XY Sense Developer Documentation

External GraphQL API and webhooks

Contents

Overview

Welcome to the XY Sense Developer Portal

XY Sense is an advanced occupancy intelligence platform that turns real-time, highly accurate, privacy-preserving sensor data into actionable insights for your office buildings. Our sensors capture 100% anonymous XY coordinates in real time to power workplace experience platforms, BMS, BI dashboards and other custom applications.

Our integration offering centres on the XY Sense API: secure access to live and historical occupancy data via Webhooks and/or GraphQL queries, with updates as frequent as every two seconds. Whether you're triggering building automation, syncing space availability, or feeding predictive models, our API is built for fast, flexible integration into your existing tech stack.

This documentation will walk you through authentication, core endpoints, webhook configuration, and common integration patterns — everything you need to get XY Sense occupancy data flowing into your application.

Ready to make your first API call? The Quickstart takes you from API key to first response in four steps. Otherwise, read on for what the platform offers and how the pieces fit together.

The two integration surfaces

XY Sense provides a GraphQL API and Webhooks, which may be used independently or together to integrate your Space, Floor Sightings, Occupancy events, and Analytics data into third-party platforms.

GraphQL API

The XY Sense GraphQL API utilises GraphQL as its query language.

The API can be used to query:

- Space data (ie/ Name and ID of all spaces on a given Floor)
- Sensor data and connection status (ie/ the position and connection status of all sensors on a floor)
- Historical Analytics and Heatmap data
- Latest or recent Occupancy information for Spaces
- Latest or recent Space Sensor readings
- Latest or recent Sightings on a Floor

XY Sense provides an interactive in-browser API Explorer that you can use to build and validate API queries and explore documentation.

Webhooks

Webhooks provide a fast and secure way to reliably stream Occupancy changes, Space Sensor readings, and Floor Sighting updates to other systems.

XY Sense batches and posts updates to any number of nominated HTTPS endpoints while queuing and tracking the delivery status of each change to guarantee at-least-once delivery in the order that the updates occurred.

Integration patterns

For most integrations, use both surfaces together:

1. Use GraphQL to discover the buildings, floors, spaces, and sensors available to your integration.
2. Register webhooks for the event types you need.
3. Store the GraphQL reference data locally and join incoming webhook events to it by ID.
4. Use GraphQL again for cache misses, reconciliation, historical data, and occasional backfills.

Prefer webhooks for ongoing occupancy, floor-sighting, and space-sensor updates. They deliver changes with lower latency and avoid continuous polling. Poll GraphQL only when webhooks cannot provide the history or query shape your workflow needs.

Need	Recommended surface
Receive near-real-time changes	Webhooks
Keep a live downstream system updated	Webhooks
Discover buildings, floors, spaces, and sensors	GraphQL
Add names, hierarchy, floor plans, or sensor associations to events	GraphQL, cached locally
Reconcile missed or uncertain data	GraphQL
Query analytics, heatmaps, or historical windows	GraphQL

Start with GraphQL API to bootstrap your reference data, then configure Webhooks for ongoing changes.

What is covered

- GraphQL query access for buildings, floors, spaces, occupancy, sightings, space sensor readings, analytics, and historical heatmap data.
- Webhook registration and event delivery for occupancy changes, floor sightings, and space sensor readings.
- Generated API reference pages from the external integration schema in this repository.

Production endpoints

- GraphQL endpoint: `https://core-api.app.xysense.io/api/external/query`
- Auth endpoint: `https://core-api.app.xysense.io/api/v1/auth`
- GraphiQL explorer: `https://core-api.app.xysense.io/ui/external/graphiql`

GraphQL API

Use GraphQL to discover the XY Sense portfolio, enrich webhook events with reference data, reconcile downstream state, and query historical or aggregated data.

For ongoing occupancy, floor-sighting, and space-sensor changes, prefer webhooks over continuous polling.

Get started

1. Follow the Quickstart, which takes you from API key to first response in four steps.
2. Read Authentication and the connection requirements for the details.
3. Explore the available data and pick the right surface with Choosing the right API.
4. Choose a GraphQL guide for your integration workflow.
5. Use the API reference for current query, mutation, and type definitions.

When to use GraphQL

- Bootstrap buildings, floors, spaces, sensors, and their relationships.
- Resolve identifiers received in webhook payloads into names and hierarchy.
- Reconcile downstream state after interruptions or processing failures.
- Query analytics, heatmaps, and bounded historical windows.
- Perform occasional backfills where webhook delivery is not appropriate.

Avoid frequent polling for data already available through webhooks. Register a webhook and maintain a local reference-data cache instead.

Endpoint

Send authenticated `POST` requests to:

`https://core-api.app.xysense.io/api/external/query`

GraphQL responses can contain errors even when the HTTP response is successful. See Errors and retries.

Quickstart

This page takes you from nothing to a successful API response. Everything else in these docs builds on these four steps.

Before you start

- Authentication uses **API keys and short-lived bearer tokens**. OAuth and service accounts are not part of the model, so there is nothing to provision on your identity platform.
- The XY Sense features you plan to use (Occupancy API, Analytics API, Webhooks) need to be enabled for your organisation. If a query returns an access-disabled error, your XY Sense representative can enable the feature for you.
- API keys and user permissions are scoped separately. Your user account needs *read/write access to the external API* to use the API Explorer and *manage API access* to create keys, and each **key** also carries its own read or read/write scope. Mutations need write scope on the key itself, in addition to your user account's permissions.

1. Create an API key

In the Admin Console, go to `https://app.xysense.io/settings/apiaccess` and create a key.

- `ReadPublicApi` suits query-only clients. Add `WritePublicApi` when you need mutations (webhook registration, test-data generation).
- The **API secret is shown once** at creation time, so store it securely.

2. Exchange the key for a bearer token

```
curl -s https://core-api.app.xysense.io/api/v1/auth \
  -H "Content-Type: application/json" \
  -d '{ "ApiKey": "<your key>", "ApiSecret": "<your secret>" }'
```

Response:

```
{
  "idToken": "eyJraWQiOiJ...",
  "expiresIn": 900
}
```

Tokens are short-lived, typically 15 minutes (`expiresIn` is in seconds). Cache the token and re-authenticate shortly before it expires, rather than fetching a new token per request. There is no separate refresh endpoint: re-authenticating with your key and secret is the refresh.

3. Run your first query

```
curl -s https://core-api.app.xysense.io/api/external/query \
  -H "Authorization: Bearer <idToken>" \
  -H "Content-Type: application/json" \
  -d '{ "query": "query Buildings { buildings { id name floors { id name } } }" }'
```

Response:


```
{
  "data": {
    "buildings": [
      {
        "id": "0b26f8f3-5d1c-4c6a-9d3e-1a2b3c4d5e6f",
        "name": "1 Example Street",
        "floors": [{ "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204", "name": "Level 12" }]
      }
    ]
  }
}
```


If you get data back, your credentials, token handling, and connectivity are all working.

4. Explore from here

- Open the interactive API Explorer. It authenticates with your user session and is the fastest way to build and validate queries.
- Read Choosing the right API to pick between live occupancy, analytics, and webhooks for your workload.
- If you have no live sensors yet, see Test data without live sensors.
- Prefer Postman? Import the Postman collection for a runnable version of these patterns.

Troubleshooting

Symptom	Where to look
`401` on the GraphQL endpoint	The token has usually expired (they last about 15 minutes). Re-authenticate and retry, and check the `Authorization: Bearer` header.
`401` on the auth endpoint	The key or secret is incorrect. Secrets are shown only at creation, so create a new key if the secret has been lost.
`You are not authorized to access the '...' field`	The API key's scope. Mutations need `WritePublicApi` on the key itself, in addition to your user account's permissions.
An access-disabled error mentioning Analytics or Occupancy API	The feature has not been enabled for your organisation yet. Your XY Sense representative can enable it.

Authentication

Bearer token authentication is used to secure the external API.

> ACCESS MODEL
>
> `ReadPublicApi` is required for queries and GraphiQL access. `WritePublicApi` is required for mutations such as webhook registration and dev-mode test mutations. These scopes are set **on the API key** and are independent of your user account's permissions, so mutations need write scope on the key itself as well as on your user.

The API uses API keys with bearer tokens. OAuth and service accounts (Azure or Google) are not part of the model, so there is nothing to provision on your identity platform.

Token lifetime

Tokens are short-lived: **typically 15 minutes**. Always use the returned `expiresIn` value (seconds) rather than assuming a fixed lifetime.

- Cache the token and reuse it until shortly before expiry, rather than authenticating per request.
- Re-authenticate with your key and secret to get a fresh token. There is no separate refresh endpoint.
- A `401` from the GraphQL endpoint usually means the token has expired.

If you need to inspect the expiry inside the token itself, decode the JWT with your platform tooling or a utility such as `jwt.io`.

Obtain your bearer token

Post a valid API key and secret to:

`https://core-api.app.xysense.io/api/v1/auth`

```
POST https://core-api.app.xysense.io/api/v1/auth
Accept: application/json
Content-Type: application/json

{
  "ApiKey": "<your key>",
  "ApiSecret": "<your secret>"
}
```

Example auth response

```
{
  "idToken": "<your token>",
  "expiresIn": 900
}
```

`expiresIn` is the token lifetime in seconds.

