

XY Sense Developer Documentation

External GraphQL API and webhooks

Contents

Overview
- Welcome to the XY Sense Developer Portal
- The two integration surfaces
- What is covered
- Production endpoints

Concepts

Available data
- Limits and retention
- Space data
- Sensor data and connection status
- Occupancy time series
- Space sensor readings
- Floor sightings
- Entry crossings
- Analytics time series
- Historical heatmaps

How sensors map to spaces
- What area sensors actually produce
- The consequences worth designing around
- How each sensor type relates to spaces
- occupancySource tells you which method produced a value
- Where to go next

How occupancy is determined
- From sightings to occupancy
- Occupancy state is a function of time since the last sighting
- Why analytics numbers differ from live ones
- Availability follows the same split
- Time is UTC, buildings are not
- Where to go next

Joining data to your own system
- Identifiers are stable, names are labels
- Mapping to your own system
- Timestamps are UTC
- The floor plan coordinate system
- Coordinates in analytics and heatmaps

Choosing a data source
- Which source for which job
- Feature enablement
- Live data vs analytics

GraphQL API
- Get started
- When to use GraphQL
- Endpoint

Quickstart
- Before you start
- 1. Create an API key
- 2. Exchange the key for a bearer token
- 3. Run your first query
- 4. Explore from here
- Troubleshooting

Authentication
- Token lifetime
- Obtain your bearer token
- Attach the bearer token

- Manage API keys
- Partner accounts
- Operational notes

Connection requirements
- Endpoint
- Headers
- TLS policy

Postman collection
- Download and import
- Run order
- Prerequisites

Errors
- HTTP status handling
- GraphQL payload errors

Limits, versioning, and support
- Rate limits
- API versioning
- Schema stability and breaking changes
- Getting help

GraphQL guides
- Discovery and test data
- Polling integrations
- Sensor status
- Heatmap

Building and floor discovery
- Typical use
- Query

Space external IDs
- Typical use
- Reading external IDs
- Availability
- Setting an external ID
- Deleting an external ID
- Names reserved by XY Sense

Test data without live sensors
- Requirements
- Generator mutations
- Generate occupancy events
- Verify the data arrived
- What to expect from generated data

Occupancy polling
- 1. List the spaces you care about
- 2. Read the current state of those spaces
- 3. Poll for changes with a moving cursor

Analytics polling
- 1. Find your buildings and how much data is ready
- 2. Pull the aggregates at the resolution you need
- 3. Join the rows back to named spaces

Floor sightings polling
- 1. Get the floor plan, its pixel size, and its scale
- 2. Fetch a recent snapshot of the floor
- 3. Poll for updates with a moving cursor

Space sensor readings polling
- 1. Discover which capabilities your sensors report
- 2. Read the latest values
- 3. Poll for changes with a moving cursor

Entry crossings polling
- 1. Find the entries on a floor and the spaces they lead into
- 2. Fetch recent crossings
- 3. Read the direction of a crossing
- 4. Poll for new crossings with a moving cursor

Sensor connection status
- Check which sensors are online
- Find what a sensor affects

Historical heatmap
- 1. Fetch the heatmap cells and the floor plan
- 2. Convert cell centres to image pixels
- 3. Render the overlay

Webhooks
- Get started
- Supported event types
- When to use webhooks
- Scope and volume
- Test your receiver

Quickstart
- Before you start
- 1. Register the webhook
- 2. Handle the first delivery
- 3. Verify it is working
- 4. Harden it
- Troubleshooting

Delivery and batching
- Request format
- The batch envelope
- Ordering and acknowledgement
- Batching
- Idempotency
- Retries and failure handling
- Recovering a backlogged receiver
- Compression
- Configuration options
- Service limits
- Managing webhooks

Event payloads
- Occupancy
- Floor sightings
- Space sensor readings

Securing your endpoint
- Custom headers
- Request signatures
- Validating a delivery
- Rotating a secret

Resolve event data with GraphQL
- Recommended integration pattern
- Resolve occupancy events
- Resolve floor sightings
- Resolve space sensor readings
- Handle reference-data changes

Webhook test events
- Generate occupancy change events
- Generate floor sighting updates
- What the generators do

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.

The rule of thumb is webhooks for ongoing changes and GraphQL for reference data, history, and reconciliation. For a dataset-by-dataset breakdown of which source fits which job, see Choosing a data source.

Start with the Quickstart to make your first call, then Choosing a data source to pick your surfaces before you build.

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`

Concepts

How the platform works, and the parts worth understanding before you model your integration. These apply to both the GraphQL API and webhooks.

- Available data — the space, sensor, occupancy, readings, sightings, analytics, and heatmap datasets, with the limits and retention for each.
- How sensors map to spaces — why area sensors have no space association, how sightings become per-space occupancy, and what that means for your integration.
- How occupancy is determined — how sightings become occupancy state, why analytics numbers differ from live ones, and which filters are applied.
- Joining data to your own system — how XY Sense identifiers, timestamps, and floor plan coordinates behave, so you can line our data up with yours.
- Choosing a data source — webhooks or GraphQL, live occupancy or analytics, and what raw history is available.

Available data

XY Sense exposes eight broad categories of data, the same whether you query them over the GraphQL API or receive them through webhooks. 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, entry crossings, 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
Entry crossings	Each pass through an entry, with its direction into or out of each space it leads to	How busy an entrance is, and which way people move
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

Limits and retention

This table is the single source of truth for request windows, retention, and freshness. The guides link here rather than repeating the numbers, so if a figure appears elsewhere in these docs and disagrees with this table, treat this table as correct and tell us.

DATASET	MAXIMUM WINDOW PER REQUEST	ROLLING RETENTION	FRESHNESS
Space and sensor reference	n/a, current state	n/a, current state	Updated as your portfolio changes
Sensor connection status	n/a, current state	n/a, current state	Near-real-time

DATASET	MAXIMUM WINDOW PER REQUEST	ROLLING RETENTION	FRESHNESS
Occupancy changes	1 hour	1,000,000 changes or 24 hours	~2 seconds
Floor sightings	1 hour	1,000,000 updates or 24 hours	~2 seconds
Space sensor readings	24 hours	1,000,000 readings or 48 hours	~10 minutes for comfort and air quality
Entry crossings	1 hour	At least 72 hours	A few seconds
Analytics, per minute	1 day	Retained long-term	Previous day
Analytics, per five minutes	1 day	Retained long-term	Previous day
Analytics, per hour	31 days	Retained long-term	Previous day
Analytics, per day	365 days	Retained long-term	Previous day
Historical heatmaps	31 days	Retained long-term	~15 minutes behind live

Analytics is processed overnight per building, so each day's figures become available from around 6 am the next morning in the building's local time. The most recent analytics you can retrieve is the previous day's.

Whichever limit is reached first applies: space sensor readings, for example, are dropped at 1,000,000 readings even if that happens inside 48 hours. Entry crossings are kept for at least 72 hours and can sometimes be returned for a little longer, so do not rely on older crossings disappearing.

Analytics and heatmap requests are scoped to one building and paged with a maximum page size of 100,000 records.

Raw events age out at the retention shown above. If your application needs event-level history beyond those windows, capture it through webhooks and store it from day one. Webhook queue depth and retry limits are listed separately under delivery and batching.

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. Occupancy is derived by merging detections across sensors and overlaying space shapes, so there is no per-sensor occupancy to query: see How sensors map to spaces.
- 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.

Request windows and retention are in Limits and retention.

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.

For why the states behave this way, including why a space can change state with nobody entering or leaving it, see How occupancy is determined.

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. Request windows, retention, and update frequency are in Limits and retention.

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. Request windows and retention are in Limits and retention.

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

Entry crossings

An entry crossing is one person passing through an entry, the line an entry sensor watches across a doorway or opening. Each crossing carries:

- the entry and the floor its sensor is installed on,
- when the sensor saw it (`collectedDate`),
- `lineDirection` , which way it crossed the line itself,
- `spaces` , each space the entry leads into with `In` or `Out` already resolved for that space.

A door between two spaces is `In` for one and `Out` for the other in the same crossing. The direction uses the entry's space mapping at the time of the crossing, so remapping an entry later does not change older crossings. An entry that is not mapped to any space has an empty `spaces` list.

Entry sensors also drive live headcount, so if you only need how many people are in a space, use occupancy. Use entry crossings when you need the individual passes, for example to see how busy an entrance is. Request windows and retention are in Limits and retention. See entry crossings polling for queries and how to read the direction.

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, so it is next-day data rather than a live feed. Request windows and availability are in Limits and retention.

Each interval applies different default filtering:

INTERVAL	DEFAULT FILTERING
Per minute	None
Per five minutes	None
Per hour	Excludes walk-bys
Per day	Excludes walk-bys and out-of-hours activity

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.

