ZephyrAB API Manual

ZephyrAB 0.1.0 (preview) · September 2026

Table of contents

1. About this API manual

ZephyrAB 0.1.0 (preview) · September 2026

This manual is for developers integrating with a ZephyrAB deployment: provisioning automation, helpdesk tooling, billing systems, and mail clients. It documents every operation the platform serves, what each one requires, and the refusals worth knowing before you hit them.

ZephyrAB exposes three API surfaces. They live on different origins on purpose, and the separation is part of the security design — do not proxy one surface onto another origin.

Surface	Base URL (convention)	Auth	What it is for
Admin (control plane)	`https://admin.example.com/admin/api`	OAuth2 bearer token	Tenants, domains, accounts, plans, delegations, queue operations, settings, webhooks, license
Self-service	`https://mail.example.com`	HTTP Basic, a session token, or an identity provider's bearer token	Account holders managing their own account: password, recovery, 2FA, app passwords, filters, encryption keys, file links
Downloads	`https://files.example.com`	Capability URL (the token in the link)	Serving link-attachment files to anyone holding the link

Base URLs are deployment-specific. This manual uses `admin.example.com`, `mail.example.com`, and `files.example.com` as placeholders; substitute your deployment's hostnames.

Conventions behind the split:

- **The admin API is mounted at `/admin/api` on the admin origin and is deliberately NOT served on the mail hostname.** A request for `/admin/api/*` on the mail host is redirected away (typically a 308 to the apex) rather than proxied. Automation must target the admin origin directly.
- **Self-service routes (`/signup`, `/recover`, `/account/*`) live on the mail origin**, same origin as webmail and JMAP, so a browser session can call them without CORS.
- **Downloads live on a separate registrable domain.** Link attachments serve attacker-authored bytes — files a stranger uploaded or mailed. Keeping them off the mail origin means those bytes can never run same-origin with a webmail session. The download host enforces this in code as well as in the proxy: a valid download token presented with the mail host's `Host` header is refused.

Mail protocols (JMAP, IMAP, POP3, SMTP submission, ManageSieve, CalDAV/CardDAV) are served from the mail origin and its standard ports; section 6 covers them.

The admin API is described by an OpenAPI 3.1 document (`docs/spec/admin-api.openapi.yaml` in the source tree). That document is the authority for request and

response shapes. Where the running server deliberately deviates from it, this manual says so at the endpoint.

2. Authentication

2.1 POST /oauth/token

The token endpoint (operation `issueToken`) is the only admin route served without a bearer token. It is form-encoded (`application/x-www-form-urlencoded`), and it supports two grants.

Client credentials (RFC 6749 §4.4) — for machine automation:

```
curl -s https://admin.example.com/admin/api/oauth/token \
  -d grant_type=client_credentials \
  -d client_id="CLIENT_ID" \
  -d client_secret="CLIENT_SECRET"
```

```
{
  "access_token": "OPAQUE_TOKEN",
  "token_type": "Bearer",
  "expires_in": 3600,
  "scope": "platform:admin"
}
```

Password grant (RFC 6749 §4.3) — for a person. The username is a mail address on the platform; the token's principal is that account's canonical address, and everything the token can do comes from delegations stored against that address (plus `platform:admin` if the deployment names the account as a platform operator).

```
curl -s https://admin.example.com/admin/api/oauth/token \
  -d grant_type=password \
  -d username="admin@customer.example" \
  -d password="ACCOUNT_PASSWORD"
```

The password-grant response additionally carries a `principal` field (the canonical address) and reports in `scope` the scopes the identity actually holds right now, read from the delegation store — an account with no delegations gets a perfectly valid token that can do nothing.

Assertion grant (RFC 7523 §2.1) — for a person whose identity lives in an identity provider and who therefore has **no ZephyrAB password**. Before this grant existed, such an operator could read their mail and reach their own settings but could not obtain an admin token at all, so a federated deployment had to keep a password on its administrators purely so they could sign in.

```
curl -s https://admin.example.com/admin/api/oauth/token \
  --data-urlencode grant_type=urn:ietf:params:oauth:grant-type:jwt-
```

```
bearer \
  --data-urlencode assertion="$ID_TOKEN"
```

The assertion is an identity token from the provider configured in `ZEPHYR_OAUTH_ISSUER` / `ZEPHYR_OAUTH_AUDIENCE` / `ZEPHYR_OAUTH_JWKS_URL`, verified exactly as the mail protocols verify one. Without a provider configured the grant answers `400 unsupported_grant_type` naming those three variables — that is deployment configuration rather than anything about a user, so it says so plainly.

The assertion proves who you are; it does not become the session. ZephyrAB verifies it, takes the identity, and mints its own opaque token. So authority is still read from delegations on every request — revoking a grant takes effect on the next call rather than when the provider's token expires — and the provider's token lifetime never becomes an admin session lifetime.

The response carries `principal` and `scope` exactly as the password grant does. There is no second-factor step: the provider is the authentication authority, and ZephyrAB does not stack its own TOTP on top of a federated sign-in.