Attach the bearer token

Once you have the token, add it to the `Authorization` header on every GraphQL request.

```
POST https://core-api.app.xysense.io/api/external/query
Authorization: Bearer <idToken>
Accept: application/json
Content-Type: application/json
Accept-Encoding: gzip

{
  "query": "query Buildings { buildings { id name } }"
}
```

Example API request body

```
{
  "query": "query Buildings($nameLike: String) { buildings(nameLike: $nameLike) { id name
} }",
  "variables": { "nameLike": "Example" },
  "operationName": "Buildings"
}
```

`variables` must be a JSON object (not a string). `operationName` is required when the document contains a named operation.

Manage API keys

Manage API keys in the Admin Console:

`https://app.xysense.io/settings/apiaccess`

Key points:

- The API key name is derived from the description and must be unique when created.
- Use `ReadPublicApi` for query-only clients.
- Use `WritePublicApi` only when the integration needs mutations such as webhook registration or dev-mode test traffic.
- Follow least privilege.
- The API secret is shown once at creation time and cannot be recovered later.

Partner accounts

If you access XY Sense as a partner, your user operates across multiple customers, while API keys, API Explorer sessions, and webhooks are all scoped to a single customer.

- Open the customer first (select the customer in the Admin Console, then choose **Open as Customer**), and create API keys or webhooks from within that customer context.
- Explorer deep links shared with you also assume a customer context, so open the customer before following them.

- Repeat key creation per customer when you integrate the same product for several customers.

Operational notes

- Refresh tokens proactively based on the returned `expiresIn` value.
- All API requests must be HTTPS and JSON encoded.
- Compression is useful for larger polling windows and change batches.

Connection requirements

Endpoint

Send GraphQL queries and mutations as HTTPS `POST` requests to:

`https://core-api.app.xysense.io/api/external/query`

Headers

Example headers for a GraphQL query request:

```
Accept: application/json
Content-Type: application/json
Accept-Encoding: br, gzip
Authorization: Bearer <token>
```

All request/response bodies are to be JSON encoded.

```
Accept: application/json
Content-Type: application/json
```

All API requests must include an authorization header. Use the bearer token returned by the authentication flow.

```
Authorization: Bearer <token>
```

If your application supports it, it is strongly suggested that you enable either Brotli or GZip compression to minimise the response size. This is especially useful when polling for occupancy changes, as the result set can grow quite large during active periods of the day.

```
Accept-Encoding: br, gzip
```

TLS policy

The API accepts TLS 1.2 or TLS 1.3 connections using the following cipher suites:

- `TLS-AES-128-GCM-SHA256`
- `TLS-AES-256-GCM-SHA384`
- `TLS-CHACHA20-POLY1305-SHA256`
- `ECDHE-ECDSA-AES128-GCM-SHA256`
- `ECDHE-RSA-AES128-GCM-SHA256`
- `ECDHE-ECDSA-AES128-SHA256`
- `ECDHE-RSA-AES128-SHA256`
- `ECDHE-ECDSA-AES256-GCM-SHA384`

- `ECDHE-RSA-AES256-GCM-SHA384`
- `ECDHE-ECDSA-AES256-SHA384`
- `ECDHE-RSA-AES256-SHA384`

After connecting, handle HTTP failures separately from errors returned in a successful GraphQL response. See Errors.

Postman collection

If you work in Postman, this collection gives you a runnable version of the core API patterns, with test scripts that chain state between requests: the auth token, your sandbox floor, the occupancy polling cursor, and the analytics paging offset are all stored in the environment automatically.

Download and import

Download collection | Download environment template

1. Download both files with the buttons above.
2. In Postman, choose **Import** and add both files.
3. Select the **XY Sense Api User** environment and fill in `coreApiKey` and `coreApiSecret` with your API key. A `ReadPublicApi` key runs every request except **Generate Occupancy Changes**, which needs `WritePublicApi`.

Run order

Run the **Getting started** folder first:

- **Fetch Auth Token** stores the bearer token in the environment. Tokens last about 15 minutes, so re-run this request when you start seeing `401` responses.
- **Fetch Space Data** lists your floors and spaces, and stores the id of a floor named `API Test Floor` for the sandbox flow. If you don't have a floor with that name, the request still shows your portfolio and every other folder works normally.

Then explore the folders in any order:

Folder	What it demonstrates
Occupancy	The full polling loop: generate sandbox events (dev-mode floors only), fetch the latest occupancy, then poll with a moving `from` cursor. Each run of the cursor request advances it, just like a production poller.
Analytics	`occupancyPerHour` with `skip` / `take` paging. Defaults to yesterday (UTC), and re-running the request pages through the results.
Space sensors	Capability and unit discovery, then the latest readings across your sensors.
Sensor health	Active area sensor installations with serial numbers and connection status.
Webhooks	Lists your registered webhooks and their delivery configuration.

Prerequisites

- Each folder needs its feature enabled for your organisation.

- **Generate Occupancy Changes** needs a dev-mode floor and a `WritePublicApi` key. See Test data without live sensors.
- Analytics data becomes available the day after collection, so a brand-new sandbox returns an empty analytics page until it has a day of history.

The collection lives alongside these docs and is updated with them, so re-download it if the schema has moved since you imported it.

Errors

Handle transport failures and GraphQL payload failures separately.

HTTP status handling

Status	Meaning	Recommended handling
`400`	Malformed request or invalid argument.	Fix the request shape or variables before retrying.
`401`	Invalid credentials or expired token.	Refresh the JWT, then retry.
`403`	Caller lacks the required API permission.	Check claims and API key scope.
`404`	Wrong route or resource path.	Verify endpoint and host.
`500`, `503`, `504`	Transient backend or upstream failure.	Retry with exponential backoff starting at 1 second.

GraphQL payload errors

A request can return `200 OK` and still include an `errors` array in the response body.

Example error response

```
{
  "errors": [
    {
      "message": "Field 'FakeField' not found on current context 'XyCoreContext'"
    }
  ],
  "data": {}
}
```

Always inspect the GraphQL response payload, even when the HTTP request succeeds.

Limits, versioning, and support

Rate limits and fair use

The API does not currently enforce hard per-key request quotas. It relies on fair use, and the patterns below will keep your integration fast and reliable:

- **Match your polling rate to how often the data changes.** Occupancy and sightings update at most every ~2 seconds, and polling `occupancyChanges` every 30 seconds with a `from` / `nextFrom` cursor is a good default. Analytics updates once a day, so a daily poll is enough.
- **Prefer webhooks for frequent changes.** If you find yourself polling often to keep up, a webhook will deliver the same data with lower latency and less work on both sides.
- **Prefer fewer, larger queries.** One building- or floor-scoped query is cheaper than many per-space queries, even if it fetches more than you need. Aliased queries in a single request execute sequentially, so batching by scope (floor, building) is faster and avoids timeouts.
- Request windows and retention limits per dataset are listed in Available data.

If you have a workload in mind that sits outside these patterns (large portfolio backfills, or polling faster than every 30 seconds), reach out before you build and we will help you find the most efficient approach.

Schema stability and breaking changes

The external GraphQL schema evolves **additively**: new fields, arguments, types, and enum values are added over time, and existing ones are not removed or repurposed without warning. Build clients that tolerate additions, so that an unknown field or a new enum value is ignored rather than treated as an error.

When a breaking change is unavoidable, XY Sense notifies registered API users ahead of time with a migration window.

The generated API reference always reflects the current production schema.

Getting help

- **Integration questions or unexpected results**: contact your XY Sense representative or reach us via support@xysense.com.
- Include your query (with variables), the full response including any `errors` array, and a timestamp. That is usually enough for us to reproduce.
- **Feature enablement** (Occupancy API, Analytics API, Webhooks) and **dev-mode floors** are provisioned by XY Sense, ask your representative or contact us via support@xysense.com.

Available data

The GraphQL API exposes seven broad categories of XY Sense data. Space and sensor data provide the reference model for your portfolio. Time-series data provides current, recent, or aggregated observations that can be joined to that model using stable identifiers.

Access to occupancy, analytics, floor sightings, historical heatmaps, and space sensor readings depends on the features enabled for your organisation.

Category	What it contains	Typical use
Space data	Buildings, floors, floor spaces, shapes, types, groups, names, and capacities	Build portfolio navigation and resolve IDs from events
Sensor data and status	Area, entry, and space sensors, installations, positions, associations, and connection state	Monitor sensor health and associate readings with spaces
Occupancy time series	Current and recent headcount and occupancy-state changes per floor space	Live availability, automation, and state reconciliation
Space sensor readings	Comfort, air-quality, light, noise, pressure, and sensor-derived occupancy readings	Environmental monitoring and space automation
Floor sightings	Detected XY positions and total headcount for a floor	Live floor views, wayfinding, and spatial applications
Analytics time series	Aggregated occupancy metrics over minute, five-minute, hourly, and daily intervals	Utilisation reporting and trend analysis
Historical heatmaps	Weighted floor-plan cells aggregated over a requested time range	Understand where activity occurred within a floor

Space data

Space data describes the physical hierarchy of your portfolio:

- Each building has one or more floors.
- Each floor has one or more floor spaces, such as desks, rooms, or zones.
- Floor spaces have a shape and space type, and can belong to space groups.
- Shapes are polygons represented by points. Point coordinates are centimetres from the top-left of the floor plan.
- Stable building, floor, and floor-space IDs connect reference data to occupancy, analytics, sightings, heatmaps, and webhook events.