Unlike the other analytics queries, heatmaps run close to live rather than next-day. Request windows and freshness are in Limits and retention. See historical heatmap for floor-plan positioning and rendering guidance.

How sensors map to spaces

If you have integrated with occupancy hardware before, you probably expect to ask a sensor for its state: one sensor covers one desk or one room, you read it, and that is the occupancy. That is how a PIR sensor works.

XY Sense area sensors work differently. There is no direct relationship between an area sensor and a space: occupancy is measured across the whole floor and then attributed to each space. Understanding that upfront makes the rest of the API fit together, and it is worth knowing before you model your side.

What area sensors actually produce

An area sensor detects people as anonymous positions within its field of view. It does not know what a desk is, which room it is looking at, or where one space ends and the next begins.

A floor is covered by however many sensors it takes, and their coverage overlaps. One person standing in an overlap is seen by more than one sensor. So detections are not usable as-is:

1. Sensors detect people as positions.
2. XY Sense stitches those detections together into a single picture of the floor, deduplicating people seen by more than one sensor as it goes.
3. Space shapes are overlaid on that picture, and the people inside each shape are counted.
4. That count, and how recently it changed, is the occupancy for that space.

This is why the API exposes **floor** sightings rather than sensor detections: by the time data reaches you, per-sensor detections have already been merged into one floor-level view with a single headcount. There is no endpoint for "what did sensor X see", because that is an intermediate state rather than an answer.

The consequences worth designing around

- **You cannot query occupancy for an area sensor**, and the schema reflects that: `AreaSensorInstallation` exposes `floor` and `floorLocation`, but no floor spaces. The omission is deliberate, not missing functionality.
- **Occupancy is a property of a space, not of a sensor.** Join everything on `floorSpaceId`. Any integration logic keyed to a sensor identity will not survive an install change.
- **Spaces can change without sensors changing.** Redrawing a space boundary, splitting a room into two zones, or retiring a desk changes occupancy immediately with no hardware change at all. Equally, sensors can be added, moved, or swapped without changing which spaces exist. Refresh your cached space reference data on a schedule and treat an unknown `floorSpaceId` as a signal to refresh.
- **Sensor health is not space health.** One offline area sensor degrades coverage across part of a floor, it does not switch one space off. Do not map a sensor's connection state onto a space's occupancy in your UI. See sensor connection status.
- **Do not rebuild occupancy from sightings yourself.** You can read floor sightings and test them against space polygons, and it will look like it works, but you will be reimplementing the space-overlay step without the filtering and state model that occupancy applies. Use the occupancy dataset and keep sightings for spatial views.

How each sensor type relates to spaces

The "no space association" rule is specific to area sensors. The three sensor types each relate to spaces in their own way:

SENSOR TYPE	ASSOCIATED WITH SPACES?	HOW IT CONTRIBUTES TO OCCUPANCY
Area sensor	No. Positioned on a floor, with no floor-space association at all	Detections are stitched and deduplicated, then spaces are overlaid
Entry sensor	Yes, indirectly. Its `entries` are associated with floor spaces	Directional movement through an entry adds to or subtracts from a count
Space sensor	Yes, directly. An installation is associated with one or more spaces	Reports readings for a space, which can include an occupancy capability

A space sensor maps most directly to a single space, and even so its readings are a separate dataset from occupancy.

`occupancySource` tells you which method produced a value

An `Occupancy` record carries an `occupancySource`, which reports how that value was arrived at: `Area`, `Entry`, `Space`, `Manual`, or `Unknown`. A floor equipped with more than one type of sensing can therefore produce occupancy for different spaces by different methods.

Treat it as diagnostic context rather than something to branch your integration on. The headcount and status mean the same thing whichever source produced them, and a space's source can change if the hardware on that floor changes.

Where to go next

- How occupancy is determined for the state model applied to these counts, and why analytics figures differ from live ones.
- Available data for the sensor and installation fields.
- Joining data to your own system for the floor plan coordinate system that positions and space shapes share.

How occupancy is determined

This page explains the model behind the numbers. It is background rather than instruction: if you are trying to get data out, start with the GraphQL quickstart or a guide.

Reading it is worth ten minutes if you are comparing API results against the XY Sense dashboard, or explaining to someone why two numbers that "should" match do not.

From sightings to occupancy

Sensors detect people as anonymous positions on a floor, expressed as XY coordinates. A detection is a **sighting**. Sightings are the raw material for everything else:

- **Floor sightings** expose those positions directly, as a full snapshot of a floor for a moment in time.
- **Occupancy** is derived by asking which sightings fall inside the shape of each floor space, then summarising that space's state. Occupancy belongs to a space rather than to any one sensor: see How sensors map to spaces.

Both are produced approximately every two seconds, which is the fastest the API and webhooks will tell you anything.

Occupancy state is a function of time since the last sighting

A space is not simply occupied or empty. Its state is decided by how recently someone was seen in it, relative to `collectedDate`, across four values with 30-second and 60-second thresholds. The exact definitions are in Available data; what follows is why the model has that shape.

`RecentlyOccupied` exists because people are not continuously visible. Someone at a desk who leans out of sensor view, or a meeting-room occupant briefly obscured, should not flip a space to empty and back again. The 30-to-60-second band absorbs that, which means a space you would call "in use" reads as either `CurrentlyOccupied` or `RecentlyOccupied` depending on the exact moment you sampled it. Treat both as occupied unless you specifically need the distinction.

Two consequences worth designing for:

- A space can change state with no one entering or leaving it, simply because 30 or 60 seconds elapsed.
- `Headcount` and `OccupancyStatus` change independently. Over 90% of occupancy events are headcount-only changes, where the status is unchanged. See `ExcludeHeadcountChanges` if you only care about occupied versus vacant.

`occupancyStatusChangeDate` tells you when the state last changed, and stays put while headcount moves around within the same state. `collectedDate` is when the observation was made, and is the field to use as the event time.

Why analytics numbers differ from live ones

Analytics is not a rolled-up copy of the live feed. It applies filters that the live feed does not, deliberately, so that reporting reflects real usage rather than raw detections.

The walk-by filter excludes activity with less than one minute of usage over a ten-minute window. Someone crossing a room to reach a window is a genuine sighting, and appears in the live data, but it is not usage of that room and would

inflate utilisation figures. Hourly and daily analytics exclude walk-bys; minute and five-minute intervals do not.

Active hours apply to daily results only. Each building has configured active hours, defaulting to 7 am to 7 pm Monday to Friday, and daily aggregates exclude activity outside them. A cleaner moving through a floor at 9 pm should not register as a working day's occupancy.

Missing intervals mean zero. Intervals with no occupancy are omitted from analytics results rather than returned as zero rows, which is a storage optimisation. When you aggregate downstream, treat an absent interval for a known space as zero occupancy, not as missing data. Averaging only the rows you received will overstate utilisation, sometimes badly.

So a live headcount and an analytics figure for the same space and hour can legitimately disagree, and the analytics number is usually the smaller one. Neither is wrong.

Availability follows the same split

Live data is available within seconds. Analytics is processed overnight per building, so the previous day's numbers appear the next morning in the building's local time. Historical heatmaps are the exception among the analytics queries and run close to live.

One thing to keep in mind about `occupancyPerMinute`: minute resolution does not mean minute latency. It is next-day data at minute granularity. Exact figures are in Limits and retention.

Time is UTC, buildings are not

Timestamps and `from` / `to` arguments are UTC, but analytics processing, active hours, and the "available next morning" behaviour are all in the building's **local** time. A building in Sydney and one in London have their overnight runs and their working day at different absolute times.

That mismatch is the most common reason API results disagree with the dashboard. When they do, check the timezone of your `from` / `to` arguments first. See Timestamps are UTC for the general rule and a worked conversion.

Where to go next

- Available data for what each dataset contains, and the limits.
- Joining data to your own system for identifier stability, timestamp semantics, and the floor plan coordinate system.
- Choosing a data source for picking the right one for your workload.

Joining data to your own system

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, time, and coordinate semantics that make that join reliable. They apply to every dataset and to both the GraphQL API and webhooks.

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 steps above store our `id` in your system. You can also do the reverse, and store your own identifier against the space in ours. See Space external IDs.

Timestamps are UTC

All API and webhook timestamps are **UTC in ISO 8601 format** (`2026-06-02T04:17:39.106Z`), and every `from` / `to` argument is interpreted as UTC. Store them as UTC on your side and convert only for display.

If results seem to disagree with the XY Sense dashboard, the timezone of your `from` / `to` arguments is the first thing to check. The dashboard presents a building's data in that building's local time, and some processing is building-local too: analytics runs overnight per building, and active hours are configured per building. See How occupancy is determined.

