiveFieldGrantDto` with no assignment-type metadata.

`GET .../inherited-holders?moduleNn=&entityNn=&fieldNn=` lists the users/groups that can read one field through a wildcard grant, as opposed to a grant on the field's own exact scope (that set is the field's own `sensitiveAccess` selector and is deliberately excluded here).

- All three query params are **required**. An earlier version let an omitted param build a scope like `undefined/employee/*`, which matched nothing and returned `200 {holders: []}` — read by the dialog as "nobody inherits this field" instead of failing with `400`.
- `fieldNn` is accepted for contract clarity but is **not** matched on: only `buildSensitiveFieldScope(moduleNn, entityNn, '*')` and `buildSensitiveFieldScope(moduleNn, '*', '*')` are queried.
- Only effective grants contribute (`state != Archived`, inside the active expiry window). A principal covered by both wildcard scopes is de-duplicated to one row.
- Each `InheritedSensitiveFieldHolderDto` carries `type` (`USER` / `GROUP`), `id`, a resolved `name`, and — users only — `email` and `avatar`.

`Since 1.8` The field designer's **Inherited Permissions** dialog (opened from a sensitive field's advanced settings, next to the Sensitive toggle) is built on this endpoint. Each holder row it lists is now clickable and opens a second, read-only panel showing that holder's resolved grant for this specific field — re-querying `GET .../by-user/{userId}` or `GET .../by-group/{groupId}` and, when the holder qualifies at more than one scope, preferring an entity-scope match over a module-wildcard one (after first discarding any grant whose expiry has passed). The panel stays read-only, same as the dialog itself; changing the grant still means editing it at its actual entity- or module-wildcard scope. Full admin-page detail — the deep link to the holder's own Sensitive Fields tab, the empty/error states — is on Sensitive Permissions [Administration], not duplicated here.

`POST /v1/sensitive-field-grants` creates one `AccessRight` per user and per group in `dto.users` / `dto.groups`, all for the same `dto.scope`.

- `dto.scope` is validated non-blank up front (`400` otherwise); a wildcard segment is valid — only "non-blank" is enforced.
- `dto.expiry` is optional. Absent or `[]` means open-ended. Otherwise it must be exactly `[start, end]`, each element `null` (open bound) or `Date.parse`-able; `start` must not be after `end`; and a present `end` must be in the **future** — a grant cannot be created already expired. Every violation is a `400`.
- A principal that already holds an Active, in-window grant for the scope is **skipped**, not duplicated, and reported back in `skippedDuplicates`.
- One `AccessRightChangedEvent` is published for the whole call, and only when at least one grant was actually created.

`PATCH /v1/sensitive-field-grants/{id}` edits **only** the expiry window. The scope and entitled entity are immutable — re-pointing a grant at a different principal could silently create a second Active grant for the same scope. `404` if the record does not exist or is not a `SENSITIVE_FIELD` grant.

`dto.expiry`	**Effect**
absent (key not sent)	Unchanged.
`[]`	**Clear** the window — the key is deleted from `value`.

`dto.expiry`	**Effect**
`[start, end]`	**Replace** the window as-is.

The key must be *deleted* rather than set to `undefined`: the MongoDB connector has no `ignoreUndefined` configured and would serialize `undefined` to a stored `null`. Every read path treats a missing key and an explicit `null` identically, so the two are interchangeable on read — but a top-level `null` is not accepted at the schema level, so a client clearing a window must send `[]`.

`DELETE /v1/sensitive-field-grants/{id}` is a permanent hard delete (`404` if not found or not a `SENSITIVE_FIELD` grant), publishing `AccessRightChangedEvent`.

`PATCH /v1/sensitive-field-grants/{id}/archive` soft-archives the record (`state → Archived`) through the same `ArchiveDataEntryCommand` path the expiry auto-archive reaction uses. Archived grants are excluded from every read path exactly as a hard delete would be — access is lost immediately — but the record survives for audit purposes and the (scope, principal) pair is free again for a fresh grant. Both removal paths remain available; which one a given screen calls is a client decision (see Sensitive Permissions [Administration]).

Option exclusion: `excludeForUserId` / `excludeForGroupId`

`SensitiveFieldSelectorComponentFetcher`, which backs the scope selector's reference-data endpoint, reads an `excludeForUserId` or `excludeForGroupId` parameter from the request body's `parameters` bag. When present, it resolves the target principal's currently-held scopes and excludes them from the returned catalogue, so an "Add grant" dialog for a fixed user or group does not offer fields they already hold. `excludeForUserId` wins if both are somehow present.

The exclusion is **gated behind system-viewer**. The fetcher is reachable from the generic `POST /v2/entities/{entity}/fields/{field}/data` endpoint, which is gated only by `canViewEntity` — a broader audience than the dedicated grants endpoints. Revealing which scopes a *specific target* already holds is therefore honored only when the **caller** is at least `isSystemViewer()`; for anyone else the exclusion is silently skipped and the full, unfiltered catalogue is returned. It never errors and never leaks the target's grants to an unprivileged caller.

How masking behaves

For a user **without** the grant — or with the grant but no active unmask session — the sensitive value is replaced with `****` consistently across the product:

Where	Behavior for a user who cannot currently read the field
List views	The column shows the mask.
Record (detail) view	The field shows the mask.
Editing	The field can still be edited "blind": the user types a new value into a masked field without ever seeing the current one. Whether what they submit is kept or discarded depends on the write-side protection described under Write path.
After-save confirmations	Any echoed value stays masked; the value the client gets back is masked on a copy, so the authoritative entry and its events keep the real value.
History	Changes to a sensitive field are shown only if the user has access to that field; otherwise the before/after values are masked.
Audit trail	The audit record masks sensitive values independently (shown as `***`).

Where	Behavior for a user who cannot currently read the field
Reports and exports	Reports (available on the web only) and exports (XLSX or PDF) read values through the same masked path, so an unauthorized user's report or export carries the masked value, not the real one.

Masking is applied when data is read, so it follows the value into every one of these surfaces automatically.

Under the hood:

- `DataEntryService.sensitiveMaskApplies(dataEntity, user)` gates whether masking does anything at all for a read: there must be an entity, it must declare at least one sensitive field, and the reader must not be an internal (system) service call.
- `DataEntryService.applyCallerSensitiveMask(entries, dataEntity, user)` is the presentation-side entry point, called from `handleResolvedValuesAfterAggregation` unless `disableHideSensitiveFields` was passed. It resolves the entity's module normalized name, reads the caller's active unmask scopes, calls `maskSensitiveFields`, and — if anything was actually revealed — publishes one batched `SensitiveDataUnmaskedEvent` for the request.
- `DataEntryService.maskSensitiveFields(entries, dataEntity, moduleNn, permissions, unmaskScopes?)` replaces each unreadable field with `SENSITIVE_DISPLAY_MASK` in place, descending into block sub-fields inside repeatable blocks. A field is revealed only when **both** `canAccessSensitiveField(permissions, scope)` is true **and** an active unmask scope matches. Its `permissions` parameter is narrowed to `Pick<IAllUserPermissions, 'sensitiveFields'>`, so a fail-closed caller can honestly pass `{sensitiveFields: []}` instead of a cast.
- **Fail closed on a module-lookup failure.** `applyCallerSensitiveMask` resolves the module through `dataModuleRepository.findById`, inside a caller whose `catch` answers *any* exception by returning the raw aggregation output. That lookup failure is therefore caught locally and masks every sensitive field with an empty grant list, rather than letting the exception propagate and silently return unmasked data.
- **Fail closed on an unmask-session lookup failure.** Reading the caller's active scopes is wrapped the same way: any error (a cache outage, say) is logged and treated as "no active session", so everything stays masked.

Masking cannot be bypassed by administrators

Sensitive-field masking is **not** overridden by system-administrator or Edit access. Even a full administrator sees the mask unless they hold the sensitive-field grant for that field *and* an active unmask session. The only exception is internal calls made by the platform backend services (or when masking is explicitly disabled via `disableHideSensitiveFields`) — used by trusted backend integrations, not by interactive users.

The client is told **which** fields are masked separately, via the entity definition's `sensitiveMaskedFields` (computed per-request by `computeSensitiveMaskedFields` / `resolveSensitiveMaskedFields`), rather than by inspecting the returned value. Because the mask sentinel is designed to be indistinguishable from a value a user could legitimately have typed, this list is the only reliable signal. It is per-request and must never be cached with the base entity definition; an absent user yields **all** sensitive fields masked.

Since 1.8, that signal is actually two disjoint lists, so the client can tell "no grant at all" apart from "grant held, but not currently unmasked": `sensitiveMaskedFields` covers a field the user has no grant for at all; `sensitiveMaskedFieldsWithUserAccess` covers a field the user *does* hold a grant for, but no active unmask session currently covers it. A field revealed by an active session appears in neither list. This is what lets the UI render a plain lock for a field the user can never see, versus an unmask ("eye") affordance for a field they hold a grant for but haven't stepped up on yet.

`Since 1.10` Both lists now also cover **encrypt-only** fields (`encrypt: true`, `sensitive: false`), which previously appeared in neither — so the client had no per-viewer signal for them at all and always drew the reveal affordance, whether or not the viewer actually held a grant. An Encrypted field never participates in an unmask session, so it is either in `sensitiveMaskedFields` (no grant → the client draws a **lock**) or, whenever a grant is held, in `sensitiveMaskedFieldsWithUserAccess` (→ the **eye** that triggers Decrypt) — never in neither list the way a session-revealed sensitive field is. The consequence a user sees is that the lock icon now also appears on an Encrypted field they cannot decrypt: the affordance is **suppressed rather than offered and then refused with a 403**. Mobile draws the same three-way distinction with three icons — an encryption glyph for Encrypted, a crossed-out eye for sensitive-and-revealable, a padlock for sensitive-with-no-grant — and the mask now propagates to card, kanban and calendar text surfaces too. Inline edit of an Encrypted field is blanked **unconditionally, grant or no grant**, because an encrypted value is opaque client-side either way; without that, the editor pre-filled the masked placeholder as if it were real content.

Client rendering of a masked value `Since 1.8`

`sensitiveMaskedFields` rides on the entry-**details** response. A plain entries-**list** response carries no such per-user list — there, the only signal a client has is the mask sentinel substituted into the field's own value. A client rendering a list must therefore infer masking from the value itself, gated on the field's own `options.sensitive` flag so that a non-sensitive field legitimately holding a string that happens to match the sentinel is never mistaken for a restricted one.

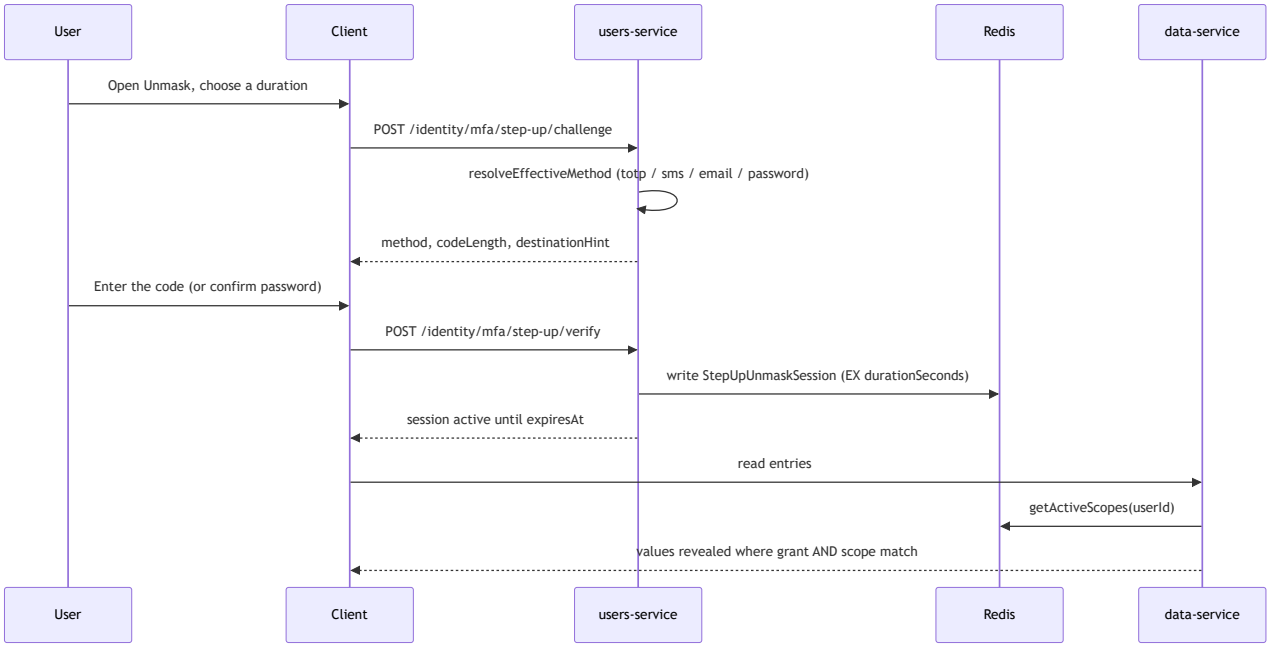
Two consequences, both now honoured by the mobile client as well as the web one:

- **A read-restricted field renders its mask, not an empty placeholder.** A masked value that cannot parse into the field's own type (a number, a date) used to degrade to "no value" on list and table surfaces, so a protected field looked identical to an unfilled one. Both the empty and non-empty rendering branches now go through the same masked-value component, so which case applies is never leaked by an empty-value glyph.
- **A masked field is inline-editable from both the details view and the list.** That matches the server contract in Write path, which never depended on the caller being able to *read* the field it writes — the blind write is safe because the server discards whatever an unauthorized caller submits, not because the client hid the control. The inline-edit gate also resolves the workflow requirement the same three-way way the full edit form does, so a caller holding **admin edit rights** on the view is no longer refused inline edit on an entry whose active step they do not personally own.

Masking is suppressed only inside the inline-edit control itself and on create, where an empty editable input is what the blind write needs.

Step-up unmask session `Since 1.8`

Holding a grant is **necessary but not sufficient** to see a live value. A granted field is still masked by default, and is revealed only while the requesting user has an active, time-boxed **unmask session** opened by re-authenticating. This sits on top of the grant model, not instead of it.



Session lifecycle

`step-up-router.ts` mounts four routes under `/identity/mfa/step-up` (JSON body parsing, domain resolution, and access-token auth applied to all): `POST /challenge`, `POST /verify`, `POST /disable`, `GET /status`. The implementation is `step-up-unmask.service.ts`.

- **`challengeUnmask(accountId, {durationSeconds, scope?, method?}, domain?)`** picks and prepares the factor.
  - `durationSeconds` must be one of the allow-listed `STEP_UP_UNMASK_DURATIONS_SECONDS` values — 120 / 300 / 600 / 1800 / 3600 / 10800 / 14400 / 21600 / 28800 (2 / 5 / 10 / 30 / 60 minutes, or 3 / 4 / 6 / 8 hours) — else `StepUpError('INVALID_DURATION')`. This list superseded an earlier 15/30/45/60/120-minute allow-list; the web and mobile duration pickers are both sourced from the same constant so they cannot drift from it.
  - `USER_NOT_FOUND` and `EXTERNAL_USER_FORBIDDEN` are rejected outright: external/SSO users cannot step up at all.
  - `resolveEffectiveMethod(available, preferred, requested?)` takes the requested method if usable, else the user's preferred one, else the first of `['totp', 'sms', 'email']`. `passkey` is explicitly **not** usable for this code-entry step-up and is skipped even if requested.
  - **Password fallback.** If no MFA method is usable (`MFA_NO_METHOD`), the domain has `allowPasswordStepUp === true` (opt-in, default `false`), and the user has a local password, the method falls back to `'password'` instead of throwing. External/SSO users were already rejected, so this only ever applies to local accounts. `allowPasswordStepUp` is a Domain core-entity field on the data-service side, synced into the users-service `Domain` projection by the data-entry created/updated handlers.
  - `sendable` methods (`sms`, `email`) are throttled by `otpRateLimiter` *before* the OTP is sent (`MFA_RATE_LIMITED`, `429`), separately from the per-verify limit — otherwise an authenticated caller could loop `/challenge` and bomb the user's inbox.
  - A pending context (method, duration, scope) is stashed keyed `step-up-pending:<accountId>` with the OTP expiry as TTL.
  - Returns `{method, sendable, codeLength, resendAfterSeconds, destinationHint?, availableMethods}`. `codeLength` is `0` for password and `6` for TOTP; `destinationHint` is the masked email/phone and is present only when `sendable`.

- `verifyUnmask(accountId, {code?, password?, method?})` verifies the pending challenge under its own rate limit. The password fallback re-verifies via `PasswordHelper.isSame` and returns `INVALID_PASSWORD` on mismatch; an MFA method delegates to its registered verifier. **Duration and scope are read back from the pending context, not from the verify request**, so a client cannot escalate the window or the scope at verify time. On success it writes the session, destroys the pending context, resets the verify rate limit, and publishes `SensitiveDataUnmaskEnabledEvent`.
- `disableUnmask(accountId)` atomically `GETDEL`s the session key, so two concurrent disables cannot both observe an active session and double-publish. `SensitiveDataUnmaskDisabledEvent` is published only when a session was actually removed; an already-expired or double-submitted disable is a silent no-op.
- `getUnmaskStatus(accountId)` returns `{active, scopes?, expiresAt?}` for the top-bar toggle and its countdown.

Session storage (cross-service contract)

The session is a JSON `StepUpUnmaskSession` (`@osos/shared/step-up-unmask.ts`) — `{scopes: string[], enabledAt: ISOstring, expiresAt: ISOstring}` — written under `stepUpUnmaskRedisKey(envPrefix, userId)`. The key is deliberately **clean** (no `oidc:` namespace segment) because users-service writes it through the raw client rather than the namespacing adapter. `scopes` is either the caller-provided `scope` from the challenge, or `STEP_UP_UNMASK_ALL_SCOPE` (`'*/*/*'`, matching every field through the standard wildcard matcher) when none was given.

users-service writes on a **dedicated duplicated connection** pinned to `stepUpSessionDb()` (`CACHE_REDIS_DB ?? 1`): its default client may sit on a different logical DB (the OIDC session/token store), and writing there would put the session somewhere data-service never looks. The key is written with `EX ttlSeconds = durationSeconds`, so it self-expires — there is no cleanup job.

`data-service`'s `UnmaskSessionService` (`SINGLETON`) is the reader. `getActiveScopes(userId)` reads the same key byte-for-byte (no re-prefixing), JSON-parses it, and returns `session?.scopes` or `undefined` if absent or expired.

Session-scoped unmask sessions `Since 1.8`

An unmask session now belongs to the **sign-in session** that opened it, not to the account as a whole. Logging out and back in, or opening a second concurrent session for the same user (a different browser, a second device), starts masked again in that new session — it does not inherit an unmask window opened elsewhere.

This closes a real leak: previously the session was keyed by account id alone, so *any* session of that account — including one opened later by anyone who could authenticate as that user, or a browser profile the original user forgot to sign out of — inherited whatever unmask window was still open, without that session ever stepping up itself.

Under the hood, both the session key and the pending-challenge key are now scoped by `(userId, sessionId)`, and every reader of the active-scopes lookup passes the caller's session id through explicitly — there is deliberately **no fallback to the old account-wide key on a miss**: an absent or wrong session id means "no active session," full stop. The one exception is a session-less caller (a service-to-service token has no session to scope to), which still falls back to the account-wide key rather than being denied outright. OTP send-rate and verify-rate limiting stay keyed by account only, on purpose — the phone/inbox/brute-force surface being protected belongs to the account, so opening extra sessions must not multiply the allowance.

Step-up keys are tenant-segmented `Since 1.10`

The unmask grant and the single-use step-up proof are written by the identity service and read by data-service through Redis. They used to carry no tenant, so in a multi-tenant deployment a grant or proof earned under one

tenant was readable under another, and no offboarding sweep could find it. Both keys are now composed with the tenancy mode:

Mode	Unmask grant	Proof
single-tenant	`<ENV prefix>step-up:unmask:<uid>[:sid:<sid>]` (unchanged)	`<ENV prefix>step-up:proof:<id>` (unchanged)
multi-tenant	`<ENV prefix>app:<appId>:step-up:unmask:…`	`<ENV prefix>app:<appId>:step-up:proof:<id>`

A grant or proof is visible only under the tenant that issued it; with no tenant in scope in a multi-tenant deployment both sides refuse before touching Redis; and offboarding a tenant removes them. The proof keeps its atomic single-use read. Single-tenant installations see no change, so no unmask sessions are lost on upgrade, and nothing changes in the entity builder or the reveal wizard.

Audited events

Three events, each with a matching `audit-service` handler that persists an audit entry:

Event	Published by	Details persisted
`SensitiveDataUnmaskEnabledEvent(userId)`	users-service, on a successful `verify`	— (enable action only)
`SensitiveDataUnmaskDisabledEvent(userId)`	users-service, on an actual disable (not a no-op)	— (disable action only)
`SensitiveDataUnmaskedEvent(userId, …)`	data-service, when `maskSensitiveFields` actually reveals at least one field	`{module, entity, fields, entryIds}`

`SensitiveDataUnmaskedEvent` is published **once per request**, batching every field revealed across every entry in that response (`entryIds` unioned across fields) rather than once per field. The publish is fire-and-forget and error-safe: a failure logs a warning and never blocks or fails the read.

Querying and filtering on sensitive fields

To prevent a user from *inferring* a masked value by filtering on it, a user-supplied query that references a sensitive field the user cannot read is **rejected** (a deliberate design decision: reject, do not silently drop). A user cannot build a condition such as "salary greater than X" against a field they are not allowed to read. This closes the side-channel where a value could be leaked through search results even while the value itself stays masked on screen.

The rejection throws `SensitiveFieldQueryNotAllowedException` (error code `SENSITIVE_FIELD_QUERY_NOT_ALLOWED`, with a `{field}` detail). Two enforcement paths mirror the two filter syntaxes:

- `assertUserCanQueryCriteria` — reuses `QueryEngineService.parseCriteria` with a sensitive context so nested `And` / `Or` / `ElemMatch` branches are walked exactly as they execute.
- `assertUserCanQueryWhere` — walks a loopback `where` tree, strips a leading `value.` prefix and numeric array-index segments, and matches a field either exactly or as the parent of a suffixed path.

Both are no-ops when there is no user, no criteria/where, or the entity has no sensitive fields. Admin-defined view / quick-filter criteria are treated as trusted system context and are **not** enforced here — only the user's ad-hoc criteria are checked. This check is **grant-based only**: it is not additionally gated by an unmask session, so a grant holder can filter on a field whose value is still masked on screen.

`Since 1.10` **Naming a protected field is no longer the same right as reading it.** In the **Administration** area, a system editor may now *name* a sensitive field in a query, a filter, or a reaction condition **without holding a read grant for it**. Before this, a masked field was dropped from the field picker unconditionally, everywhere — so a manager with

no grant of their own saw none of an entity's sensitive fields when designing a query, even in the admin screens where they were designing the query for other people. Three things did **not** change:

- **An ordinary end user still needs the grant** to reference the field; the end-user query builder keeps the old rule.
- **Nothing about which values anyone sees changes.** The masking path, both masked-field lists, the step-up session, and every element policy are untouched — this widens only who may *reference* a field in criteria.
- **An Encrypted field stays rejected either way**, in both areas, ahead of this check entirely — see What an encrypted field refuses.

What decides "admin context" is the **area the request came through, not the caller's role**: a system editor browsing the end-user side (through a data view, say) is refused exactly what an ordinary end user is refused for the same query. That is what lets the client's own field list and the server guard agree without either restating the other's rule, and it fails closed — an unbound authorizer, absent permissions, or any thrown error all resolve to "not admin".

Time-bounded (expiring) grants

A sensitive-field grant can carry a **time-bounded validity window** instead of lasting until it is manually revoked. When granting access, you can set a start date and/or an end date:

- Leaving both dates empty makes the grant **never expire** (the default).
- Setting **only an end date** makes the grant effective from now until that date.
- Setting **both** defines the full validity window; before the start date the grant exists but confers nothing.

When a grant reaches its end date, OSOS automatically **archives** the grant record via a pre-configured timer reaction, and the revocation takes effect immediately: the user-permission cache is cleared, the field's "restricted" state is recomputed, and the archived grant is excluded from every subsequent permission query — so the user loses access the moment it expires, with no manual step.

A grant can also be archived or deleted manually at any point before its expiry; both have the same immediate effect on access. Re-adding a principal whose grant was archived or expired creates a fresh Active record rather than being a silent no-op.

Data model

Field flag: `options.sensitive`

Defined on `FieldOptions` (`field-options.model.ts`):

Field	Type	Default	Meaning
`sensitive`	`boolean`	`false`	When true, read access to this field's value is restricted to users/groups holding a matching grant.

Transient selection: `sensitiveAccess`

`DataEntityField.sensitiveAccess` is a `SensitiveFieldAccess` (`sensitive-field-access.model.ts`):

```
class SensitiveFieldAccess {
  users: ReferenceProperty[];   // users granted read access
  groups: ReferenceProperty[];  // groups granted read access
}
```

This selection is **request-only**. It is never stored on the entity document:

- On **write** (create/update field or entity), it is synced into the `access_right` collection and then stripped from the field before persistence.
- On **read** (fetch field/entity definition), it is recomputed from `access_right` and re-attached on the fly.

Grant storage: the `access_right` collection

Each granted (user|group, scope) pair is one `AccessRight` document with:

- `value.permission_type.id` = `SENSITIVE_FIELD`
- `value.scope` = the scope string
- `value.entitled_entity` = `{dataEntity: USER|GROUP, dataEntry: <principalId>}`
- `value.expiry` = optional `[startIso, endIso]` validity window; `null` or absent means the grant never expires
- `state` = `Active`, plus standard creation/update info and `store = PUBLIC`.

The write path pins two configuration constants when it persists a grant: the record's `store` is the `PUBLIC_STORE` reference (id `'PUBLIC'`), and the accompanying `AccessRightChangedEvent` is published under the root space, `ROOT_SPACE_ID` (`'ROOT'`), so the grant and its cache-invalidation event are always scoped to the public store at the space root rather than to a tenant sub-space.

Reads of grants ignore records that are `Archived` or outside their expiry window, so re-adding a principal whose grant was archived or expired creates a fresh Active record rather than silently doing nothing. Grant queries are backed by a compound index on `(value.permission_type.id, value.scope)` (migration `1.8.0-a25`) and use an `^`-anchored regex for index-friendly prefix matching.