Start with building and floor discovery, then use the generated `Building`, `Floor`, and `FloorSpace` type references for field-level detail.

Sensor data and connection status

Floors can contain area sensor, entry sensor, and space sensor installations. Each installation records its floor position and references the physical sensor.

Associations differ by sensor type:

- Area sensor installations have no direct floor-space association.
- Entry sensor installations expose entries that track forward and backward movement. Entries are associated with floor spaces so movement contributes to live headcount and occupancy state.
- Space sensor installations can be associated with multiple floor spaces. Individual capabilities can be ignored for an installation or floor-space association.

Connection status is available for each sensor. An offline sensor does not always indicate a fault: check the floor's `currentFloorStatus`, because sensors on floors being installed or reviewed may not yet be expected to remain online.

See sensor connection status for practical queries.

Occupancy time series

Occupancy represents the headcount and occupancy state for a floor space at a point in time. Updates are produced approximately every two seconds and can be accessed as:

- The latest occupancy for a floor space.
- Recent occupancy changes across accessible spaces.
- Occupancy changes since a timestamp, optionally filtered by building, floor, or floor space.

Each request can cover up to one hour. The rolling history contains up to the most recent 1,000,000 changes or 24 hours of data.

Occupancy states are:

- `CurrentlyOccupied`: one or more sightings within 30 seconds of `collectedDate`.
- `RecentlyOccupied`: one or more sightings between 30 and 60 seconds before `collectedDate`.
- `NotOccupied`: no sightings within 60 seconds of `collectedDate`.
- `Unknown`: no occupancy information is available for the space.

Use webhooks for ongoing occupancy changes. Use occupancy polling for bootstrap, replay, and reconciliation windows.

Space sensor readings

Space sensor readings contain current and recent observations from space sensors. Depending on the installed model and enabled features, capabilities can include:

- Occupancy
- Temperature
- Humidity
- Ambient and relative light

- Ambient noise
- Barometric pressure
- Carbon dioxide
- Volatile organic compounds
- Particulate matter at 1, 2.5, and 10 micrometres

The exact capabilities available to you depend on the sensor models installed and on which capabilities are enabled per space, so discover them at runtime with the `spaceSensorCapabilities` query rather than assuming a fixed set. Each capability includes its measurement `unit` and the sensor types that support it.

Readings can be fetched by floor space or physical sensor, as the latest values or as changes since a timestamp. A request can cover up to 24 hours, and the rolling history contains up to the most recent 1,000,000 readings or 48 hours of data. Comfort and air-quality readings are typically updated approximately every 10 minutes.

Use webhooks for ongoing reading changes and space sensor readings polling for capability discovery, latest values, history, and reconciliation.

Floor sightings

Floor sightings contain the detected XY positions and total headcount for a floor at a point in time. Coordinates are centimetres from the top-left of the floor plan, and updates are produced approximately every two seconds.

Sightings can be fetched as the latest observations per floor or as changes since a timestamp. Each request can cover up to one hour, and the rolling history contains up to the most recent 1,000,000 updates or 24 hours of data.

Use webhooks for ongoing sighting updates and floor sightings polling for recent snapshots and reconciliation.

Analytics time series

Analytics data contains aggregated occupancy metrics for each floor space, including minimum, maximum, and time-weighted average occupant counts and occupied duration.

Analytics is processed overnight for each building. The previous day's data becomes available at approximately 6 am the following day in the building's local time.

Interval	Maximum range per request	Default filtering
Per minute	1 day	None
Per five minutes	1 day	None
Per hour	31 days	Excludes walk-bys
Per day	365 days	Excludes walk-bys and out-of-hours activity

Requests are scoped to a building and paged with a maximum page size of 100,000 records. The walk-by filter excludes activity with less than one minute of usage over ten minutes. Daily results also apply each building's configured active hours, which default to 7 am to 7 pm, Monday to Friday.

See analytics polling for query and pagination examples.

Historical heatmaps

Historical heatmaps aggregate activity into weighted cells across a floor plan. Each cell contains its centre point, aggregation window, and a relative event weight.

Data is available with an approximate 15-minute delay. A request can aggregate a maximum window of 31 days. See historical heatmap for floor-plan positioning and rendering guidance.

Choosing the right API

Each data surface is designed for a different job. A few minutes here will help you pick the right one for your workload before you build.

Which API for which job

You want	Use	Instead of
Live occupancy pushed to your system	Webhooks (`OccupancyChanges`)	Polling analytics
Live/recent occupancy on demand	`occupancyChanges` / `occupancy` queries (polling guide)	`occupancyPerMinute`, which is next-day analytics
Utilisation reporting over days/weeks/months	Analytics queries (`occupancyPerHour`, `occupancyPerDay`, ...)	Storing and aggregating every webhook event yourself
Where activity happened on a floor plan over a period	Historical heatmap	Reconstructing it from sightings
Raw event-level history beyond the retention windows below	Webhooks, captured and stored on your side	

The API focuses on recent events plus long-term aggregates. Occupancy changes are retained for up to 24 hours (or the most recent 1,000,000 changes), space sensor readings for up to 48 hours, and floor sightings for up to 24 hours. Beyond those windows the API serves aggregates (analytics and heatmaps). If your application needs raw events long-term, capture them via webhooks and store them from day one, since raw events age out of the API after these windows.

Feature enablement

Each data surface is a feature that XY Sense enables for your organisation, and enablement can vary per building and floor. Before building against a surface, confirm with your XY Sense representative that it is enabled where you need it.

Surface	Feature
Occupancy queries and `OccupancyChanges` webhooks	Occupancy API
Floor sighting queries and `FloorSightings` webhooks	Sightings API
Space sensor reading queries and `SpaceSensorReadings` webhooks	Space Sensor Readings API
Analytics queries and historical heatmaps	Analytics API
Test-data generator mutations	Dev mode (per floor)

When a feature is not enabled, queries return a readable error naming the feature and the affected floors (for example `Analytics Api access disabled for floor ...`), and webhook registration for the matching event type is

declined with a similar message. Running the Quickstart queries is a quick way to smoke-test your access.

Live data vs analytics

- **Occupancy changes, sightings, and sensor readings** are near-real-time, with updates as often as every ~2 seconds.
- **Analytics** (`occupancyPerMinute`, `occupancyPerFiveMinutes`, `occupancyPerHour`, `occupancyPerDay`) is processed overnight per building. The previous day's data becomes available at approximately **6 am the following day in the building's local time**. `occupancyPerMinute` is minute-resolution analytics rather than a live feed, and like all analytics it becomes available the next day.
- **Historical heatmaps** run approximately 15 minutes behind live.

Timestamps are UTC

All API timestamps are **UTC in ISO 8601 format** (`2026-06-02T04:17:39.106Z`), and all `from`/`to` arguments are interpreted as UTC. If API results seem to disagree with the dashboard, the timezone of your `from`/`to` arguments is the first thing to check.

Worked example: to request analytics for Tuesday 3 June 2026 in Sydney (AEST, UTC+10), query `from: "2026-06-02T14:00:00Z"` `to: "2026-06-03T14:00:00Z"`.

Field meanings:

- `collectedDate` (occupancy) is when the underlying observation was made. Use this as the event time.
- `occupancyStatusChangeDate` (occupancy) is when the space last changed occupancy *state*. It stays the same while headcount varies within the same state.
- `collectedAt` (space sensor readings) has the same meaning as `collectedDate`. The name differs for historical reasons only.

Missing analytics rows mean zero

Analytics intervals with no occupancy are omitted from results rather than returned as zero rows (a deliberate storage optimisation). When aggregating downstream, treat absent intervals for a known space as zero occupancy rather than missing data.

Related: results reflect the **walk-by filter** (brief pass-through detections are excluded from occupied time) and any configured active-hours filtering. See Available data for the metric definitions.

Paging analytics results

Analytics queries return offset pages (`skip`/`take`, max page size 100,000 records). A year of hourly data for a large portfolio spans multiple pages, so loop until `hasNextPage` is false:

```graphql
query HourlyPage($buildingId: ID, $from: Date!, $to: Date!, $skip: Int, $take: Int) {
  occupancyPerHour(buildingId: $buildingId, from: $from, to: $to, skip: $skip, take:
$take) {
    totalItems
    hasNextPage
    items {
      floorSpaceId
      date
      hour
      avgOccupants
      maxOccupants
      occupiedSeconds
    }
  }
}
```

First page: `{ "skip": 0, "take": 50000 }`. While the response has `"hasNextPage": true`, request the next page with `{ "skip": 50000, "take": 50000 }`, and so on. `totalItems` lets you size the loop up front.