Form fields accepted: `grant_type`, `client_id`, `client_secret`, `username`, `password`, `assertion`, `scope`. Any other grant type answers `400 unsupported_grant_type` naming the three supported grants.

Every failure is answered identically — a wrong password, an unknown address, a locked account, a forged assertion, and an identity with no mailbox here all return the same status and body. That is deliberate: anything else makes the endpoint an oracle for which administrator addresses exist. The audit log distinguishes them; the caller cannot.

Two-factor accounts and the password grant. The token endpoint itself has no TOTP field. If the account has a second factor enrolled, a raw password is refused exactly like a wrong one. The supported path is: mint a session token first with `POST /account/session` on the mail origin (which takes the TOTP `code`), then present that session token in the `password` field here. A session token is a full-strength credential — it was minted after both factors — and is accepted anywhere a password is.

2.2 Token properties

- **Lifetime is 3600 seconds.** The response's `expires_in` is authoritative. Fetch a new token rather than trying to refresh; there is no refresh grant.
- **Tokens are opaque** — 256 bits from the OS random source, stored by their SHA-256 so there is no signing key to manage and no comparison to get wrong. Nothing about the token is parseable; do not try to read a scope or an expiry out of it.
- **A token is valid on every node in the cell.** Tokens live in the cell's FoundationDB, so an admin API behind a load balancer needs no sticky sessions: obtain a token from one frontend and spend it on any other. They also survive a restart of the node that issued them.
- **Rotating the client secret revokes every token minted against it, fleet-wide and at once.** A machine token records a fingerprint of the credential that minted it and is

checked against the node's current configuration on every request. So changing `ZEPHYR_ADMIN_CLIENT_SECRET` is a real revocation — no restart, no waiting out an hour of TTL. Plan for it: a running script's token stops working the moment an operator rotates.

- **The outstanding-token cap is 1024 per principal**, and past it that principal's own earliest-expiring token is evicted. One client in a retry loop can therefore only push out its own tokens, never another operator's.
- **A node with no cluster store configured keeps tokens in its own memory instead.** That fails safe — a node-local store can only ever honour fewer tokens, never more — but it brings back the old behaviour, so on such a deployment issue and spend against the same host. The server logs which store it is using at startup.
- **Token responses are served with `Cache-Control: no-store`.** Do not cache them anywhere shared.

2.3 Scope down-scoping

The optional `scope` parameter (space- or comma-delimited) narrows the token below what the client is granted — least privilege for a one-off script:

```
curl -s https://admin.example.com/admin/api/oauth/token \
  -d grant_type=client_credentials \
  -d client_id="CLIENT_ID" \
  -d client_secret="CLIENT_SECRET" \
  -d scope="helpdesk:read"
```

Asking for **less** is allowed. Asking for a scope the client does not hold, or a name the taxonomy does not define, is `400 invalid_scope` — reachable only after the credential was accepted, so it is safe to be specific. The granted scope is always echoed in the response, because a down-scoped token is otherwise indistinguishable from a full one.

The mask narrows **scopes only**. Subjects (which tenant, which domain) come from the client's configuration and its stored delegations, and are re-read on every request — a token cannot pin, widen, or outlive them.

2.4 The scope taxonomy

Every operation's requirement is an ANY-OF list: holding one listed scope is enough.

Scope	Meaning
`platform:admin`	Full platform administration. **Implies every other scope** — an operation listing only narrower roles is also satisfied by `platform:admin`. May only ever be granted platform-wide; binding it to a tenant or domain is refused.
`reseller:admin`	Manage owned tenants
`tenant:admin`	Manage one tenant

`domain:admin`	Manage assigned domains
`helpdesk:read`	Read-only support access
`helpdesk:write`	Support actions (password reset; ticket reference required)
`compliance:read`	Audit / hold read access
`compliance:write`	Restores, holds
`mailops:read`	Queue / trace read
`mailops:write`	Queue actions (cancel, retry, pause)

2.5 Subjects: a grant is scope + subject

A grant is a `(scope, subject)` pair. The subject is the whole platform, one tenant, or one domain. Every operation that names a tenant, domain, account, or delegation resolves that resource to its **stored** owner and requires a grant whose subject covers it. Two consequences:

- **Cross-subject denial is the resource's own 404**, byte-identical to the answer for a resource that does not exist. A caller who may not touch tenant B cannot use the difference between "yours" and "nobody's" to enumerate the platform. The audit log records which it was; the response never does.
- **Cross-tenant listings require a platform-wide grant** whatever the scope. `GET /tenants`, `GET /domains`, and the plan catalogue span tenants; a tenant- or domain-bound grant reaching them gets an honest `403` saying a platform-wide grant is required (there is no individual resource to hide behind a 404 there).

Subjects are live; the scope mask is pinned. The token pins only the down-scope mask its holder asked for. Delegations are read from storage on every request, so:

- Revoking a delegation takes effect on the caller's **next request**, not when their token expires.
- A delegation **added** after a token was issued is picked up by that token immediately.