A later idempotent migration, `1.8.0-a31`, re-asserts the cumulative `AccessRight` structure (`permission_type` / `scope` / `expiry`) in a single read-modify-send. Its two predecessors persisted their entity change with a fire-and-forget bus send, so an environment that ran them without a data-service consuming the queue recorded the migrations as applied while `access_right` never gained the `scope` / `expiry` structure — leaving the scope picker permanently empty with no self-healing. `a31` carries a higher version so it runs on the next migrate, and is a no-op on an already-converged database.

Resolved permissions: `sensitiveFields`

When a user's permissions are resolved (`access-rights.service.ts`), all `SENSITIVE_FIELD` grant scopes reachable through the user's own + group grants are flattened, de-duplicated, and exposed as a string array `IAllUserPermissions.sensitiveFields`. Every runtime access check reads only this array — see Access checks. Holding a matching scope here is **necessary but no longer sufficient** to see a live value; the step-up unmask session is the second gate.

Masks

Four constants live in `libs/sensitive-field.ts`, three of them still in use:

Constant	Value	Used for	Since
`SENSITIVE_DISPLAY_MASK`	`****__DISPLAY_SENSITIVE__****`	The value actually substituted into a masked field's `value` on read. **Not** the plain `****` an earlier version of this page described.	`1.8`
`SENSITIVE_STARS`	`****`	What a masked value looks like to a *human* — substituted only when a value is rendered into prose (an after-save message, a button-confirmation dialog, a redirect URL), never into the entry's own `value` payload.	`1.8`

Constant	Value	Used for	Since
`AUDIT_SENSITIVE_MASK`	`***`	Permanent mask in audit/history payloads.	`1.8`
`SENSITIVE_MASK` (legacy)	`****__MAKE_SENSITIVE__****`	**Never emitted on read.** Retained only as a detector for data poisoned while it was still in use.	`1.8`

The read path deliberately returns an unguessable sentinel rather than a value a user could plausibly type by hand, so the write path can tell an untouched masked field a client echoed back apart from a value the client actually changed — see Write path. Because the sentinel is 26 characters, not four asterisks, any code path that substitutes a masked field's value into human-facing text rather than the JSON `value` payload has to translate it back to plain stars first; `renderSensitiveMaskAsStars` does that translation, called centrally from `TokenService.replaceTokenInText` so every text-substitution site (after-save messages, button-confirmation dialogs, redirect URLs) is covered in one place.

Historical note. Earlier in the 1.8 line, a masked field read back the internal sentinel `SENSITIVE_MASK` verbatim, and the *only* write-side protection was the validator restoring the old value when it saw that exact string echoed back — write safety depended entirely on a client behaving. That sentinel was replaced by `SENSITIVE_DISPLAY_MASK` above; a validator restore still fires on either string, so a client still round-tripping the old sentinel remains protected.

The legacy sentinel survives in the source for two narrow reasons, never as a value the system produces on read:

1. `data-entry-validator.service.ts` restores the stored value if it sees `value === SENSITIVE_MASK` **or** `value === SENSITIVE_DISPLAY_MASK` — this is the restore branch of the validator (see Write path).
2. It is used as a **poison-data detector**: migration `1.8.0-a34` and the indexing-service `containsLegacySensitiveMask` guard both match the legacy string to find or reject data that leaked while the old sentinel was live. Neither matches the current `SENSITIVE_DISPLAY_MASK`, which is distinctive and unguessable enough to be safe as a detector on its own if a future migration needs one.

The index-poisoning bug and its fix

Worth knowing when reading this code: `DataEntryWriterService` used to re-read a just-saved entry through the **masked** read path and publish *that* masked object as `DataEntryCreatedEvent` / `DataEntryUpdatedEvent`. Since indexing-service derives `DataEntryIndex.meta.mainTitle` from that payload, saving an entry whose title field is sensitive, as a user without the grant, persisted the mask into the shared index — and because a relation resolves its label straight from that index (which cannot be re-masked per reader), the mask came back to *every* reader, including grant holders. The fix has three parts:

- `DataEntryWriterService.resolveSavedEntryUnmasked()` — every after-save re-read (create, update, patch, upgrade, virtual refresh, post-transaction resolution, batch create with related entries) resolves with `disabledStages.hideSensitiveFields`, so event payloads and derived writes always carry authoritative values.
- `DataEntryService.maskEntryForResponse()` — masks a **deep copy** of the saved entry for the HTTP response, leaving the event-payload entry untouched. Masking is a per-reader presentation concern and must only ever be applied on the way out to that specific reader.
- Migration `1.8.0-a34` repairs rows that already drifted: it synchronously nulls every `DataEntryIndex.meta.mainTitle` matching the **legacy** sentinel pattern, then asynchronously publishes a `DataEntityFieldMetaUpdatedEvent` per affected entity so indexing-service rebuilds the real title. Step 2 is best-effort and unconfirmed from the migration itself — affected entity ids are logged individually so a rebuild can be re-triggered.

Access checks

`SensitiveFieldsService` (the primary source) provides pure static helpers plus one instance method:

`canAccessSensitiveField(permissions, fieldScope): boolean`

Binary check, **no admin bypass**: returns true iff at least one grant in `permissions.sensitiveFields` matches `fieldScope` via `matchesSensitiveScope` (wildcard-aware). A user with no `sensitiveFields` array (or an unauthenticated request) is treated as having no grants and therefore no access.

This is the *grant* check only. A live entry read needs an additional active-unmask-session check on top of it; `canAccessSensitiveField` alone still governs the write guard and the query guard, neither of which is session-gated. Reading audit history is the exception — Since 1.8 it requires both the grant and an active session, the same as a live read; see Reading audit history.

`filterQueryableFieldsForUser(fields, permissions, moduleNn, entityNn)`

Filters a flat field list, dropping sensitive fields the user cannot access; non-sensitive fields always pass. Intended for the query-builder field-list so the UI only offers fields the user may filter on. (Verified as an exported static helper; no production caller of it was found — see Known gaps.)

`getSensitiveFieldOptions(search?, excludeScopes?)` — the option catalogue

Instance method returning one `SensitiveFieldOptionDto` per sensitive field across the whole space (all entities/modules), for the scope selector. It runs a Mongo aggregation over `DataEntity` that:

1. `$lookup`s the parent `DataModule` for localized module context.
2. Collects **top-level** sensitive fields (excluding Block container fields).
3. Collects **block sub-fields** marked sensitive, emitting a dotted `block.subfield` normalized name; orphaned block sub-fields (no Block field references the internal entity) are skipped, mirroring `getAllFields()`.
4. Optionally filters by `search` against field **names** across all registered language codes. The term is regex-escaped (`\\$&`) before interpolation to prevent invalid-regex crashes and ReDoS.
5. Optionally excludes any scope in `excludeScopes`, used by the "Add grant" dialogs to hide fields a target user or group already effectively holds — see Option exclusion. `Since 1.8`

Each `SensitiveFieldOptionDto` carries `id`, `fieldName` / `fieldNormalizedName`, `entityName` / `entityNormalizedName`, and `moduleName` / `moduleNormalizedName` (localized names are `LocalizedValue`).

The DTO is served through the generic reference-field-data endpoint via `SensitiveFieldSelectorComponentFetcher` (component name `SensitiveFieldSelectorComponent`), bound to the `scope` field of the `AccessRight` entity (migration `1.8.0-a24`). The fetcher returns the flat catalogue verbatim; the `search` term comes from the request body.

Reading audit history

`DataEntryHistoryService.maskSensitiveHistoryFields(history, dataEntity, user)` replaces the `old` / `new` values of sensitive fields in each `Update` history record's `additionalData` with `AUDIT_SENSITIVE_MASK`. **Since 1.8**, this reveals a field only when **both** the grant (`canAccessSensitiveField`) and an active unmask session cover it — the same grant-and-session rule a live read applies. An earlier rule here was grant-only, which meant anyone holding a grant could read a field's past values in its history without ever opening an unmask session — a way around the live-read rule, since the value stayed hidden on the entry itself but was exposed one screen over, in its history. It only replaces `old` / `new`

when the property is actually present, so a first-write record (only `new`) does not gain a spurious masked `old`. A fast path returns unchanged history when the entity has no sensitive fields.

Separately, in `audit-service`, `audit-base-handler.ts` masks sensitive values **before persisting** the audit record (using its own local `AUDIT_SENSITIVE_MASK = '***'`), and **fails closed**: if sensitive-field metadata cannot be loaded, every value in the payload is masked rather than risk leaking a sensitive value into the immutable audit log.

Write path (permission-enforced preservation) `Since 1.8`

Because an unreadable field reads back as a mask sentinel, a naive write-back would let a user echo it and unintentionally overwrite the real value, or seed an arbitrary one. The design for closing that gap is enforced by **permission**, not by client behavior:

- `DataEntryService.preserveSensitiveFieldsOnWrite(newValue, oldValue, dataEntity, permissions)` — for every sensitive field the caller cannot read, discards whatever was submitted and restores the stored value. On **create** (no `oldValue`) the field is dropped entirely rather than seeded.
- For a sensitive sub-field inside a **repeatable block**: on create the sub-field is stripped from every submitted row; on update, values are restored **by array index**, because rows carry no stable id. If the submitted row *count* differs from the stored count — an insert or delete that a caller without the grant cannot safely be reconciled against — the write is **rejected** with `SensitiveFieldWriteNotAllowedException` (`{field}` detail) rather than risk dropping or misattaching a protected value. An equal-length reorder is undetectable without a row id, but it is a data-integrity edge, never a disclosure.
- **Deliberately no admin bypass** was ever part of the design. Sensitive-field access is grant-based and the read mask does not exempt admins either, so exempting writes would reopen the hole the guard exists to close. Unconditional system writes are meant to go through internal service calls instead.

Constraints (what can be marked sensitive)

`FieldService.validateAndNormalizeSensitive` enforces, when `options.sensitive === true`:

- **Rejected** on reference fields (`DataTypeEnum.Entity`) and Block container fields (`DataTypeEnum.Block`) → throws `SensitiveFieldNotAllowedException` (error code `SENSITIVE_FIELD_NOT_ALLOWED`).
- **Forced off**: `options.searchable = false` and `options.showAsTitle = false` (a sensitive value must not be indexed for search nor used as the entry title).

Block **sub-fields** may still be marked sensitive individually — only the Block *container* field itself cannot be.

The entity's main title field cannot be marked Sensitive `Since 1.8`

`Since 1.8` The two directions are now both closed:

- Marking a field sensitive **forces** `options.showAsTitle` **off**, as above. A field cannot be sensitive *and* the entry title.
- The entity's **View Options** → **Main Title** picker now **excludes every field whose** `options.sensitive` **is true**, block sub-fields included — the exclusion filter is tree-aware. Before this, the picker only excluded the configured sub-title field (and anything nested under it) and did not consider sensitivity at all, so an already-sensitive field could still be *selected* as the main title even though the backend rule intended otherwise.

The reason is disclosure, not tidiness: the main title is rendered **unmasked** across list and detail surfaces, because it is what row labels and reference labels display. A sensitive field used as the title would therefore leak its value to every

user who can see the entry in a list.

The sub title, by contrast, is allowed to be sensitive `Since 1.8`

This is easy to get backwards, so read the behavior itself rather than assuming it mirrors the main-title rule above: **a sensitive field may be the sub title.** Earlier in the 1.8 line this was rejected in both directions, mirroring the main-title restriction exactly — picking an already-sensitive field as sub title, or marking the configured sub-title field sensitive, both failed, on the reasoning that the card, badge, calendar, mini-card, and tree faces all render a sub title as plain text too. That restriction was removed before this line shipped: those surfaces now mask a sensitive sub title themselves (the same `****` treatment every other field renderer already applies to a sensitive value), instead of the field being barred from the role. If you find an older description asserting sub titles are barred, it reflects that intermediate, no-longer-current state — not what the platform enforces today.

Mapped fields inherit sensitivity `Since 1.8`

A **mapped field** — a field on one entity configured to mirror a field on another entity through a relation (`mapping.enabled`, `mapping.sourceFieldId`) — automatically carries the source field's sensitivity and grant holders with it. Marking the source field sensitive, or changing who can read it, propagates onto every field mapped from it, so a mapped field ends up masked and granted by the same machinery as any other sensitive field, rather than needing special-casing on read.

Propagation runs in both directions, because either side can change first:

- **When the mapping entity itself is saved**, its mapped fields' sensitivity and grant holders are re-derived from their source fields before the save completes.
- **When the source field changes without the mapping entity being touched**, the change is pushed out to every entity that maps from it — chained through multi-hop mappings (if entity `C` mirrors `B`, which itself mirrors `A`, updating `B` in the same pass makes it the source for the next hop onto `C`), with a cycle guard so a mapping loop cannot spin forever. Deleting a field that other entities mapped from also runs the propagation, so a mapped field whose source no longer exists stops being sensitive and its mirrored grants are torn down.

A propagation failure is logged and swallowed rather than failing the edit that triggered it, and is recoverable by re-saving either entity.

Encrypted fields `Since 1.10`

Marking a field **Sensitive** changes what a reader is *shown* — the stored value stays plaintext in the database. **Encrypt** is a second, separate option that changes what is *stored*: the value is written as ciphertext (AES-256-GCM), and it stays ciphertext for every caller. There is no "system bypass" of encryption.

Encrypt is its own option, and the two are mutually exclusive

`options.encrypt` sits beside `options.sensitive` on the field, and in the field editor the **Encrypt** toggle is Sensitive's own sibling block — rendered immediately after **Sensitive** and before **Searchable**, not nested inside it. A field may carry **at most one of the two**. `FieldService.validateAndNormalizeEncrypt` rejects `{sensitive: true, encrypt: true}` with `SensitiveAndEncryptMutuallyExclusiveException` (code `SENSITIVE_AND_ENCRYPT_MUTUALLY_EXCLUSIVE`, message *"A field cannot have both sensitive and encrypt enabled at the same time."*, detail `{field}`).

Two practical notes on that rule:

- **The error code has no localized copy** — it is absent from the error dictionary, so on the rare path where it does reach a user, it arrives untranslated.
- **In the editor the rule is a silent, symmetric auto-untoggle, not a validation error.** Turning Sensitive on switches Encrypt off; turning Encrypt on switches Sensitive off; whichever the designer just switched on wins. Neither toggle is ever disabled because of the other, and there is no client-side validator or message for the rule at all — the server's untranslated message is the only feedback if a client somehow submits both. A legacy field stored with both flags true resolves to **Sensitive** when the editor loads it.

Turning Encrypt on also **clears Searchable** — the server force-sets `options.searchable = false` on save, keeping ciphertext out of the shared full-text index — and empties the inherited-holders selector.

Grants are **not** a second mechanism. An Encrypted field is authorized through the same transient `sensitiveAccess` selector and the same `SENSITIVE_FIELD` grant records described above, on the same `module/entity/field` scope. The field editor's user/group picker is simply duplicated under the Encrypt toggle so grants stay reachable when Encrypt is on without Sensitive; it writes the **same** grant, not a second one. The admin-side consequence — an encrypt-only field now being offered in the Add-grant dialogs at all — is on Sensitive Permissions [Administration].

Which field types can be encrypted

Four kinds can never be encrypted. The server does not reject them — it **silently normalizes `encrypt` back to `false`**:

Cannot be Encrypted	How it is recognized
Reference / relation field	`dataType === DataTypeEnum.Entity`
Block container	`dataType === DataTypeEnum.Block`
Switch	`dataType === Json` with a `SwitchComponent` edit component
Category	`dataType === Json` with a `CategoryComponent` edit component

Everything else is encryptable: text, number, date, time, date-time, boolean, `blob` (files and images), icon, localized JSON, GeoJSON, plain JSON, object JSON, and SVG icon.

The two exclusion lists are deliberately different shapes — in both directions:

- `sensitive` excludes only reference and Block fields. **A Switch or a Category field can therefore be marked Sensitive but never Encrypted.**
- In the field editor, a **Mapped** field is hidden from the Sensitive toggle but **not** from the Encrypt toggle. That asymmetry is intentional and is called out as such in the component; the backend has no mapped-field rule for Encrypt either. What it does have is a narrower conflict rule: a continuously-syncing mapped field (`ParentOnly` / `TwoWay`) may not be mapped **from** an Encrypted or mid-transition source (`FIELD_MAPPED_FROM_ENCRYPTED_SOURCE`).

Because the normalization above runs **before** the mutual-exclusion check, one of those four kinds submitted with both flags true is normalized, never rejected — its `encrypt` is already `false` by the time the check runs, so the exception never fires for them.

Two adjacent constraints mirror the Sensitive ones: an Encrypted **or mid-transition** field cannot be the entity's main title (`ENCRYPTED_FIELD_CANNOT_BE_MAIN_TITLE` — unlike the mutual-exclusion code, this one *is* localized), while the **sub title** is still allowed to be Encrypted, because the faces that render a sub title mask it themselves. The main-title check reads the *persisted* transition state, not whatever the request body claims.

How the encryption works

Enough detail to plan a deployment and to recognize a stored value — not a cryptography specification:

- **Cipher: AES-256-GCM.** Each value is encrypted under a fresh 12-byte IV with a 16-byte authentication tag; the envelope is `iv:authTag:ciphertext`, each segment hex, joined by `:`. Decryption rejects any envelope that is not exactly three parts, whose IV is not 12 bytes, or whose tag is not 16 bytes, and pins the tag length explicitly — a deliberate hardening against a short, and therefore forgeable, tag being accepted.
- **Two layers, and no KDF.** The **master key** is read straight from the `FIELD_ENCRYPTION_MASTER_KEY` environment variable — 64 hex characters, or base64, decoding to exactly 32 bytes. There is no PBKDF2 / HKDF / scrypt step: the environment value *is* the master key. It wraps a randomly generated 32-byte **data-encryption key (DEK)**, and the DEK is what encrypts field values.
- **One active key per data-service database; one master key per deployment (`local` provider).** The active key is the single document flagged `isDefault: true` in the `EncryptionKeys` collection (an index-enforced flag, `Since 1.10` — it replaced an earlier "the document literally named `default`" convention) — never the raw key material, only a wrapped DEK or, in `direct` mode, no DEK at all. There is no per-tenant, per-entity, per-field or per-user key, and general key *rotation within the same provider* is still out of scope. What **`Since 1.10`** adds is a different capability — the ability to *switch which provider/mode is active* — see Encryption providers below; that is not rotation, and it does not re-encrypt anything already stored. A missing default key is a loud failure (`DEFAULT_ENCRYPTION_KEY_NOT_FOUND`) rather than a silent write of an unencrypted value that merely looks encrypted, and the provider is resolved from the key's own record, so a key created under one provider can never be processed by another. Retiring a key never makes its data unreadable — decryption resolves keys by id from the ciphertext itself, so a retired (non-default) key must never be deleted while it still has surviving ciphertext.
- **A missing master key is a hard failure** at startup or first use — unless *both* `NODE_ENV=development` and `ALLOW_INSECURE_FIELD_ENCRYPTION_KEY=true` are set, in which case an all-zero development key is used. That fallback is never reachable in production.
- **At rest, a value is `<keyId>:<iv>:<tag>:<ciphertext>`** — four colon-separated segments, the key id being everything before the *first* colon. **A multi-value field is encrypted element-wise**: one composed string per array element, not one ciphertext for the whole array. A malformed stored value raises `MALFORMED_ENCRYPTED_VALUE`, whose payload carries only the raw length — never the raw string.
- **Nothing ever detects ciphertext by looking at it.** Whether a given entry's stored value already reflects the field's current `encrypt` setting is a plain integer comparison — `entry.entityVersion >= options.encryptTransitionVersion` — so a string, an array and a block sub-field all behave identically, and a plaintext value can never be coincidentally misread as ciphertext. A write-path guard additionally stops an already-encrypted value being encrypted a second time when an unchanged entry is re-saved.

What an encrypted field refuses

An encrypted value is opaque to the database, so every operation that would need to read *into* it is refused outright rather than silently producing nonsense:

Operation	On an Encrypted field
Full-text search	Off — `searchable` is forced off when Encrypt is saved.
Filter / query criteria, and loop-back `where`	**Rejected unconditionally** (`SENSITIVE_FIELD_QUERY_NOT_ALLOWED`), ahead of any grant or admin-area check. Holding the grant does not help.
Reaction conditions and criteria	**Rejected**, under their own code `ENCRYPTED_FIELD_NOT_ALLOWED` — "*This field is Encrypted and cannot be used in a reaction condition.*" Encrypted fields are also filtered out of every reaction picker, top-level and internal-entity alike.

Operation	On an Encrypted field
Aggregates on a summary	**Refused outright** — the aggregate does not appear at all, not even masked: an aggregate over ciphertext is meaningless rather than secret. The aggregate picker hides Encrypted fields for the same reason.
Entity main title	Not allowed (`ENCRYPTED_FIELD_CANNOT_BE_MAIN_TITLE`).
Reports	Stripped from both the report field catalogue and the export column set.
Notification and email tokens	A field, expression, or pointer token resolving to an Encrypted field resolves to an empty string instead. (A `Fixed` token is literal text and is unaffected.)
Document export templates, and mapped-field data copies	Masked unconditionally, before any expression is evaluated or anything is rendered.
Grouping	**Does not fail — it collapses.** See below.

Grouping is the one that behaves differently, and deliberately so. A view grouped on a field that later becomes Encrypted (or Sensitive) does not break: the level collapses into **one group headed** `****`, carrying the correct total count and every matched entry. There is no per-viewer variant — a grant holder with an active unmask session sees the same single group as an anonymous visitor — because a saved grouping level is *shared* configuration, not a per-request choice, and grouping by a protected value is useful to nobody. The masked group key carries no field path at all, so no pipeline stage can unwind, look up, convert, or order by it. This replaced an earlier design that returned a `400` naming the field as sensitive — which disclosed more than a single `****` bucket does, and which the user could do nothing about. For the same reason the grouping *picker* no longer filters sensitive fields out at all.

Decrypting a value

An Encrypted field is **never** revealed through the classic masking path — `maskSensitiveFields` refuses it outright, grant and session notwithstanding. It has exactly one reveal mechanism:

```
POST /v2/entities/{entity}/entries/{entryId}/fields/{field}/decrypt
Body: { "stepUpProof"?: string, "rowIndex"?: number }
200 → { "value": <decrypted value> }
```

There is no request body requirement beyond the proof. It is a **POST-as-read**: a method-level voter replaces the class-level edit gate and checks `canViewEntity`, not `canEditEntity`, because the call fetches and reveals a value — it never writes one.

The handler runs in this order:

1. Resolve the entity and the field — `EntityFieldNotExistException` if absent.
2. Check the caller may read **this specific entry** (`ForbiddenToReadEntryException`); the class-level voter alone would let a system-wide viewer read any entity's data.
3. Refuse a repeatable field unless `rowIndex` is given — `DecryptNotSupportedForRepeatableFieldException` / `DECRYPT_NOT_SUPPORTED_FOR_REPEATABLE_FIELD`.
4. Refuse a password-rendered field before the entry is even fetched — `DecryptNotSupportedForPasswordFieldException` (`Since 1.10`) — a password field's stored value is a one-way hash, so there is nothing to decrypt.
5. Refuse a field with nothing to decrypt — `FIELD_NOT_ENCRYPTED`. A field mid a **toggle-off** whose entry has not yet been upgraded deliberately **stays reachable**: that is the documented recovery path, see Toggling Encrypt.

6. **The field grant.** `canAccessSensitiveField` against the field's own `module/entity/field` scope; failure is a `403` (`SensitiveFieldDecryptForbiddenException` / `SENSITIVE_FIELD_DECRYPT_FORBIDDEN`, *"Permission denied. You do not have access to decrypt this field."*).

7. `Since 1.10` **— a fresh, correctly-bound, single-use step-up proof.** `StepUpProofService.consumeDecrypt(body?.stepUpProof, {userId, sessionId, entityId, entryId, fieldId, rowIndex})` atomically redeems the proof (Redis `GETDEL`) and re-compares every bound field. A missing, expired, already-spent, or mismatched proof throws `SensitiveFieldDecryptStepUpRequiredException` — the same exception whether the proof was never supplied or was wrong, so the response never distinguishes "you forgot to verify" from "your verification didn't match this exact request." See Step-up unmask, Decrypt, and WorkflowAction below for how the proof is minted.

8. Decrypt via `readEncryptedFieldValue`. A `null` / `undefined` value, or one that doesn't parse cleanly, passes through the same way every other Encrypted field type does — decryption never invents a value.

9. Publish `SensitiveDataDecryptedEvent`.

The grant and the proof are both required — neither alone is enough, and there is **no admin exemption**: unlike `WorkflowAction` (below), which skips step-up entirely for an admin execution, Decrypt applies the check to every caller, precisely so a grant holder is never contradicted by letting an admin bypass it silently. A denied caller (no grant) may retry the identical request indefinitely and always gets the same `403` — there is nothing to guess there, so no lockout applies to the grant check itself. A caller who *has* the grant but keeps failing the step-up challenge (wrong code/password) hits the same account-scoped OTP/MFA rate limit the classic Unmask flow shares — see Step-up unmask session above.

Every successful decrypt is audited under its **own event name** — not as an extra unmask record — carrying who did it, the module, entity and fields, and the entries involved. **The decrypted value itself is never recorded.** This is a separate audit record from the step-up identity check itself, which is recorded too — see Step-up unmask, Decrypt, and WorkflowAction below. See Monitoring: Logs & Audit [Administration › Monitoring: logs & audit] and the step-up-verify record [Administration › Monitoring: logs & audit] for what the administrator sees, and Account Security & Sensitive Data Reveal [Using OSOS › Account Security & Revealing Protected Values] for what the person clicking the icon experiences.

Step-up unmask, Decrypt, and WorkflowAction: three purposes, one mechanism `Since 1.10`

`Since 1.10` the platform generalized what had been a two-purpose step-up mechanism (`Unmask`, pre-existing since 1.8; `WorkflowAction`, added in the same release — see Workflow model and API reference: step-up verification before executing an action) into three, by adding `Decrypt`. All three are members of one `StepUpPurpose` enum (`packages/osos-shared/src/step-up-proof.ts`; see also `core-reference-enums.md`), share the same four `users-service` endpoints (`POST /identity/mfa/step-up/{challenge,verify,disable}`, `GET .../status`) and the same pending-challenge Redis slot keyed by account + OIDC session — but purpose is fixed at challenge time and the three are **never interchangeable**: `verifyUnmask` branches on `pending.purpose`, and a proof or session minted for one purpose can never satisfy another.