IDs, mapping, and coordinates

Most integrations join XY Sense data to an external system: a workplace app, a BMS, a digital twin, or a BI model. This page covers the identifier and coordinate semantics that make that join reliable.

Identifiers are stable, names are labels

- Every entity (building, floor, floor space, sensor, entry) has a GUID `id`. IDs are stable for the life of the entity, so they are the right key to store in your system.
- Names, descriptions, and space types are display metadata and can be edited at any time. Treat them as labels to show, rather than keys to join on.
- Reference data changes as workplaces change: spaces are added, renamed, recategorised, and retired. Refresh your cached reference data periodically (daily is a good default) and treat an unknown `floorSpaceId` in an event as a signal to refresh.

Mapping to your own system

1. Pull the portfolio once with a discovery query (see Building and floor discovery).
2. Match each XY Sense space to your system's equivalent. A shared naming convention makes the first pass automatic, and names can then change freely afterwards because your stored link is by `id`.
3. Store the XY Sense `id` against your entity and join all events, polling results, and analytics rows on it.

The floor plan coordinate system

Every floor has a `floorPlanUrl` (the floor plan image) and a `scale`.

- Coordinates are expressed in **centimetres from the top-left corner** of the floor plan, with `x` increasing to the right and `y` increasing downward.
- Floor space **shapes** are polygons of points in this coordinate system.
- **Floor sightings** are `x`/`y` points in the same system.
- `scale` is the floor plan's resolution in **centimetres per image pixel**. To place a coordinate on the floor plan image, divide by `scale`:

```
pixelX = x / scale
pixelY = y / scale
```

The same conversion works in reverse (`cm = pixels * scale`) when translating a position picked on the image back to sensor coordinates.

Coordinates in analytics and heatmaps

Historical heatmap cells use the same centimetre coordinate system, so a heatmap overlay and a live sightings overlay can share one rendering path. See Historical heatmap for a worked rendering example.

GraphQL guides

Use these task-oriented examples as implementation starting points for querying and enriching XY Sense data.

Discovery and test data

- Building and floor discovery
- Test data without live sensors

Polling integrations

- Occupancy polling
- Analytics polling
- Floor sightings polling
- Space sensor readings polling

Sensor status

- Sensor connection status

Heatmap

- Historical heatmap

Building and floor discovery

Use this query when you need the portfolio skeleton before doing anything else.

Typical use

- Show a building or floor picker.
- Resolve the `floorId` required by occupancy, sightings, or heatmap queries.
- Fetch `floorPlanUrl` and `scale` for overlays and floor-based visualizations.

Query

Open in API Explorer

Example query

```
query BuildingAndFloorIndex {
  buildings {
    id
    name
    floors(isLive: true) {
      id
      name
      floorPlanUrl
      scale
    }
  }
}
```

Example response

```
{
  "data": {
    "buildings": [
      {
        "id": "0b26f8f3-5d1c-4c6a-9d3e-1a2b3c4d5e6f",
        "name": "1 Example Street",
        "floors": [
          {
            "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
            "name": "Level 12",
            "floorPlanUrl": "https://cdn.xysense.io/floorplans/level-12.png",
            "scale": 1.8
          }
        ]
      }
    ]
  }
}
```

IDs are GUIDs. Some other guides shorten them to readable slugs (such as `desk-ne-101`) to keep examples compact.

Test data without live sensors

You do not need live sensors (or live occupants) to build an integration. XY Sense can provide a sandbox floor in **dev mode**, where you generate synthetic occupancy and sighting traffic on demand. Generated events flow through the same pipeline as real sensor data: they appear in polling queries *and* are delivered to registered webhooks.

> **GETTING A DEV-MODE FLOOR**
>
> Dev mode is enabled per floor by XY Sense. Ask your XY Sense representative to set up a sandbox floor (or enable dev mode on a test floor) for your integration.

Requirements

Generator mutations require **write access on the API key** (`WritePublicApi`), not just on your user account. If you get `You are not authorized to access the 'generateOccupancyEvents' field`, the key's scope in the Admin Console is the first thing to check.

A brand-new sandbox has **no data until you generate some**. If your first `occupancyChanges` poll returns empty results, generate events first.

Generator mutations

Mutation	What it does
`generateOccupancyEvents`	Toggles occupancy on randomly selected spaces on the floor. `includeHeadcountChanges` adds headcount-only events.
`toggleOccupancyStatus`	Toggles a single space between occupied and not occupied.
`updateOccupancyStatus`	Sets a specific space to a specific status, source, headcount, and collection time.
`generateFloorSightings`	Emits random sightings for a floor, optionally continuously for `durationSecs`.
`addFloorSightings`	Sets exact sighting coordinates for a floor, useful for deterministic floor-plan tests.

Generate occupancy events

Open in API Explorer

```
mutation generateOccupancyEvents {
  generateOccupancyEvents(floorId: "<floor guid>", count: 1, includeHeadcountChanges:
true) {
    floorSpaceId
    headcount
    occupancyStatus
    previousOccupancyStatus
    collectedDate
  }
}
```

Example response


```
{
  "data": {
    "generateOccupancyEvents": [
      {
        "floorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1",
        "headcount": 0,
        "occupancyStatus": "NotOccupied",
        "previousOccupancyStatus": "RecentlyOccupied",
        "collectedDate": "2026-06-02T04:17:39.1066221Z"
      }
    ]
  }
}
```


Verify the data arrived

- **Polling**: run `occupancyChanges(includeLatest: true)` for the floor and confirm the generated events appear. See Occupancy polling.
- **Webhooks**: register a webhook first, then generate events and watch your receiver. There is a full walkthrough in Webhook test events.
- **Postman**: the Postman collection includes the generate mutation and the polling loop as runnable requests.

Notes

- Generated events are indistinguishable from real events downstream, so you can build your parsing, cursor handling, and idempotency against them with confidence.
- Occupancy states still follow the normal state model (`CurrentlyOccupied` → `RecentlyOccupied` → `NotOccupied` over 30/60-second thresholds), so allow a minute if you are asserting on state transitions.
- Analytics is processed overnight like everything else, so it cannot be generated on demand. Test analytics queries against a floor that had (real or generated) activity the previous day.

Occupancy polling

Use occupancy polling when you need pull-based updates, replay windows, or downstream reconciliation.

REQUIRES THE OCCUPANCY API FEATURE

The Occupancy API feature is enabled by XY Sense per organisation and can vary per building and floor. If your queries return an access-disabled error, your XY Sense representative can enable it. See Feature enablement.

Fetch space information

Open in API Explorer

Example query

```
query OccupancyFloorSpaces($floorId: ID!) {
  floorSpaces(floorId: $floorId) {
    id
    name
    capacity
    floor {
      id
      name
    }
    spaceType {
      id
      name
    }
  }
}
```

Example variables

```
{
  "floorId": "floor-l12"
}
```

Example response


```
{
  "data": {
    "floorSpaces": [
      {
        "id": "desk-ne-101",
        "name": "Desk NE-101",
        "capacity": 1,
        "floor": {
          "id": "floor-l12",
          "name": "Level 12"
        },
        "spaceType": {
          "id": "space-type-desk",
          "name": "Desk"
        }
      }
    ]
  }
}
```


Fetch latest occupancy

Open in API Explorer

Example query

```
query LatestOccupancy($floorId: ID!) {
  floorSpaces(floorId: $floorId) {
    id
    name
    latestOccupancy {
      floorSpaceId
      headcount
      occupancyStatus
      previousOccupancyStatus
      collectedDate
      occupancyStatusChangeDate
    }
  }
}
```

Example variables


```
{
  "floorId": "floor-l12"
}
```

Example response


```
{
  "data": {
    "floorSpaces": [
      {
        "id": "desk-ne-101",
        "name": "Desk NE-101",
        "latestOccupancy": {
          "floorSpaceId": "desk-ne-101",
          "headcount": 1,
          "occupancyStatus": "CurrentlyOccupied",
          "previousOccupancyStatus": "RecentlyOccupied",
          "collectedDate": "2026-06-02T04:17:39.1066221Z",
          "occupancyStatusChangeDate": "2026-06-02T04:17:39.1066221Z"
        }
      }
    ]
  }
}
```


Fetch occupancy changes

Open in API Explorer

Example query

```
query OccupancyChanges($floorIds: [ID!], $from: Date!, $to: Date!) {
  occupancyChanges(floorIds: $floorIds, from: $from, to: $to, includeLatest: false,
includeHeadcountChanges: true) {
    from
    to
    nextFrom
    data {
      floorSpaceId
      headcount
      occupancyStatus
      previousOccupancyStatus
      collectedDate
      occupancyStatusChangeDate
    }
  }
}
```

Example variables


```
{
  "floorIds": [
    "floor-l12"
  ],
  "from": "2026-06-02T04:00:00Z",
  "to": "2026-06-02T05:00:00Z"
}
```

Example response


```
{
  "data": {
    "occupancyChanges": {
      "from": "2026-06-02T04:00:00Z",
      "to": "2026-06-02T05:00:00Z",
      "nextFrom": "2026-06-02T04:17:39.1066221Z",
      "data": [
        {
          "floorSpaceId": "desk-ne-101",
          "headcount": 0,
          "occupancyStatus": "NotOccupied",
          "previousOccupancyStatus": "RecentlyOccupied",
          "collectedDate": "2026-06-02T04:17:39.1066221Z",
          "occupancyStatusChangeDate": "2026-06-02T04:17:39.1066221Z"
        }
      ]
    }
  }
}
```


Filter occupancy changes

Use explicit floor-space filters when you only care about a subset of desks, rooms, or zones.