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"`.

Which timestamp means what

- `collectedDate` is when the underlying observation was made. Use it as the event time for occupancy and sightings.
- `occupancyStatusChangeDate` is when a space last changed occupancy *state*. It stays put while headcount varies within the same state, so it is not an event time.

- `collectedAt` (space sensor readings) means the same as `collectedDate`. The name differs for historical reasons only.
- `Created` and `Sent` on a webhook batch describe the delivery, not the observation: `Created` is when the batch started filling and `Sent` changes on every retry. See delivery and batching.

The floor plan coordinate system

Every floor has a `floorPlanUrl` (the floor plan image), a `floorPlanMetadata` object, 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.
- `floorPlanMetadata.type` is how the plan is represented. It is `Raster` today. A vector representation may be added later.
- `floorPlanMetadata.widthPixels` and `floorPlanMetadata.heightPixels` are the raster image dimensions. Use them to size a canvas or overlay. They match the image at `floorPlanUrl`, so you do not need to download the file and read its headers. Both are null when the plan is not a raster image.
- `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.

Choosing a data source

XY Sense exposes the same underlying data in several ways, each suited to a different job. A few minutes here will help you pick the right one for your workload before you build.

Which source 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
How busy an entrance is, and which way people go through it	`entryCrossings` query (polling guide)	Inferring it from occupancy changes
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: raw events are kept for a rolling window, and beyond it the API serves aggregates (analytics and heatmaps). The exact windows per dataset are in Limits and retention. 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.

Feature enablement

Each source above is gated by a feature that XY Sense enables for your organisation, and enablement can vary per building and floor. Before building against one, confirm with your XY Sense representative that it is enabled where you need it. The feature names below are entitlements rather than separate endpoints, which is why one feature covers both the queries and the matching webhook.

DATA	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
Entry crossing queries	Entry Crossings API

DATA	FEATURE
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, so it is **next-day data, not a live feed**. Note that `occupancyPerMinute` is next-day analytics too: minute resolution means minute granularity, not minute latency.
- **Historical heatmaps** are the exception among the analytics queries and run close to live.

Exact availability for each is in Limits and retention.

Analytics also applies filtering that live data does not, so the two can legitimately disagree for the same space and hour. See How occupancy is determined.

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 where to get it with Choosing a data source.
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 a data source 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 generated from the description with a short random suffix, so descriptions don't need to be unique.
- 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 key name and secret are shown together once at creation time. The secret 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.
`422`	Query over the depth, field selection or alias limits.	Split the query into smaller ones; retrying it unchanged fails the same way.
`429`	Rate limit exceeded.	Wait for the seconds in the `Retry-After` header, then retry; back off exponentially if it repeats. See Rate limits.
`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

The API enforces the limits below. A request that exceeds one is rejected rather than run.

LIMIT	VALUE	APPLIES TO
Requests per API key	300 per minute	Every query request to the external API.
Occupancy time-series per API key	45 per minute	`occupancyPerMinute`, `occupancyPerFiveMinutes`, `occupancyPerHour`, `occupancyPerDay` and their `...ForPreviousDay` variants. Each selection counts, so a request that selects three of these fields uses three.
Aliases of one occupancy time-series field	5 per request	Each of the occupancy time-series fields above.
Entry crossings per API key	60 per minute	`entryCrossings`. Each selection counts, so a request that selects it under three aliases uses three.
Query depth	15 levels of nesting	Every request.
Field selections	2,000 per request, after fragments are expanded	Every request.

Per-minute limits use a fixed one-minute window: the count resets at the start of each window rather than sliding.

When a limit is exceeded

The request is rejected and the response carries an `errors` entry naming the limit, for example:

- `Rate limit exceeded for policy 'apiBudget'.` - the per-key request limit,
- `Rate limit exceeded for policy 'occupancyPerPeriod'.` - the occupancy time-series limit,
- `Rate limit exceeded for policy 'heavyQuery'.` - the entry crossings limit,
- `Query exceeds maximum allowed depth of 15` or a similar message for the depth, field selection and alias limits.

A rate-limit rejection returns HTTP status `429 Too Many Requests` with a `Retry-After` header giving the seconds to wait before retrying. A query rejected for depth, field selections or aliases returns `422 Unprocessable Content`. Other errors keep their usual status codes; see Errors. When you hit a rate limit, wait for the `Retry-After` period before retrying, and back off exponentially rather than retrying in a tight loop. A query rejected for depth, field selections or aliases will fail the same way every time, so split it rather than retrying it.

Staying within the limits

The patterns below keep an integration well inside the limits, and fast:

- **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.
- **Fetch occupancy time-series by floor or building, not per space**, so a full refresh stays within the 45-per-minute occupancy limit.
- Request windows and retention limits per dataset are listed in Available data.

If you have a workload that needs more than these limits allow (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.

API versioning

There is no version in the API URL, and no version header to send. A single current version of the GraphQL schema is served, and integrations do not pin to a version.

That works because the schema only grows: see Schema stability and breaking changes for what is additive, what counts as breaking, and how breaking changes are communicated. If we ever need to introduce versioning, it will be announced well ahead of time along with a migration path for existing integrations.

Deprecated fields

Where a field, argument, or type is superseded, we keep the original working and mark it **deprecated** rather than removing it, so existing integrations keep running. This is how we avoid needing an API version.

A deprecation is marked in the schema itself and carries a reason naming what to use instead. The API currently has a few, for example:

- `location` and `locations` are deprecated in favour of `building` and `buildings`,
- on `SpaceGroup`, `spaces` and `totalSpaces` are deprecated in favour of `floorSpaces` and `totalFloorSpaces`.

If you are building a new integration, do not use deprecated fields. They are the likely candidates for removal in future, and removal is a breaking change, so it would come with advance notice and a migration window. Choosing the current field now saves you that migration. If you maintain an existing integration, there is no urgency to move, but moving off a deprecated field when you next touch the code is worthwhile.

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. Extending rather than changing the API is a

deliberate choice, and it is why there is no API version for you to pin.

Changes we make without notice, and that your client should tolerate:

- new fields and types in the schema,
- new optional arguments on existing fields,
- new values in an existing enum,
- new event types and new fields in webhook payloads,
- marking an existing field as deprecated, which does not change its behaviour (see Deprecated fields).

Changes we treat as breaking, and do not make without notice:

- removing or renaming a field, argument, type, or enum value,
- changing the type or nullability of an existing field,
- changing the meaning of an existing field, or the units it is expressed in,
- making a previously optional argument required.

To stay compatible, request only the fields you use, ignore fields you do not recognise, and handle an unfamiliar enum value as a default case rather than an error.

When a breaking change is unavoidable, XY Sense notifies registered API contacts ahead of time with a migration window, so you have time to adjust before the change takes effect. Planned maintenance that affects API availability is communicated the same way, ahead of the window.

Make sure we hold the right technical contact for your integration, so these notices reach the person who maintains it rather than a commercial contact.

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, Entry Crossings API, Webhooks) and **dev-mode floors** are provisioned by XY Sense, ask your representative or contact us via support@xysense.com.

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
- Space external IDs
- Test data without live sensors

Polling integrations

- Occupancy polling
- Analytics polling
- Floor sightings polling
- Space sensor readings polling
- Entry crossings 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`, `floorPlanMetadata`, 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
      floorPlanMetadata {
        type
        widthPixels
        heightPixels
      }
      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",
            "floorPlanMetadata": { "type": "Raster", "widthPixels": 4200, "heightPixels":
2800 },
            "scale": 1.8
          }
        ]
      }
    ]
  }
}
```


All identifiers are GUIDs. The examples throughout these guides use the same building, floor, and spaces, so IDs can be followed from one page to the next.

Space external IDs

An external ID is your own identifier for a space, stored against that space in XY Sense. Where joining data to your own system covers keeping our `id` on your side, external IDs are the other direction: keeping your key on ours, so the mapping between your codes and our spaces lives with the space instead of in a table you maintain.

A space can hold any number of external IDs, each under a different name. The name is how you group them, so use one name consistently across your portfolio — `acme.roomCode` for every room code, rather than a different name per building.

You can set your own external IDs on spaces only. Floors, buildings and locations do not support them through this API.

Typical use

- Keep your room or asset code with the space in XY Sense, and read it back whenever you fetch spaces from the API.
- Recognise a space from an occupancy webhook. Payloads carry our `FloorSpaceId`, not your external IDs, so include `externalIds` in the space reference data you cache and match events on `FloorSpaceId`. See Resolve event data with GraphQL.
- Map the same space into two systems at once, under two names.

Reading external IDs

`externalIds` returns the external IDs you can manage on a space. Use `externalId(name: "...")` instead when you already know the name and want the value alone. Names XY Sense manages are left out of both, as described below.

Open in API Explorer

Example query