Purpose	Status	Mechanism	Binding	TTL
`Unmask` (`'unmask'`)	Pre-existing, **Since 1.8**	A reusable **session** in Redis, opened for a caller-chosen duration, scope wildcard-matchable up to global `*/*/*`	Account + OIDC session + scope	Caller-chosen, from the allow-list in Reveal durations

| `Decrypt` (`'decrypt'`) | **New,** `Since 1.10` | A **single-use proof** — exactly one field reveal, then destroyed on redemption | `{entityId, entryId, fieldId, rowIndex?}` — every field re-compared on redemption | 120 seconds | | `WorkflowAction` (`'workflow-action'`) | **New,** `Since 1.10` | A **single-use proof per entry** — batchable: one factor check can mint a `proofs: {entryId: proof}` map covering a whole selection | `{entityId, entryId, actionId}` × N, one binding per entry | Same 120 seconds, per proof |

Design details that hold across all three purposes:

- **The proof is the secret.** For `Decrypt` / `WorkflowAction`, minting generates a `crypto.randomUUID()` and stores it, with its binding, at a proof-scoped Redis key. Redemption uses `GETDEL` — atomic, so exactly one caller can ever win — and a **mismatched presentation still burns the proof** rather than leaving it available to retry.
- **Minting refuses a session-less caller.** A `client_credentials`/ROPC caller (no OIDC session) is refused a `Decrypt` or `WorkflowAction` challenge outright — such a proof could never be redeemed safely or audited. `Unmask` carries no such restriction, since a session-less caller there simply falls back to an account-wide grant.
- `Decrypt` **uniquely always offers the password fallback, independent of the tenant's** `allowPasswordStepUp` **domain setting.** `Unmask` and `WorkflowAction` only fall back to the account password when the caller has no usable MFA method **and** the domain has explicitly opted in. `Decrypt` ignores that setting and always allows the fallback (still gated on the user having no usable MFA method and holding a local password) — a grant holder must never be left with literally no way to ever reveal a field they hold a grant for, on a default-configured domain that never enabled the MFA-less fallback for anything else.
- **A** `WorkflowAction` **challenge may batch;** `Decrypt` **never does.** A `WorkflowAction` batch is capped at 100 bindings, every binding sharing the same `entityId` / `actionId` (one action, many entries — never a mixed batch), with no duplicate `entryId`; the batch's effective expiry is the **first** proof's `expiresAt`, since the batch is only usable while its earliest member still is. `Decrypt` accepts only a single binding — one reveal per proof, every time.
- **Audit.** Both new purposes record their identity-check outcome on the existing MFA audit trail via dedicated helpers, **distinct from, and in addition to**, the data-entry audit event a successful reveal or workflow-action execution separately produces. See Monitoring: Logs & Audit [Administration › Monitoring: logs & audit].

Recovery-code regenerate/view is a fourth, separate, simpler mechanism — not a `StepUpPurpose`**.** `POST /identity/mfa/totp/recovery-codes/regenerate` takes a code or password directly in the request body and verifies inline, the same pattern the pre-existing TOTP-disable endpoint uses. It shares no Redis session, no single-use proof, and no `StepUpPurpose` value with the three purposes above — do not describe it using step-up vocabulary. See Account Security & Sensitive Data Reveal [Using OSOS › Account Security & Revealing Protected Values] for the user-facing flow.

Full workflow-action detail — the `stepUp` config property on a workflow action, the endpoints, the bulk batching contract end to end, and the refusal shape — is in Workflow model and API reference: step-up verification before executing an action. The comparison below covers `Decrypt` specifically, against the classic `Unmask` session:

	Sensitive: step-up unmask	**Encrypted: Decrypt** `Since 1.10`
Precondition	grant **and** an active step-up session covering the scope	grant **and** a fresh, unspent, correctly-bound single-use proof
Step-up (OTP/MFA/password)	Yes — opens a session	**Yes,** `Since 1.10` — mints a proof good for exactly one reveal
Session vs. proof	Redis session, caller-chosen duration, scoped to the OIDC session, reusable for every matching field until it expires	Redis single-use proof, 120s TTL, bound to one exact `{entity, entry, field, rowIndex?}`, destroyed on first use whether or not it matched

	Sensitive: step-up unmask	**Encrypted: Decrypt** `Since 1.10`
Reach of one reveal	every scope the session covers, in practice all of them (wildcard-matchable up to global)	exactly one field, on one entry (+ row, for a block-repeatable field)
How the value arrives	the next read comes back unmasked	returned inline by the call itself
Repeatable fields	supported	**rejected** unless `rowIndex` is given
Audit event	`SENSITIVE_DATA_UNMASKED`; no separate step-up-verify row for this purpose	`SENSITIVE_DATA_DECRYPTED` **and** a `Decrypt`-tagged step-up-verify row (MFA audit trail)
Admin exemption	N/A — Unmask *is* the per-user session	**None.** Unlike `WorkflowAction`, which skips step-up for an admin execution, Decrypt applies to every caller.

What the client does, `Since 1.10`**.** Both web and mobile now open a step-up prompt before calling decrypt, on every tap — not just the first. Web puts the Decrypt and WorkflowAction prompts on one shared dialog shell but deliberately does **not** route Decrypt through the same helper `WorkflowAction` uses, because that helper applies the admin exemption — which must never apply to Decrypt. Mobile opens the same generalized verification prompt the workflow-action gate uses. Neither client offers the reveal affordance at all for a password-rendered field — the permanent lock shows instead, matching the backend guard in step 4 above.

Toggling Encrypt: a versioned breaking change

There is **no data-backfill migration** behind the Encrypt toggle. Turning it on — or off — is treated as a versioned breaking schema change and resolved through the ordinary **Upgrade Records** flow:

1. Changing `encrypt` adds a `Field encryption changed` reason to the new entity version's breaking-change list. It is added **even when another reason already applies** (a type change, say), so the stamping in step 2 always happens.
2. Every affected field is stamped with `encryptTransitionVersion` (the new entity version) and `pendingEncryptUpgrade = true`. Neither is admin-facing, and both are re-stamped from the persisted field on every field update regardless of what the request body sends — so a client that omits them cannot silently erase a transition in progress.
3. Entries below that version become **Obsolete**, and the administrator resolves them through Upgrade Records as usual. For the whole window the field is excluded from every builder surface.
4. After each successful upgrade the platform counts the entries still below the transition version — **ignoring archived entries**, which would otherwise stop the count ever reaching zero and leave the flag stuck forever — and clears `pendingEncryptUpgrade` when none are left.
5. **Auto Repair is refused when the toggle is being turned OFF.** `SYSTEM_BEHAVIOR_NOT_ALLOWED_FOR_ENCRYPTION_TOGGLE_OFF`: *"Auto Repair cannot be used to resolve this field while its Encrypt toggle-off is still in progress. Choose Fixed, Expression, Dynamic, or Not Set instead."* The stored value is still real ciphertext, and once `encrypt` is `false` nothing downstream re-encrypts or masks it — Auto Repair would promote the raw ciphertext to the entry's real value going forward. **The documented recovery is to call Decrypt on the entry — which stays reachable precisely for this case — and supply the plaintext as the dialog's** `Fixed` **resolution.** Auto Repair remains available in the toggle-**on** direction, which is the intended "just encrypt it" choice.

Re-toggling Encrypt while a transition is still pending **is allowed**: an earlier guard that rejected it was removed after launch.

Encryption providers and keys `Since 1.10`

Which key service enciphers field values (`local` , HashiCorp Vault Transit or Azure Key Vault), the `wrap` and `direct` key modes, provider configuration, activation, adopting a provider change on a running install, and the chart wiring are operator concerns, documented in Field encryption keys and providers. Nothing on this page changes with the provider: grants, masking, Decrypt and the refusals behave the same.

Sensitive-data access policy `Since 1.10`

A grant answers "may *this reader* see this field?". But some things run, or are generated, on someone's behalf — a reaction, an export template, a summary — and there the question is different: "what should *this element* be allowed to disclose when it runs?". That is a separate, element-level setting: the **Sensitive Data Access Policy**.

Policy	What it means
Prohibited	Never reveal, whoever executes, opens, or generates the element.
Privileged	Always reveal, independently of the executor's own grants. **The default.**
User-Based	Decide per execution, from the actual executor's own sensitive-field grants.
Auto	A system run behaves as Privileged; a user-triggered run behaves as User-Based.

It is stored as `sensitiveDataAccessPolicy` on the element. **An absent value resolves as `Privileged`**, so every reaction, template and summary that predates the feature keeps exactly the behavior it had before the setting existed — nothing pre-existing changed, and no backfill migration was needed. An *unrecognized* stored value resolves to mask: the resolver fails closed.

Where it applies — and where it deliberately does not:

Element	Policy available?
Reaction	Yes — all four values.
Document export template	Yes — all four values.
Summary (dashboard tile / statistic query)	Yes, **except `Auto`** — a summary has no automatic or scheduled run distinct from its live rendering, so the option is not offered at all.
Report	**Deliberately not.** A report keeps following each viewer's own sensitive-field permission instead.
Views, plain queries, entry lists	Not part of this policy at all.

The control appears on the reaction, template and summary forms **only when the entity actually declares at least one sensitive field** (block sub-fields included); when it is hidden it is disabled rather than left silently blocking Save.

Three rules are enforced when the element is saved or run:

- **`Auto` on a summary** is rejected — *"The 'Auto' option is not available on a summary"*.
- **`User-Based` together with an automatic reaction trigger** — a system executor, a timer, or an SLA-exceeded event — is rejected: *"The 'User-Based' policy cannot be selected with an automatic trigger type"*. There is no human executor whose grants could be consulted.
- A reaction whose policy masks a value it then needs to **act on** *fails* (`SENSITIVE_DATA_ACCESS_PROHIBITED`) rather than persisting `****`. Only the five rendering actions — send notification, send email, send SMS, generate file from template, announce event — tolerate a masked value passing through them.

A summary the policy denies is hidden entirely, not masked. There is no chart, no card, and no `****` : the request is a `403` and the dashboard renders *"You do not have permission to view this summary"* in the tile's place. Under `User-`

`Based` the executor must hold a grant for **every** sensitive field the summary reads — a partially-revealed aggregate is still a leak — and that now includes the fields the summary's own **source query** filters on, not only the aggregated and grouped ones. (Contrast the per-field rule for an ordinary aggregate a *reader* simply has no grant for: that one stays present under its display name showing `****`, because a vanished aggregate was itself a disclosure.)

End-to-end flow summary

1. A designer marks a field `sensitive` and picks users/groups in the field editor, or an admin grants access later through the grants API. The scope is `module/entity/field`, or a wildcard scope for an inherited grant.
2. On save, `syncSensitiveFieldAccess` (field editor) or `createSensitiveFieldGrants` (admin API) writes `SENSITIVE_FIELD` `AccessRight` records for the scope and publishes `AccessRightChangedEvent`.
3. When permissions are resolved, matching grant scopes flow into `permissions.sensitiveFields`.
4. To actually *see* a value, the user must additionally open a step-up unmask session covering the scope; that session belongs to their specific sign-in session, not just their account.
5. On entry read, `maskSensitiveFields` reveals a field only when both the grant and an active unmask scope match; otherwise it substitutes the mask sentinel. Revealed fields publish a batched `SensitiveDataUnmaskedEvent` for audit. The entity definition reports masked field names in `sensitiveMaskedFields` (no grant) and `sensitiveMaskedFieldsWithUserAccess` (grant held, session not covering it).
6. On query, unreadable sensitive fields in user criteria/where cause a `SensitiveFieldQueryNotAllowedException` (grant-only check, no session gate).
7. On audit read, sensitive `old` / `new` values are replaced with `***`, now requiring both the grant **and** an active unmask session covering the scope; the audit writer masks, fail-closed, before persisting.
8. If the field mirrors another entity's field (a **mapped field**), its sensitivity and grant holders are kept in step with the source field automatically, in both directions — see Mapped fields inherit sensitivity.
9. `Since 1.10` If the field is **Encrypted** rather than Sensitive, steps 4–6 do not apply at all: the value is ciphertext at rest, every builder surface refuses it (grouping collapses instead), and the only way to see it is a per-entry Decrypt call gated on the grant **and** a fresh single-use step-up proof (`Since 1.10`) — no reusable session, but no longer verification-free either — audited under its own event name plus a separate step-up-verify record.
10. `Since 1.10` A reaction, an export template, or a summary additionally carries its own sensitive-data access policy, deciding what *that element* may disclose when it runs. The default, `Privileged`, is exactly what all three did before the setting existed.
11. `Since 1.10` The Decrypt reveal and a gated `WorkflowAction` execution both go through the same generalized three-purpose step-up mechanism — see Step-up unmask, Decrypt, and WorkflowAction.

Reference: key symbols

Symbol	File	Role	Since
`buildSensitiveFieldScope`	`libs/sensitive-field.ts`	Build a `module/entity/field` scope.	`1.8`
`matchesSensitiveScope`	`libs/sensitive-field.ts`	Wildcard segment match.	`1.8`
`SENSITIVE_DISPLAY_MASK`	`libs/sensitive-field.ts`	The 26-character sentinel actually returned on read.	`1.8`

Symbol	File	Role	Since
`SENSITIVE_STARS` / `renderSensitiveMaskAsStars`	`libs/sensitive-field.ts`	Human-facing `****` and the text-substitution sanitizer that produces it.	`1.8`
`SENSITIVE_MASK` (legacy) / `AUDIT_SENSITIVE_MASK`	`libs/sensitive-field.ts`	Legacy sentinel detector / audit mask.	`1.8`
`SensitiveFieldsService.canAccessSensitiveField`	`services/sensitive-fields.service.ts`	Binary grant check (no admin bypass).	`1.8`
`SensitiveFieldsService.filterQueryableFieldsForUser`	`services/sensitive-fields.service.ts`	Drop inaccessible sensitive fields from a list.	`1.8`
`SensitiveFieldsService.getSensitiveFieldOptions`	`services/sensitive-fields.service.ts`	Space-wide catalogue (optional `excludeScopes`).	`1.8`
`AccessRightsService.syncSensitiveFieldAccess`	`services/access-rights.service.ts`	Declarative grant diff + event (field-editor path).	`1.8`
`AccessRightsService.listSensitiveFieldGrants`	`services/access-rights.service.ts`	Space-wide Active grants list.	`1.8`
`AccessRightsService.listUserSensitiveFieldGrants`	`services/access-rights.service.ts`	A user's direct + group-inherited grants.	`1.8`
`AccessRightsService.listGroupSensitiveFieldGrants`	`services/access-rights.service.ts`	A group's direct grants.	`1.8`
`AccessRightsService.getInheritedSensitiveFieldHolders`	`services/access-rights.service.ts`	Wildcard-scope holders for one field.	`1.8`
`AccessRightsService.createSensitiveFieldGrants` / `updateSensitiveFieldGrant` / `deleteSensitiveFieldGrant` / `archiveSensitiveFieldGrant`	`services/access-rights.service.ts`	Grants-management API mutations.	`1.8`
`AccessRightsService.getSensitiveFieldGrants`	`services/access-rights.service.ts`	Read grants under an entity scope prefix.	`1.8`
`AccessRightsService.removeSensitiveFieldAccess`	`services/access-rights.service.ts`	Tear down grants at or under a scope.	`1.8`
`DataEntityService.attachSensitiveFieldsAccess`	`services/data-entity.service.ts`	Repopulate `sensitiveAccess` on read.	`1.8`
`DataEntityService.computeSensitiveMaskedFields`	`services/data-entity.service.ts`	Per-request masked field-name lists (`sensitiveMaskedFields` + `sensitiveMaskedFieldsWithUserAccess`).	`1.8`
`DataEntityService.propagateMappedFieldSensitivity` / `propagateSensitivityToDependentMappedFields`	`services/data-entity.service.ts`	Mapped-field sensitivity inheritance, both directions.	`1.8`
`DataEntryService.maskSensitiveFields` / `applyCallerSensitiveMask` / `maskEntryForResponse`	`services/data-entry.service.ts`	Grant + session read-side masking.	`1.8`
`DataEntryWriterService.resolveSavedEntryUnmasked`	`services/data-entry-writer.service.ts`	Unmasked after-save re-read.	`1.8`
`UnmaskSessionService.getActiveScopes`	`services/unmask-session.service.ts`	Read a user's active, session-scoped step-up scopes (fails closed).	`1.8`
`challengeUnmask` / `verifyUnmask` / `disableUnmask` / `getUnmaskStatus`	`users-service/src/services/step-up-unmask.service.ts`	Step-up session lifecycle.	`1.8`

Symbol	File	Role	Since
`stepUpUnmaskRedisKey(envPrefix, userId, sessionId)` / `StepUpUnmaskSession` / `STEP_UP_UNMASK_ALL_SCOPE` / `STEP_UP_UNMASK_DURATIONS_SECONDS`	`packages/osos-shared/src/step-up-unmask.ts`	Cross-service session contract; session-scoped keying and the current duration set are both this release.	`1.8`
`PermissionTypeEnum.SensitiveField`	`packages/osos-shared/src/models/enums.ts`	`'SENSITIVE_FIELD'`.	`1.8`
`FieldService.validateAndNormalizeEncrypt`	`services/field.service.ts`	Normalize the four excluded kinds, reject `sensitive + encrypt`, force `searchable = false`.	`1.10`
`options.encrypt` / `encryptTransitionVersion` / `pendingEncryptUpgrade`	`models/data-entity/field-options.model.ts`	The Encrypt toggle and its two server-owned transition companions.	`1.10`
`aesGcmEncrypt` / `aesGcmDecrypt`	`libs/aes-gcm.ts`	AES-256-GCM envelope (`iv:tag:ciphertext`), 16-byte tag pinned; shared with TOTP secrets.	`1.10`
`getFieldEncryptionMasterKey`	`libs/field-encryption-master-key.ts`	Read and validate `FIELD_ENCRYPTION_MASTER_KEY` (32 bytes, hex or base64).	`1.10`
`writeEncryptedFieldValue` / `readEncryptedFieldValue` / `isFieldValueMigrated`	`libs/encrypted-field-value.ts`	At-rest `<keyId>:<iv>:<tag>:<ciphertext>` composition and the version-compare migration test.	`1.10`
`DataEntriesController.decryptField`	`controllers/data-entries.controller.ts`	`POST …/fields/{field}/decrypt` — grant **and** single-use step-up proof, one field, one entry.	`1.10`
`SensitiveDataDecryptedEvent`	`packages/osos-shared/src/messages/events/sensitive-data/`	The dedicated decrypt audit event; never carries the value.	`1.10`
`DataEntity.getFieldsExcludedFromBuilders`	`models/data-entity/data-entity.model.ts`	`encrypt || pendingEncryptUpgrade` — the one definition nearly every refusal traces back to.	`1.10`
`SensitiveDataAccessPolicyEnum` / `resolveSensitiveDataAccessDecision`	`packages/osos-shared/src/sensitive-data-access-policy.ts`	The four policies, the default `Privileged`, and the fail-closed resolver.	`1.10`
`SensitiveFieldsService.canQuerySensitiveField`	`services/sensitive-fields.service.ts`	Query capability = admin area **or** grant; deliberately weaker than the read rule.	`1.10`
`StepUpPurpose` (`Unmask` / `Decrypt` / `WorkflowAction`)	`packages/osos-shared/src/step-up-proof.ts`	Discriminates the three step-up purposes; fixed at challenge time, never interchangeable.	`1.10`
`mintDecryptProof` / `mintWorkflowActionProof(s)` / `auditDecryptStepUp` / `auditWorkflowActionStepUp`	`users-service/src/services/step-up-unmask.service.ts`	Single-use proof minting + MFA-audit-trail recording per purpose.	`1.10`
`stepUpProofRedisKey` / `stepUpProofMatchesBinding` / `validateStepUpBatch` / `STEP_UP_PROOF_TTL_SECONDS` / `STEP_UP_PROOF_MAX_BATCH`	`packages/osos-shared/src/step-up-proof.ts`	Cross-service single-use-proof contract; 120s TTL, 100-entry batch cap.	`1.10`

Symbol	File	Role	Since
`StepUpProofService.consume` / `consumeDecrypt`	`data-service/src/services/step-up-proof.service.ts`	Atomic `GETDEL` proof redemption + binding comparison.	`1.10`
`DecryptNotSupportedForPasswordFieldException`	`data-service/src/exceptions/`	Rejects Decrypt for a password-rendered field before the entry is fetched.	`1.10`
`VaultEncryptionProvider` / `AzureKeyVaultProvider`	`data-service/src/encryption-providers/`	HashiCorp Vault Transit / Azure Key Vault providers, `wrap` and `direct` modes.	`1.10`

Related pages

- Protect a field [Build Apps] — the builder's guide to marking fields Sensitive or Encrypted.
- Sensitive Permissions [Administration] — the administrator-facing procedure for granting, editing, and revoking these grants.
- Permissions & Access Control [Administration › Permissions & access] — the object-level permission ladder this dimension sits beside.
- Entity model and API reference — entity and field definitions.
- Workflow model and API reference — the `WorkflowAction` step-up purpose, batching, and refusal shape.
- Reference: Enums — the `StepUpPurpose` enum definition.
- Monitoring: Logs & Audit [Administration › Monitoring: logs & audit] — what an administrator sees for a Decrypt reveal and its step-up identity check.
- Field encryption keys and providers and Configuration Reference — encryption keys and environment variables.
- Account Security & Sensitive Data Reveal [Using OSOS › Account Security & Revealing Protected Values] — the end-user-facing description of the reveal wizard and the Decrypt prompt.

Module model and API reference

This page is for **integrators and platform engineers** who create, read or delete modules through the API, build navigation from the sidebar endpoint, or need the exact authorization and deletion rules.

> **Managing modules in the UI?**
>
> Builders should start with Modules [Build Apps].

All module APIs live in the **data-service**, mounted under `/v1/data-module` (plus the sidebar under `/v2/side-bar`). Every endpoint requires authentication via the OSOS Auth2 strategy.

Overview

A module is a lightweight grouping object. It does **not** embed its entities; instead, each `DataEntity` carries a `module` reference pointing back to the module that contains it. Deleting a module is therefore blocked while any entity still points at it.

Two kinds of modules exist:

- **Regular modules** — created by developers, fully editable and deletable. Each groups a set of related functions into an integrated unit (for example, a module dedicated to task management, or one for human resources).
- **Implicit modules** (`isImplicit: true`) — system-managed containers that cannot be edited or deleted through the API. The core/system module (`00000000-0000-0000-0000-000000000000`) is treated as a restricted module and is excluded from normal listings unless explicitly requested.

When a new module is created, OSOS automatically provisions a default **dashboard** and a default **landing page** for it. A module also carries a **sharing level** (Private/Public/Shared) and an **order** that determines its position in the sidebar.

The normalized name

The normalized name is uniquely indexed and pattern-validated. The portal sends the stored normalized name on every edit, so it cannot be changed there after creation; `PATCH /v1/data-module/{id}` does persist a new `normalizedName` (see Update by id), which risks a duplicate-name conflict and breaks anything that binds to the old name — change it only deliberately.

Dependencies between modules

A module is an independent, self-contained unit that executes its functions without depending directly on other modules. Where cross-module interaction is needed, modules communicate through a **virtual (proxy) entity** rather than a direct dependency.

The core module

The core module is a default, system-provisioned module that holds the essential entities OSOS needs to set up its environment. It is created automatically, and users cannot modify its information or create entities under it the way they can with other modules. Internally it is the restricted module id `00000000-0000-0000-0000-000000000000`, filtered out of listings by default and protected by the restricted-module authorization path.

Data model

`DataModule` extends `Identified<UUID>` (giving it a string `id`) and implements `IName` and `ISpecificationVersion<1>`.

Field	Type	Required	Notes
`id`	`UUID` (string)	assigned	UUID v4; generated server-side on create if omitted.
`specificationVersion`	`1`	yes	Fixed literal `1`, defaults to `1`.
`normalizedName`	`string`	yes	Machine name; **unique index**. Validated against `NormalizedNameValidation` (min/max length + pattern).
`name`	`LocalizedValue`	yes	Localized display name (per-language strings).
`description`	`NullableLocalizedValue`	no	Localized, nullable.
`icon`	`Icon`	yes	Object with a required, non-empty `name` string.
`isImplicit`	`boolean`	yes	Marks a system-managed module.
`creationInfo`	`ActionInfo`	no*	Stamped server-side on create (who/when, plus optional delegatee).
`updateInfo`	`ActionInfo`	no*	Re-stamped server-side on every update.
`sharing`	`ApplicationSharing`	yes	Controls cross-space visibility (see Sharing).

* `creationInfo` and `updateInfo` are declared `required: false` on the model but are always populated by the service; they are excluded from request bodies.

Icon

`Icon` is a small embedded object (`idInjection: false`, nullable at the schema level) with a single required `name: string`. The controller additionally rejects an empty or whitespace-only icon name at write time.

Action stamping

On create, the service builds an `ActionInfo` with `doneAt` (server time), `doneBy` (a `ReferenceProperty` to the `USER` entity with the acting user's id), and an optional `delegatee` reference when the request is made on behalf of another user (`user.delegatee.dataEntry`). Both `creationInfo` and `updateInfo` are set to this value at creation; `updateModule` overwrites `updateInfo` with a fresh stamp (no delegatee) on each update.

Sharing

`sharing` is an `ApplicationSharing` object:

Field	Type	Notes
`level`	`CategoryComponentEntryValue<ApplicationSharingLevelEnum>`	The sharing level.
`spaces`	`ReferenceProperty[]`	Spaces the module is shared with (used for `PRIVATE` / `SHARED`).

`ApplicationSharingLevelEnum` (from `@osos/shared`):

Value	Meaning
`PUBLIC`	Visible to all spaces.

Value	Meaning
`PRIVATE`	Visible only to the listed `spaces`.
`SHARED`	Visible to the owning space and the listed `spaces`.

`DataModuleService.getSpaceModules(spaceId)` resolves the modules visible to a space using exactly these rules: any `PUBLIC` module, plus `PRIVATE` or `SHARED` modules whose `sharing.spaces.dataEntry` includes the space id.

Long-standing — per-module sharing (module/entity sharing) predates the 1.2 line and is baseline for all supported versions; no availability badge is shown.