Open in API Explorer

Example query

```
query FilteredOccupancyChanges($floorSpaceIds: [ID!], $from: Date!, $to: Date!) {
  occupancyChanges(floorSpaceIds: $floorSpaceIds, from: $from, to: $to, includeLatest:
false, includeHeadcountChanges: true) {
    nextFrom
    data {
      floorSpaceId
      headcount
      occupancyStatus
      collectedDate
    }
  }
}
```

Example variables

```
{
  "floorSpaceIds": [
    "desk-ne-101",
    "room-12a"
  ],
  "from": "2026-06-02T04:00:00Z",
  "to": "2026-06-02T05:00:00Z"
}
```

Example response

```
{
  "data": {
    "occupancyChanges": {
      "nextFrom": "2026-06-02T04:52:10.5411902Z",
      "data": [
        {
          "floorSpaceId": "desk-ne-101",
          "headcount": 1,
          "occupancyStatus": "CurrentlyOccupied",
          "collectedDate": "2026-06-02T04:52:10.5411902Z"
        },
        {
          "floorSpaceId": "room-12a",
          "headcount": 4,
          "occupancyStatus": "CurrentlyOccupied",
          "collectedDate": "2026-06-02T04:49:58.1178410Z"
        }
      ]
    }
  }
}
```

Notes

- The maximum request window for occupancy changes is one hour.
- The rolling retention is the most recent 1,000,000 events or 24 hours.
- Persist `nextFrom` and use it for the next incremental request.

Analytics polling

Use analytics polling when you need aggregated utilisation metrics instead of raw occupancy changes.

> **REQUIRES THE ANALYTICS API FEATURE**
>
> The Analytics API feature is enabled by XY Sense per organisation and can vary per building and floor. If your queries return an access-disabled error, your XY Sense representative can enable it. See Feature enablement.

Fetch building information

Open in API Explorer

Example query

```
query AnalyticsBuildings {
  buildings {
    id
    name
    analyticsAvailableTo
    floors(isLive: true) {
      id
      name
      scale
      floorPlanUrl
    }
  }
}
```

Example response

```
{
  "data": {
    "buildings": [
      {
        "id": "bldg-melb-hq",
        "name": "Melbourne HQ",
        "analyticsAvailableTo": "2026-06-02T00:00:00Z",
        "floors": [
          {
            "id": "floor-l12",
            "name": "Level 12",
            "scale": 0.065,
            "floorPlanUrl": "https://cdn.xysense.io/floorplans/floor-l12.png"
          }
        ]
      }
    ]
  }
}
```

Fetch analytics data

Analytics time series are available in multiple granularities. Use the interval query that matches your reporting need. This example uses hourly aggregation.

Open in API Explorer

Example query

```
query OccupancyPerHour($floorId: ID!, $from: Date!, $to: Date!) {
  occupancyPerHour(floorId: $floorId, from: $from, to: $to, take: 1000) {
    items {
      floorSpaceId
      periodStartUtc
      periodStartLocal
      minOccupants
      maxOccupants
      avgOccupants
      occupiedMinutes
      occupiedHours
    }
    hasNextPage
    totalItems
  }
}
```

Example variables

```
{
  "floorId": "floor-l12",
  "from": "2026-06-01T00:00:00Z",
  "to": "2026-06-02T00:00:00Z"
}
```

Example response


```
{
  "data": {
    "occupancyPerHour": {
      "items": [
        {
          "floorSpaceId": "desk-ne-101",
          "periodStartUtc": "2026-06-01T01:00:00Z",
          "periodStartLocal": "2026-06-01T11:00:00+10:00",
          "minOccupants": 0,
          "maxOccupants": 1,
          "avgOccupants": 0.63,
          "occupiedMinutes": 38,
          "occupiedHours": 0.63
        }
      ],
      "hasNextPage": false,
      "totalItems": 1
    }
  }
}
```


Fetch space information

Pull floor-space metadata so the analytics rows can be joined back to named spaces, types, and capacities.

Open in API Explorer

Example query

```
query AnalyticsFloorSpaces($floorId: ID!) {
  floorSpaces(floorId: $floorId) {
    id
    name
    capacity
    spaceType {
      id
      name
      globalSpaceCategory {
        id
        name
      }
    }
  }
}
```

Example variables


```
{
  "floorId": "floor-l12"
}
```

Example response


```
{
  "data": {
    "floorSpaces": [
      {
        "id": "desk-ne-101",
        "name": "Desk NE-101",
        "capacity": 1,
        "spaceType": {
          "id": "space-type-desk",
          "name": "Desk",
          "globalSpaceCategory": {
            "id": "category-workstation",
            "name": "Workstation"
          }
        }
      }
    ]
  }
}
```


Notes

- Analytics queries allow up to 31 days per request.
- Analytics are not available until approximately 6am the following day in the building's timezone.
- Pick `occupancyPerMinute`, `occupancyPerFiveMinutes`, `occupancyPerHour`, or `occupancyPerDay` based on the resolution you need.

Floor sightings polling

Use floor sightings polling when you need XY coordinates for observed people on a floor.

> **REQUIRES THE SIGHTINGS API FEATURE**
>
> The Sightings API feature is enabled by XY Sense per organisation and can vary per building and floor. If your queries return an access-disabled error, your XY Sense representative can enable it. See Feature enablement.

Fetch floor information

Open in API Explorer

Example query

```
query SightingFloor($floorId: ID!) {
  floor(id: $floorId) {
    id
    name
    floorPlanUrl
    scale
  }
}
```

Example variables

```
{
  "floorId": "floor-l12"
}
```

Example response

```
{
  "data": {
    "floor": {
      "id": "floor-l12",
      "name": "Level 12",
      "floorPlanUrl": "https://cdn.xysense.io/floorplans/floor-l12.png",
      "scale": 1.8
    }
  }
}
```

Fetch recent floor sightings

If you omit `from`, the API returns the last minute for each matching floor.

Open in API Explorer

Example query

```
query RecentFloorSightings($floorIds: [ID!]) {
  floorSightingChanges(floorIds: $floorIds, includeLatest: true) {
    nextFrom
    data {
      floorId
      headcount
      collectedDate
      sightings
    }
  }
}
```

Example variables

```
{
  "floorIds": [
    "floor-l12"
  ]
}
```

Example response

```
{
  "data": {
    "floorSightingChanges": {
      "nextFrom": "2026-06-02T07:36:11.964156Z",
      "data": [
        {
          "floorId": "floor-l12",
          "headcount": 3,
          "collectedDate": "2026-06-02T07:36:11.964156Z",
          "sightings": [
            { "x": 1842, "y": 971 },
            { "x": 2510, "y": 1164 },
            { "x": 2732, "y": 1189 }
          ]
        }
      ]
    }
  }
}
```

Fetch floor sightings updates

Open in API Explorer

Example query

```
query FloorSightingUpdates($floorIds: [ID!], $from: Date!, $to: Date!) {
  floorSightingChanges(floorIds: $floorIds, from: $from, to: $to, includeLatest: false) {
    from
    to
    nextFrom
    data {
      floorId
      headcount
      collectedDate
      sightings
    }
  }
}
```

Example variables

```
{
  "floorIds": [
    "floor-l12"
  ],
  "from": "2026-06-02T07:30:00Z",
  "to": "2026-06-02T07:40:00Z"
}
```

Example response

```
{
  "data": {
    "floorSightingChanges": {
      "from": "2026-06-02T07:30:00Z",
      "to": "2026-06-02T07:40:00Z",
      "nextFrom": "2026-06-02T07:36:11.964156Z",
      "data": [
        {
          "floorId": "floor-l12",
          "headcount": 2,
          "collectedDate": "2026-06-02T07:36:11.964156Z",
          "sightings": [
            { "x": 1842, "y": 971 },
            { "x": 2510, "y": 1164 }
          ]
        }
      ]
    }
  }
}
```

Notes

- Floor-sighting payload coordinates are in centimeters from the top-left of the floor plan.

- The floor `scale` field is the plan's resolution in centimetres per image pixel, so divide coordinates by `scale` to get image pixels. See IDs, mapping, and coordinates.
- Persist `nextFrom` for incremental polling.

Space sensor readings polling

Use this flow for comfort, air quality, and sensor-derived occupancy measurements from space sensors.

> **REQUIRES THE SPACE SENSOR READINGS API FEATURE**
>
> The Space Sensor Readings API feature is enabled by XY Sense per organisation and can vary per building and floor. If your queries return an access-disabled error, your XY Sense representative can enable it. See Feature enablement.