```
query SpaceExternalIds {
  floorSpaces(floorId: "a2961d46-971f-4ce1-a66e-7ee7bdb2a204") {
    id
    name
    externalIds {
      name
      value
    }
  }
}
```

Example response

```
{
  "data": {
    "floorSpaces": [
      {
        "id": "7c9e6679-7425-40de-944b-e07fc1f90ae7",
        "name": "Meeting Room 4",
        "externalIds": [
          {
            "name": "acme.roomCode",
            "value": "\"L12-MR-04\""
          }
        ]
      }
    ]
  }
}
```

Values are stored and returned as JSON. A plain string therefore comes back quoted, as above, and needs parsing before you use it.

Availability

Space external IDs are on by default, and can be switched off per customer. If they have been switched off for yours, the mutations return:

```
{ "errors": [{ "message": "The external ID API is not enabled for this customer." }] }
```

The reads do not error. `externalIds` comes back as an empty list and `externalId(name: "...")` comes back `null`, so neither result on its own means a space has none set. Talk to your account contact if you need it back on.

Setting an external ID

`setSpaceExternalId` adds the external ID if that name is not yet set on the space, and replaces the value if it is. It returns `true` on success.

```
mutation SetSpaceExternalId {
  setSpaceExternalId(spaceId: "7c9e6679-7425-40de-944b-e07fc1f90ae7", name:
"acme.roomCode", value: "L12-MR-04")
}
```

Both this and the delete below need an API key with write access. See Authentication.

Deleting an external ID

`deleteSpaceExternalId` takes the current `value` as well as the name. The delete only goes ahead if the stored value matches, which stops a stale request removing an ID that has since changed.

```
mutation DeleteSpaceExternalId {
  deleteSpaceExternalId(spaceId: "7c9e6679-7425-40de-944b-e07fc1f90ae7", name:
"acme.roomCode", value: "L12-MR-04")
}
```

Names reserved by XY Sense

Some external ID names belong to XY Sense. They are not returned by the read queries above and cannot be written, so they are invisible through this API entirely:

- anything beginning with `$`, such as `$bim` and `$bacnet`
- `serraview`, `Airthings` and `Kaiterra`, which predate the `$` convention and are already in use

Matching on the named ones is exact and case-insensitive, so a name of your own like `serraview.ourRef` is fine. Writing to a reserved name is rejected:

```
{
  "errors": [
    {
      "message": "External identifier '$bim' is XY Sense managed and cannot be edited
manually."
    }
  ]
}
```

Pick a name with a prefix of your own, such as `acme.`, and the reservation will never affect you.

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.

What to expect from generated data

- 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.

1. List the spaces you care about

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

Example response


```
{
  "data": {
    "floorSpaces": [
      {
        "id": "5c8a1d92-4b6e-4f03-9a71-2d4e6f8a0b13",
        "name": "Desk NE-101",
        "capacity": 1,
        "floor": {
          "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
          "name": "Level 12"
        },
        "spaceType": {
          "id": "7d2e4f60-8a91-4c23-b5d6-0e1f2a3b4c56",
          "name": "Desk"
        }
      }
    ]
  }
}
```


2. Read the current state of those spaces

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

Example response


```
{
  "data": {
    "floorSpaces": [
      {
        "id": "5c8a1d92-4b6e-4f03-9a71-2d4e6f8a0b13",
        "name": "Desk NE-101",
        "latestOccupancy": {
          "floorSpaceId": "5c8a1d92-4b6e-4f03-9a71-2d4e6f8a0b13",
          "headcount": 1,
          "occupancyStatus": "CurrentlyOccupied",
          "previousOccupancyStatus": "RecentlyOccupied",
          "collectedDate": "2026-06-02T04:17:39.1066221Z",
          "occupancyStatusChangeDate": "2026-06-02T04:17:39.1066221Z"
        }
      }
    ]
  }
}
```


3. Poll for changes with a moving cursor

Each response carries a `nextFrom`. Persist it and send it as the next request's `from`: that is the cursor, and it is what stops you re-reading or skipping changes. Request windows and retention are in Limits and retention.

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": [
    "a2961d46-971f-4ce1-a66e-7ee7bdb2a204"
  ],
  "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": "5c8a1d92-4b6e-4f03-9a71-2d4e6f8a0b13",
          "headcount": 0,
          "occupancyStatus": "NotOccupied",
          "previousOccupancyStatus": "RecentlyOccupied",
          "collectedDate": "2026-06-02T04:17:39.1066221Z",
          "occupancyStatusChangeDate": "2026-06-02T04:17:39.1066221Z"
        }
      ]
    }
  }
}
```

Narrowing the poll to specific spaces

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": [
    "5c8a1d92-4b6e-4f03-9a71-2d4e6f8a0b13",
    "6e0b4c85-1f27-4d93-a8b5-3c7d9e0f1a24"
  ],
  "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": "5c8a1d92-4b6e-4f03-9a71-2d4e6f8a0b13",
          "headcount": 1,
          "occupancyStatus": "CurrentlyOccupied",
          "collectedDate": "2026-06-02T04:52:10.5411902Z"
        },
        {
          "floorSpaceId": "6e0b4c85-1f27-4d93-a8b5-3c7d9e0f1a24",
          "headcount": 4,
          "occupancyStatus": "CurrentlyOccupied",
          "collectedDate": "2026-06-02T04:49:58.1178410Z"
        }
      ]
    }
  }
}
```

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.

1. Find your buildings and how much data is ready

Open in API Explorer

Example query

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

Example response


```
{
  "data": {
    "buildings": [
      {
        "id": "0b26f8f3-5d1c-4c6a-9d3e-1a2b3c4d5e6f",
        "name": "1 Example Street",
        "analyticsAvailableTo": "2026-06-02T00:00:00Z",
        "floors": [
          {
            "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
            "name": "Level 12",
            "scale": 1.8,
            "floorPlanMetadata": { "type": "Raster", "widthPixels": 4200, "heightPixels":
2800 },
            "floorPlanUrl": "https://cdn.xysense.io/floorplans/level-12.png"
          }
        ]
      }
    ]
  }
}
```


2. Pull the aggregates at the resolution you need

Pick the interval that matches your reporting need: `occupancyPerMinute`, `occupancyPerFiveMinutes`, `occupancyPerHour`, or `occupancyPerDay`. This example uses hourly aggregation. The window each interval allows per request, and when a day's data becomes available, are in Limits and retention.

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": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
  "from": "2026-06-01T00:00:00Z",
  "to": "2026-06-02T00:00:00Z"
}
```

Example response

```
{
  "data": {
    "occupancyPerHour": {
      "items": [
        {
          "floorSpaceId": "5c8a1d92-4b6e-4f03-9a71-2d4e6f8a0b13",
          "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
    }
  }
}
```

Paging through large results

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

```
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
      periodStartLocal
      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.

Intervals with no occupancy are omitted

Empty intervals are left out of the results rather than returned as zero rows, so treat an absent interval for a known space as zero occupancy rather than missing data. Averaging only the rows you received will overstate utilisation. Why that is.

3. Join the rows back to named spaces

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

Example response


```
{
  "data": {
    "floorSpaces": [
      {
        "id": "5c8a1d92-4b6e-4f03-9a71-2d4e6f8a0b13",
        "name": "Desk NE-101",
        "capacity": 1,
        "spaceType": {
          "id": "7d2e4f60-8a91-4c23-b5d6-0e1f2a3b4c56",
          "name": "Desk",
          "globalSpaceCategory": {
            "id": "3b9c7e14-2d58-4a6f-8b70-9c1d2e3f4a5b",
            "name": "Workstation"
          }
        }
      }
    ]
  }
}
```

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.

1. Get the floor plan, its pixel size, and its scale

Sighting coordinates are centimetres from the top-left of the floor plan. `floorPlanMetadata` describes the plan: `type` is `Raster` today, and `widthPixels` and `heightPixels` are the raster image size. `scale` is the plan's resolution in centimetres per image pixel, so divide by it to place a sighting on the image. See the coordinate system.

Open in API Explorer

Example query

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

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",
      "floorPlanMetadata": { "type": "Raster", "widthPixels": 4200, "heightPixels": 2800
},
      "scale": 1.8
    }
  }
}
```


2. Fetch a recent snapshot of the floor

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": [
    "a2961d46-971f-4ce1-a66e-7ee7bdb2a204"
  ]
}
```

Example response


```
{
  "data": {
    "floorSightingChanges": {
      "nextFrom": "2026-06-02T07:36:11.964156Z",
      "data": [
        {
          "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 }
          ]
        }
      ]
    }
  }
}
```


3. Poll for updates with a moving cursor

Persist `nextFrom` and send it as the next request's `from`. Request windows and retention are in Limits and retention.

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": [
    "a2961d46-971f-4ce1-a66e-7ee7bdb2a204"
  ],
  "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": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
          "headcount": 2,
          "collectedDate": "2026-06-02T07:36:11.964156Z",
          "sightings": [
            { "x": 1842, "y": 971 },
            { "x": 2510, "y": 1164 }
          ]
        }
      ]
    }
  }
}
```

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.