DTOs

`DataModuleCreateDto`

Extends `DataModule` and adds two write-only helpers used by the create request:

Field	Type	Notes
`entities`	`UUID[]`	Optional, defaults to `[]`.
`permissions`	`Permission[]`	Optional. Stripped from the module document and stored separately as access rights (see below).

`LiteModule` / `LiteModuleWithViews`

Trimmed projections used by navigation payloads: `id`, `specificationVersion`, `name`, `icon` — and, for `LiteModuleWithViews`, an array of `LiteDataViews`. `SidebarDto` (v1) is composed of these plus system views and a service-status block.

`ExportingDataModuleDto` / `ExportingDataModulesDto`

Packaging DTOs used when exporting a module for sharing across spaces: a module reduces to its `id` plus an array of `ExportingEntityDto` (`dataEntities`); `ExportingDataModulesDto` wraps a list of them. A `ForbiddenToExportModule` exception guards the export path.

REST API

Base path: `/v1/data-module`. All routes are authenticated. Roles and voters are noted per route.

Method & path	Purpose	Authorization
`POST /v1/data-module`	Create a module	Role `SystemDeveloper`
`GET /v1/data-module/count`	Count modules	Role `SystemViewer` or `Restricted`
`GET /v1/data-module`	List modules	Role `SystemViewer` or `Restricted`
`GET /v1/data-module/{id}`	Get one module	`moduleVoterFactory(0)`
`PATCH /v1/data-module/{id}`	Update a module	`moduleVoterFactory(0)`
`DELETE /v1/data-module/{id}`	Delete a module	`moduleVoterFactory(0)`
`DELETE /v1/data-module/{id}/permanent`	Delete a module and everything in it (`Since 1.10`, background job)	Role `SystemAdmin` (license must include administration)

Create (`POST /v1/data-module`)

Request body is a `DataModuleCreateDto` with `id`, `creationInfo`, and `updateInfo` excluded (they are server-assigned).

Flow (`create` → `moduleChecker` → `DataModuleService.createModule`):

1. **Validation** (`moduleChecker`), in order:
   - `languagesService.validateName(name)` — may throw `NameEnIsRequired`, `NameEnIsShort`, `NameEnIsLong`.
   - `checkIcon` — throws `DataModuleIconIsRequired` if `icon.name` is empty.
   - `checkDuplicateNormalizedName` — throws `ModuleDuplicateException` if another module already uses the same `normalizedName`.
   - `checkDuplicateNamesFromDatabase` — throws `ModelWithGivenNameExistException` if a localized name collides.
2. `permissions` is pulled off the DTO and passed separately (not stored on the module document).
3. The service assigns `id` (UUID v4 if absent), sets `specificationVersion: 1`, stamps `creationInfo` / `updateInfo`, and persists via `DataModuleRepository`.
4. Publishes `DataModuleCreatedEvent` on the service bus.
5. If `permissions` were provided, stores them as access rights (`EntitlementEnum.Module`, keyed by the new module id). A failure here is logged and re-thrown.

Returns the created `DataModule`.

```
POST /v1/data-module
Content-Type: application/json

{
  "normalizedName": "hr_module",
  "name": { "en": "Human Resources", "ar": "الموارد البشرية" },
  "description": { "en": "HR entities and views" },
  "icon": { "name": "users" },
  "isImplicit": false,
  "sharing": {
    "level": { "id": "PUBLIC", "name": { "en": "Public" } },
    "spaces": []
  },
  "permissions": []
}
```

Count (`GET /v1/data-module/count`)

Query params:

- `where` — standard LoopBack `where` filter on `DataModule`.
- `includeCoreModule` (boolean) — when falsy (default), the core module id `00000000-0000-0000-0000-000000000000` is excluded via `neq('id', ...)`.

Counting goes through `countWithAccessRight`, which scopes the result to what the caller can actually see (see Access-right scoping).

List (`GET /v1/data-module`)

Query params:

- `filter` — standard LoopBack filter on `DataModule`.
- `includeCoreModule` (boolean) — same core-module exclusion as `count`.

Results come from `findWithAccessRightInSpace`, using an English collation (`{collation: {locale: 'en'}}`) for ordering. As with count, results are scoped to the caller's access rights.

Get by id (`GET /v1/data-module/{id}`)

`getModule` applies the caller's sharing-level filter (`spaceService.getFilterBySharingLevel`) on top of the `id` match and reads via `findOneInSpace`. If nothing matches, it throws `EntityNotFoundError` (404). Any supplied `filter` excludes `where`.

Update by id (`PATCH /v1/data-module/{id}`)

Partial body (`DataModule`, `partial: true`) with `id`, `creationInfo`, and `updateInfo` excluded. Flow:

1. `getModule(id)` — 404 if not visible.
2. `moduleChecker(..., id)` — same validation as create, but duplicate checks ignore the module's own id (`neq: selfId`).
3. `DataModuleService.updateModule` writes `name`, `description`, `normalizedName`, `icon`, `sharing`, and a fresh `updateInfo`, then re-reads the document.
4. Publishes `DataModuleUpdatedEvent`, carrying a `changes` diff. The diff is computed by `identifyChanges`, which currently tracks **only** `sharing` changes (old/new values).

Returns the updated `DataModule`.

> The service persists `normalizedName` on update, but it remains the module's stable key. The portal presents it as fixed after creation, and renaming it at the API level risks a duplicate conflict — change it only deliberately.

Delete by id (`DELETE /v1/data-module/{id}`)

Guards, in order:

1. `getModule(id)` — 404 if not visible.
2. If `isImplicit`, throw `ImplicitModuleException` — implicit modules can't be modified or deleted.
3. If any `DataEntity` still references the module (`findInSpace` where `module == id`), throw `ModuleHasEntities` — move or delete its entities first.
4. `DataModuleService.deleteModule` deletes the document and publishes `DataModuleDeletedEvent`.

Returns `204` (no content). To delete a module that still holds content, use the permanent delete below.

Permanent delete (`DELETE /v1/data-module/{id}/permanent`) `Since 1.10`

The route above only removes an **empty** module. The permanent delete removes a module **and everything in it**, as a background job. It is system-admin only (`allowedRoles: [SystemAdmin]`; the license must include administration).

Guards, in code order (`ModulePurgeService.assertPurgeable`):

Check	Refusal
The module is visible to the caller	`404`
It is not the core module and not implicit	`400 MODULE_PURGE_NOT_ALLOWED`

Check	Refusal
No delete is already running (a failed or stalled run is resumed instead)	`409 MODULE_BEING_DELETED`
Every entity in it can be purged by the entity permanent delete	`400 MODULE_PURGE_NOT_ALLOWED` with `details.entityIds`
Neither the module nor any entity or page in it was installed from a package	`403`
No entity in it is locked by another user	`409 LOCKED_OBJECT`
Nothing **outside** the module still references the module or anything in it (dependency-service `GET /v1/external-dependents/DataModule/{id}`)	`409 MODULE_HAS_DEPENDENTS` (`details.count`), or `503 DEPENDENCY_CHECK_UNAVAILABLE` when the check cannot answer

The last check is the "unlink first" rule: the portal dialog lists those references, each with an **Open** link, and enables Delete only when none remain.

Response. The route answers **before** the work is done, with `200 {success: true, result: null}` (the OpenAPI document declares `202`). It marks the module with `purgeInfo` and `purgeStatus: {step: 'queued'}` through a conditional write (`409 UPDATE_CONFLICT` if another request marked it first), archives the module's sidebar link and the links under it, sends `PurgeDataModuleCommand` and publishes `DataModulePurgeStateChangedEvent` (`deleting`). It publishes no `DataModuleUpdatedEvent`.

The job. `ModulePurgeService.run` claims the module with a fresh run id and performs idempotent steps, recording each in `purgeStatus.step`:



- `entities` purges every entity of the module (archived ones and those in any space included) through the entity permanent delete, counting down `entitiesLeft`.
- `functions` archives the module's functions, so the functions service deprovisions them; `grants` removes the module grants and the sensitive-field rights under the module's name, then refreshes the permission and profile epochs.
- `verify` counts what is left and fails with `MODULE_PURGE_INCOMPLETE` if something was added meanwhile; `event` publishes `DataModuleDeletedEvent` carrying the ids it deleted; `module` deletes the document.
- A failed step is recorded as `purgeStatus.error {step, code, message}` and announced as `failed`. **The module stays marked, and calling Delete again resumes it** (Delete also resumes a run whose status has not moved for 10 minutes). There is no cancel.

While a module is being deleted data-service refuses to edit it, create an entity or virtual entity in it, move an entity into it, or create a page in it (`409 MODULE_BEING_DELETED`); `purgeInfo` and `purgeStatus` are not accepted in create/update bodies. The portal polls `GET /v1/data-module/{id}` every 3 s (a `404` means the delete is done), and the web-socket service pushes `modules/delete-state {moduleId, state}` to every open session of the tenant, so module pages show **Deleting…** or **Delete failed** live.

> **Known limit:** the purge runs inside a bus handler, and data-service consumes one bus message at a time by default, so a long purge holds up that pod's other bus work.

Sorting the Modules and Entities lists `Since 1.10`

The admin **Modules** page, the **System Entities** page and each module's **Entities** tab have the same multi-level **Sort** control as an entity's Queries tab. The sort fields are **Newest created**, **Last modified** and **Name A–Z**; the default is newest created first, and **Reset** restores it.

Both lists page on the server, so the chosen sort is sent to the existing list routes as a LoopBack `order` with an `id` tie-break — there is no new endpoint. Each list remembers its own sort (the Modules page, the System Entities page and every module's Entities tab separately) for the browser session: it survives a refresh, is cleared on sign-out, and is reset when you navigate into the list. Changing the sort keeps the applied filter and returns to page 1.

Authorization

Roles

`create` requires `SystemDeveloper`. `count` / `find` require `SystemViewer` or `Restricted`. The item-level routes (`findById`, `updateById`, `deleteById`) use the module voter instead of a fixed role.

Module voter

`moduleVoterFactory(moduleParamIndex = 0, bodyParamKey?, ignoreRestricted?)` builds an authorizer for a route:

- Internal service calls are allowed outright; if a `view` is already in context, the voter abstains.
- It reads the module id from the argument at `moduleParamIndex` (or from `bodyParamKey` within an object body). If the id is missing/null, it abstains and lets the default provider decide.
- If the id is a **restricted module** (the core module), decisions route through `canViewRestrictedModule` / `canEditRestrictedModule` (unless `ignoreRestricted`).
- Otherwise, **view** is granted via `canViewModuleIncludingEntityGrants(moduleId)` and **edit** via `canEditModule(moduleId)`. Read access is intentionally broader than edit: a user who only holds a grant on an *entity inside* the module can view the module as a navigation container, but editing still requires a real module grant.

Since 1.8 — allowing an entity-only grant to open its parent module (`canViewModuleIncludingEntityGrants` and the `entityParentModuleIds` plumbing) first shipped in the 1.8 line.

Access-right scoping

Both the list and count paths delegate to `DefaultCrudWithSharingAndAccessRightRepository` (used only by the module and entity repositories), which composes **two orthogonal filters** into the Mongo `where`:

1. Space/sharing filter (`getWhereSharing`) — a row is visible only if `sharing.level == Public`, **or** its level is `Private` / `Shared` **and** `sharing.spaces.dataEntry` includes the current `spaceId`. This is the list/count-time counterpart of the per-item `getFilterBySharingLevel` check applied by the get-by-id route.

2. Access-scope filter — the repository calls `accessRightsService.getAllModuleIdsUserCanView()` and branches:

- `AdminAccess` → no filtering (full space, ordered `_id DESC` on list).
- `EditAccess` / `DeveloperAccess` / `ViewAccess` → everything **except** restricted modules (`nin('id', restrictedModuleIds)`).
- Resource-scoped result `{moduleIds, entityIds, entityParentModuleIds}` → for modules, restrict to `moduleIds ∪ entityParentModuleIds` (granted modules plus the parent modules of granted entities, so those parents appear as navigation containers; sibling entities stay hidden). This scope `where` is imposed on the filter so resource-scoped users (for example `Restricted`) only receive what they were granted.

Lifecycle events

The service publishes to the service bus (`@node-ts/bus-core`) on every mutation, with `correlationId` set to the request trace id and sticky attributes carrying `userId` and `spaceId` :

Event	Published by	Payload
`DataModuleCreatedEvent`	`createModule`	Created module + acting user id.
`DataModuleUpdatedEvent`	`updateModule`	Updated module + user id + `changes` (sharing diff).
`DataModuleDeletedEvent`	`deleteModule`	Deleted module + user id.

These events are consumed elsewhere to synchronize the navigation **links** (a link entity is maintained for each module, page, view, and report).

Navigation and the sidebar

Modules are the backbone of portal navigation. The sidebar endpoint returns the links the user can see, grouped so that each module acts as a container for its views and pages.

`GET /v2/side-bar`

`SideBarController.getViewsForSidebarV2` (authorization is skipped at the route; scoping happens inside the services). Inputs:

- Header `x-domain-id` , header `x-space-id` .
- Query `include` (`SidebarIncludeDataEnum` , default `All`):

Value	Name	Returns
`0`	`All`	`links` + `serviceViewsStatus` .
`1`	`Links`	`links` only (`serviceViewsStatus: null`).
`2`	`OnlyService`	`serviceViewsStatus` only (`links: []`).

Response is a `SidebarDtoV2` :

Field	Type	Notes
`links`	`GenericLink[]`	Navigation links (modules and their children).
`serviceViewsStatus`	`SidebarServiceStatus | null`	Service-catalog / requests / inbox status + counters.
`isRestricted`	`boolean`	Whether the caller holds the system-level `Restricted` floor.
`restrictedAccess`	`{hasModuleAccess, hasEntityAccess}`	Only meaningful when `isRestricted` ; drives the restricted UI.

`restrictedAccess` is computed from the user's `module` / `entity` permissions, each flag true when at least one grant is `>= ViewAccess` .

Since 1.4 — the v2 sidebar `include` options and the service-status model first shipped in the 1.4 line. The v1 `SidebarDto` / `LiteModuleWithViews` shape and the links machinery predate the 1.2 line and are baseline.

`SidebarServiceStatus`

Booleans `hasServiceCatalog` , `hasRequests` , `hasInbox` , `hasObserverRequests` and counters `onGoingRequests` , `onGoingInbox` , `onGoingObserverRequests` .

End-user view discovery `Since 1.8`

`GET /v1/views/end-user/modules`

`DataViewsController.findModulesForEndUser` closes a real gap. Every other listing that touches modules or views answers an **authoring** question and is gated accordingly: `GET /v1/data-module` (above) needs role `SystemViewer` or `Restricted`, and the data-service views-by-module endpoint needs a grant on the module or entity and still returns **every published view** in that module, whether or not the caller can actually open it. A user holding nothing but **view** grants — no module or entity grant at all — could not discover their own views through any of these, which made a module-first view picker effectively admin-only.

This endpoint asks the end-user question instead — which views may I actually open, and which module is each one in — and returns `LiteModuleWithViews[]` (see `LiteModule` / `LiteModuleWithViews` above): the modules containing at least one view the caller can open, each with those views. It is also the **first** endpoint that actually serves this shape over HTTP — the v1 `SidebarDto` embeds it too, but no v1 sidebar route is mounted any more; only `GET /v2/side-bar` is live (see above).

Authorization is `@authorize({skip: true})` on purpose: the filtering inside `DataViewService.getModulesWithAccessibleViews` *is* the authorization, and it fails closed — an anonymous caller only gets views with anonymous access enabled. It mirrors the same end-user accessibility check the single-view endpoint uses (anonymous, then authenticated, then an explicit view grant or system-developer-or-above for a restricted view), with one deliberate difference: a view with **neither** access mode enabled is excluded here rather than falling through and still being returned, since listing a view no configured flow can open would be misleading in a picker.

Candidates are scoped to **published, non-implicit** views in the requesting space, grouped under the module resolved through the view's **entity** (not the view's own, possibly stale, `module` snapshot), and sorted by name for a stable picker order.

Exceptions

Exception	Error code	When
`DataModuleIconIsRequired`	`DATA_MODULE_ICON_REQUIRED`	Icon name empty on create/update.
`ModuleDuplicateException`	`MODULE_DUPLICATE`	`normalizedName` already used.
`ModuleHasEntities`	`MODULE_HAS_ENTITIES`	Delete attempted while entities reference the module.
`ImplicitModuleException`	`MODULE_IS_IMPLICIT`	Modify/delete attempted on an implicit module.
`InvalidModuleId`	`INVALID_MODULE_ID`	Provided module id is not valid (404).
`EntitiesNotFound`	`ENTITY_NOT_FOUND`	One or more referenced entities do not exist.
`ForbiddenToExportModule`	`CANNOT_EXPORT_MODULE`	Export not permitted (403).
`ModulePurgeNotAllowedException`	`MODULE_PURGE_NOT_ALLOWED`	`Since 1.10`. Permanent delete of the core or an implicit module, or of one holding an entity the entity purge refuses (400, `details.entityIds`).
`ModuleHasDependentsException`	`MODULE_HAS_DEPENDENTS`	`Since 1.10`. Permanent delete while items outside the module reference it (409, `details.count`).

Exception	Error code	When
`ModuleBeingDeletedException`	`MODULE_BEING_DELETED`	`Since 1.10`. Edit, create-in, move-into or permanent delete of a module whose delete is running (409).
`DependencyCheckUnavailableException`	`DEPENDENCY_CHECK_UNAVAILABLE`	`Since 1.10`. The dependency check could not answer (503, not retried).

Name-validation errors (`NameEnIsRequired`, `NameEnIsShort`, `NameEnIsLong`, `ModelWithGivenNameExistException`) are raised by `LanguagesService` during `moduleChecker`.

Practical notes for builders

- **Create requires `SystemDeveloper`.** Viewing and listing need only `SystemViewer` / `Restricted`, but results are always access-scoped.
- **The normalized name is your stable key** — it is uniquely indexed and pattern-validated; pick it deliberately, since renaming risks a duplicate conflict.
- **Empty the module before deleting — or delete it permanently.** The plain delete route rejects a non-empty module with `ModuleHasEntities`; `Since 1.10` a system administrator can instead delete it permanently with everything in it, after unlinking every reference from outside the module.
- **Don't touch implicit/core modules** — the core module id `00000000-0000-0000-0000-000000000000` is filtered out of listings by default and protected by the restricted-module voter path; implicit modules reject edit and delete outright.
- **Update diffs are sharing-only** — the `changes` payload on `DataModuleUpdatedEvent` currently reflects `sharing` changes only, even though other fields are persisted.

Behaviour history

- Switching modules while browsing a module's **Entities** tab used to carry over the previously viewed module's filter; the list now scopes to the module you switched to.

Field configuration reference

This page is for **integrators and developers** who read or write entity definitions or entries through the API, or who need to know exactly how the platform stores and enforces a field's settings. It covers the storage model behind the field types, the stored form of the settings the builder shows, and the technical notes for individual field types.

> **Designing fields in the UI?**
>
> Builders should start with Choose a field type [Build Apps] and Settings every field shares [Build Apps] — they cover the same features in the labels the entity designer shows. Field types that the designer does not offer are on Internal field types.

Storage types and field kinds

It helps to keep two distinct concepts apart:

- **Storage data type (`dataType`)** — a small, closed enum that describes how a value is *stored and cast* (text, number, date, boolean, blob, entity reference, JSON, and so on). This is the field's storage contract.
- **Field kind** — the semantic field a builder actually chooses (plain text, long text, password, phone, image, reference, category, switch, map, …). The platform recognizes a kind by inspecting both the `dataType` **and** the configured edit component (and sometimes further options).

A single storage type backs many kinds: `Text` , for example, is the storage type behind plain text, long text, password, and phone fields, each distinguished by its edit component. Keeping the two layers separate explains why two fields that store the same underlying value can behave and render very differently.

An intermediate role layer

Between the roughly fifteen primitive storage types (`DataTypeEnum`) and the roughly thirty-five UI field kinds sits a thin layer of **semantic-role tokens** (such as `MAIN_TITLE` , `SUB_TITLE` , `PICTURE` , `CATEGORY`) that tag a field with the presentation role it plays rather than how it stores or edits its value. This layer is marked in-code as temporary, so treat it as an implementation detail that may be folded away later; rely on the storage type and the field kind as the stable contract.

Storage data types

The storage layer is a closed enum of data types, including text, number, date, time, date-time, boolean, blob (files and images), entity references, icons, nested blocks, localized JSON, GeoJSON (maps), and JSON. Inline SVG (`SVGIcon`) was added **Since 1.8**. On save, submitted values are coerced toward their storage type by a per-type caster; types without a registered caster pass through unchanged.

Field kinds and their storage types

The kinds a builder can pick in the entity designer, with the label the designer shows and the storage type behind each:

Designer label	Storage type (`dataType`)	Notes
Text	`Text`	Also offered as a display-only variant

Designer label	Storage type (`dataType`)	Notes
Long-text	`Text`	Display-only variant; rich-text modes, see Rich text
Localized Text	`LocalizedJson`	Display-only variant
Localized Long Text	`LocalizedJson`	Markdown only; display-only variant
Password	`Text`	Stored as a one-way hash, see Password
Phone	`Text`	See Phone formats
Number	`Number`	
Sequence Field	`Number`	Auto-increment, see Sequence
Year	`Number`	
Date	`Date`	
Month	`Date`	Min/max as month numbers, see Month
Month-year	`Date`	
Time	`Time`	
Date-time	`DateTime`	
Boolean	`Boolean`	
Switch	`Boolean`	
Category	`Json`	See Category
File, Image, Document, Video	`blob`	See File constraints
Icon	`Icon`	
Map	`GeoJson`	See Map
Reference	`Entity`	See Reference data sources
User	`Entity`	References the user entity, see User pointers
Application Reference	`Entity`	See Application Reference
Mapped Field	the source field's type	See Mapped field update modes
Block	`Block`	See Block rows

Kinds registered as hidden — JSON, SVG Icon, Page Reference, Number Enum and the Sensitive Field Selector — and the Data Query Selector control are not offered by the designer; they are described on Internal field types.

Capability flags

When a kind is registered it declares a set of **capability flags** that govern how the platform lets that kind be used — independent of any single field's configuration:

- **`supportUniqueOption`** — the kind may be marked unique (enforced by a database unique index).
- **`supportSearchable`** — the kind can participate in full-text search (`searchIndex.fullTextSearch`).
- **`supportAdvancedDefaultValue`** — the kind accepts expression- or pointer-based default values, not just a literal.
- **`fieldQueryOptions`** — which filter operators and pointers the kind exposes in the query builder.
- **`overwriteFieldOption`** — per-kind overrides for searchable / queryable / display-only / auto-increment behavior.

These flags explain behavior that is otherwise surprising: a **JSON** field is neither searchable nor queryable (model data you need to filter on as real fields instead), a **Sequence** field is always auto-increment and locked read-only, while **Number** and **Boolean** fields are queryable.

Which kinds can be marked **Unique** (`supportUniqueOption`): Text, Long-text, Localized Text, Localized Long Text, Phone, Number, Date, Time, Date-time, Month, Month-year, Year, Reference, User (and the hidden Page Reference and Number Enum). Which kinds support **Multiple** (`supportMultipleOptions`): File, Image, Document, Video, Category and Reference with a configurable minimum and maximum; User and Application Reference without one.

Validation and cardinality

Several concerns apply regardless of the kind you choose; the per-kind pages reference them rather than repeat them:

- **Validation** — every field carries a validation model (required, min/max, regex, file extensions, and localized-value rules). Localized-value validation modes were added **Since 1.4**. Custom expression-based validations run on top of the built-in rules.
- **Cardinality** — any field can be made *multi-value* (an array of its own values) and/or *repeatable* (the field or block repeats). Index-safe handling when repeatable items are removed landed **Since 1.6**. `Since 1.10` , a **category** field can be multi-value too, and it is the only kind whose multi-value minimum/maximum count is enforced on save — see Category: multiple values. Also `Since 1.10` , each row of a repeatable block carries a stable `__id` , so per-row locked and sensitive values follow the row rather than its position — see Block: stable row ids. A save through a view keeps the block columns the view does not grant (`Since 1.4`) and, from release 1.10.1, restores them only for rows matched by `__id` — see Block: saving rows through a view.
- **Calculated values** — a field can be computed from an expression, driven by calculation triggers and an execution scope; the scope (create-only, update-only, or always) gates every trigger, including a field-change (`OnFieldsChanged`) trigger, not just save-time ones.
- **Sensitive fields** — a field can restrict read access to granted users or groups, returning a mask to everyone else. **Since 1.8.** Reference and block fields cannot be marked sensitive.
- **Encrypted fields** — a field can instead be stored as ciphertext at rest and revealed one value at a time by a decrypt action. **Since 1.10.** Encrypt is a *separate* option from Sensitive, and the two are mutually exclusive on the same field. See Encrypted fields.
- **Uniqueness and display roles** — field names are unique within an entity, and title/summary roles (title, main number, main date, and similar) may be held by at most one field each.

Calculated values

On the per-kind pages, what the **Default Value** → **Advanced** toggle configures — a script expression plus its triggers — is stored on the field as `options.calculatedValueOptions` :

```
{
  "expression": "value.price * value.quantity",
  "calculationTriggers": [
    {
      "type": 3,
      "triggerOptions": { "watchedFields": ["price", "quantity"] }
    }
  ],
  "executionScope": 2
}
```

Property	Required	Type	Meaning
`expression`	no	string	the expression evaluated to produce the field's value
`calculationTriggers`	**yes**	array	when to recompute; each trigger `type` may appear at most once
`calculationTriggers[].type`	**yes**	number	a `CalculationTriggerEnum` value (table below)
`calculationTriggers[].triggerOptions`	no (nullable)	object	`{ "watchedFields": [...] }` — normalized names of the fields watched by field-based triggers
`executionScope`	no	number	an `ExecutionScopeEnum` value (table below)

The trigger types are the four options the UI describes as page load, form submission, entry change, and field change (`CalculationTriggerEnum`):

Value	Name	Fires
`0`	`OnFormOpened`	when the form opens; server-side it applies to creation only
`1`	`OnSave`	when the entry is saved
`2`	`OnEntryChanged`	when the entry changes
`3`	`OnFieldsChanged`	when a field listed in `triggerOptions.watchedFields` actually changes value

The execution scope decides which saves recompute the value (`ExecutionScopeEnum`):

Value	Name	Recomputes
`0`	`NewEntries`	on create only
`1`	`ExistingEntries`	on update only
`2`	`AllEntries`	on every save

When `executionScope` is absent, the server-side save-time gate never passes: the field is not recalculated on save at all. Set it explicitly on any field you expect the server to keep current.