Fetch sensor capability information

Open in API Explorer

Example query

```
query SpaceSensorCapabilities {
  spaceSensorCapabilities {
    id
    name
    description
    unit
  }
}
```

Example response

```
{
  "data": {
    "spaceSensorCapabilities": [
      {
        "id": "9bbebc44-654a-4496-a055-5a892008a94d",
        "name": "Temperature",
        "description": "Ambient temperature",
        "unit": "degrees Celsius"
      },
      {
        "id": "47c873e6-3c60-41f8-9d17-96e5639b3bde",
        "name": "Humidity",
        "description": "Relative humidity",
        "unit": "percent"
      }
    ]
  }
}
```

Fetch latest space sensor readings

If you omit `from`, the API returns the latest readings for each matching space sensor.

Open in API Explorer

Example query

```
query LatestSpaceSensorReadings($floorIds: [ID!]) {
  spaceSensorReadings(floorIds: $floorIds, includeLatest: true, comfortOnly: false) {
    nextFrom
    data {
      spaceSensorId
      capabilityName
      value
      collectedAt
    }
  }
}
```

Example variables

```
{
  "floorIds": [
    "floor-l12"
  ]
}
```

Example response

```
{
  "data": {
    "spaceSensorReadings": {
      "nextFrom": "2026-06-02T07:36:11.964156Z",
      "data": [
        {
          "spaceSensorId": "sensor-01",
          "capabilityName": "Temperature",
          "value": "22.4",
          "collectedAt": "2026-06-02T07:36:11.964156Z"
        },
        {
          "spaceSensorId": "sensor-01",
          "capabilityName": "Humidity",
          "value": "48.2",
          "collectedAt": "2026-06-02T07:36:11.964156Z"
        }
      ]
    }
  }
}
```

Fetch space sensor reading changes

Open in API Explorer

Example query

```
query SpaceSensorReadingChanges($floorIds: [ID!], $from: Date!, $to: Date!) {
  spaceSensorReadings(floorIds: $floorIds, from: $from, to: $to, includeLatest: false,
comfortOnly: false) {
    from
    to
    nextFrom
    data {
      spaceSensorId
      capabilityName
      value
      collectedAt
    }
  }
}
```

Example variables

```
{
  "floorIds": [
    "floor-l12"
  ],
  "from": "2026-06-02T07:00:00Z",
  "to": "2026-06-02T08:00:00Z"
}
```

Example response

```
{
  "data": {
    "spaceSensorReadings": {
      "from": "2026-06-02T06:00:00Z",
      "to": "2026-06-02T07:00:00Z",
      "nextFrom": "2026-06-02T06:50:12.331245Z",
      "data": [
        {
          "spaceSensorId": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3",
          "capabilityName": "Temperature",
          "value": "22.4",
          "collectedAt": "2026-06-02T06:50:12.331245Z"
        },
        {
          "spaceSensorId": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3",
          "capabilityName": "Humidity",
          "value": "48.2",
          "collectedAt": "2026-06-02T06:50:12.331245Z"
        }
      ]
    }
  }
}
```

Fetch sensor associations by space or sensor

Use floor-space metadata when you need to render or label which spaces a sensor installation serves.

Open in API Explorer

Example query

```graphql
query SpaceSensorAssociations($floorId: ID!) {
  floor(id: $floorId) {
    id
    name
    floorSpaces {
      id
      name
      spaceSensorInstallations {
        id
        name
        spaceSensorId
      }
    }
    spaceSensorInstallations {
      id
      name
      spaceSensorId
      floorSpaces {
        id
        name
      }
      ignoredCapabilities {
        id
        name
      }
    }
  }
}
```

Example variables

```json
{
  "floorId": "floor-l12"
}
```

Example response


```
{
  "data": {
    "floor": {
      "id": "floor-l12",
      "name": "Level 12",
      "floorSpaces": [
        {
          "id": "room-12a",
          "name": "Room 12A",
          "spaceSensorInstallations": [
            {
              "id": "ssi-1",
              "name": "Room 12A Sensor",
              "spaceSensorId": "sensor-01"
            }
          ]
        }
      ],
      "spaceSensorInstallations": [
        {
          "id": "ssi-1",
          "name": "Room 12A Sensor",
          "spaceSensorId": "sensor-01",
          "floorSpaces": [
            {
              "id": "room-12a",
              "name": "Room 12A"
            }
          ],
          "ignoredCapabilities": [
            {
              "id": "47c873e6-3c60-41f8-9d17-96e5639b3bde",
              "name": "Humidity"
            }
          ]
        }
      ]
    }
  }
}
```


Filter space sensor reading changes

Open in API Explorer

Example query

```
query FilteredSpaceSensorReadingChanges($spaceSensorIds: [ID!], $from: Date!, $to: Date!)
{
  spaceSensorReadings(spaceSensorIds: $spaceSensorIds, from: $from, to: $to,
includeLatest: false) {
    nextFrom
    data {
      spaceSensorId
      capabilityName
      value
      collectedAt
    }
  }
}
```

Example variables

```
{
  "spaceSensorIds": [
    "sensor-01",
    "sensor-02"
  ],
  "from": "2026-06-02T07:00:00Z",
  "to": "2026-06-02T08:00:00Z"
}
```

Example response

```
{
  "data": {
    "spaceSensorReadings": {
      "nextFrom": "2026-06-02T06:50:12.331245Z",
      "data": [
        {
          "spaceSensorId": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3",
          "capabilityName": "CarbonDioxide",
          "value": "641",
          "collectedAt": "2026-06-02T06:50:12.331245Z"
        }
      ]
    }
  }
}
```

Notes

- The maximum request window is 24 hours.
- The documented rolling retention is the most recent 1,000,000 readings or 48 hours.
- Use `comfortOnly: true` when you want comfort and air-quality readings without occupancy-style sensor output.

Sensor connection status

Use this flow to check whether sensors are online and to understand what spaces each sensor affects.

Fetch sensor connection information

Open in API Explorer

Example query

```
query FloorSensorStatus($floorId: ID!) {
  floor(id: $floorId) {
    id
    name
    currentFloorStatus {
      status
      startDate
    }
    entrySensorInstallations {
      id
      name
      entrySensor {
        id
        externalId
        isVirtual
        connection {
          isOnline
          isOnlineChangeDate
          source
        }
      }
    }
    spaceSensorInstallations {
      id
      name
      spaceSensor {
        id
        externalId
        type {
          id
          name
        }
        connection {
          isOnline
          isOnlineChangeDate
          source
        }
      }
    }
  }
}
```

Example variables

```
{
  "floorId": "floor-l12"
}
```

Example response

```
{
  "data": {
    "floor": {
      "id": "floor-l12",
      "name": "Level 12",
      "currentFloorStatus": {
        "status": "Live",
        "startDate": "2026-05-14T00:00:00Z"
      },
      "entrySensorInstallations": [
        {
          "id": "esi-1",
          "name": "North Entry",
          "entrySensor": {
            "id": "entry-sensor-01",
            "externalId": "ENT-01",
            "isVirtual": false,
            "connection": {
              "isOnline": true,
              "isOnlineChangeDate": "2026-06-02T06:58:00Z",
              "source": "Heartbeat"
            }
          }
        }
      ],
      "spaceSensorInstallations": [
        {
          "id": "ssi-1",
          "name": "Room 12A Sensor",
          "spaceSensor": {
            "id": "space-sensor-01",
            "externalId": "SS-01",
            "type": {
              "id": "7187aa0d-6cc9-4c5d-8661-3861d745e664",
              "name": "Comfort"
            },
            "connection": {
              "isOnline": false,
              "isOnlineChangeDate": "2026-06-02T06:50:00Z",
              "source": "Heartbeat"
            }
          }
        }
      ]
    }
  }
}
```

Fetch floor spaces associated with an entry sensor

Open in API Explorer

Example query

```
query EntrySensorFloorSpaces($installationId: ID!) {
  entrySensorInstallation(id: $installationId) {
    id
    name
    entries {
      id
      name
      externalId
      entryFloorSpaces {
        invertEntryDirection
        floorSpace {
          id
          name
        }
      }
    }
  }
}
```

Example variables

```
{
  "installationId": "esi-1"
}
```

Example response


```
{
  "data": {
    "entrySensorInstallation": {
      "id": "esi-1",
      "name": "North Entry",
      "entries": [
        {
          "id": "entry-1",
          "name": "Inbound lane",
          "externalId": "ENTRY-IN-01",
          "entryFloorSpaces": [
            {
              "invertEntryDirection": false,
              "floorSpace": {
                "id": "room-12a",
                "name": "Room 12A"
              }
            }
          ]
        }
      ]
    }
  }
}
```


Fetch floor spaces associated with a space sensor

Open in API Explorer

Example query

```
query SpaceSensorFloorSpaces($installationId: ID!) {
  spaceSensorInstallation(id: $installationId) {
    id
    name
    ignoredCapabilities {
      id
      name
    }
    spaceSensor {
      id
      externalId
      type {
        id
        name
      }
    }
    spaceSensorInstallationFloorSpaces {
      ignoredCapabilities {
        id
        name
      }
      floorSpace {
        id
        name
      }
    }
  }
}
```

Example variables

```
{
  "installationId": "ssi-1"
}
```

Example response


```
{
  "data": {
    "spaceSensorInstallation": {
      "id": "ssi-1",
      "name": "Room 12A Sensor",
      "ignoredCapabilities": [
        {
          "id": "47c873e6-3c60-41f8-9d17-96e5639b3bde",
          "name": "Humidity"
        }
      ],
      "spaceSensor": {
        "id": "space-sensor-01",
        "externalId": "SS-01",
        "type": {
          "id": "7187aa0d-6cc9-4c5d-8661-3861d745e664",
          "name": "Comfort"
        }
      },
      "spaceSensorInstallationFloorSpaces": [
        {
          "ignoredCapabilities": [
            {
              "id": "47c873e6-3c60-41f8-9d17-96e5639b3bde",
              "name": "Humidity"
            }
          ],
          "floorSpace": {
            "id": "room-12a",
            "name": "Room 12A"
          }
        }
      ]
    }
  }
}
```


Notes

- An offline sensor is not always an incident. Check the floor status first.
- Entry sensors affect occupancy through `entries` and their associated `entryFloorSpaces`.
- Space sensors can be associated with more than one floor space and can ignore selected capabilities at install or per-space level.