1. Discover which capabilities your sensors report

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"
      }
    ]
  }
}
```

2. Read the latest values

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": [
    "a2961d46-971f-4ce1-a66e-7ee7bdb2a204"
  ]
}
```

Example response

```
{
  "data": {
    "spaceSensorReadings": {
      "nextFrom": "2026-06-02T07:36:11.964156Z",
      "data": [
        {
          "spaceSensorId": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
          "capabilityName": "Temperature",
          "value": "22.4",
          "collectedAt": "2026-06-02T07:36:11.964156Z"
        },
        {
          "spaceSensorId": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
          "capabilityName": "Humidity",
          "value": "48.2",
          "collectedAt": "2026-06-02T07:36:11.964156Z"
        }
      ]
    }
  }
}
```

3. Poll for changes with a moving cursor

Persist `nextFrom` and send it as the next request's `from`. Request windows and retention are in Limits and retention.

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": [
    "a2961d46-971f-4ce1-a66e-7ee7bdb2a204"
  ],
  "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": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
          "capabilityName": "Temperature",
          "value": "22.4",
          "collectedAt": "2026-06-02T06:50:12.331245Z"
        },
        {
          "spaceSensorId": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
          "capabilityName": "Humidity",
          "value": "48.2",
          "collectedAt": "2026-06-02T06:50:12.331245Z"
        }
      ]
    }
  }
}
```

Finding which spaces a sensor serves

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

Open in API Explorer

Example query

```
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

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

Example response


```
{
  "data": {
    "floor": {
      "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
      "name": "Level 12",
      "floorSpaces": [
        {
          "id": "6e0b4c85-1f27-4d93-a8b5-3c7d9e0f1a24",
          "name": "Room 12A",
          "spaceSensorInstallations": [
            {
              "id": "9c1d5e83-2b47-4f60-a915-6d8e0f1a2b37",
              "name": "Room 12A Sensor",
              "spaceSensorId": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46"
            }
          ]
        }
      ],
      "spaceSensorInstallations": [
        {
          "id": "9c1d5e83-2b47-4f60-a915-6d8e0f1a2b37",
          "name": "Room 12A Sensor",
          "spaceSensorId": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
          "floorSpaces": [
            {
              "id": "6e0b4c85-1f27-4d93-a8b5-3c7d9e0f1a24",
              "name": "Room 12A"
            }
          ],
          "ignoredCapabilities": [
            {
              "id": "47c873e6-3c60-41f8-9d17-96e5639b3bde",
              "name": "Humidity"
            }
          ]
        }
      ]
    }
  }
}
```


Narrowing the poll to specific sensors

Filter by sensor when a receiver only needs part of the estate. Use `comfortOnly: true` when you want comfort and air-quality readings without occupancy-style sensor output.

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": [
    "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
    "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": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
          "capabilityName": "CarbonDioxide",
          "value": "641",
          "collectedAt": "2026-06-02T06:50:12.331245Z"
        }
      ]
    }
  }
}
```

Entry crossings polling

Use entry crossings when you need each individual pass through an entry, for example to count how busy a doorway is or to see which way people are moving. Each crossing is one person passing through one entry, with the direction resolved for every space the entry leads into. If you only need how many people are in a space right now, use occupancy instead: entry sensors already drive its headcount.

> **REQUIRES THE ENTRY CROSSINGS API FEATURE**
>
> The Entry Crossings API feature is enabled by XY Sense per organisation and can vary per building and floor. A query that covers any floor without it fails rather than leaving that floor out, so if only some floors have it, filter to them with `floorIds` or `buildingIds`. If your queries return an access-disabled error, your XY Sense representative can enable it. See Feature enablement.

1. Find the entries on a floor and the spaces they lead into

An entry is a line drawn across a doorway or opening, watched by an entry sensor. `shape` is the line, in centimetres from the top-left of the floor plan. `entryFloorSpaces` lists the spaces the entry leads into, so you can label crossings by space name and decide which entries to poll.

Open in API Explorer

Example query

```
query EntryCrossingEntries($floorId: ID!) {
  floor(id: $floorId) {
    id
    name
    entrySensorInstallations(isActive: true) {
      id
      name
      entries {
        id
        name
        shape
        entryFloorSpaces {
          floorSpaceId
          floorSpace {
            name
          }
        }
      }
    }
  }
}
```

Example variables

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

Example response


```
{
  "data": {
    "floor": {
      "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
      "name": "Level 12",
      "entrySensorInstallations": [
        {
          "id": "0c9d4b7e-2f61-4a8e-b3d5-6e1f9a2c7b40",
          "name": "Lobby sensor",
          "entries": [
            {
              "id": "5f3c8e21-7a4d-4c1b-9e62-0d8b7a1f4c35",
              "name": "Meeting room 12.04 door",
              "shape": [
                { "x": 1820, "y": 940 },
                { "x": 1910, "y": 940 }
              ],
              "entryFloorSpaces": [
                { "floorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1", "floorSpace": {
"name": "Meeting room 12.04" } },
                { "floorSpaceId": "7f10b2c9-5e83-4d26-a1f4-3b9c8e6d2a17", "floorSpace": {
"name": "Level 12 open plan" } }
              ]
            }
          ]
        }
      ]
    }
  }
}
```


2. Fetch recent crossings

If you omit `from`, the API returns the last minute of crossings for the matching floors.

Open in API Explorer

Example query

```
query RecentEntryCrossings($floorIds: [ID!]) {
  entryCrossings(floorIds: $floorIds) {
    nextFrom
    data {
      id
      entryId
      floorId
      collectedDate
      lineDirection
      spaces {
        floorSpaceId
        direction
      }
    }
  }
}
```

Example variables

```
{
  "floorIds": [
    "a2961d46-971f-4ce1-a66e-7ee7bdb2a204"
  ]
}
```

Example response

```
{
  "data": {
    "entryCrossings": {
      "nextFrom": "2026-06-02T07:36:12.001Z",
      "data": [
        {
          "id": "8e4b1c02-9d37-4f5a-b6e1-2c7a9f0d3b58",
          "entryId": "5f3c8e21-7a4d-4c1b-9e62-0d8b7a1f4c35",
          "floorId": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
          "collectedDate": "2026-06-02T07:36:09.412Z",
          "lineDirection": "Forward",
          "spaces": [
            { "floorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1", "direction": "In" },
            { "floorSpaceId": "7f10b2c9-5e83-4d26-a1f4-3b9c8e6d2a17", "direction": "Out" }
          ]
        }
      ]
    }
  }
}
```

3. Read the direction of a crossing

Use `spaces[].direction` to tell whether a crossing went into or out of a space:

- `In` or `Out` **is resolved for you, per space.** A door between two spaces, like the meeting room door above, is `In` for one space and `Out` for the other in the same crossing, so counting by space never counts a crossing twice.
- **It uses the mapping at the time of the crossing.** If an entry is later mapped to different spaces, older crossings keep the direction they had when they happened.
- `spaces` **is empty when the entry is not mapped to any space.** `lineDirection` is then the only direction available.

`lineDirection` is `Forward` or `Backward` relative to the entry line itself. Crossings without a known direction are never recorded.

4. Poll for new crossings with a moving cursor

Persist `nextFrom` and send it as the next request's `from`. Filter with `floorIds`, `buildingIds` or `entryIds` to narrow the results. A `floorIds` filter matches crossings from sensors on that floor and crossings into spaces on that floor. Crossings are only recorded when the feature is enabled on the entry sensor's floor, so a crossing into a space on this floor from a sensor on a floor without the feature is never returned, and one from a sensor on a floor with it includes all its spaces, wherever they are.

Open in API Explorer

Example query

```
query EntryCrossingUpdates($floorIds: [ID!], $entryIds: [ID!], $from: Date!, $to: Date) {
  entryCrossings(floorIds: $floorIds, entryIds: $entryIds, from: $from, to: $to) {
    from
    to
    nextFrom
    data {
      id
      entryId
      floorId
      collectedDate
      lineDirection
      spaces {
        floorSpaceId
        direction
      }
    }
  }
}
```

Example variables