Where the expression is evaluated

The same stored expression is evaluated in two places: the form evaluates it live as the triggers fire, **and** the server evaluates it again inside save-time validation. For writes that arrive through the API, the server-computed result is applied in exactly two cases:

- the field is **locked or hidden** — the submitted value is overwritten by the calculated result (a lock never suppresses the field's own calculation; the two are designed to pair); and
- the field is **omitted** from the payload and has no default value — the calculated result fills it.

An explicit value submitted for a calculated field that is neither locked nor hidden wins over the calculation. Server-side recalculation also has gates: the field needs at least one trigger other than `OnFormOpened`, an `executionScope` matching the operation (absent scope — never), and an `OnFieldsChanged` trigger additionally requires that a watched field's value actually changed. `OnFormOpened` never fires on update; on create it does trigger server evaluation (even without an `executionScope`) to fill omitted values, while for locked or hidden fields the value the form already computed is preserved.

The practical rule for API callers: you may omit a calculated field whenever it is locked or hidden, or its triggers and scope let the server fill it; a field whose only trigger is `OnFormOpened`, or whose `executionScope` is unset, is not recalculated on save — supply the value yourself.

The designer maps onto this payload as follows: **Advanced** turns on `options.isAdvancedDefaultValue`; **Type expression** is `expression`; **Execution Triggers** (shown only while **Lock Field** is on) is the first entry of `calculationTriggers`, with **On page load** = `0`, **On save** = `1`, **On entry change** = `2` and **On field change** = `3` (**Watched fields** fills `triggerOptions.watchedFields`); **Execution Scope** is `executionScope`, with **New entries** = `0`, **Existing entries** = `1` and **All entries** = `2`. Choosing a trigger resets the scope to `0`, and with **On page load** only `0` is offered.

Locked fields on the server

Locking is enforced on the server, not just in the form. A save that submits a new value for a locked field — whether from a user, a reaction, or a direct API call — still succeeds, but the submitted value is silently discarded and the field keeps its stored value (or its default value when the entry is created). The check does not distinguish who is writing, so it applies equally to form saves, direct API calls, and automations; it is bypassed only when the caller explicitly skips validation or when the field is unlocked through a system button's overridable locked fields. The field's own calculated value is unaffected: it is still computed and applied while the field is locked, which is why pairing "Lock Field" with a calculated value is the standard way to model a field that the system maintains.

In the designer, a locked field that is also unconditionally required must have a default value, and with **Advanced** on the expression is mandatory. Block fields cannot be locked.

Encrypted fields

Encrypt (`options.encrypt`) is its own field option, not a stronger setting of **Sensitive**. Sensitive changes what a reader is *shown* while the stored value stays plaintext; Encrypt changes what is *stored*, writing the value as ciphertext. A field may carry **at most one of the two** — saving both raises `SENSITIVE_AND_ENCRYPT_MUTUALLY_EXCLUSIVE`, and in the field editor switching one on silently switches the other off.

Which kinds can be encrypted. Four kinds can never be, and the server does not reject them — it silently normalizes `encrypt` back to `false`:

Encryptable?	Field kinds
Never	Reference / relation, block container, **switch**, **category**.
Yes	Everything else — text and text-like, number, date/time/date-time, boolean, file, image, video, document, icon, SVG icon, localized text, JSON, object JSON, map (GeoJSON).

Note that this list is **not** the same as the Sensitive one, in both directions. Sensitive excludes only reference and block fields, so **a switch or a category field can be Sensitive but never Encrypted**.

Turning Encrypt on clears Searchable. The server force-sets `options.searchable = false` on save, so an encrypted value never reaches the shared full-text index — the same thing Sensitive already does, and the reason `supportSearchable` (see Capability flags) is not the whole story for an encrypted field. Encrypt also bars the field from being the entity's main title, and — because the stored value is opaque — from filters, query criteria, reaction conditions, summary aggregates and reports.

Two further properties travel alongside `encrypt` on the field's options, `encryptTransitionVersion` and `pendingEncryptUpgrade`. Neither is set by a builder: they are server-owned bookkeeping for the Upgrade Records transition that toggling Encrypt starts. The full mechanics — the encryption itself, the decrypt action, the transition, and what an encrypted field refuses — are on Sensitive Fields [Build Apps › Protect a field].

Mapped fields. A mapped field's `sensitive` and `encrypt` flags are mirrored from its source field for every update mode, so the designer shows neither toggle on a mapped field; any value set on the mapped field itself is overwritten from the source on save.

Read-only rendering and inline editing

Every editable kind has a matching **view component** that renders its stored value read-only across the contexts where entries appear — **Details**, **Table**, **Card**, and **Badge**. So a field is described by a pair: an edit component for forms and a view component for display (for example, number has a formatted-currency view, date has a relative "from now / to now" period view, and boolean has a badge view).

The **entry content renderer** assembles these view components into the final entry layout, honoring the entity's display option and the view's style. It supports several layouts: a simple container, a block container, entries-columns, and tabs.

Where a field is editable, the renderer also offers **inline editing** in place: click (or press Enter) to edit, Esc to cancel, Ctrl-S to save, and auto-save when you click outside the field. Editability is checked backend-authoritatively — in particular for aggregator-entity fields and workflow-step fields, the server has the final say on whether an inline edit is allowed.

Notes by field type

Rich text: Long-text and Localized Long Text

The designer's **Type** setting on a Long-text field maps onto the stored options: **Normal** (none of the flags), **Markdown** = `isRichText`, **Rich Text** = `isRichTextV2`, **Rich with code** = `isRichTextV2` plus `allowCodeEditor`. A Localized Long Text field offers **Markdown** only.

For longer multilingual content, the multilingual text field can be edited with formatting instead of plain text. Two rich-editing modes are available:

- `isRichText` — a lightweight **markdown** mode.
- `isRichTextV2` — an advanced rich editor with a fuller formatting toolbar.
- `allowCodeEditor` — additionally lets the user switch to editing the underlying source (code) directly.

Rich-text editing uses a roosterjs-based editor. Its toolbar offers Bold, Italic, Underline, Strikethrough, Heading, ordered/unordered lists, Quote, Code, Table, and Clear Format (the markdown mode omits Underline). When an entry is viewed, the stored markdown is parsed and the resulting HTML is DOM-sanitized before it is rendered.

Phone formats

The phone input includes a searchable country dropdown so users can pick the country and its dialing code before entering the number. Country data (names, flags, and dialing codes) is sourced from standard ISO country information, and the dial code is shown separately from the number. When no country is selected, the input defaults to the United Arab Emirates (`ae`); a different default country (ISO code) can be configured for the field.

Stored phone values can be presented in any of the following standard formats:

Format	Description
`E164`	The international standard: a leading `+`, country code, then the number with no spaces or separators (e.g. `+97150XXXXXXX`).

Format	Description
`INTERNATIONAL`	The country code with human-readable spacing (e.g. `+971 50 XXX XXXX`).
`NATIONAL`	The number as written within its own country, without the country code (e.g. `050 XXX XXXX`).
`RFC3966`	A URI-style form prefixed with `tel:` (e.g. `tel:+971-50-XXX-XXXX`).

By default, phone values are displayed in `INTERNATIONAL` format when rendered in read-only contexts (details, tables, and cards).

Map

The map field is a Leaflet-based selector. Beyond clicking directly on the map to place a marker, you can search for a place by name: the field performs geocoding and autocomplete through Nominatim and shows a results list, and selecting a result moves the marker to that location. The value is stored as GeoJSON, which the backend validates at boot time via its GeoJSON validator.

> ### Decrypted display uses full width · Since 1.10
>
> If a Map field is also marked **Encrypt** (see Encrypted fields) and a user reveals it through the decrypt action, the revealed map — the address input plus the interactive selector — now renders at its normal full width in entry details. Before this fix it rendered squeezed to roughly half width, the narrow layout meant for a masked or locked value's short placeholder text; a masked or locked Map still uses that narrow layout, unaffected.

File constraints

This section lets you restrict what can be uploaded to the file field:

1. **Accepted File Types:** Limit the field to specific file types by listing the allowed extensions (the `accept` setting). Users are then only able to browse and select matching files.
2. **Maximum File Size:** Set the largest size (`maximumSize`) a single uploaded file may reach. The size is checked in the browser before upload, so oversized files are rejected immediately, and the file is also cast and sanitized on the server after it is received.
3. **Minimum / Maximum Count:** When the field accepts more than one file, you can require a minimum number and cap the maximum number of files per entry.

Note: Uploaded files are displayed with a file-type icon that matches the file's MIME type, alongside the file name and its size shown in a human-readable format.

In the designer these are **Acceptable File Types** (File fields), **Max file size** and, when the field is **Multiple** or **Repeatable**, **Total files size**. Both sizes default to 20 MB and **Max file size** cannot be set above 20 MB.

Password

A password field's stored value is a one-way hash, not reversible ciphertext — this holds even if the field also has **Encrypt** turned on, since the password hashing always wins over encryption for this field kind. There is nothing to reveal, so the field never offers a decrypt/eye action and always renders as a plain `********` placeholder, with no lock icon either (a lock icon would suggest access could be granted, which is never true for a password value). The masking also covers the hover tooltip on table/card/badge views, not just the visible text.

Date pointers and relative display

Filter Pointers

Instead of a fixed calendar date, a date field can use **filter pointers** — relative-date tokens that are resolved at run time so queries and default values stay dynamic (they always track the current date rather than a value frozen when the field was configured).

The available date pointers are:

- **Now** — the current date (and time, where the field carries time).
- **Today** — the current day.
- **StartOfDay**, **StartOfWeek**, **StartOfMonth**, **StartOfYear** — the start of the current day, week, month, or year.

These pointers can be used in two places:

1. **Queries / filters** — for example, "entries whose date is on or after **StartOfMonth**" keeps a filter pointing at the current month without editing it each month.
2. **Default values** — a new entry's date field can default to **Today** or **Now** instead of a hard-coded date.

Relative Date-Period View

For read-only display, the date field also offers a relative **date-period** renderer that shows how far a stored date is from the present as a *fromNow* / *toNow* expression (for example, "3 days ago" or "in 2 weeks") rather than an absolute date.

In the query builder the pointers are offered as `@now`, `@today`, `@StartOfDay`, `@StartOfWeek`, `@StartOfMonth` and `@StartOfYear`. In an expression, the scope helpers `scope.today()` and `scope.StartOfMonth()` return dates.

Month: minimum and maximum

A month field also accepts a minimum and maximum value constraint, entered and validated as a plain month number (1–12) — not as a date — so, for example, a `{min: 1, max: 12}` configuration is valid on a month field the same way it is on a number field. If no minimum or maximum is set, any month is accepted.

Reference data sources

Beyond **Entities** and **Lists**, a reference field's data source can also be bound to a **view**, a **module**, a saved **query**, or a **collection**. This lets the dropdown draw its options from a filtered view, a module's objects, the result of a saved query, or a stand-alone collection instead of a whole entity.

Filter pointers are relative tokens you can apply to keep the option list scoped to the current user's context instead of hard-coding values. The available pointers depend on the chosen source and include:

- **Me** — the current user.
- **MyGroups** — the groups the current user belongs to.
- **MyRoles** — the current user's roles.
- **MyOrgUnit** — the current user's organizational unit.

Parameterized queries: when the source is a query, you can pass parameters into it so the fetched options react to other field values or to runtime context.

Preserve value on edit: the reference field can keep its stored value when an entry is edited, even if that value would no longer match the current source filter, so previously-linked entries are not silently cleared.

In the designer, the **Data Source** section offers **Entity Source** (one or more entities, added with **Add source**, each optionally narrowed by a saved query in **Entries source**) and **List Source** (**Choose List Item**). In the query builder the pointers above are offered as `@me`, `@myGroups`, `@myRoles` and `@myOrgUnit`.

User pointers

Because the user field references the USER entity, it also exposes relative pointers that resolve dynamically against the current user rather than a fixed list:

- **Me** (`@me`): Resolves to the currently signed-in user.
- **MyGroupsUsers** (`@myGroupsUsers`): Resolves to the users who belong to the current user's groups.

These pointers keep the field dynamic per user and can be used in default values and filter conditions.

Application Reference

When the field value is displayed, the platform resolves and attaches extra information about the referenced object, including its top-level parent (the owning application), its display name, and whether the target is an application or a service. This lets views show a meaningful title and link for the selected object.

Category: user-managed options `Since 1.10`

Normally only a developer edits a Category field's option list, from the field settings above. Two settings let end users add or rename options themselves from an entry's create or edit card, without touching the field settings. Both must be on:

Setting	Where you set it	Effect
Allow users to manage options (`options.allowUsersToManageOptions`)	The Category field's settings	Required for any change an end user makes to the options. Only allowed on a Category field whose options are **not** mapped from another field; saving it on a mapped field is refused (`INVALID_USER_OPTION_PERMISSIONS`).
Add category options / **Edit category options** (`allowAddCategoryOptions` / `allowEditCategoryOptions`)	The view's card settings, per operation (create card, edit card), in `editConfiguration.supportedOperations[]`	Whether end users may add or edit options in that view's card. **Link create and edit card settings** keeps the two operations in step.

When a user saves an entry with option changes, the client sends them in the entry body (create, update and batch create) as `categoryOptionChanges`: an object keyed by the field's full normalized name (a block's inner fields as `block.field`), each value shaped `{baseOptionIds?, create?: [{id, label, color?}], update?: [{id, label, color?, base: {label, color?}}]}`. Labels are per-language strings, the default colour is `#81C784`, and the client chooses the new option ids.

The server checks every field **before** the entry is written. Through a view, the view must belong to this entity, its card must allow the operation, and the field must allow user management and be editable in that view; without a view, the caller must be able to edit the entity. An entity another user has locked, or one installed from a package, is refused. The options are then added to the entity definition with guarded updates and one `DataEntityUpdatedEvent` is published.

Error	When
`403 CATEGORY_OPTION_CHANGE_NOT_ALLOWED`	The field, the view's card or the caller does not allow this change.
`400 INVALID_CATEGORY_OPTION_CHANGE`	An unknown field, an empty label, or another malformed change.

Error	When
`409 CATEGORY_OPTION_CONFLICT`	Someone else changed or deleted the option meanwhile (`update.base` is the optimistic-concurrency token).
`409 CATEGORY_OPTIONS_CHANGED`	`baseOptionIds` no longer matches the stored option set.

Category: multiple values `Since 1.10`

A Category field can hold **several options in one entry**. Turn on **Multiple** in the field's value settings (the platform-wide `options.multiValue` flag); you can also set an optional minimum and maximum number of selections. Entry editors then show the picks as removable chips, and table cells show every value. The mobile app supports the same behaviour.

- **Stored shape.** A single Category field stores `{id}`; a multi-value one stores an **array** `[{id}, …]` in selection order (a repeatable multi-value category is an array of arrays). The default value follows the same shape: a multi-value default is an array of `{id}` whose ids exist among the field's options and respect the minimum and maximum.
- **Counting rules.** Every selected `id` must be one of the field's options. The minimum (`multiValue.min.value`) and maximum (`multiValue.max.value`) are enforced for Category fields: an empty selection never breaks the minimum, because emptiness is what **Required** decides (so a minimum of 1 does not silently make the field required); a missing minimum means 0, and a missing maximum, or one of 0 or less, means unlimited.
- **Turning Multiple on is a breaking change.** The entity's version is bumped and existing entries become obsolete until upgraded; the automatic repair wraps each stored single value into a one-item array. Until an entry is upgraded, saving it for any other reason still succeeds, because every save wraps a leftover single value. See Versioning and breaking changes [Build Apps › Create and design an entity].
- **Turning Multiple off** keeps only the first value picked and drops the rest; an empty selection becomes empty (`null`).
- **The switch is refused while the field is relied on.** If a view or statistic query of the same entity uses the field for a Kanban board, a grouping, a sort, a dashboard or a timeline, the save fails with `400` `CATEGORY_FIELD_MULTI_VALUE_IN_USE` , and `details.usages` lists each one (`{type, id, name}`) so you can remove those usages first. Views on a *related* entity that reach the field through a relation are not checked.
- **Where a multi-value category cannot be used.** Anything that assumes one value per entry refuses it with `400` `MULTI_VALUE_CATEGORY_FIELD_NOT_ALLOWED` (`details.usage`): the entity's **Main Category** or **Subcategory**, a Kanban column field, a default sort, a dashboard group-by and the timeline attribute mapping. It is also left out of a view's grouping choices.
- **Exports and filters.** Template exports print a multi-value category as its labels joined per language, and the `In` / `NotIn` filter operators accept a single category value as well as a list.

Category: option display roles

Each option added to a Category field can be assigned a display role, so it is shown as a **category**, a **sub-category**, or a **title** when entries are rendered. The options of a Category field are backed by an internal category collection maintained per entity. When a Category field is mapped from another field, its available options update automatically whenever that source field changes.

Mapped field update modes

The designer's **Update Mode** list maps onto `mapping.updateMode` :

Designer label	Value	Behaviour
Local Update	`Local` = 0	Filled from the source automatically and editable; edits stay on this entry and are not written back. The designer's default.
Never	`Never` = 1	Filled from the source the first time only; later changes to the source are not copied. The field must be locked (the server rejects the definition otherwise).
From Parent Only	`ParentOnly` = 2	Follows the source: when the source entry's value changes, the change is copied to this field. The field must be locked.
2 way	`TwoWay` = 3	Editable in both places; a change on either side is written to the other. The mapped field's **Required** setting must match the source field's.

Rules the server enforces when the entity is saved: a `Never` or `ParentOnly` mapped field must be locked; a `TwoWay` mapped field's required setting must equal the source field's; and when the mapped field sits in a repeatable block but its reference field does not, only `Never` and `ParentOnly` are allowed. Only `ParentOnly` and `TwoWay` fields receive changes from the source entry after it is updated.

Sequence: auto-increment configuration

The designer's **Auto increment settings** map onto `autoIncrementOptions`: **Starting value** = `startAt`, **Increment by** = `incrementBy`, **Conditional Sequencing** = `conditionalIncrement`, and each **Key** / **Value** pair (added with **Add Key**) is an entry of `keys` with `keyName` and an expression `value` (`calculationMethod` 2). Once the counter has been used, the keys are read-only.

Beyond the starting value (`startAt`) and the increment amount (`incrementBy`), the auto-increment configuration (`autoIncrementOptions`) supports two advanced behaviors:

- **Keyed increment (`keys`):** Scopes the counter to a combination of key fields, so a separate running sequence is kept for each distinct key combination instead of a single global counter. For example, an invoice number can count independently per branch or per year rather than sharing one sequence across all entries.
- **Conditional increment (`conditionalIncrement`):** Advances the sequence only when a defined condition is met, so entries that do not satisfy the condition do not consume a number.

For an end-to-end worked example that combines both behaviors, see the formatted-reference steps below.

The worked "formatted reference" example below is the integrator view of the builder recipe in Sequence [Build Apps › Sequence Field].

Step 1 — a year-keyed sequence

Configure the sequence field's auto-increment options (`autoIncrementOptions`) with conditional increment enabled and a single key whose expression resolves to the year:

```
{
  "autoIncrementOptions": {
    "enabled": true,
    "key": "lc-reference",
    "startAt": 1,
    "incrementBy": 1,
    "conditionalIncrement": true,
    "keys": [
      {
        "keyName": "year",
        "calculationMethod": 2,
        "value": "new Date().getFullYear()"
      }
    ]
  }
}
```

`calculationMethod` selects where the key's value comes from:

`calculationMethod`	**Source**
`0`	Fixed value
`1`	Field
`2`	Expression (default)
`3`	Pointer

There is no separate condition language: with `calculationMethod: 2`, the key's `value` holds ordinary JavaScript evaluated against the entry, with `value` bound to the entry's values and helper functions such as `now()` and `today()` in scope.

How the key behaves at runtime:

- The key's result becomes part of the counter lookup: the next number is drawn with an atomic upsert matched on the counter plus each key's value, so every distinct year keeps its own running count, starting at `startAt` and advancing by `incrementBy`. When the first entry of a new year is created, that year's counter row does not exist yet and starts fresh at `startAt` — the yearly reset requires no maintenance step.
- With `conditionalIncrement` enabled, the platform forces the field's uniqueness off (without it, an auto-increment field is forced `unique: true`). Duplicate numbers within a year are prevented by the counter's atomic update, not by the uniqueness flag.
- When the entity definition is saved, `startAt`, `incrementBy`, and `keys` are copied onto the platform's counter document, which is the runtime source of truth; editing `keys` afterwards only propagates to the counter while it is still unused. Decide the key structure before real entries exist.

Step 2 — the locked companion field

Add a second, locked field — for example a text field with the normalized name `ref` — to carry the formatted value, and give it a calculated value. The example assumes the sequence field's normalized name is `seq`:

```
{
  "calculatedValueOptions": {
    "expression": "value.ref ? value.ref : (value.seq ? 'LC-' + new Date().getFullYear() + '-' + String(value.seq).p
adStart(4, '0') : null)",
    "calculationTriggers": [{ "type": 1 }],
    "executionScope": 2
  }
}
```

Two configuration points are load-bearing:

- **The lock is required for the calculation to apply.** The server writes a calculated result onto the entry only when the field is locked (or hidden); an unlocked field keeps whatever was submitted. The platform treats the pairing as intended: any field that has `calculatedValueOptions` is automatically reported to forms as locked.
- `executionScope` **must be set.** When it is absent, the server never recalculates the field on save, and the reference would never be composed. Use `1` or `2` so the recalculation reaches entries that already exist.

`calculationTriggers[].type`	**Trigger**
`0`	On form opened (form-context only: for locked fields the server keeps the submitted value)
`1`	On save
`2`	On entry changed
`3`	On fields changed (watches `triggerOptions.watchedFields`)

`executionScope`	**Recalculates on**
`0`	Create only (new entries)
`1`	Update only (existing entries)
`2`	Create and update (all entries)
unset	Never recalculated server-side

The idempotent guard

The leading `value.ref ? value.ref : …` is what makes the expression safe to re-run. The on-save trigger fires on every later save of the entry; without the guard, each save would recompose the reference, and a save made after the year boundary would rebuild it with the new year — silently turning an issued `LC-2026-0042` into `LC-2027-0042`. With the guard, the first composed value is frozen: every later recalculation finds `value.ref` already set and returns it unchanged, so the reference cannot drift.

The inner `value.seq ? … : null` guard matters just as much, because of the timing described next: on create, the expression evaluates while the sequence field is still empty. Without this check, the expression would compose `LC-2026-undefined` — and the outer guard would then freeze that broken value permanently. Returning `null` instead makes the calculation count as failed, and the platform then leaves the field unset on create (and keeps the old value on update) — exactly what the recipe needs. The reference materializes on the first later save, once the sequence number exists.

Both expressions in this recipe — the counter key and the companion's calculation — are plain JavaScript evaluated against the entry, so if the composed year must always match the counter's year, derive both from the same date field on the entry rather than from the current date.

Timing: the sequence number is assigned after validation

This ordering is platform behavior, and it shapes the whole recipe. When an entry is created, the platform validates it first — and server-side evaluation of calculated fields happens inside that validation — then draws the sequence number and writes it onto the entry, and only then stores the entry. A calculated expression evaluated during the create request therefore always sees the sequence field as `undefined`; do not expect the formatted reference in the create response. Creation is also the only moment a sequence number is assigned — updates never (re)assign it — so on any later save the number is present and a save-triggered recalculation can read it.

A counter failure during creation is logged but does not abort the save: the entry is created with an empty sequence field. The `null` branch of the guard keeps the companion field empty in that case too, rather than freezing a malformed reference.

Entries created through the API

The sequence number itself needs no client: it is assigned server-side on every create, whichever channel created the entry. The formatted reference is different. Because the number is drawn after calculated fields are evaluated, no channel — form or API — can compose the reference inside the create request itself, so an API-created entry starts with an empty `ref`. It receives its value in one of three ways:

1. **A later save** whose trigger and `executionScope` match (as configured above): the server recomposes the field itself, with no client involvement. This is the recommended route — issue a follow-up save after creating the entry, or let the next natural update do it.
2. **The caller passes the pre-formatted value.** For a locked field, the server only reliably keeps a submitted value when the field's trigger is on-form-opened (`type: 0`) — the one form-context-only case, where the server computes nothing and trusts the value the form (or the API caller) submitted — or when the field is unlocked through a system button's overridable locked fields. Under other trigger configurations, the server-side calculation supersedes the submitted value.
3. **An out-of-band backfill** that writes the reference directly.

Locked and never required

A sequence field is always locked: the platform requires every auto-increment field to be locked and rejects the field definition otherwise (see Locked fields on the server). With `conditionalIncrement` off, an auto-increment field is forced `unique: true`.

`Since 1.10` A sequence field can never be required: the platform refuses a "required" verdict on an auto-increment field from every source that could produce one — the field's own Required toggle, the view's own required-fields list, and a workflow step that marks the field's enclosing Block mandatory. The sequence number is assigned by the platform only after the entry passes validation, and the field is never shown on the entry form, so nothing a caller does could satisfy a "required" check on it.

Block: required sub-fields in a hidden block `Since 1.10`

A **Required** sub-field nested inside a Block is suspended whenever the Block itself is hidden — whether the Block was hidden by its own hide-field condition or by a workflow step that hides it. This applies at any nesting depth inside the Block, and for a **repeatable** Block it is resolved separately for each repetition: one repeated instance of the Block can be hidden (for example, an expression that only hides the first row) while another repetition stays visible and enforces its own required sub-fields normally.

Block: decrypting a field inside a repeatable block `Since 1.10`

An **Encrypted** field (see Encrypted fields) nested inside a **repeatable** Block has one stored value per repetition, so revealing it now takes a row selector: the decrypt action addresses one specific repetition rather than the field as a whole. Decrypting such a field without specifying which repetition, or specifying one that does not exist, is refused. The decrypt mechanism itself — the encrypt-at-rest model, access grants, and the decrypt action in general — is documented on Sensitive Fields [Build Apps › Protect a field]; this note covers only the repeatable-block-specific behavior.

Block: stable row ids `Since 1.10`

Each row of a **repeatable** Block now carries a reserved key, `__id` (a uuid), so the server can tell rows apart across an update. Before this change, per-row state (locked values, sensitive values, the entry history diff) was carried across an update **by position**, so it could not follow a row that moved.