Historical heatmap

Use historical heatmap data when you want aggregated movement density over a selected time window.

> **REQUIRES THE ANALYTICS API FEATURE**
>
> The Analytics API feature is enabled by XY Sense per organisation and can vary per building and floor. If your queries return an access-disabled error, your XY Sense representative can enable it. See Feature enablement.

Fetch floor plan information and heatmap data

Open in API Explorer

Example query

```
query FloorHeatmap($floorId: ID!, $from: Date!, $to: Date!, $resolutionCms: Int) {
  floor(id: $floorId) {
    id
    name
    floorPlanUrl
    scale
  }
  historicalHeatmap(floorId: $floorId, from: $from, to: $to, resolutionCms:
$resolutionCms) {
    from
    to
    center
    weight
  }
}
```

Example variables

```
{
  "floorId": "floor-l12",
  "from": "2026-06-01T00:00:00Z",
  "to": "2026-06-01T23:59:59Z",
  "resolutionCms": 100
}
```

Example response


```
{
  "data": {
    "floor": {
      "id": "floor-l12",
      "name": "Level 12",
      "floorPlanUrl": "https://cdn.xysense.io/floorplans/floor-l12.png",
      "scale": 1.8
    },
    "historicalHeatmap": [
      {
        "from": "2026-06-01T00:00:00Z",
        "to": "2026-06-01T23:59:59Z",
        "center": {
          "x": 1842,
          "y": 971
        },
        "weight": 74
      },
      {
        "from": "2026-06-01T00:00:00Z",
        "to": "2026-06-01T23:59:59Z",
        "center": {
          "x": 2510,
          "y": 1164
        },
        "weight": 41
      }
    ]
  }
}
```


Plot heatmap cells on the floor plan

The essential conversion is from centimetres in the API response to pixels in the displayed floor plan. `scale` is the plan's resolution in centimetres per image pixel, so divide by it (see IDs, mapping, and coordinates):

```
const heatmapPoints = response.data.historicalHeatmap.map((cell) => ({
  x: cell.center.x / response.data.floor.scale,
  y: cell.center.y / response.data.floor.scale,
  value: cell.weight,
}))
```

Display heatmap data

- Load the floor plan from `floor.floorPlanUrl`.
- Use `floor.scale` to convert API coordinates into rendered image pixels.
- `weight` is an integer representing accumulated activity in the cell. Normalise by the maximum weight in the response when your heatmap library expects a 0 to 1 intensity.
- Keep the floor plan and heatmap request scoped to the same floor and time window.

Notes

- The maximum heatmap request window is 31 days.
- `resolutionCms` sets the cell size of the heatmap grid in centimetres. Larger values return fewer, coarser cells.
- Historical heatmap availability is delayed by roughly 15 minutes.

Webhooks

Use webhooks for near-real-time delivery of change events.

For most integrations, webhooks should be the primary source of ongoing changes. Use GraphQL to bootstrap and refresh the building, floor, space, and sensor reference data needed to interpret each event. See Resolve webhook events with GraphQL.

Supported event types

- `OccupancyChanges`
- `FloorSightings`
- `SpaceSensorReadings`

Delivery model

- Events are queued per webhook, preserving order.
- A `2xx` response acknowledges the batch.
- Retries back off up to 10 minutes for up to 24 hours.
- Webhooks focus on recent changes and support up to 1,000,000 queued events.

Key requirements

- Endpoints must be reachable over HTTPS.
- Treat delivery as at-least-once and idempotently process events downstream.
- Expect batched payloads.
- Preserve the raw request body if you validate signatures.

Scope and volume

- **A webhook receives events for your entire organisation**: every building and floor you have access to. Webhooks are not filtered per building or floor, so filter downstream by `floorSpaceId` / `floorId` where a receiver only needs one site.
- **Plan for volume.** A large portfolio can produce on the order of 1–2 million occupancy events per weekday (tens of events per second at peak). Sizing your receiver for this up front, and acknowledging batches quickly while processing asynchronously, keeps it comfortably ahead of the queue. Over 90% of occupancy events are headcount-only changes, so if you only need occupied/vacant state the `ExcludeHeadcountChanges` config option below reduces volume dramatically.

Batching semantics

Payloads are JSON **arrays of events**. Read every element:

- A single batch can contain **multiple events for the same space** (e.g. two headcount changes in the same delivery window), so process the array in order rather than keeping the first match per space.
- Batch size and flush delay are configurable per webhook (`maxBatchSize`: 100–1000 events, `maxBatchDelayMs`: 1 second–5 minutes). A batch is sent when either limit is reached.
- Events are queued per webhook in order; a `2xx` response acknowledges the whole batch.

Configuration options

`addWebhook` / `updateWebhook` accept `configOptions` as key/value pairs:

Key	Default	Effect
`ExcludeHeadcountChanges`	`false`	Occupancy webhooks only: drop events where only the headcount changed.
`GroupByLocation`	`false`	Group batches by location.
`OnlyComfortReadings`	`false`	Space-sensor webhooks only: deliver only comfort readings.
`EnableBatching`	`true`	Disable to receive one event per request (not recommended at volume).

Example: `"configOptions": [{ "key": "ExcludeHeadcountChanges", "value": "true" }]`

Manage webhooks over their lifecycle

Everything you can do in the Admin Console is also available via GraphQL (requires `WritePublicApi` on the key):

- `webhooks` / `webhook(id:)`: list and inspect registered webhooks, including delivery metrics.
- `updateWebhook`: change the endpoint, batching, headers, secret, or config. Set `enabled: false` to pause delivery.
- `deleteWebhook`: remove a webhook.
- `resetWebhookConsumer`: skip to the latest queued event. Use this to recover a receiver that has fallen a long way behind: reset, then backfill the gap via GraphQL polling queries.
- `resetWebhookMetrics`: clear delivery metrics.

Service limits

- Batch size range: `100` to `1000`
- Batch delay range: `1 second` to `5 minutes`
- Retry behavior: failed deliveries are retried with backoff (up to 10 minutes between attempts) for up to `24 hours`. Events that exhaust the retry window are dropped from the webhook queue, and can be recovered afterwards with the corresponding GraphQL history query, subject to its retention window.
- Queue depth: up to `1,000,000` events per webhook.

Example payloads

Occupancy webhook payload


```
[
  {
    "floorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1",
    "headcount": 2,
    "occupancyStatus": "CurrentlyOccupied",
    "previousOccupancyStatus": "RecentlyOccupied",
    "collectedDate": "2026-06-02T04:17:39.1066221Z",
    "occupancyStatusChangeDate": "2026-06-02T04:17:39.1066221Z"
  }
]
```


Floor sightings webhook payload


```
[
  {
    "floorId": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
    "headcount": 3,
    "collectedDate": "2026-06-02T07:36:11.964156Z",
    "sightings": [
      { "x": 1842, "y": 971 },
      { "x": 2510, "y": 1164 },
      { "x": 2732, "y": 1189 }
    ]
  }
]
```


Space sensor readings webhook payload


```
[
  {
    "spaceSensorId": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3",
    "capabilityName": "Temperature",
    "value": "22.4",
    "collectedAt": "2026-06-02T07:36:11.964156Z"
  },
  {
    "spaceSensorId": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3",
    "capabilityName": "Humidity",
    "value": "48.2",
    "collectedAt": "2026-06-02T07:36:11.964156Z"
  }
]
```


Authenticate webhook requests

You can protect downstream receivers with either custom headers, request signatures, or both.

Custom headers

Configure one or more fixed headers such as an API token and validate them on receipt.

Signature model

If a signature secret is configured, each request includes `X-XY-Signature` containing a timestamp and HMAC SHA-256 signature over:

```
<timestamp>.<raw json payload>
```

Validate signed requests

The signature scheme, precisely:

- `t` is the send time as **Unix epoch seconds**.
- The signed content is the UTF-8 bytes of `{t}.` followed by the **raw, decompressed request body**. Decompress first if you enabled compression, and use the body bytes as received rather than re-serialized JSON.
- `s` is the **HMAC-SHA-256** of that content, keyed with the UTF-8 bytes of your signature secret, encoded as **uppercase hex**.
- The secret must be 64–256 characters and cannot be retrieved after it is set, so store it when you create the webhook.