```
{
  "floorIds": [
    "a2961d46-971f-4ce1-a66e-7ee7bdb2a204"
  ],
  "entryIds": [
    "5f3c8e21-7a4d-4c1b-9e62-0d8b7a1f4c35"
  ],
  "from": "2026-06-02T07:30:00Z",
  "to": "2026-06-02T07:40:00Z"
}
```

Example response

```
{
  "data": {
    "entryCrossings": {
      "from": "2026-06-02T07:30:00Z",
      "to": "2026-06-02T07:40:00Z",
      "nextFrom": "2026-06-02T07:40:00.001Z",
      "data": [
        {
          "id": "8e4b1c02-9d37-4f5a-b6e1-2c7a9f0d3b58",
          "entryId": "5f3c8e21-7a4d-4c1b-9e62-0d8b7a1f4c35",
          "floorId": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
          "collectedDate": "2026-06-02T07:36:09.412Z",
          "lineDirection": "Forward",
          "spaces": [
            { "floorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1", "direction": "In" },
            { "floorSpaceId": "7f10b2c9-5e83-4d26-a1f4-3b9c8e6d2a17", "direction": "Out" }
          ]
        }
      ]
    }
  }
}
```

Things to handle when polling:

- **`from` and `to` are when a crossing reached XY Sense, not `collectedDate`.** Crossings a sensor sends late, for example after a network outage, appear in a later page than their `collectedDate` suggests. Always page with `nextFrom` rather than computing the next window from `collectedDate`.
- **The same crossing can be returned more than once.** Drop duplicates using `id`, which stays the same for a given crossing.
- **Delivery is best effort.** A crossing can occasionally be missing, so treat counts as close estimates rather than an exact audit.
- **Polling again within a second of `nextFrom` returns an empty page** with the same `nextFrom`. Wait and try again.

Request windows, retention and rate limits are in Limits and retention and Limits, versioning, and support.

Sensor connection status

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

Check which sensors are online

An offline sensor is not always an incident. Check the floor's `currentFloorStatus` first, because sensors on floors still being installed or reviewed are not yet expected to stay online.

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

Example response

```
{
  "data": {
    "floor": {
      "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
      "name": "Level 12",
      "currentFloorStatus": {
        "status": "Live",
        "startDate": "2026-05-14T00:00:00Z"
      },
      "entrySensorInstallations": [
        {
          "id": "b6f1c3d7-9e42-4a58-8c60-1d2e3f4a5b6c",
          "name": "North Entry",
          "entrySensor": {
            "id": "e41a7b52-6c39-4d81-9f0a-2b3c4d5e6f70",
            "externalId": "ENT-01",
            "isVirtual": false,
            "connection": {
              "isOnline": true,
              "isOnlineChangeDate": "2026-06-02T06:58:00Z",
              "source": "Heartbeat"
            }
          }
        }
      ],
      "spaceSensorInstallations": [
        {
          "id": "9c1d5e83-2b47-4f60-a915-6d8e0f1a2b37",
          "name": "Room 12A Sensor",
          "spaceSensor": {
            "id": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
            "externalId": "SS-01",
            "type": {
              "id": "7187aa0d-6cc9-4c5d-8661-3861d745e664",
              "name": "Comfort"
            },
            "connection": {
              "isOnline": false,
              "isOnlineChangeDate": "2026-06-02T06:50:00Z",
              "source": "Heartbeat"
            }
          }
        }
      ]
    }
  }
}
```

Find what a sensor affects

Which spaces a sensor influences depends on its type. Entry sensors affect occupancy indirectly, through their `entries` and the `entryFloorSpaces` those entries are associated with. Space sensors can be associated with more than one floor space, and can have individual capabilities ignored at the installation or per-space level.

Spaces served by 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": "b6f1c3d7-9e42-4a58-8c60-1d2e3f4a5b6c"
}
```

Example response


```
{
  "data": {
    "entrySensorInstallation": {
      "id": "b6f1c3d7-9e42-4a58-8c60-1d2e3f4a5b6c",
      "name": "North Entry",
      "entries": [
        {
          "id": "c72d9f81-3a64-4e59-8b12-5d6e7f8a9b0c",
          "name": "Inbound lane",
          "externalId": "ENTRY-IN-01",
          "entryFloorSpaces": [
            {
              "invertEntryDirection": false,
              "floorSpace": {
                "id": "6e0b4c85-1f27-4d93-a8b5-3c7d9e0f1a24",
                "name": "Room 12A"
              }
            }
          ]
        }
      ]
    }
  }
}
```


Spaces served by 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": "9c1d5e83-2b47-4f60-a915-6d8e0f1a2b37"
}
```

Example response


```
{
  "data": {
    "spaceSensorInstallation": {
      "id": "9c1d5e83-2b47-4f60-a915-6d8e0f1a2b37",
      "name": "Room 12A Sensor",
      "ignoredCapabilities": [
        {
          "id": "47c873e6-3c60-41f8-9d17-96e5639b3bde",
          "name": "Humidity"
        }
      ],
      "spaceSensor": {
        "id": "3f7a2c19-8d64-4b05-9e13-7a2b5c8d0e46",
        "externalId": "SS-01",
        "type": {
          "id": "7187aa0d-6cc9-4c5d-8661-3861d745e664",
          "name": "Comfort"
        }
      },
      "spaceSensorInstallationFloorSpaces": [
        {
          "ignoredCapabilities": [
            {
              "id": "47c873e6-3c60-41f8-9d17-96e5639b3bde",
              "name": "Humidity"
            }
          ],
          "floorSpace": {
            "id": "6e0b4c85-1f27-4d93-a8b5-3c7d9e0f1a24",
            "name": "Room 12A"
          }
        }
      ]
    }
  }
}
```

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.

1. Fetch the heatmap cells and the floor plan

`resolutionCms` sets the grid cell size in centimetres: larger values return fewer, coarser cells. The maximum aggregation window, and how far behind live heatmaps run, are in Limits and retention.

Open in API Explorer

Example query

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

Example variables

```
{
  "floorId": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
  "from": "2026-06-01T00:00:00Z",
  "to": "2026-06-01T23:59:59Z",
  "resolutionCms": 100
}
```

Example response


```
{
  "data": {
    "floor": {
      "id": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
      "name": "Level 12",
      "floorPlanUrl": "https://cdn.xysense.io/floorplans/level-12.png",
      "floorPlanMetadata": { "type": "Raster", "widthPixels": 4200, "heightPixels": 2800
},
      "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
      }
    ]
  }
}
```


2. Convert cell centres to image pixels

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 the coordinate system):

```
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,
}))
```

3. Render the overlay

- Load the floor plan from `floor.floorPlanUrl`. For a raster plan, `floorPlanMetadata.widthPixels` and `floorPlanMetadata.heightPixels` are that image's dimensions.
- `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.

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.

Get started

1. Follow the Quickstart to register a webhook and receive your first delivery.
2. Read Delivery and batching for the envelope, ordering, retry, and idempotency model your receiver has to handle.
3. Use Event payloads as the field-level reference for each event type.
4. Secure the endpoint with Securing your endpoint.
5. Join event identifiers to names and hierarchy with Resolve event data with GraphQL.

Supported event types

WEBHOOK TYPE (GRAPHQL)	PAYLOAD `TYPE`	DELIVERS
`OccupancyChanges`	`Occupancy`	Occupancy status and headcount changes per floor space
`FloorSightings`	`FloorSightings`	Anonymous sighting coordinates per floor, per time block
`SpaceSensorReadings`	`SpaceSensorReading`	Environmental readings per space sensor capability

The enum you pass to `addWebhook` and the `Type` value in the delivered payload are not the same string. Switch on the payload `Type` when routing events.

When to use webhooks

- Streaming ongoing occupancy, sighting, or sensor-reading changes into another system.
- Triggering building automation or space-availability updates on change rather than on a timer.

Use GraphQL instead for reference data, backfills, reconciliation after an outage, and analytics or historical queries.

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). Size your receiver for this up front, and acknowledge batches quickly while processing asynchronously. Over 90% of occupancy events are headcount-only changes, so if you only need occupied/vacant state the `ExcludeHeadcountChanges` option reduces volume dramatically.

Test your receiver

If your floor is in dev mode, you can generate synthetic occupancy and floor-sighting traffic. See Webhook test events.

Quickstart

This page takes you from nothing to a webhook delivering events to your endpoint.

Before you start

- Your endpoint must be reachable over **HTTPS** and respond within **10 seconds**. See Delivery and batching.
- 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 cannot register a webhook, your XY Sense representative can enable the feature. See Feature enablement.
- Registering via GraphQL needs `WritePublicApi` on the API key. See the GraphQL quickstart.

1. Register the webhook

Register via the Admin Console at `https://app.xysense.io/settings/webhooks`, or programmatically with `addWebhook`.

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": {
      "id": "1c9c1e7a-38f6-4a9a-8f4c-6a1f0f2d3e4b",
      "name": "Occupancy stream",
      "endpoint": "https://example.com/hooks/xy",
      "enabled": true,
      "webhookType": "OccupancyChanges"
    }
  }
}
```

The signature secret cannot be read back after it is set, so store it when you create the webhook.