What this means when you design or integrate:

- **It is not a field.** `__id` never appears in the entity's field definitions, so exports, indexing, table views and filter pickers do not see it. The `__` prefix is reserved: no field you define can start with it.
- **Echo the ids you receive.** A row whose `__id` matches a stored row is treated as the same row; a row with an unknown id is a new row; stored rows that no submitted row claims are removed. The web and mobile clients do this for you.
- **Clients that send no ids at all still work.** If a submitted array carries no `__id` on any row, every row is matched to the stored row at the same position, as before. If some rows carry ids and others do not, the id-less ones are treated as new rows and never take over a stored row's values. A save **through a view** is stricter: see the next section.
- **Stored rows get their ids on upgrade** (`Since 1.10`, from release 1.10.1). Migration `1.10.0-a52` (`stamp-repeatable-block-row-ids`) gives every stored row of every top-level repeatable Block, in every space, an `__id`. Ids already present are kept, the write records no history, and re-running it changes nothing. Before it, a stored row without an id was given a fresh one each time the web form opened it, so a save could never pair that row with storage.
- **The server owns the id.** A well-formed uuid you send is kept; the server mints one for a row with no id, a malformed id, or a repeat of an earlier id. Two rows with the same id in one submitted array are rejected, and `__id` cannot be changed through a patch.
- **Related behaviour.** Entry upgrade transformations after a breaking change now apply to **every** row of a repeatable Block, the entry history compares rows by id, and the block-item reaction actions (create / update block item) match rows by id as well. Expressions in field rules inside a repeatable Block run per row (`scope.index`).

Block: saving rows through a view

A view's edit operation lists the field paths the user may write. A repeatable Block's rows arrive as one array that **replaces** the stored array, so without a repair every column the view does not grant would vanish from every row on save. The server repairs it: for each repeatable Block it pairs the submitted rows with the stored rows and copies the **stored** values of every path the view does not let the user edit into the matching row. Those values always come from storage, never from the payload.

- **Keeping the ungranted columns** is `Since 1.4`: it shipped in the 1.4, 1.6 and 1.8 patch releases (1.4.119, 1.6.69, 1.8.28) and in 1.10.1. On the 1.4, 1.6 and 1.8 lines rows carry no ids, so a submitted row is paired with a stored row by its **unchanged granted values** (same position, or any position when the save removed rows). Those lines also keep the empty rows of a block the save did not change.
- **Pairing by row id** is `Since 1.10`, from release 1.10.1 (not in 1.10.0): on the 1.10 line and later the pairing uses `__id` only.

What an API client writing through a view needs to know (1.10.1 and later):

1. **Send each existing row back with its `__id`.** Ungranted columns are restored only for a row whose submitted `__id` matches a stored row. A pair made by position alone gets nothing restored. Reordering rows, or deleting rows above one, is safe as long as the ids travel with the rows.
2. **A row without an id beside rows that carry ids is a new row.** It gets nothing restored, so its ungranted columns are empty, and the server mints its id. A row added in a deleted row's place therefore cannot inherit that row's locked values.
3. **An array with no ids at all** is still paired by position for the locked-value check, so older integrations keep working, but a view save restores **nothing** for those rows. Send the ids.

4. **An id the server does not know** is a new row. On a view save the id you chose is not kept: `__id` is not in any view's field list, so the view filter drops it and the server mints one. (On a create or full update outside a view, a well-formed unique uuid you send is kept.)
5. **Rows you leave out are removed.** A view can remove rows only when it grants at least one field inside the Block; a view that grants no field of the Block never touches the stored Block.
6. **Duplicate ids in one array are refused** with `400 ENTRY_VALIDATION_ERROR` (reason `duplicate __id in <field>: <ids>`), and no patch operation may address `__id`.

Version notes

Past behaviour that changed, kept here so the guides can describe only current behaviour:

- **Percentage precision on calculated values (1.8).** Previously only typed values respected a percentage field's precision, because rounding happened as the user entered the number. A calculated result went straight to display with its full floating-point precision, so a field configured for 0 or 1 decimal places could render something like `33.333333333333336%`. Both percentage display styles are covered — the plain percentage label and the *Progress Bar with Percentage* label; the progress bar's own fill is unaffected.
- **Required sub-fields inside a hidden block (1.10).** Before this fix, a required sub-field inside a hidden Block could still block save with a "Field is required!" error, even though the user was never shown the field and had no way to fill it in.
- **Sequence fields and Required (1.10).** Before this fix a sequence field marked required (through its toggle, a view's required-fields list, or a workflow step making its Block mandatory) blocked **every** create with a "Field is required!" error.
- **Decrypted Map fields (1.10).** A revealed Map field — the address input plus the interactive selector — used to render squeezed to roughly half width, the narrow layout meant for a masked or locked value's short placeholder text; it now renders at its normal full width. A masked or locked Map still uses the narrow layout.
- **Repeatable value popover.** The popover the entry-details / table / card views use to show a repeatable field's full value list was fixed at a set height regardless of how few values it actually held, so a field with only one or two entries still opened a mostly-empty box. It now sizes to its content, up to the same cap. Display-only; the stored value shape is unchanged.
- **Mapped fields and Encrypt.** The designer does not show the Encrypt toggle on a mapped field (earlier documentation said it did): the flag is mirrored from the source field.

Internal field types

This page is for **integrators and developers** who meet these field types in an entity definition read through the API, in a platform-provided entity, or in a package. None of them is offered in the entity designer's field list, so a builder cannot add them; they exist for the platform's own entities and configuration screens.

> ### Building an entity in the UI?
>
> The field types a builder can add are listed on Choose a field type [Build Apps]. How fields are stored is on Field configuration reference.

Type	Storage type	Why it is internal
JSON	`ObjectJson`	Registered as hidden
Data Query Selector	`Entity`	A control used by configuration screens; not registered as a field type
SVG Icon	`SVGIcon` (`Since 1.8`)	Registered as hidden
Page Reference	`Entity`	Registered as hidden
Number Enum	`Number`	Registered as hidden

The settings these fields carry — **Lock Field**, **Hide Field**, **Required**, **Display on table** — behave as described on Settings every field shares [Build Apps].

JSON

The JSON field lets an entry store an arbitrary structured object as raw JSON. It provides a plain JSON editor so advanced users can capture data that does not fit the standard field types — for example, a configuration blob or a nested object returned by an integration.

Name	Description	Data Type
JSON	A field that stores an arbitrary object as raw JSON.	ObjectJson

- JSON fields are **not searchable and not queryable** — their contents cannot be used in the search box or in query criteria. Model data you need to filter on as real fields instead.
- The raw JSON editor exists only on **web**; on **mobile**, JSON (ObjectJson) fields render nothing in both edit and view.
- A JSON field accepts no advanced (expression) default value and cannot be unique.
- Shown in a list or detail view, the JSON content can be opened in a dedicated dialog for readable inspection.
- Settings it carries: **Lock Field** (with an optional script expression as the lock condition), **Hide Field** (with a script expression), and show in list.

Data Query Selector

The Data Query Selector field lets an entry point to a saved data query. It is used where a record needs to carry a reference to a stored filter/criteria definition — for example, to drive a dynamic list or a parameterized source elsewhere in the configuration.

Name	Description	Data Type
Data Query Selector	A field that allows selecting a saved data query.	Entity

- It is exposed as a control the platform uses in specific configuration screens, but it is not registered in the standard field picker when designing an entity.
- When displayed, the selected query is resolved to its name for the field's title.
- Settings it carries: **Lock Field** (with an optional script expression as the lock condition) and **Hide Field** (with a script expression).

SVG Icon `Since 1.8`

The SVG Icon field lets an entry store a scalable vector (SVG) graphic. Unlike the regular Icon field, which points to an entry from the platform's built-in icon set, the SVG Icon field holds the SVG markup itself, so it can render sharp at any size and carry custom artwork.

Name	Description	Data Type
SVG Icon	A field that stores a scalable vector (SVG) graphic.	SVGIcon

- Any SVG content is passed through a sanitizer before it is stored or rendered.
- It has no default value.
- Settings it carries: **Lock Field** (with an optional script expression), **Hide Field** (with a script expression), and show in list.

Page Reference

The Page Reference field lets an entry point to a page (a generic view). It is a specialized reference field whose data source is the platform's pages, so it is used to associate a record with a specific page in the system.

Name	Description	Data Type
Page Reference	A field that allows selecting a page as a reference.	Entity

- It supports **Multiple** (without a minimum or maximum), can be unique, and accepts an advanced (expression) default value.
- Settings it carries: **Lock Field** (with an optional script expression), **Hide Field** (with a script expression), and show in list.

Number Enum

The Number Enum field lets an entry choose one value from a predefined set of numbered options. The user picks a label from a list, and the field stores the corresponding number. This is useful when a record needs a constrained numeric choice — for example, a priority level or a rating — rather than a free-form number.

Name	Description	Data Type
Number Enum	A field that allows selecting one option from a set of numbered choices.	Number

- Because the stored value is a number, it can still be filtered and sorted like other Number fields.
- It can be unique, supports **Multiple** (without a minimum or maximum), and accepts an advanced (expression) default value.

- Settings it carries: **Lock Field** (with an optional script expression), **Hide Field** (with a script expression), **Required** (with an optional script expression as the condition), and show in list.

MCP Server for AI Agents

> ⚠ **Early access — `osos-mcp` 0.2.0**
>
> This module is at an **early stage** and is **not yet suitable for reliable standalone work**. Use it as a supervised assistant against a non-production tenant.
>
> No tool has yet been verified against a live tenant with a real credential — the test suite runs against recorded API shapes, and the shapes and the running platform have already been observed to disagree in places. The tool names are a public contract but are **not frozen** until 1.0.0.
>
> **This page describes `osos-mcp` 0.2.0.** Check the module's own changelog before relying on anything here against a newer build.

The MCP server lets an AI agent work with OSOS through its HTTP API — discovering a tenant's data model, querying and writing records, reading permissions, and driving workflows — using the Model Context Protocol, which is what most agent tooling speaks.

What it is

OSOS's gateway serves several hundred HTTP routes across its services; about 400 of them are typed operations described by the services' OpenAPI documents, and those are what this server can reach. An agent cannot usefully be handed 400 tool definitions: the descriptions alone would crowd out the task. So the server presents a small, task-shaped catalogue — around thirty tools covering the high-value surface — plus a discovery pair that reaches every remaining operation on demand.

It runs in two shapes:

- **stdio** — a local process a desktop agent launches, configured from a file or environment.
- **Streamable HTTP** — a deployed service where each session carries its own tenant and credential, so one process can serve many tenants.

It is a *client* of the platform. It holds no data, adds no permissions, and cannot do anything the credential it is given could not already do.

Maturity

Read the notice above before planning any work around this module. In particular:

Version	0.2.0
Stability	Early access. Tool names may change before 1.0.0.
Live verification	None yet. The suite runs on recorded shapes.
Suitable for	Supervised, interactive use against a non-production tenant.
Not suitable for	Unattended automation, production data, or anything where a silently wrong answer matters.

Documentation for this module — here and in the certification programme — states the version it describes. When the module is updated, that version is what tells you whether a page still applies.

License requirement

The MCP server requires a valid OSOS license; it refuses requests otherwise. See Your license [Administration › Your license & plan limits].

What it can do

Grouped by what an agent is trying to achieve:

Area	Examples
Connection	Check reachability and credential; report the tenant's languages.
Schema discovery	List applications and entities; describe an entity with its fields, relations, views and your effective access in one call.
Reading data	Count, query and fetch records, through an entity or through a view.
Operation discovery	Search all ~400 operations and invoke one by name.
Files	Upload an attachment; read a file's metadata.
People and access	List users and groups; read effective permissions; probe whether an action is allowed.
Process	Read and act on workflow state; comments; notifications; wait on long-running jobs.

Write operations are gated: depending on configuration they require an explicit confirmation step, and a read-only session does not register them at all.

See the Tools reference.

Where to go next

- Quickstart — running, configured and answering, in a few minutes.
- Tools reference — the catalogue.
- Calling the HTTP API — the API underneath, if you are building a client of your own rather than using this one.

MCP Quickstart

> ⚠ **Early access — `osos-mcp` 0.2.0**
>
> This page describes `osos-mcp` **0.2.0**, which is early access and not suitable for reliable standalone work. Use a non-production tenant.

From nothing to an agent answering a question about your tenant.

Before you start

- Node 24 and pnpm.
- The tenant's **public** URL. Its hostname is how OSOS identifies the tenant, so an IP address, `localhost`, or an in-cluster address will not work — the server rejects them with an explanation rather than letting you discover it as an opaque error.
- An **application token**. A tenant administrator creates an **application user** (a user of type *Application* — see Adding an application user [Administration › Create an API token for an integration]), and the platform emails the new token to the administrator who created it; it is never shown on screen and cannot be read back later (if it is lost, regenerate it). The token carries the permissions of that application user, so give each integration its own.
- A tenant with a **valid license**. See Your license [Administration › Your license & plan limits].

Install and build

Get the `osos-mcp` package from your OSOS vendor — it is not published to a public registry — and run these in its folder:

```
pnpm install
pnpm build
```

Configure a tenant

Configuration resolves across three layers — per-request values, environment variables, and a named profile — and it resolves **per field**, so a deployment can pin the tenant centrally and take the credential per request.

The smallest working configuration is two values:

```
OSOS_BASE_URL=https://your-tenant.example.com
OSOS_APP_TOKEN=<the token you copied>
```

For anything beyond a quick trial, prefer a named **profile**, so what you pass around is a name rather than a secret. A profile can also read its secret from an environment variable or a secret manager rather than storing it.

Two rules are worth knowing because they are deliberately not "last value wins":

- A credential is **bound to the tenant it was configured for**. Naming a different tenant in a request will not quietly reuse a credential belonging to another one.
- Restrictions only accumulate. A request can make a session **more** restricted; it can never make it less.

Wire it into a client

Most agent tools take a block like this:

```
{
  "mcpServers": {
    "osos": {
      "command": "node",
      "args": ["/path/to/osos-mcp/dist/index.js"],
      "env": {
        "OSOS_BASE_URL": "https://your-tenant.example.com",
        "OSOS_APP_TOKEN": "…",
        "OSOS_LOCALE": "en",
        "OSOS_READ_ONLY": "true"
      }
    }
  }
}
```

Starting read-only is the sensible default for a first session: write tools are then not registered at all, so the agent cannot attempt one.

Verify

Ask the agent to run the connection check. It answers four separate questions, which is what makes a failure diagnosable:

1. **Is the tenant reachable?** — needs no credential, so it isolates a wrong URL.
2. **What languages does the tenant have?** — also credential-free, and confirms you reached the tenant you meant.
3. **Is the credential accepted?**
4. **Is the license valid?**

A wrong base URL fails the first. A bad token passes the first two and fails the third. A license problem passes the first three and fails the fourth. Then try `list applications` and `describe entity`.

If something fails

Symptom	Most likely cause
Refused before anything runs, mentioning a license	The tenant has no valid license. Ask your vendor; see Your license [Administration › Your license & plan limits].
The tenant cannot be identified	The base URL is an IP, `localhost`, or an in-cluster address, or the hostname is not the registered one.
Authentication fails but the tenant is reachable	The token is wrong, expired in practice, or was archived. Note that the platform caches validations for a few minutes, so a token created seconds ago may not work yet.
A read returns an empty list you did not expect	Listing an entity's records requires **edit**-tier access, not view. An empty result and "not permitted" can look identical.
Configuration changes refused	The license may not include administration. Ask your vendor.

MCP Tools Reference

> ⚠ **Early access — `osos-mcp` 0.2.0**
>
> The catalogue below is the one shipped by `osos-mcp` **0.2.0** — 27 tools. Tool names are a public contract but are **not frozen** until 1.0.0, so a newer build may add to this list and, before 1.0.0, may rename within it. The module's own repository carries the generated, always-current reference with full parameter detail.

How to read this

Each tool is either a **read** or a **write**. Reads are safe to call freely. Writes modify the tenant, and depending on configuration require an explicit confirmation step; in a read-only session they are not registered at all, so the catalogue an agent sees is a truthful statement of what that session can do.

Everything here is subject to two limits the module does not control: the credential's own permissions, and the tenant's license [Administration › Your license & plan limits].

Connection

Tool	Kind	Purpose
Check connection	read	Reachability, tenant languages and credential — reported separately so a failure is diagnosable.
Platform version	read	The backend version. Needs no credential; the cheapest proof the base URL is right.

Schema discovery

Tool	Kind	Purpose
List applications	read	The applications (modules) in the space.
List entities	read	Entities, optionally scoped to one application.
Describe entity	read	Entity, fields, relations, views and your effective access — in **one** call. The most efficient way to understand an entity.
List entity fields	read	Field types, required and unique flags, reference targets, category options.
List views	read	Views for an entity or an application.
Describe view	read	Columns, default filter, quick filters, permissions.

Reading data

Tool	Kind	Purpose
Count records	read	Cheap. Worth calling before a broad query.
Query records	read	Filter, free-text search, projection, paging.
Get record	read	One record by id.
Query records through a view	read	Applies the view's access rules and column set.

Tool	Kind	Purpose
List related records	read	Traverse a relation from a record.

Paging is by page size and offset; there are no cursors, and the server caps the page size so a broad query cannot swamp the agent.

Operation discovery

Tool	Kind	Purpose
Search operations	read	Search all ~400 API operations by name, path or summary. Collapses versions of the same operation to the newest by default.
Invoke operation	*varies*	Call one by name. Its risk is worked out per operation, so the same rules apply as to a named tool.

These two are why the catalogue can stay small. Invoke takes an operation **name**, never a URL — the method and path come from the recorded API description, so it cannot be pointed elsewhere.

Files

Tool	Kind	Purpose
Upload file	write	Upload from a path or inline content; returns the id for attaching to a record.
Get file	read	Metadata and a download link. Binary content is never streamed into the agent.

People and access

Tool	Kind	Purpose
List users	read	The people directory, with each user's state.
List groups	read	Groups and their members, with the group type.
Get permissions	read	Effective access for the current or a named user.
Check access	read	Whether one action on one record is allowed — the cheapest way to answer "may I?" before a write.

Permissions are a ladder of grants rather than scopes, and one level is a hard denial that overrides any other grant. **Check access** exists so an agent can explain a refusal instead of retrying it.

Process

Tool	Kind	Purpose
Get workflow status	read	Current step, eligible actions, owners.
Execute workflow action	write	Act on a record's workflow.
List comments	read	Comments on a record, with mentions resolved to names.
Add comment	write	Post a comment.
List notifications	read	The credential's own notifications.
Await job	read	Poll a long-running job to completion. There are no callbacks; polling is the pattern.

A refused workflow action is frequently the record's *current state* forbidding it, not a permissions problem — the tools report that distinction, because confusing the two sends you hunting for access you already have.

Records and Lists: Behaviour Reference

This page is for integrators and developers who need the exact behaviour behind the end-user pages on records and lists. It collects the technical detail that those pages leave out, in one place, and links to the API reference for each area. The end-user pages are: Browse a List [Using OSOS › Browse a List and Change How It Looks], Search, Filter, Sort and Group [Using OSOS › Search, Filter, Sort and Group a List], Make a View Your Own [Using OSOS], Create and Edit Records [Using OSOS], Lists [Using OSOS › Lists: Keep Records Together] and Export a List [Using OSOS].

Display types

A view's displays ("faces") are stored as numeric view types:

Value	Type	On-screen name
`0`	`TableView`	**Table view**
`1`	`CardView`	**Badge view**
`2`	`CalenderView`	**Calendar view**
`3`	`KanbanBoard`	**Kanban view**
`4`	`Timeline`	**Timeline view**
`5`	`TreeView`	**Tree view**
`6`	`MiniCard`	Not in the web portal's display menu; used by counter widgets. The mobile app renders it as an empty area.

The web portal offers a display only when the view enables it and the entity suits it: Badge needs an ADT-type entity, Calendar a date field, Kanban a Kanban field, Timeline a timeline configuration, Tree a tree reference field. Timeline and Tree are not offered while the drafts list is shown.

Search

- One full-text index per entry covers every field marked **searchable**. Category fields contribute their option **labels**, reference fields the linked record's **title**, lookup collections their display value, and localized text every configured language.
- The text index is built with `default_language: 'none'`: no stemming and no stop-word removal. Matching is on whole tokens and case-insensitive.
- Syntax follows MongoDB `$text`: space-separated words are **OR**ed, a quoted string is an exact phrase, a leading `-` excludes a word.
- Indexing is asynchronous: a create or update queues a re-index of that entry, and a change to an entry that other entries reference re-indexes them when the reference field is searchable. A new or changed entry can take a moment to be found.
- Encrypted fields are never indexed.
- Reference and user pickers match the typed text against the main title in every language of the tenant, case-insensitively, anywhere in the title, with regex metacharacters escaped (since 1.6).

Full pipeline: Engine: Indexing & Search.

Filters

Operators and their labels

Engine operator	Label in the Filter panel
`Equal` / `NotEqual`	**=** / **!=**
`LessThan` / `LessThanOrEqual`	**<** / **<=**
`GreaterThan` / `GreaterThanOrEqual`	**>** / **>=**
`In` / `NotIn`	**in** / **not In**
`Contains` / `DoesNotContain`	**contains** / **doesn't contain**
`StartsWith` / `DoesNotStartWith`	**starts with** / **doesn't start with**
`Like` / `IsNotLike`	**is like** / **isn't like**
`IsEmpty` / `IsNotEmpty`	**is empty** / **is filled**
`Under`	**under**
`And` / `Or` (groups)	**All conditions are met** / **Some conditions are met**

Field operators: `Size` (**Count of values**), `Sum`, `Avg` (**Average**), `Min`, `Max`, `First` (**First Value**), `Last` (**Last Value**), `Any` (**Any Value**, the default). Value sources: `Fixed`, `Pointer` (for example `@me`, `@today`), `Dynamic` (**Field**), `Expression`, `Parameter`.

- `Contains`, `StartsWith` and `Like` and their negations are case-insensitive.
- `In` / `NotIn` accept reference values and choice values directly.
- `IsEmpty` / `IsNotEmpty` also test **presence** (since 1.8): `IsNotEmpty` matches only a field that exists and holds a value. A field inside a repeatable block with no rows is absent, so it is empty. The test applies across the whole block, not per row.
- `Under` filters a tree entity to the descendants of a node (since 1.2).

Full model: Queries [Build Apps › Queries & filters] · Criteria and field-path dialects · Engine: Query & Aggregation.

How a list request is assembled

1. The structured filter and any applied saved-query criteria are combined with **And** and compiled into a database query.
2. A full-text match stage is added first if a search term was sent and the view allows search.
3. The sort is translated, with the entry id appended as a tiebreaker.
4. View restrictions (stripped conditions, forced active-only and so on) are applied last so they can't be bypassed.

View settings that strip conditions

When a view switches a capability off, the matching conditions are **silently removed** from the request rather than rejected:

View setting	Effect when off
Allow search	The search term is ignored.
Allow filter on current step	No filtering on the current workflow step.
Allow filter on previous steps	No filtering on workflow history.

View setting	Effect when off
Allow metadata filter	No filtering on creation or update information or the record id.
Allow details metadata filter	No filtering on who created or updated the record.
Allow current-user filter	No “belongs to me” filtering.
Allow current-user-groups filter	No “belongs to my groups” filtering.
Show obsolete / invalid	Obsolete or invalid entries are forced out.
Show archived entries	Only active entries are returned.

See Data Views & Faces [Build Apps].

Sensitive and encrypted fields in criteria

- A criterion on a sensitive field the caller holds no grant for is **rejected** with an error, not silently dropped. The guard also covers localized sub-paths (for example `salary.en`) and reference sub-paths (for example `manager.dataEntry`).
- Since 1.10 a sensitive field the caller does hold a grant for stays in the end-user field list even before an unmask session is opened.
- In the admin area (an entity’s **Queries** tab, reaction conditions) the field picker offers every sensitive field regardless of the builder’s own grant, because naming a field in a criterion discloses nothing. The filter value box is editable for a sensitive field on web and mobile.
- An encrypted field can never be filtered, searched or sorted, for any caller.

See Sensitive Fields: querying and filtering and Queries: encrypted and sensitive fields [Build Apps › Queries & filters].

Sorting and paging

- Sort is expressed as `"<field> ASC"` or `"<field> DESC"`; several entries sort by priority. The entry id is always appended as a tiebreaker so offset paging is stable. The default for an entries list is descending by modification date.
- List requests page with `limit` and `skip` (or `offset`).
- **View-scoped reads are capped** (since 1.8): a request through a view without a `limit` gets the ceiling as its limit, and a larger `limit` is clamped to it. The ceiling is `MAX_ENTRIES_RESULT_LIMIT`, default 10,000. See Reading Data.
- A parameterized saved query applied as a quick filter (`appliedQuickFilters`) has its parameters resolved from the request (since 1.8).

See Configuration reference for the ceiling variable.

Grouping and totals

- Groupable fields exclude multi-value fields, repeating fields and fields inside a repeatable block, `_metaData.*` fields, and lookups whose stored value is not a plain reference. A single-valued reference is groupable.
- The end-user grouping picker offers sensitive fields but hides encrypted ones. A grouping level on a sensitive or encrypted field is **accepted and collapsed** into one group keyed `****` for every caller, regardless of grants; a stored level on an encrypted field still collapses. The picker’s info tooltip still says protected fields are unavailable, which no longer matches this behaviour.

- A user's own grouping and sort are kept in the browser's session storage for the session; a configuration change to the view's grouping drops a user's own levels.

Full detail: Entity Aggregates & Entry Grouping [Build Apps › Totals & grouping].

Saved queries

- A query's name must be unique per entity; a duplicate is rejected with `ITEM_DUPLICATE`.
- The query model carries a `visibility` of `Private`, `Organization` or `Public`, and optional `parameters`. The portal's **Save Query** dialog asks only for a name: the query is created with a `relatedTo` of the current user and view, which is what lists it under the view in that user's sidebar.
- Saving through a view requires the view's **Allow saved queries**; otherwise the request fails with `SaveQueryNotAllowedException`. The portal also hides **Save Query** for anonymous users and on the drafts list.
- Applying a saved query that already has an id and saving again updates it; saving a filter derived from a parameterized query creates a new one.

API: DataQuery reference.