To validate:

1. Extract `t` and `s` from `X-XY-Signature` (format `t=<epoch seconds>,s=<hex>`).
2. Compute `HMAC_SHA256(secret, "<t>." + rawBody)` and hex-encode it.
3. Compare against `s` using a constant-time comparison (case-insensitively, or uppercase yours).
4. Reject stale timestamps. Five minutes is a reasonable tolerance for replay protection.

Worked example

```python
import hashlib, hmac

secret = b"my-signature-secret-of-at-least-64-characters-abcdefghijklmnopqrs"
raw_body = b'[{"floorSpaceId":"ac52e43d-7c21-4464-ab21-d0ace07a94b1","headcount":2}]'
t = "1608766071"

signed_content = t.encode() + b"." + raw_body
expected = hmac.new(secret, signed_content, hashlib.sha256).hexdigest().upper()
# compare `expected` to the `s` value with hmac.compare_digest
```

Example request with signature header

```
POST https://<your endpoint>
Content-Type: application/json; charset=utf-8
X-XY-Signature:
t=1608766071,s=13EEAE8BEDA256A9098554AECA718096504B143C1D51CAC2D1555F38210B09A2

<json payload>
```

Register your webhook

You can register webhooks via the Admin Console at `https://app.xysense.io/settings/webhooks` or programmatically through GraphQL.

Each webhook type requires the matching API feature (Occupancy API, Sightings API, or Space Sensor Readings API) to be enabled on at least one of your floors, and events are only delivered for floors where it is enabled. If you are unable to register your Webhook, your XY Sense representative can enable the feature. See Feature enablement.

GraphQL request

Open in API Explorer

Example query

```
mutation AddWebhook($headers: [InputPair!]) {
  addWebhook(
    name: "Occupancy stream"
    webhookType: OccupancyChanges
    endpoint: "https://example.com/hooks/xy"
    email: "integrations@example.com"
    additionalHeaders: $headers
    signatureSecret: "<64-256 char secret>"
    enabled: true
  ) {
    id
    name
    endpoint
    enabled
    webhookType
  }
}
```

Example variables

```
{
  "headers": [
    {
      "key": "X-Partner-Token",
      "value": "mytoken"
    }
  ]
}
```

Example response

```
{
  "data": {
    "addWebhook": {
      "name": "Occupancy stream",
      "endpoint": "https://example.com/hooks/xy",
      "additionalHeaders": [
        {
          "key": "X-Partner-Token",
          "value": "mytoken"
        }
      ],
      "signatureSecret": "[redacted]",
      "enabled": true
    }
  }
}
```

Test your receiver

If your floor is in dev mode, you can generate synthetic occupancy and floor-sighting traffic. The full request and response examples are in Webhook test events.

Resolve event data with GraphQL

Webhook payloads contain stable identifiers and changing measurements. Use the GraphQL API to resolve those identifiers into the names, hierarchy, floor plans, and sensor associations needed by your application.

Recommended integration pattern

1. Bootstrap reference data from GraphQL before processing events.
2. Cache buildings, floors, floor spaces, sensors, and their relationships locally by ID.
3. Process webhooks idempotently and join each event to the cached reference data.
4. Query GraphQL on a cache miss, then update the cache.
5. Refresh reference data periodically and use GraphQL to reconcile after an interruption.

Join events against your local cache rather than querying GraphQL once per event. Webhook batches can contain many events for the same entity, so per-event lookups add latency and unnecessary API traffic.

Event type	Event identifier	GraphQL lookup	Reference data to cache
`OccupancyChanges`	`floorSpaceId`	`floorSpace(id:)`	Space name, capacity, type, floor, and building
`FloorSightings`	`floorId`	`floor(id:)`	Floor name, building, floor plan URL, and scale
`SpaceSensorReadings`	`spaceSensorId`	`spaceSensor(id:)`	Sensor name, type, installations, floors, and floor spaces

Resolve occupancy events

Use `floorSpaceId` from the occupancy event to identify the affected desk, room, or zone.

Open in API Explorer

Example query

```
query ResolveOccupancySpace($floorSpaceId: ID!) {
  floorSpace(id: $floorSpaceId) {
    id
    name
    capacity
    spaceType {
      id
      name
    }
    floor {
      id
      name
      building {
        id
        name
      }
    }
  }
}
```

Example variables

```
{
  "floorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1"
}
```

Example response

```
{
  "data": {
    "floorSpace": {
      "id": "ac52e43d-7c21-4464-ab21-d0ace07a94b1",
      "name": "Meeting Room 12.04",
      "capacity": 8,
      "spaceType": { "id": "0f3f9c6a-2f6d-4f0a-9a5b-8f6a1f9f2b31", "name": "Meeting Room"
},
      "floor": {
        "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
        "name": "Level 12",
        "building": { "id": "0b26f8f3-5d1c-4c6a-9d3e-1a2b3c4d5e6f", "name": "1 Example
Street" }
      }
    }
  }
}
```

Resolve floor sightings

Use `floorId` to fetch the floor plan and scale required to display sighting coordinates. Coordinates are measured in centimetres from the top-left of the floor plan.

Open in API Explorer

Example query

```
query ResolveSightingFloor($floorId: ID!) {
  floor(id: $floorId) {
    id
    name
    floorPlanUrl
    scale
    building {
      id
      name
    }
  }
}
```

Example variables

```
{
  "floorId": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204"
}
```

Example response

```
{
  "data": {
    "floor": {
      "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
      "name": "Level 12",
      "floorPlanUrl": "https://cdn.xysense.io/floorplans/level-12.png",
      "scale": 1.8,
      "building": { "id": "0b26f8f3-5d1c-4c6a-9d3e-1a2b3c4d5e6f", "name": "1 Example
Street" }
    }
  }
}
```

Resolve space sensor readings

Use `spaceSensorId` to identify the sensor and the active installations and floor spaces it serves. The event's `capabilityName` identifies the measured property, such as temperature or humidity.

Open in API Explorer

Example query

```
query ResolveSpaceSensor($spaceSensorId: ID!) {
  spaceSensor(id: $spaceSensorId) {
    id
    name
    type {
      id
      name
    }
    installations(isActive: true) {
      id
      name
      floor {
        id
        name
        building {
          id
          name
        }
      }
      floorSpaces {
        id
        name
      }
    }
  }
}
```

Example variables

```
{
  "spaceSensorId": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3"
}
```

Example response


```
{
  "data": {
    "spaceSensor": {
      "id": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3",
      "name": "SS-12-041",
      "type": { "id": "6a1f2b3c-4d5e-6f70-8192-a3b4c5d6e7f8", "name": "Space Sensor Mini"
},
      "installations": [
        {
          "id": "4d1c2b3a-5e6f-7081-92a3-b4c5d6e7f809",
          "name": "Level 12 North",
          "floor": {
            "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
            "name": "Level 12",
            "building": { "id": "0b26f8f3-5d1c-4c6a-9d3e-1a2b3c4d5e6f", "name": "1 Example
Street" }
          },
          "floorSpaces": [{ "id": "ac52e43d-7c21-4464-ab21-d0ace07a94b1", "name": "Meeting
Room 12.04" }]
        }
      ]
    }
  }
}
```


Handle reference-data changes

Treat GraphQL reference data as mutable. Spaces can be renamed, sensors can be moved, and installations can become inactive. Keep event records keyed by their XY Sense identifiers and update descriptive fields from GraphQL rather than treating names as identifiers.

Webhook test events

Use these mutations to validate webhook receivers or downstream ingestion without waiting for live traffic.

> **DEV-MODE ONLY**
>
> These mutations are only available for floors in dev mode, and require `WritePublicApi` **on the API key**. See Test data without live sensors for how to get a dev-mode floor and the full list of generator mutations.

Generate occupancy change events

Open in API Explorer

Example query

```
mutation generateOccupancyEvents {
  generateOccupancyEvents(floorId: "<floor guid>", count: 1, includeHeadcountChanges:
true) {
    floorSpaceId
    headcount
    occupancyStatus
    previousOccupancyStatus
    collectedDate
  }
}
```

Example response

```
{
  "data": {
    "generateOccupancyEvents": [
      {
        "floorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1",
        "headcount": 0,
        "occupancyStatus": "NotOccupied",
        "previousOccupancyStatus": "RecentlyOccupied",
        "collectedDate": "2023-04-19T04:17:39.1066221Z"
      }
    ]
  }
}
```

Generate floor sighting updates

Open in API Explorer

Example query

```
mutation generateSightings {
  generateFloorSightings(floorId: "<floor guid>", sightingCount: 20, durationSecs: 30)
}
```

Example response

```
{
  "data": {
    "generateFloorSightings": true
  }
}
```

Notes

- The occupancy mutation toggles randomly selected spaces on the target floor.
- The floor-sightings mutation can emit sightings continuously for the requested duration.
- Use the regular polling queries or webhook receiver to verify that the generated traffic arrived where expected.