2. Handle the first delivery

Deliveries are `POST` requests with `Content-Type: application/json; charset=utf-8`. The body is a **batch envelope**, not a bare array of events:

```
{
  "Id": "3f1b8f0e-2c44-4a1d-9f52-6b3f0a2d1c77",
  "Type": "Occupancy",
  "CustomerId": "2b7f9e10-4c3a-4b8e-9a11-7d2e5f6c8b90",
  "Created": "2026-06-02T04:17:40Z",
  "Sent": "2026-06-02T04:17:41Z",
  "Data": [
    {
      "FloorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1",
      "OccupancyStatus": "CurrentlyOccupied",
      "Headcount": 2,
      "CollectedDate": "2026-06-02T04:17:39.1066221Z",
      "OccupancyStatusChangeDate": "2026-06-02T04:17:39.1066221Z",
      "PreviousOccupancyStatus": "RecentlyOccupied"
    }
  ]
}
```

Iterate `Data` and return `2xx` to acknowledge the whole batch. Field names are **PascalCase**. See Delivery and batching for the full envelope reference.

3. Verify it is working

- Inspect delivery metrics with the `webhook(id:)` query.
- If your floor is in dev mode, generate traffic on demand with Webhook test events.

4. Harden it

- Add request signing or custom headers.
- Make processing idempotent — delivery is at-least-once.

- Cache reference data and resolve identifiers via GraphQL rather than querying per event.

Troubleshooting

SYMPTOM	WHERE TO LOOK
Registration rejected for a webhook type	The matching API feature is not enabled on any floor. See Feature enablement.
`You are not authorized to access the '...' field`	The API key needs `WritePublicApi`, in addition to your user account's permissions.
No deliveries arriving	Check `enabled` on the webhook, and its delivery metrics via `webhook(id:)`. A webhook that fails continuously is disabled automatically.
Deliveries stopped after an outage	The queue holds up to 1,000,000 events and retries for 24 hours. If your receiver is far behind, use `resetWebhookConsumer` and backfill the gap through GraphQL.
Receiver only sees the first event	The body is an envelope object, not an array. Iterate `Data`.
Timeouts under load	Deliveries time out after 10 seconds per attempt. Acknowledge the batch and process asynchronously.

Delivery and batching

Request format

Each delivery is an HTTPS `POST` to your endpoint with `Content-Type: application/json; charset=utf-8`, plus any custom headers and the `X-XY-Signature` header you have configured. See Securing your endpoint.

Each attempt has a **10 second timeout**. Anything other than a `2xx` response — including a timeout or a connection failure — is treated as a failed delivery and retried.

The batch envelope

The request body is a **JSON object wrapping the events**, not a bare array. Field names are **PascalCase**.

```
{
  "Id": "3f1b8f0e-2c44-4a1d-9f52-6b3f0a2d1c77",
  "GroupId": "0b26f8f3-5d1c-4c6a-9d3e-1a2b3c4d5e6f",
  "Type": "Occupancy",
  "CustomerId": "2b7f9e10-4c3a-4b8e-9a11-7d2e5f6c8b90",
  "Created": "2026-06-02T04:17:40Z",
  "Sent": "2026-06-02T04:17:41Z",
  "Data": [
    {
      "FloorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1",
      "OccupancyStatus": "CurrentlyOccupied",
      "Headcount": 2,
      "CollectedDate": "2026-06-02T04:17:39.1066221Z",
      "OccupancyStatusChangeDate": "2026-06-02T04:17:39.1066221Z",
      "PreviousOccupancyStatus": "RecentlyOccupied"
    }
  ]
}
```

FIELD	TYPE	NOTES
`Id`	string (GUID)	Batch identifier. See Idempotency for when it is stable across retries.
`GroupId`	string	The location the batch was grouped by. **Only present when `GroupByLocation` is enabled**; the field is omitted otherwise.
`Type`	string	`Occupancy`, `FloorSightings`, or `SpaceSensorReading`. Note these differ from the GraphQL `webhookType` enum values.
`CustomerId`	string (GUID)	Your organisation. Useful when several XY Sense organisations post to one receiver.
`Created`	string (ISO8601)	UTC time the batch started filling.
`Sent`	string (ISO8601)	UTC time this attempt was sent. Changes on every retry. Compare with `Created` to measure delivery delay.

FIELD	TYPE	NOTES
`Data`	array	The events. See Event payloads.

Null fields are omitted rather than sent as `null`, so treat any field documented as optional as absent rather than null.

Ordering and acknowledgement

- Events are queued **per webhook** and delivered in order.
- A `2xx` response acknowledges the **whole batch**. There is no partial acknowledgement.
- Because ordering is preserved, a batch your receiver keeps rejecting **blocks everything behind it**. Return `2xx` once you have durably accepted a batch and handle parse or business errors out of band, rather than rejecting the delivery.

Batching

- A batch is sent when either `maxBatchSize` (100–1000 events) or `maxBatchDelayMs` (1 second–5 minutes) is reached.
- A single batch can contain **multiple events for the same entity** — for example two headcount changes for one space within the same window. Process `Data` in order rather than keeping the first or last match per entity.

Idempotency

Delivery is **at-least-once**. A batch that times out or fails after your receiver has already committed it will be sent again, so processing must be idempotent.

`Id` identifies the batch, but its stability across retries depends on configuration:

- **Default (`GroupByLocation` off):** `Id` is generated once per batch and is stable across all retry attempts. You can dedupe on it.
- **`GroupByLocation` on:** a fresh `Id` is generated for each retry attempt, so it will **not** catch a redelivered batch.

For a scheme that works in both cases, dedupe on the event itself: the entity identifier plus its timestamp (`FloorSpaceId` + `CollectedDate`, `FloorId` + `CollectedDate`, or `SpaceSensorId` + `CapabilityName` + `CollectedAt`) is unique per event.

Retries and failure handling

- Failed attempts are retried with exponential backoff (2 seconds, 4, 8, ...), capped at **10 minutes** between attempts.
- Retries continue for up to **24 hours**.
- If an endpoint is still failing after 24 hours, the webhook is **stopped and set to `enabled: false`**. Its queued events are not delivered. Re-enable it with `updateWebhook` once the endpoint is healthy, then backfill the gap with the corresponding GraphQL history query, subject to its retention window.

Recovering a backlogged receiver

If your receiver has fallen a long way behind, use `resetWebhookConsumer` to skip to the latest queued event, then backfill the gap through GraphQL polling queries. See Resolve event data with GraphQL.

Compression

Set `compressionType` on the webhook to `Gzip` or `Deflate` to have request bodies compressed. Your receiver must decompress the body, and must sign against the **decompressed** bytes when validating signatures. Compression is off by default.

Configuration options

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

KEY	DEFAULT	APPLIES TO	EFFECT
`ExcludeHeadcountChanges`	`false`	`OccupancyChanges`	Drop events where only the headcount changed. Cuts volume by roughly 90%.
`GroupByLocation`	`false`	Occupancy, space sensors	Send one batch per location and populate `GroupId`. Note the `Id` caveat.
`OnlyComfortReadings`	`false`	`SpaceSensorReadings`	Deliver only comfort readings.
`EnableBatching`	`true`	`FloorSightings` only	Set to `false` to receive one event per request. Ignored for occupancy and space-sensor webhooks, which are always batched.

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

Service limits

LIMIT	VALUE
Batch size	100 to 1000 events
Batch delay	1 second to 5 minutes
Per-attempt timeout	10 seconds
Retry backoff cap	10 minutes
Total retry window	24 hours, after which the webhook disables
Queue depth per webhook	1,000,000 events

Managing webhooks

Everything available 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, compression, or config. Set `enabled: false` to pause delivery.
- `deleteWebhook`: remove a webhook.
- `resetWebhookConsumer`: skip to the latest queued event.
- `resetWebhookMetrics`: clear delivery metrics.

Event payloads

Every delivery wraps its events in a batch envelope. The shapes below are the objects inside the envelope's `Data` array. Field names are **PascalCase**, timestamps are UTC ISO 8601, and null fields are omitted.

To turn the identifiers in these payloads into names, hierarchy, and floor plans, see Resolve event data with GraphQL.