Writes

- **Full update and partial patch.** The portal saves a form as a full update of the changed values, and inline edits as a JSON Patch (field-by-field operations).
- **Batch create** (`POST /v1/entities/{entityId}/entries/batch`) creates one main entry plus related entries in one call, linking each related entry back through its relation's target field. Through a view it needs the view's `AllowCreateNewEntries` operation, and every related entry's `relationId` must be one of the view's relations. It does **not** need `allowSaveInvalidEntries` (the quick-save setting); since 1.8 that requirement, which wrongly blocked correctly configured views, is gone.
- **Save conflicts in the portal.** Entries carry an `updateIdentifier`. The web portal sends it on a full edit only when its runtime configuration flag `OPTIMISTIC_LOCKING_ENABLED` is on (off by default); the API itself checks the token whenever a client sends it. Inline edits from the portal do not send it. See Writing Data: update conflicts.
- **Masked values on write.** A sensitive value the user can't read arrives as the mask; the portal's full edit form keeps that control disabled until the user selects it, which nulls it so the typed value (even empty) is sent. The inline editor keeps masked controls enabled and starts them empty. A mask sent back unchanged never overwrites the stored value. See Sensitive Fields: write path [Build Apps › Protect a field].
- **Entry cap.** When a tenant's plan limit is passed by more than 10%, creates (including drafts, draft submission and batch creates) are refused with an English-only message naming the limit, the stored count and the refusal threshold. Archived entries count as a quarter. See Writing Data: write refusals added in 1.10.
- **Repeatable-block rows** carry their own id from release 1.10.1, and the upgrade backfills ids for rows saved earlier; a view that grants only some block columns gets the other columns restored from the stored row with the same id. On the 1.4, 1.6 and 1.8 lines rows are matched by unchanged values instead.
- **Proxy entities.** Single-entry create, edit and save-as-draft through a proxy view are routed to the host entity when the proxy's bound view allows it; bulk writes are refused. See Virtual Entities: write-through.

User lists (collections)

The portal calls a collection a **List**.

- **Model.** A collection has a localized `name` (required) and `description`, an `icon`, one `dataEntity`, `items`, a structure type, a `visibility` (`Organization`, `Private` or `Public`) and sharing settings. Each one keeps who created and last changed it, and an `updateIdentifier`.
- **Structure types.** `Flat` and `Tree` are validated; `Reference` and `Multi` are defined but reserved. Flat items are entry references, de-duplicated on write. Tree items carry value, index, level and parent, and are validated (every node has a value and level, no sibling order clash, no missing parent, no self parent).
- **Two kinds.** Entity-level collections belong to the entity. Personal collections are created inside a view whose configuration allows user collections; they are tagged with the view and the owner, and are listed as "my collections" (optionally per view) and "shared with me". Category (internal) collections back Category fields and are managed by builders.
- **Names.** Unique per entity, and for a personal collection within the owner's collections in that view. A `normalizedName` is generated when not supplied, and must be unique when supplied (since 1.6).
- **Implicit items and collections** are system-maintained: an edit that changes or drops an implicit item is rejected, and an implicit collection can't be deleted or have its structure type changed.
- **Set and unset.** One operation either adds (`set`) or removes (`unset`) entries, named explicitly or chosen with select-all against a search or filter; if both are sent, `set` wins. The response reports `matchedRecords`, `adjustedRecords`, `totalRecordsBeforeOperation` and `totalRecordsAfterOperation`. Creating a collection also accepts select-all, resolved at creation time.
- **Sharing.** Only personal collections can be shared, and only with users or groups that can access the collection's view; a view that is not authenticated can't be shared at all. The owner manages sharing; a recipient can re-share only when the collection is an authenticated, restricted shared collection and they are among its shared users or groups.
- **Duplicate** gives a new id and normalized name, records the source collection, drops all sharing, and is owned by the caller.
- **Delete.** Personal collections can be deleted only by their owner, within their view. A collection used by an entity can't be deleted. Deleting never deletes entries.
- **Browsing** a collection through a view applies the view's rules (archived, invalid, search), excludes virtual entries unless asked, and supports count, tags, quick filters and relation quick-view.
- **Concurrency.** A save whose `updateIdentifier` no longer matches is refused as a conflict.

API: DataCollections reference · Reading a collection.

Exports

- **Quick export** (**Export Entries**) requires the view's entries list and its **Allow export** setting. It writes XLSX or CSV of every entry matching the request (search, filter, quick filters with their parameter values, tags, filter-first values), or of up to 150 selected ids. Column order follows the user's table columns, else the view's fields. The file is named `<list name>_<yyyy-MM-dd_HH-mm>.<ext>` in the user's local time.
- **Limits.** Up to `MAX_ENTRIES_RESULT_LIMIT` (default 10,000) rows are returned synchronously. Larger exports for signed-in users run in the background, page by page, capped at `MAX_BACKGROUND_EXPORT_ROWS` (default 100,000), and deliver a "file ready" notification; a capped file's name says `first-<n>`. A failed background export delivers neither a file nor a notification. Anonymous callers get the first `MAX_ENTRIES_RESULT_LIMIT` rows only.
- **CSV.** UTF-8 with a byte-order mark, comma-separated. Text beginning with `=`, `+`, `-` or `@` is prefixed with an apostrophe; numbers and numeric-looking text are left alone. A block field becomes `Block › Field` columns; in XLSX it gets a merged two-row header. No group headings or totals are written.

- **Value formatting** is shared with the XLSX list export: references as titles, dates as `dd/MM/yyyy`, date-times as `dd/MM/yyyy hh:mm`, a ranged Time field as `from - to` (since 1.10), passwords as `****`. A value that can't be formatted leaves its cell blank instead of failing the file (since 1.10).
- **XLSX list export** (`GET /v1/entities/{entity}/views/{id}/entries/export/xlsx`, with `fileName`, `filter`, `filterFirst`, `include_virtual`, `selectedColumns` and `collectionId`) is what **Export list** calls, on the web and in the mobile app.
- **Reports** export PDF or XLSX, never CSV; a page offers **Export to PDF** only when its `allowExportPDF` setting is on. See Reports: export renditions.
- **Administrators** have the same Excel or CSV export on an entity's own page: Data Modeling: quick export of an entity's entries [Build Apps › Create and design an entity].

API: Entity entries export reference.

Use an API token

This page is for developers who connect a script, a service or an automation tool to OSOS without a person signing in. You will get an API token, send it with your requests, keep its access narrow, and rotate it without downtime.

An API token is the simplest credential the platform offers: no login screen and no token exchange. Send it in the `x-app-token` header on every request. Each request then runs as the token's **application user**, with exactly that user's permissions.

When to use an API token

Your caller	Use
A person in a browser or mobile app	OAuth sign-in (authorization code) — see Authentication & OAuth
A trusted first-party script that holds a user's password	The OAuth password grant
A service, script or automation acting as itself	**An API token**. Not OAuth client credentials: that token has no user, so it can reach almost nothing

The full decision, including what an API token cannot express, is in Choosing a credential.

Get a token

A tenant administrator creates the token: they add an **application user**, and the platform emails the token to them once. There is no API to create a token and no screen that shows one. Ask your administrator to follow Create an API token for an integration [Administration], and to give the application user access to exactly the data your integration needs.

You receive from them:

- the token — a 64-character hexadecimal string;
- the tenant's host, for example `https://acme.example.com`;
- what the application user is allowed to read and change.

Store the token in a secret store or an environment variable, never in source code.

Send the token

Put the token in the `x-app-token` header and call the tenant's own host:

```
export BASE="https://acme.example.com"
export TOKEN="<your-api-token>"

curl -s "$BASE/api/v2/entities/<entity>/entries/count" \
  -H "x-app-token: $TOKEN"
```

```
{ "success": true, "result": { "count": 42 } }
```

The tenant comes from the host you call; you never send a tenant id. Every other convention — paths, versions, the response envelope, errors — is the same as for any credential: see Calling the HTTP API, and the Backend quickstart for a guided first call.

What the token can do

A token has **no scopes of its own**. It authenticates as its application user, and the platform then applies that user's system level, space membership and permissions, exactly as for a person. So:

- To limit an integration, limit its application user. For data you only read, ask for a view shared with the application user, and read through it (`/api/v1/v/{viewId}/…`, see Calling the HTTP API). **View** on the entity itself is not enough: it shows the design, not the records, and listing the entity's records then returns an empty list, not an error. Ask for **Edit** only where you write.
- Give each integration its own application user and token. You can then revoke one without touching the others, and the record history shows which integration made each change.
- A token never grants more than its user has. A `403` on a request, or an empty list where you expect records, means the application user lacks access to that data, not that the token is wrong.

Expiry

The administrator chooses, per application user, whether the token never expires or expires on a date. The token stops working **at the end of the expiry date**. An expired token is refused like any other invalid token (see Errors); the fix is for the administrator to extend the **Expiry date** on the application user, not to regenerate.

`Since 1.10` administrators can set the expiry date. Before 1.10 the field was hidden and every token was created without an expiry.

Rotate a token without downtime

An administrator with the system **Admin** level can **regenerate** the token. The application user, its permissions and its history stay the same; only the secret changes. `Since 1.8`

1. The administrator selects **Regenerate Token** on the application user. The new token is emailed to that administrator — it is never returned by the API.
2. **Both tokens work during a grace period** — 24 hours by default; your operator may change it.
3. Deploy the new token to your integration within that window. After it, the old token is refused.

Design your integration to read its token from configuration, so a rotation is a configuration change, not a release. If the email with the new token is lost, regenerate again before the old token's grace period ends.

Regeneration is refused for an archived application user. Rotation for a function's own token works differently: the new token goes straight to the function — see Functions.

Revocation and its delay

A token stops working when the administrator **deactivates** or **archives** its application user, or when it expires.

The gateway remembers a successful check for a short time — **five minutes by default** — so a revoked token can keep working until that time runs out. Plan for this delay when you revoke a leaked token: rotate or revoke first, then watch for calls that still succeed within the window.

How this delay interacts with OAuth sessions, and how to design around it, is in Token lifecycle.

Errors

Status	`error.code`	Meaning	What to do
`400`	`INVALID_APP_TOKEN_API_ERROR`	The token is unknown, expired, deactivated or archived.	Do not retry. Check the token value and ask the administrator about the application user.
`403`	depends on the operation	The token is valid, but its application user lacks access to this data.	Ask the administrator to grant the access.
`500`	`APP_TOKEN_VALIDATION_REQUEST_FAILED`	The gateway could not reach the identity service to check the token.	Retry with backoff.

Errors use the platform's standard error body. The full catalogue and the retry policy are in Errors and retries.

Technical details

- The token is a 64-character hexadecimal secret. The platform stores only a hash of it, which is why no one can show it again.
- A request with `x-app-token` is authenticated by that header. A request without it falls back to the normal `Authorization: Bearer` path, so the two credentials never conflict.
- Each time the platform checks the token (not on every cached request), it records a "last used" time.

Good to know

- **Shown once, by email only.** If the token is lost, the administrator regenerates it; there is no way to read it again.
- **No per-token scopes.** All authorization comes from the application user.
- **Acting for a person is a different mechanism.** An API token always acts as its own application user. For one person acting for another during an absence, see Delegation [Administration]; for a third-party app acting for a signed-in user, use OAuth.

> ### Go deeper
>
> For administrators: Create an API token for an integration [Administration] · For operators: Gateway operations — app-token-auth · For platform engineers: Identity, OAuth and API tokens — internals

Integration FAQ

Common questions from **integrators and professional developers** — calling the HTTP API, authenticating scripts and services, handling error codes, reacting to events, writing functions, and driving packages through the API. Questions are grouped by topic; select a question to expand its answer.

> **Building in the designer?**
>
> Questions about modeling data, views, workflows, reactions, pages and moving an app through the **Packages** screens are in the Build apps FAQ [Build Apps].

Extending and integrating

Is there an HTTP API, and how is it versioned?

Yes — the platform exposes a large HTTP API reached through a single gateway, with all paths under `/api`. Operations are **versioned in the URL path** (`/v1/…`, `/v2/…`, `/v3/…`); versions coexist rather than replacing each other, and newer versions usually add capability, so prefer the highest version a resource offers. Multi-tenancy is header-scoped, so the same operation returns different data per space. Successful JSON responses are wrapped in `{success, result}`, with one exception on 1.10: a `GET` under `/v3/` returns the bare payload — see `GET /api/v3/…` is raw again.

How do I authenticate a script or integration without a user login?

Use an API token (also called an application token). The program sends it in the `x-app-token` header, and the gateway resolves it to the owning application user. The token value is shown **only once** on creation — copy it then, because it cannot be retrieved later. To revoke a token, archive it (subject to a short caching delay). Give each integration its own token so you can revoke one without affecting the others.

Do API tokens carry their own permissions?

No. An API token has no separate permission set or scopes — it inherits the access of the **application user** that owns it. To limit what an integration can do, create a dedicated application user with exactly the permissions that integration needs and issue the token under that user.

How are uploaded files stored?

Files (in file, image, and document fields, and in comment attachments) are uploaded to a storage service, which returns an **id**; the record stores that id, not the bytes. Permitted attachment types are configurable by an administrator **per view** (with a default size limit around 1 MB) rather than a fixed set. Uploads coming from end users are validated (extension allowlist, real-content magic-byte checks, and a PDF embedded-JavaScript scan).

How do I get an OpenAPI document for a single view?

Turn on **Swagger document** in that view's display options; only then does `GET /api/v1/views/{viewId}/openapi.json` publish a small OpenAPI 3.0 document scoped to that view. Watch one deliberate inversion: **that response is not wrapped in the `{success, result}` envelope** — parse the body as a

bare OpenAPI document — while the document it returns *does* describe the envelope, because every operation it documents answers inside one. Two other limits are worth knowing: fetching the entity-schema operation always needs a token, even on an anonymous view; and the entry `value` schema is built from the view's own field selection and is **not** filtered by sensitivity, so a sensitive or encrypted field the view displays appears there by name, type and cardinality (never by value). See The per-view OpenAPI document. *(Since 1.10)*

Why does a `GET /api/v3/…` call return a bare body without `{success, result}`?

Because on the 1.10 line a `GET` whose path starts with `/v3/` is sent as the raw payload again, for the external integrations that read `/v3` bodies directly; `/v3` writes, `v1` / `v2` and unversioned JSON routes keep the `{success, result}` envelope, and errors stay enveloped. A 1.8 deployment still envelopes `/v3` reads up to and including 1.8.21, so a client that talks to both lines should unwrap `result` only when the body carries `success: true`. See `GET /api/v3/…` is raw again. *(Since 1.10)*

What do `404 SPACE_ERROR` and `400 SPACE_CONTEXT_MISSING` mean?

`404 SPACE_ERROR` comes from the gateway when the request host matches no space — it was `400` before 1.10 — and any `x-space-id` the client sent is discarded first, so use a host registered for a space. `400 SPACE_CONTEXT_MISSING` means a service was called directly without a usable `x-space-id` while space-scope enforcement is on; it does not happen through the gateway, and a direct caller must send exactly one non-blank `x-space-id`. See Tenancy- and space-related refusals. *(Since 1.10)*

Can a browser load a stored file in a plain `<img>` without a bearer token?

Yes: `POST /api/storage/v2/files/links` signs short-lived links for up to 100 file ids, and `GET /api/storage/v2/files/:id/content?exp=…&sig=…` streams the file with the signature as the only credential. Treat each link as a **bearer capability** — anyone holding one can fetch the file until it expires — so never log links or put them in shareable URLs. Delivered first to the 1.6 line and to 1.8 as an update. See Signed, cacheable content links. *(Since 1.6)*

How do I start sign-in directly with one specific external provider?

Add the authorization-request parameter `idp_hint=<provider name>` to `/auth`: when the login page would be shown, the identity service redirects straight to that **active** external provider if it is linked to the calling domain, and ignores unknown, inactive or unlinked names. The value is the provider's unique `name`, not its display label, and the portal's `/signin?provider=<name>` deep link feeds this parameter. Separately, a domain whose only way in is one external provider skips the login page for every client by default. See Starting sign-in with a named provider. *(Since 1.10)*

How do I give a function an API key or another secret?

On the development line, add a row to the function's **Secrets** block, with a `key` (the environment-variable name: letters, digits and `_`, not starting with a digit) and a `value`, through an ordinary save of the function's entry; there is no separate secrets API. The value is stored encrypted and masked when read back. The function reads it as an environment variable, for example `process.env.MY_API_KEY`, once a new revision starts: a change is synced without a rebuild and rolls a new revision, while a pod that is already running keeps the old value. See Function secrets and per-function identity. *(Since dev)*

Who does a function act as when it calls the platform API?

On the development line, as **its own application user** (issuer `function:<functionId>`), whose token the function receives as `OSOS_API_TOKEN`. That user starts with no groups, roles or admin access, so the function can do only what you grant that user explicitly. Uploading the function's code again does not rotate the token, and an administrator who regenerates it gets no e-mail: the new token is handed straight to the function. See The per-function identity. *(Since dev)*

Why does my integration not receive an updated event after a write?

On the development line, a write whose entry was deleted before data-service re-read it publishes **no** created or updated event: the write is reported as a failure, and the deletion has its own `DataEntryDeletedEvent`. An event that is published still always carries `dataEntry`. So do not wait indefinitely for an updated event on an entry that may have been deleted. See Data Model & Events. *(Since dev)*

Why did a save fail with `400 FILE_EXPIRED`, and how should my client attach files?

Because the save named a file id that storage cannot find — usually a **temporary** upload older than a day, or an id that never existed. Since 1.10 storage keeps a ledger of which saved record uses which file, and the robust pattern is the one the web portal uses: upload with `?temp=true`, save the record with the returned descriptor within the day, and the save **promotes** the file to permanent storage under the same id. Only ids **new** to the stored record are checked, so re-saving an old record never fails on files it already held; `details` lists each missing file with its field. The fix is always to upload again and re-save. If storage cannot be reached while a save adds a new file, the save answers a retryable `503 STORAGE_SERVICE_UNAVAILABLE` instead. Permanent uploads still work as before, but they are not tracked. See Temp upload, then save and FILE_EXPIRED and storage-unavailable. *(Since 1.10)*

Will the platform delete a file that nothing references any more?

Only a **tracked** file, and only after every former owner confirms it. The orphan file sweeper considers files that were promoted from a temp upload and later released; a file uploaded straight to permanent storage, or a temp file that never reached a save, is never swept. A file must have been orphaned for a grace period (7 days by default), and any `listed` or `unknown` answer from a former owner keeps it. In the service the sweeper is off by default; the deployment chart turns it on in **dry-run** mode, so it only logs what it would delete until an operator switches dry run off. Deleting a package version or an export never deletes its ZIP. See Orphan file sweeper and File reference ledger. *(Since 1.10)*

Can my client set `x-service` or `x-view-id` on a request through the gateway?

No. Through the gateway, a client-supplied `x-service` is removed and set to `API Gateway`, and a client-supplied `x-view-id` is cleared before any access check and set again only from the `{viewId}` of a `/api/v1/v/{viewId}/…` path — so to act through a view, use that path form. This joins the long-standing removal of `x-shared-key` and the other internal headers. A service reached **directly** has none of this protection. See What the gateway strips and Service and view headers are removed too. *(Since 1.10)*

Why does a call that used to fail with `500` now answer a `4xx`?

Because refusals of the caller's input or entitlement now carry their own status and a stable `error.code`. Examples: `400 INVALID_POINTER` for a pointer such as `@me` or `@today('-1d')` that is not a known pointer function, `400 MISSING_REQUIRED_PARAMETER` for an entry create sent with no body at all, `403 ACCESS_OBJECT_NO_AUDIENCE`, `404 QUERY_NOT_FOUND`, and `404` for a space that does not exist. Separately, a `4xx` from another OSOS service is now relayed with that service's own status rather than flattened to `500` — that one also shipped in 1.8 patches (`Since`

`1.8`). Branch on `error.code` and treat the status as the coarse signal. See Typed refusals that used to answer 500 and Write refusals added in 1.10. *(Since 1.10)*

What does `503 TENANT_IN_MAINTENANCE` mean, and what should my client do?

In a multi-tenant deployment, it means the tenant's data is being **restored** from a backup, or an operator is holding the tenant for a **move** to another cell. Every request for the tenant — including identity's `/auth`, `/token` and refresh — answers `503` with `Retry-After`; wait that long and retry rather than signing the user out. Bus consumers **park** the tenant's messages meanwhile and replay them afterwards, with one deliberate exception: a message first parked during a restore that rewound the data is **dropped**, so an integration that relies on an event for an outside effect must treat events from inside a restore window as possibly never delivered. Single-tenant installations have no maintenance state. See Delivery while a tenant is under maintenance and Write refusals added in 1.10. *(Since 1.10)*

Can I back up, restore or export a tenant through the API?

Yes, as the tenant's own **system administrator** in a multi-tenant deployment — there is **no web screen for it in 1.10**, and in a single-tenant deployment every route answers `501 BACKUP_NOT_SUPPORTED_IN_SINGLE_MODE`. Each start answers `202` and you poll: `POST /api/v1/admin/backups`, `POST /api/v1/admin/exports`, and a restore in two steps — a preview that returns a single-use `confirmationToken`, then the restore with that token. From the start of a restore the tenant answers `503 TENANT_IN_MAINTENANCE` to everything except the `GET` routes of this family. An export is downloaded through a single-use link (at most five per export) and holds the tenant's databases **without user accounts**. Starts count against per-day quotas (`429` with `Retry-After` when exceeded), and there is no cancellation. See Tenant backups, restores and exports and Long-running work. *(Since 1.10)*

Why does my client receive real-time notifications from every space?

Because notifications are **not space-scoped on the socket**: the `notification` push reaches the user's room whatever space it was raised in. The shipped web client drops a pushed notification that belongs to a space other than the active one, but that filter is client-side only. A custom client should filter on the entry's `value.space` itself. See Real-time. *(Since 1.10)*

Does the V2 end-user web need its own OAuth client?

No. When a deployment serves the end-user web (`END_USER_ENABLED=true`, at `END_USER_PATH`, `/v2` by default), the standard public web client also allows `https://<domain><END_USER_PATH>/auth-callback` for every domain of every tenant — the same `client_id`, with post-logout URIs unchanged. With the end-user web off, no such redirect URI exists. This is the routing only: the V2 web app itself is still `Since dev`. See Authentication & OAuth. *(Since 1.10)*

Data and information modeling — API and errors

Can I move an entity to another module?

Yes, as a system administrator. Since 1.10 moving an entity (`PATCH /v1/entities/{id}/module`) is system-admin only, and `POST /v1/entities/move` moves 1 to 50 entities into one module in one request, reporting each as moved or failed; virtual entities can be moved too. When an entity moves, its child views and collections follow it; the move is refused into the protected core module and into a module that is being permanently deleted (`409 MODULE_BEING_DELETED`). The entity's normalized name and type stay fixed through the move. See Admin move, bulk move and what a move repairs [Build Apps › Create and design an entity]. *(Since 1.10)*

Are there limits on how much data or how many records I can store?

Only if the tenant's plan sets one — and since 1.10 a plan can. Individual field types still impose their own size or count limits, but the platform-wide limits are now plan limits, and a limit that is not set is not enforced:

- **Entries** — counted daily and weighted (active and draft entries count 1, archived 0.25; only user-built entities count). A warning starts at 80 % of the cap; creating an entry — single, batch, draft or through a proxy — is refused with `403 ENTRY_CAP_EXCEEDED` once the total reaches the **hard limit** (the cap plus 10 %), unless overage is granted. Reading, editing and exporting are never refused.
- **Entities** — creating one beyond the limit is `403 ENTITY_CAP_EXCEEDED`; implicit and core-module entities do not count.
- **Spaces** — `403 SPACE_CAP_EXCEEDED` on any write that adds a space beyond the limit.
- **Storage** — an upload at or across the limit is `403 STORAGE_CAP_EXCEEDED` from the storage service.
- **Reaction runs per month** — a run over the limit fails no request; it is recorded as `refused` and none of its actions run.

A package install that would pass the entries limit is refused up front. See Plan limits: usage and refusals, Entity cap and, for what an administrator sees, Plan limits [Administration › Your license & plan limits]. *(Since 1.10)*

Can I permanently delete an entity or a whole module?

Yes, since 1.10, as a system administrator. `DELETE /v1/entities/{id}/permanent` irreversibly removes a Custom, Request or virtual entity and everything it owns, and `DELETE /v1/data-module/{id}/permanent` removes a module and everything in it as a background job; the ordinary delete routes are unchanged. Both follow an "unlink first" rule — they refuse with `409 ENTITY_HAS_DEPENDENTS` / `409 MODULE_HAS_DEPENDENTS` while anything outside still references them, and refuse resources installed from a package with `403`. See Permanent delete (purge) [Build Apps › Create and design an entity] and Modules — Permanent delete [Build Apps › Modules]. *(Since 1.10)*

Can I add my own database indexes to an entity?

Yes. The entity builder's **Indexes** tab lets an administrator define up to **10** extra MongoDB indexes per entity — 1 to 8 fields each, optionally **Unique**, with an optional partial filter — to speed up a frequent filter or sort or to enforce uniqueness across fields. A unique index must carry a partial filter that excludes empty values, because MongoDB indexes a missing value as `null`; the tab is not offered for core-module, `Request` or virtual entities. See Entity indexes (Indexes tab) [Build Apps › Create and design an entity]. *(Since 1.10)*

Fields and data types — API and errors

Can a Category field hold several values, and what happens when I turn that on?

Yes, since 1.10: turn on **Multiple** in the field's value settings, optionally with a minimum and maximum number of selections, and the field stores an array `[{id}, …]` instead of a single `{id}`. Turning it on is a **breaking change** — the entity's version is bumped and existing entries are wrapped into one-item arrays when upgraded — and the switch is refused with `400 CATEGORY_FIELD_MULTI_VALUE_IN_USE` while a Kanban board, grouping, sort, dashboard or timeline relies on the field. A multi-value category can never be the main category, a Kanban column field, a default sort or a dashboard group-by (`400 MULTI_VALUE_CATEGORY_FIELD_NOT_ALLOWED`). See Multiple values [Build Apps › Category field]. *(Since 1.10)*

Can end users add or edit a Category field's options themselves?

Yes, from an entry's create or edit card, when two settings line up: **Allow users to manage options** on the Category field (not allowed on a field whose options are mapped from another field), and **Add category options** / **Edit category options** on the view's create or edit operation. The changes travel with the entry write as `categoryOptionChanges` and are saved together with the entry; a conflicting concurrent change is refused (`CATEGORY_OPTION_CONFLICT` or `CATEGORY_OPTIONS_CHANGED`) and the user reloads. See Letting users manage options [Build Apps › Category field]. *(Since 1.10)*

Working with data — views, faces, and entries — API and errors

How do I let end users reorder repeated values?

Turn on `allowRepeatableReorder` — a **per-operation** view-configuration option that sits on each entry in the edit configuration's supported-operations list, not on the field definition. So the same field can be reorderable on Create and fixed on Edit within one view, or reorderable on one view and not on another. It becomes selectable only once the operation's field list contains at least one repeatable target (a repeatable field or a repeatable block), and it resets itself to off if you remove the last eligible field. When it is on, each repetition gains a drag handle on the create or edit form; the handle appears only when there is more than one repetition, and dragging marks the form dirty so the new order is saved with the entry.

Both clients honour this setting. The mobile app reads the same per-operation flag off the view configuration and renders its own drag handle on each repetition beyond the first, so a view configured for reorder behaves the same in the web portal and on the phone. (Repeatable reorder was web-only when it first shipped; that is no longer the case.) See Data views [Build Apps › Data Views & Faces]. *(Since 1.8)*

Why did saving a view or page answer `409 UPDATE_CONFLICT`?

Because someone else saved it after you opened it. A view or page is saved as a whole document, guarded by its `updateIdentifier`. Before 1.10 only an up-front read compared the tokens, so two saves that both passed it could overwrite each other; now the write itself is conditional, and the second of two concurrent saves loses with a `409`. Re-read and save again. Two other view-save refusals are now typed rather than `500`: enabling entry edits with no supported operation is `400 DATA_VIEW_INVALID_ALLOWED_OPERATIONS`, and asking whether a caller may open a view, page or report that enables **neither** audience is `403 ACCESS_OBJECT_NO_AUDIENCE`. See Saving a view: conflicts and refusals [Build Apps › Data Views & Faces]. *(Since 1.10)*

Can I rename a view or page that was installed from a package, or change its icon?

Yes. A view or page an installation wrote is locked, but its **name, description and icon** can be changed in the editor (not the name of an implicit view), which shows the package's value beside yours. Your label is kept across package updates; the package's own value is recorded in `packageLabels`, which a client cannot write. Through the API, send the labels on `PATCH /api/v1/entities/{entity}/views/{id}` or `PATCH /api/v1/generic-views/{id}` with the current `updateIdentifier` (`409 UPDATE_CONFLICT` otherwise); `description: null` clears the description, and `icon: null` clears the icon **only** with `?clearIcon=true`. Publishing state, display order, audience and access rights also stay yours. Anything else is still `403 PACKAGE_RESOURCE_READ_ONLY`. See Installed views [Build Apps › Data Views & Faces] and Update flow: concurrency and installed pages [Build Apps › Pages & Widgets]. *(Since 1.10)*

Queries, search, and sorting — API and errors

How does a parameterized quick query get its values?

Since 1.10 each applied quick query carries its own values: end users get a **Query parameters** dialog, and the API takes `quickFilterParameters` keyed by query id (`{"<queryId>": {"<parameterName>": value}}`), falling back to the flat `parameters` object so existing Grafana panels keep working. A missing value fails the request with `422 QUICK_FILTER_PARAMETER_VALUE_MISSING`, and a parameterized quick query can never be applied by default. See Parameterized quick queries [Build Apps › Queries & filters]. *(Since 1.10)*

Services and workflows — API and errors

How do I hand a workflow step off to an external system?

Use an **External Step** (`StepType.ExternalStep`). Its `ExternalStepConfig` is required and carries a **URL**, a **Method** (`GET` or `POST`), and **Open As** — how the destination is presented to the owner. A `POST` step may also carry a **Custom Payload**; by default the request sends the entry's details plus the step's actions. Embed the callback with the `{{@returnUrl}}` token in the URL or the payload — use `{{@returnUrl:actionId}}` when the step has more than one action, since a bare token is auto-substituted only when there is exactly one. The step's owner must resolve to exactly one specific user (group, dynamic, and expression owners are rejected at save), and the step completes only when the external system calls back. See Workflow model and API reference. *(Since 1.8)*

Why did a workflow action come back with `409 UPDATE_CONFLICT`?

Because someone else changed the entry's workflow first. Since 1.10 every workflow write — executing an action, starting or completing a sub-workflow, inserting an ad-hoc step, changing the owner or the step — is saved only if the workflow is still in the state it was validated against, and a lost race answers `409 UPDATE_CONFLICT` with nothing recorded or published. Reload the entry and decide again; a step-up proof is given back for its remaining lifetime so the retry can use it. See Concurrency guard on workflow writes. *(Since 1.10)*

Why do executing a hidden action and taking an owned step now answer `422`?

Because they are refusals of the request, not server faults. Executing an action that is hidden on the entry answers `422 WORKFLOW_ACTION_EXECUTION_NOT_ALLOWED`, and taking ownership of a step that already has an owner answers `422 CURRENT_STEP_ALREADY_OWNED`; both used to be `500`. It is deliberately `422`, not `403`, because the web client sends every `403` to its access-denied page. Client code that matched on the `500` should match on the error code instead. See Typed refusals that used to answer 500. *(Since 1.8)*

Reactions and automation — API and errors

Can a reaction call an external system or run my own code?

Yes. A reaction's **Call API** action (`CallApi`, 90) sends an outbound HTTP request (GET/POST/PATCH/PUT/DELETE) and captures the response, and the **Execute Function** action runs a serverless function you have deployed. Reach for these when you need external network calls or custom logic beyond the built-in, no-code action catalogue.

What replaced the "UI Event" toggle on a custom action trigger?

Two settings. The trigger is now named **Custom Action** in the builder (it is still `CustomEvent` in the code and the API), and it carries:

- **Custom action level** (`customActionType`, `CustomActionTypeEnum`) — a dropdown with three values. **UI** (*UI action*) means the action is invoked from a user interface, i.e. a system button, and is the equivalent of the old *UI Event* toggle being on. **External** (*External action*) can be invoked from another reaction's Execute Custom Action [Build Apps › Reactions & automation] in any module, and **Local** (*Local action*) only from inside its own module; reaction-service refuses an Execute Custom Action that points at a UI action. The values were first shipped as `Module` and `Internal` and renamed by a one-way migration in the same 1.8 cycle (`Module` → `Local`, `Internal` → `External`).
- **Contextual** (`contextual`) — a toggle, on by default, stored on the trigger and shown in the editor. No runtime behaviour is attached to it yet; it is a forward-looking marker, not a switch that changes how the reaction runs today.

Two rules follow from the type: the **Show to user** options (execution indicator and its On Start / On Success messages) apply only when the type is **UI** — the service refuses them on an External or Local trigger — and asking the user for input no longer depends on the type. Buttons are unaffected on the wire: the system-button DTO still exposes a plain boolean `uiEvent`, now **derived** from the type being UI rather than read from a stored flag.

Existing reactions were all migrated to type **UI** with Contextual on, and the retired flag removed. The migration is one-way, so review any trigger that should now be External or Local and change its type by hand. See Custom action type and Contextual [Build Apps › Reactions & automation]. *(Since 1.8)*

Can a reaction run against an entry from a different entity?

No. A reaction runs **as its own entity** — its actions resolve field paths and write through that entity's definition — and the engine now enforces that explicitly. An entry whose entity differs from the reaction's own entity is **skipped** with a warning in the service log instead of being run through the wrong entity's field definitions. For related-entity triggers, entries pulled in through a relation are filtered per trigger down to that trigger's own entity *before* they are fetched, which matters for a **reference field that accepts more than one entity**: previously every reaction sharing that relation received every referenced entry, whatever entity it belonged to. And a trigger whose defining entity cannot be resolved — a deleted entity, say — is skipped with a warning rather than failing the whole run.

No new error is raised for any of this. A foreign entry or an unresolvable entity is a logged skip, not a failure, so nothing appears in the reaction's execution history. If a reaction seems not to have fired for some entries, look for skip warnings in the reaction-service log rather than for a failed action result. See Entity isolation [Build Apps › Reactions & automation]. *(Since 1.8)*

Can a reaction grant or remove access permissions?

Yes, since 1.10, with the **Set Permission** and **Remove Permission** actions, on a System, Module, entity, view, page or report resource for users or groups; `Restricted` and `None` are never grantable, and Set never lowers a higher existing grant. A reaction cannot grant more than the person who saves it: each permission action is checked against the saver's own access at save time, and a refusal is `PERMISSION_ACTION_EXCEEDS_SAVER_ACCESS`. See Set / Remove Permission actions [Build Apps › Reactions & automation]. *(Since 1.10)*

Why was saving a reaction refused with `REACTION_AUTHORING_FORBIDDEN`?

Because creating, updating or deleting a reaction needs effective **Developer-or-higher** access on its entity, and the caller does not have it. The refusal is a `403`. Ask for Developer on the entity (or its module), or have someone who

holds it save the reaction. See Typed refusals that used to answer 500 and Reactions & Automation [Build Apps › Reactions & automation]. *(Since 1.8)*

Why did a reaction run show as *refused*, with none of its actions run?

Because the tenant reached its plan's **monthly reaction-run limit** (`maxReactionRunsPerMonth`). The limit is enforced inside the automation service and fails no request: the run is recorded with status `refused` (code `REACTION_RUN_CAP_EXCEEDED`, with the limit, the count used and the period), and none of its actions run. Find such runs with `outcome=refused` on the run history, `status=refused` on the reaction logs, and the `refusedCount` the reaction list returns beside `executedCount`. A single-tenant deployment does not meter reaction runs. See Plan limits: usage and refusals. *(Since 1.10)*

Can a reaction send a WhatsApp message?

Yes, with the **Send WhatsApp** action, which sends one of the space's **approved WhatsApp message templates**. The template list comes from `GET /api/reaction/v1/whatsapp/templates` (System View access or above), which answers `409 whatsapp_not_configured` until the space's WhatsApp business account is set up and leaves out authentication-category templates. The account is configured in the System Settings entity — there is no dedicated admin screen in 1.10 — and each message's delivery status is recorded in the WhatsApp message log. See WhatsApp and, for the action itself, Reactions & Automation [Build Apps › Reactions & automation]. *(Since 1.10)*

Why didn't my `ReactionFailed` reaction fire when the monthly run limit stopped a run?

Because a refused run is not a failed run. Once the tenant has used its monthly run allowance, the run is recorded as `refused`, none of its actions run, and no "this reaction finished" event is published, so neither a `ReactionFailed` nor a `ReactionSucceeded` reaction fires on it. Watch for refused runs instead, with `outcome=refused` on the run history or the reaction's `refusedCount`. See Monthly run limit and refused runs. *(Since 1.10)*

Why can't my Call API action reach an internal host?

Because a Call API action refuses private and internal destinations: private, loopback, link-local (cloud metadata included), CGNAT, multicast and reserved addresses, IPv4 or IPv6. Host names are checked when they resolve, so a public name that points at a private address is refused too, and every redirect is checked again. Only the operator of your deployment can open a particular private destination, through the `CALL_API_ALLOWED_PRIVATE_DESTINATIONS` allow-list, so ask them. Other limits apply to every call: HTTP(S) only, 30 seconds for the whole exchange, and a 10 MiB body cap. See Call API: what the platform allows [Build Apps › Reactions & automation]. *(Since 1.8)*

Buttons — API and errors

Why was a system button silently removed when I saved a view?

Five system-button ids are **implicit** — `VIEW`, `EDIT`, `ARCHIVE`, `CREATE`, and `DUPLICATE`. They are not stored buttons; they are derived from the view's own configuration. If your save payload still carries one the current configuration no longer permits — the classic case is an `EDIT` or `DUPLICATE` button left behind after you switched the matching operation off — the save now **succeeds and prunes the disallowed implicit button** instead of failing with a "system button not found" error. Only those five ids are eligible: custom system buttons and template buttons are never pruned, and a reference to one that does not exist is still an error. See Data views [Build Apps › Data Views & Faces]. *(Since 1.8)*

Permissions, sharing, and sensitive data — API and errors

Is there an API for managing sensitive-field grants?

Yes — the `/v1/sensitive-field-grants*` endpoints, which back the administrator's **Sensitive Permissions** page and the per-user / per-group tabs. `GET` returns every active grant space-wide, or a single user's direct plus group-inherited grants, or a group's own grants, or the principals holding a wildcard grant that covers one field. `POST` creates grants (reporting created rows and skipped duplicates), `PATCH` updates the expiry only, and there are separate hard-delete and archive routes. They introduce **no new storage** — `AccessRight` remains the single source of truth, and these are new read/write shapes over the same records the field editor writes. Unlike the field editor, they can author wildcard scopes and validity windows. See Sensitive fields [Build Apps › Protect a field]. *(Since 1.8)*

Why is granting a permission refused with `PERMISSION_GRANT_EXCEEDS_CALLER_ACCESS`?

Because below system **Admin** you cannot grant a level above your own effective level on the target, nor change or remove a grant that is already above it. The refusal is a `403` whose details carry `callerLevel`, `requestedLevel` and `currentLevel`. Your effective level counts your system grants, your direct and group grants on the target, and what you inherit from its module or entity; under a delegation, the delegation's permissions count. The same ceiling already applies to reactions that set permissions. See A grant cannot exceed your own level [Administration › Permissions & access]. *(Since 1.8)*

Pages and interfaces — API and errors

What is the `layouts` property on `GET /v2/views/end-user/{id}`, and can I use it?

For a view on the **Layout builder** display template, that route resolves the SHELL entries the template names (`layout_list`, `layout_view`, `layout_edit`) into `layouts` — `{listEntry, viewEntry, editEntry}`, each `{id, name, definition}` or `null` when nothing usable is linked. The template stores only references, so a layout edit reaches every linked view without re-saving it, and anonymous visitors still receive the definition; the v1 route does not carry `layouts`. The backend half is in 1.10, but **no shipped client uses it yet**: the Layout builder template and the shell builder that read `layouts` are still `Since dev`. See Layout references on the end-user route [Build Apps › Data Views & Faces]. *(Since 1.10)*

Packaging formats and APIs

How do I move an application between environments (dev → staging → production)?

Use packaging. It bundles selected resources — modules, entities, views, queries, relations, reactions, reports, pages, buttons, collections, functions, and more — plus everything they depend on into a single ZIP that you export from one environment and import into another. Import is a guided, session-based flow (validate → configure mappings → execute), and it remaps source ids to target ids as it goes. Always run the export **analysis** first and resolve external-dependency warnings before shipping.

What is left out when I export a package?

Some resources are filtered out even if selected: **archived** entities (and their nested subtree), **user-scoped** resources, and **implicit** entities/reactions. Core identity entities (Users, Groups, Org Units, Working Hours) are never written into the package — they exist independently in every environment and are handled as references, which is

where principal mapping and mapping profiles come in. Only hard dependencies are guaranteed to travel with a package; soft (optional) references may be nulled on import.

Why did picking an import target for a shared user or group also re-point another resource's reference?

Because both resources were bound to the same **row-level** decision for that principal. When the same user, group, or org unit is referenced from more than one **resource** in one import — a workflow step owner on an entity and the user a view is restricted to, say — each resource's copy of the reference is now retargetable independently: the wizard recognises that a principal renders under more than one resource row, and lets each resource's group read and write only its own occurrences. A principal confined to a single resource is unaffected and keeps using the row-level decision, which is also what a stored mapping profile remembers.

The row-level default is kept when one group diverges. An occurrence-level choice is only ever a *partial* override layered on the row-level decision, and the server rejects an item-level remap carrying no target of its own. So the first target you choose for a shared principal also establishes the row's default, and a later group diverges *against* it rather than replacing it — which is what makes a choice made during Configure survive through to Execute. **Auto-create** on a principal that spans resources always applies at the row level, since auto-created entries are keyed by the principal, not by where it appears. See Packaging. *(Since 1.8)*

Can I import a schema 1.2 package into an environment that hasn't been upgraded?

Only if the target is on **1.10 or later**. A pre-1.10 deployment still accepts `1.0` and `1.1` alone, and a package declaring `1.2` is refused there with a blocking `INCOMPATIBLE_SCHEMA` before any target-space lookup runs. The allow-list is not a range check, so "newer" does not mean "tolerated". This is why export stamps the **lowest** version that describes the package — `1.2` only when it actually carries the `field-mapping-hints.json` sidecar, `1.1` otherwise — so an export with no mapped fields still imports into an environment that has not been upgraded. The operator rule that follows is **upgrade the target environment before the source**. The same holds for `1.3` and `1.4` — declared by a published package version and by any package that carries a README — which a pre-1.10 target also refuses with `INCOMPATIBLE_SCHEMA`; see What 1.3 and 1.4 add. See Schema versioning. *(Since 1.10)*

Can I publish a template to the marketplace and install apps from an in-app store?

Yes, in 1.10: a **package** owns modules, its **published versions** (and its **store page**) go through an internal review inbox before reaching the marketplace, and the **in-app store** installs from the catalog. The marketplace is **off until an operator enables it** (every marketplace route answers `503 MARKETPLACE_DISABLED` meanwhile). Once it is on:

- **Browsing needs no connection**; installing does, and the first install connects the environment after you confirm.
- **Connecting and publishing are separate.** Connecting makes the environment install-only; publishing also needs a **publisher**, set up afterwards, and the `marketplacePublish` plan entitlement.
- **Rights come from three system groups** — publishers, reviewers and installers. Publishing and reviewing need group membership **even for a system administrator**.
- **The publisher terms are accepted at approval**, by the reviewer, not when the request is sent.
- **An installed app is updated in place** to a newer listed version; you no longer uninstall and reinstall.

See Packages, Marketplace & the In-App Store. *(Since 1.10)*

Why can't I edit or export a resource that was installed from a package?

Because an installation locks what it created: data-service and reaction-service refuse an update or delete with `403 PACKAGE_RESOURCE_READ_ONLY` for every caller, system administrators included, a plain import over it is blocked with

`TARGET_INSTALLED`, and a plain export refuses it with `403 EXPORT_INSTALLED_RESOURCE`. A few things stay yours on a locked view or page — its published state, display order, audience and access rights, and its name, description and icon — and an update of the package keeps them. To change anything else, **fork** the installation (`POST /v1/installations/{id}/fork`): its resources become editable, but they no longer follow the package, receive no updates, and can never be exported or published. See Install mode and Fork and uninstall. *(Since 1.10)*

Who may export a package, and who may import one?

Since 1.10, exporting is a **developer** action: a system grant of Developer or higher may export any roots, a system grant of None refuses, and otherwise every root needs Developer or higher on the module or entity that owns it. Importing is the opposite — validate, configure, execute, retry and cancel all require a system-level administrator grant, because import writes skip per-resource grants. See Export access and Import access and execution guards. *(Since 1.10)*

Do I need a marketplace connection to browse the in-app store?

No. When the environment is not connected — never connected, or revoked — the catalog, the publishers and the app pages are read anonymously from the marketplace's public catalog. Only **installing** needs a live connection: on **Install**, an environment that is not connected yet asks you to confirm in a *Connect to the marketplace* dialog, connects, and then installs. Connecting that way creates no publisher. The store still needs the marketplace to be enabled by an operator, the ROOT space, and the `install` permission. See The store routes and Installing from the store connects first. *(Since 1.10)*

We are connected to the marketplace — why is sending a version for review refused?

Because connecting does not make you a publisher. A new connection is **install-only**; preparing a publish request is refused with `409 MARKETPLACE_PUBLISHER_REQUIRED` until someone **sets up publishing** (`POST /v1/marketplace/publisher`), choosing a publisher name and, optionally, a store address. Set-up needs a system administrator or a `publish` holder and the `marketplacePublish` plan entitlement; afterwards only a system administrator can rename the publisher or change the store address. Two other refusals are common at this point: `403 MARKETPLACE_PUBLISH_NOT_IN_PLAN` when the plan does not include publishing, and `409 PACKAGE_OWNED` when another marketplace publisher owns the app. See Connecting and setting up publishing. *(Since 1.10)*

Why is *Send for review* disabled, with a "publishing revoked" warning?

Because the marketplace has stopped your publisher from publishing, without ending the connection. The package page shows the date and the marketplace's reason as written, and the connection card shows a *Publishing revoked* tag; new versions and store-page changes cannot be sent until the marketplace restores publishing. Installing and updating apps keep working. Your environment does not decide this: the marketplace refuses the submissions (`502 MARKETPLACE_REFUSED`), so the remedy lies with the marketplace. Available from release 1.10.2. See Publishing revoked by the marketplace. *(Since 1.10)*

I am a system administrator — why can't I send a package for review or approve one?

Because publishing and reviewing come from **group membership alone**. A system administrator holds only `install` and `update` without belonging to a group; `publish` needs membership of `MARKETPLACE_PUBLISHERS` and `review` of `MARKETPLACE_REVIEWERS`, otherwise the answer is `403 MARKETPLACE_PERMISSION_REQUIRED`. Add yourself to the group: membership is cached for 60 seconds, but a refused system administrator re-reads their groups once without the cache, so there is no wait. The **publisher terms** are asked for at approval — the reviewer accepts them in the approve dialog when the answer is `409 MARKETPLACE_TERMS_REQUIRED` — not when the request is sent. See The three marketplace groups and Publish requests and the review inbox. *(Since 1.10)*

How do I move an installed app to a newer version?

Update it **in place** — no uninstall and reinstall. An installed store app with a newer listed version offers **Update** on its installation page and on its store app page; both open an update preview with the release notes between the two versions. Through the API: stage an update session (`POST /v1/installations/{id}/update-sessions`), then run it (`POST /v1/installations/{id}/update`), with a retry route for a failed update. The update routes need an environment administrator in the ROOT space **and** the `update` permission (the installers group). A forked installation cannot be updated (`409 INSTALLATION_FORKED`), and neither can a version that is not newer (`409 UPDATE_NOT_NEWER`). See Updating an installed app in place. *(Since 1.10)*

An install failed — how do I recover without leaving half an app behind?

What a failed install created stays locked, so nothing is left unprotected. Its installation page shows the way forward: **retry** from its import session while that session exists, or **upload the same package file again**, which takes the failed installation over. When a compatible **newer version** is available, **Update** is offered instead: it overwrites what the failed run created and installs the rest, so a failure caused by the package itself no longer needs an uninstall first. Uninstall remains possible from `failed` , and a stopped uninstall offers *Retry*. Before installing at all, the store's install check (`POST …/install-check`) says whether a version would install now, and which names collide. See Installed list and installation page and Store install check. *(Since 1.10)*

How do I tell a publisher that their app failed to install?

From the installation page: a failed store install, or a stopped store update, offers **Report to *publisher***. It first previews exactly what will be sent (the package and version, the error and the failed resources of the run), and you can add a note and choose whether to share your name and e-mail. You need to be an environment administrator in the ROOT space with a live marketplace connection; the publishing entitlement is not needed. If the error changed since the preview, the send is refused with `409 REPORT_STALE` , so preview again. Available from release 1.10.1. See Reporting a failed install or update to its publisher. *(Since 1.10)*

Can I uninstall an installation I have forked?

Yes. Uninstall is allowed from `active` , `failed` , `forked` or `uninstalling` , and needs an environment administrator. On a forked installation the dialog warns that **your own changes to the forked resources are deleted with them**. Either way, the dialog lists what is deleted, the entries users created that go with it and the links from outside the package; **Uninstall** stays disabled until those outside links are removed, and the server refuses a call that skips the dialog with `409 UNINSTALL_BLOCKED_BY_OUTSIDE_LINKS` . A module that still holds entities your users created is kept. See Fork and uninstall. *(Since 1.10)*

How do I find failed or uninstalled installations?

On the **Installed** page (`/admin/packages/installed`), filtered from the toolbar: a **Status** select (*Installed*, *Failed*, *Uninstalled*) and a **Source** select beside the search and sort. Both start empty, so every status and source shows until you pick one; there is no *All* option. Each row shows who installed the package and, for a store install, its publisher. The API equivalent is `GET /v1/imports` with `state` (`installed` , `failed` or `uninstalled`) and `source` . (This page said earlier that 1.10 shows status tabs; the toolbar selects are in the `1.10.0` release.) See Installed list and installation page and Store and Installed polish. *(Since 1.10)*

What do the package history and export records keep, and does deleting a version or an export free its ZIP?

The package page's **history** sidebar records the package's life: created, edited (with what changed), README edited, versions created, deprecated, undeprecated and deleted, and the review events. Every export writes an **export record** (`GET /v1/exports`, listed on the Build page's *Exports* tab), from which the same export can be reopened and started again.

Deleting either **removes the record only — the ZIP stays in storage**, because tenant backups list stored files without copying them. A version can be deleted only if the store never received it: one that was sent for review or is installed somewhere is refused (`VERSION_SENT`, `VERSION_INSTALLED`), and a deleted version's number is never reused. See Export records and Build page and package page. *(Since 1.10)*

Why does an import stop at *Choose a field* for a mapped field?

Because the import could not work out which field on the target parent entity the mapped field copies from, and **the installer now decides**. The import first matches the source field by name; when that fails, validation reports the field as `unmatched` or `unnamed` and raises the blocking error `MAPPED_FIELD_SOURCE_UNDECIDED`, so `canProceed` stays `false`. Pick **Copy from** a field on the target parent — only fields of the same type, with compatible relation targets and, for two-way mappings, the same required setting are offered — or **Don't copy**. Execute and retry check again (`409 MAPPED_FIELD_SOURCE_UNDECIDED`). A store install or update that runs without the wizard returns the field among the choices you must make on the import page. The earlier non-blocking `MAPPED_FIELD_SOURCE_UNRESOLVED` warning is not in 1.10. See Mapped fields: the installer decides and the import API. *(Since 1.10)*

Can a package hold pages that belong to no module?

Yes. A **global page** — one with no module or entity scope — can be a root of a package, alone or beside modules, so a package may hold only pages. A package save naming a page that belongs to a module or entity is `422 PACKAGE_PAGE_NOT_GLOBAL`, and publishing refuses a page that has since moved into one (`PUBLISH_PAGE_NOT_GLOBAL`) or that was itself installed or forked from another package. A statistic-query widget whose data points outside the package installs **unconfigured** instead of failing the install. See Global pages in packages [Build Apps › Pages & Widgets]. *(Since 1.10)*

Do a function's secrets travel in a package?

No. On the development line a packaged function carries its secret **names** only: every value is exported as `null`, and the function's own `OSOS_API_TOKEN` and identity are left out, because the encrypted values are bound to the source environment's keys. The install strips them again, so even an older package cannot plant them. Set the values in each target environment after installing. See Function secrets stay in their environment. *(Since dev)*