Occupancy

Delivered by an `OccupancyChanges` webhook. Envelope `Type` is `Occupancy`.


```
{
  "Id": "3f1b8f0e-2c44-4a1d-9f52-6b3f0a2d1c77",
  "Type": "Occupancy",
  "CustomerId": "2b7f9e10-4c3a-4b8e-9a11-7d2e5f6c8b90",
  "Created": "2026-06-02T04:17:40Z",
  "Sent": "2026-06-02T04:17:41Z",
  "Data": [
    {
      "FloorSpaceId": "ac52e43d-7c21-4464-ab21-d0ace07a94b1",
      "OccupancyStatus": "CurrentlyOccupied",
      "Headcount": 2,
      "CollectedDate": "2026-06-02T04:17:39.1066221Z",
      "OccupancyStatusChangeDate": "2026-06-02T04:17:39.1066221Z",
      "PreviousOccupancyStatus": "RecentlyOccupied"
    }
  ]
}
```


FIELD	TYPE	NOTES
`FloorSpaceId`	string (GUID)	The desk, room, or zone. Resolve with `floorSpace(id:)`.
`OccupancyStatus`	string	`CurrentlyOccupied`, `RecentlyOccupied`, `NotOccupied`, or `Unknown`.
`Headcount`	integer	People currently detected in the space.
`CollectedDate`	string	When the underlying sightings were collected.
`OccupancyStatusChangeDate`	string	When `OccupancyStatus` last changed. Equals `CollectedDate` on a status change.
`PreviousOccupancyStatus`	string	Optional. Absent when there is no prior status.

`OccupancyStatus` follows the same 30/60-second state model as the GraphQL field, and `FloorSpaceId` is a space rather than a sensor. See How occupancy is determined and How sensors map to spaces.

An event is emitted when either the status or the headcount changes. Over 90% of events are headcount-only changes, where `OccupancyStatus` equals `PreviousOccupancyStatus`; set `ExcludeHeadcountChanges` to suppress them.

Floor sightings

Delivered by a `FloorSightings` webhook. Envelope `Type` is `FloorSightings`.


```
{
  "Id": "8a7d5e2c-9f11-4b3a-88c0-2e4f6a1b7d93",
  "Type": "FloorSightings",
  "CustomerId": "2b7f9e10-4c3a-4b8e-9a11-7d2e5f6c8b90",
  "Created": "2026-06-02T07:36:12Z",
  "Sent": "2026-06-02T07:36:13Z",
  "Data": [
    {
      "FloorId": "a2961d46-971f-4ce1-a66e-7ee7bdb2a204",
      "CollectedDate": "2026-06-02T07:36:11.964156Z",
      "Sightings": [
        { "X": 1842, "Y": 971 },
        { "X": 2510, "Y": 1164 },
        { "X": 2732, "Y": 1189 }
      ],
      "Headcount": 3
    }
  ]
}
```


FIELD	TYPE	NOTES
`FloorId`	string (GUID)	Resolve with `floor(id:)` for the floor plan, pixel size, and scale.
`CollectedDate`	string	Start of the time block the sightings belong to.
`Sightings`	array	Anonymous positions, in **centimetres from the top-left of the floor plan**.
`Headcount`	integer	Convenience count, always equal to `Sightings.length`.

Each event is a full snapshot of the floor for that time block, not a delta. An empty `Sightings` array means nobody was detected.

Sightings are floor-level and already merged across sensors, so a position is not attributable to a sensor. To place them on a floor plan, see the coordinate system.

Space sensor readings

Delivered by a `SpaceSensorReadings` webhook. Envelope `Type` is `SpaceSensorReading`.

```
{
  "Id": "c4e2b1a0-7d68-4f5c-9a3b-1e2d3c4b5a69",
  "Type": "SpaceSensorReading",
  "CustomerId": "2b7f9e10-4c3a-4b8e-9a11-7d2e5f6c8b90",
  "Created": "2026-06-02T07:36:12Z",
  "Sent": "2026-06-02T07:36:13Z",
  "Data": [
    {
      "Value": "22.4",
      "CollectedAt": "2026-06-02T07:36:11.964156Z",
      "SpaceSensorId": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3",
      "CapabilityName": "Temperature"
    },
    {
      "Value": "48.2",
      "CollectedAt": "2026-06-02T07:36:11.964156Z",
      "SpaceSensorId": "8bb1ba0b-6ea8-4635-86ef-3d57a7de4df3",
      "CapabilityName": "Humidity"
    }
  ]
}
```


FIELD	TYPE	NOTES
`Value`	string	Always a string, including for numeric readings. Parse according to the capability.
`CollectedAt`	string	When the sensor took the reading. Note this field is `CollectedAt`, not `CollectedDate`.
`SpaceSensorId`	string (GUID)	Resolve with `spaceSensor(id:)`.
`CapabilityName`	string	The measured property, for example `Temperature` or `Humidity`. Discover the set via `spaceSensor(id:)`.

One event is emitted per capability, so a single sensor reporting several capabilities produces several events in the same batch.

Securing your endpoint

Endpoints must be reachable over HTTPS. You can additionally authenticate deliveries with custom headers, request signatures, or both.

Custom headers

Set `additionalHeaders` on the webhook to send one or more fixed headers, such as a shared API token, and validate them on receipt. Headers are sent verbatim on every delivery.

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

Custom headers prove the caller knows a shared secret. They do not prove the body is unmodified — use signatures for that.

Request signatures

Set `signatureSecret` on the webhook and every delivery includes an `X-XY-Signature` header:

```
X-XY-Signature:
t=1780373861,s=76807D53FD29B7D00510CE1CCF10D963BB46D560E184AA0A2093192A8E0E92B1
```

The scheme, precisely:

- `t` is the send time as **Unix epoch seconds**. It changes on every retry attempt.
- The signed content is the UTF-8 bytes of `{t}.` followed by the **raw, decompressed request body**.
- `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.

> **SIGN THE BYTES YOU RECEIVED**
>
> Sign the raw body exactly as it arrived, not a re-serialized copy of the parsed JSON. Round-tripping through a JSON parser changes whitespace, key order, and number formatting, and the signature will not match. If you enabled compression, decompress first and sign the decompressed bytes.

Validating a delivery

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.
4. Reject stale timestamps. Five minutes is a reasonable tolerance for replay protection.

Worked example

```
import hashlib, hmac

secret = b"my-signature-secret-of-at-least-64-characters-abcdefghijklmnopqrs"

# the raw bytes as received, not a re-serialized dict
raw_body = (
    b'{"Id":"3f1b8f0e-2c44-4a1d-9f52-6b3f0a2d1c77","Type":"Occupancy",'
    b'"CustomerId":"2b7f9e10-4c3a-4b8e-9a11-7d2e5f6c8b90",'
    b'"Created":"2026-06-02T04:17:40Z","Sent":"2026-06-02T04:17:41Z",'
    b'"Data":[{"FloorSpaceId":"ac52e43d-7c21-4464-ab21-d0ace07a94b1",'
    b'"OccupancyStatus":"CurrentlyOccupied","Headcount":2}]}'
)
t = "1780373861"

expected = hmac.new(secret, t.encode() + b"." + raw_body,
hashlib.sha256).hexdigest().upper()
# 76807D53FD29B7D00510CE1CCF10D963BB46D560E184AA0A2093192A8E0E92B1

assert hmac.compare_digest(expected, received_signature.upper())
```

Example request

```
POST https://<your endpoint>
Content-Type: application/json; charset=utf-8
X-Partner-Token: mytoken
X-XY-Signature:
t=1780373861,s=76807D53FD29B7D00510CE1CCF10D963BB46D560E184AA0A2093192A8E0E92B1

{"Id":"3f1b8f0e-...","Type":"Occupancy","Data":[ ... ]}
```

Rotating a secret

Call `updateWebhook` with a new `signatureSecret`, or `clearSignatureSecret: true` to stop signing altogether. There is no overlap window: the next delivery is signed with the new secret, so accept either the old or the new value on your receiver during the changeover.

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, building, and external IDs if you set them
`FloorSightings`	`FloorId`	`floor(id:)`	Floor name, building, floor plan URL, pixel size, and scale
`SpaceSensorReadings`	`SpaceSensorId`	`spaceSensor(id:)`	Sensor name, type, installations, floors, and floor spaces

Webhook payload fields are PascalCase; GraphQL response fields are camelCase. See Event payloads.

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, `floorPlanMetadata` (representation and, for raster plans, pixel size), 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
    floorPlanMetadata {
      type
      widthPixels
      heightPixels
    }
    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",
      "floorPlanMetadata": { "type": "Raster", "widthPixels": 4200, "heightPixels": 2800
},
      "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
  }
}
```

What the generators do

- 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.