Delegations are created with `POST /tenants/{tenantId}/delegations` (section 4.11). The creator must already hold the scope it is handing out, on a subject covering the target — the route cannot be used to climb. Platform subjects are refused outright: platform-wide authority is configured on the server, never delegated at runtime.

2.6 Failed-token rate limiting

Failed token requests are rate-limited per source IP (default: 10 failures per 15-minute window). A throttled request is answered **exactly like a bad credential** — `401` with `invalid_client` (or `invalid_grant` for the password grant), same body, no `Retry-After`. This is deliberate: a `429` would tell a credential guesser when its guesses stopped being evaluated. If your automation starts getting 401s with a credential you believe is good, stop, wait out the window, and check the credential out of band rather than retrying in a loop.

The same one-answer rule applies inside the password grant: wrong password, unknown address, and locked account are indistinguishable to the caller.

2.7 Using the token

Send it as a bearer header on every admin request:

```
curl -s https://admin.example.com/admin/api/tenants \
  -H "Authorization: Bearer OPAQUE_TOKEN"
```

Refusal shapes you will meet:

- **401** — missing, malformed, unknown, or expired token. One indistinguishable answer for all four, with `WWW-Authenticate: Bearer`.
- **403 `insufficient scope`** — authenticated, but no listed scope. Deliberately distinct from 401: re-fetching a token would not help. The response names the required scopes, and `WWW-Authenticate` carries `error="insufficient_scope"`.
- **404** — the resource does not exist, **or** it exists and your grant's subject does not cover it. You cannot tell which.
- **503** — the node has no admin credential configured at all (fail-closed).

2.8 Federated identity: bearer tokens from an external provider

Everything above is the **admin** control plane's own credential. This subsection is a different thing entirely: where the deployment is pointed at an external identity provider, ZephyrAB will accept **that provider's** JWTs as proof of who a mailbox holder is. The two never mix — a provider's token is not an admin token, and no down-scoping, delegation or subject rule from §2.1–2.7 applies to it.

ZephyrAB is an OAuth **resource server** and nothing more. It validates tokens somebody else issued and maps a claim to an account. There is no `/authorize`, no consent screen, no refresh rotation, no client registry and **no token-exchange or impersonation endpoint**. Two consequences an integrator should plan around: against a standalone deployment with no external provider a client cannot do OAuth at all (use an application password), and there is no way to ask this server for a token that speaks for an arbitrary mailbox.

Where a bearer token is accepted.

Surface	How it is carried
JMAP (`/jmap`, `/.well-known/jmap`, upload, download, EventSource, WebSocket)	`Authorization: Bearer <jwt>` — RFC 6750, which RFC 8620 §8.2 names as JMAP's OAuth carrier
Self-service `/account/*` (section 5)	`Authorization: Bearer <jwt>` — every authenticated route, `POST /account/session` included
IMAP · POP3 · SMTP submission · ManageSieve	SASL `XOAUTH2` or `OAUTHBEARER` (RFC 7628)

CalDAV / CardDAV	**Not accepted.** Basic only — password, session token, or a `dav`-scoped application password
Admin API	**Not accepted**, deliberately. It is a control plane for operators, not a mailbox surface, and its authorization model is ZephyrAB's own

Bearer is decided by the scheme, once, and is never a fallback from Basic. A token that fails verification is not then tried as a password. Without that rule a client misconfigured against the wrong provider would drive its user's account into the lockout with a credential that was never a password. The converse holds too: a password is never tried as a token.

`POST /account/session` takes a bearer token, and that is why there is no impersonation endpoint. An application that has just authenticated somebody through the provider can hand that person's own token to the session route and get back an ordinary 12-hour mail session — the same credential webmail uses, good on JMAP, IMAP and the admin password grant. The requirement people usually reach for a token-exchange endpoint to satisfy ("mint a session for the account we just authenticated") is therefore met with the **user's own** token, and nothing in the system ever needs the ability to mint a session for an arbitrary mailbox. That is the capability worth not building, and it is not built.

No second factor is demanded on that path. The provider is the authentication authority and applies whatever multi-factor policy the organisation set; stacking ZephyrAB's own TOTP on top would refuse every federated user who had ever enrolled, with no way to satisfy it. The same reasoning means our TOTP is never additionally required of a bearer caller anywhere.

Operator configuration — issuer, audience, JWKS URL, claim name — is in `docs/runbooks/identity-federation.md` rather than here. Two properties of it that change how a client should behave: setting some of those variables but not all is refused **at startup**, so a deployment cannot half-enable federation; and the JWKS cache is install-never-clear, so an identity-provider outage stops new sign-ins but does not invalidate tokens already issued.

The claim carrying the mailbox is configurable. The operator sets `ZEPHYR_OAUTH_ADDRESS_CLAIM` (default `email`), and the value in that claim must contain an `@` or the token is refused. Aliases resolve to the canonical account exactly as they do everywhere else. Two rules worth designing your broker around: an address the provider itself marks `email_verified: false` is not an identity, and **accounts are never auto-provisioned from a token** — a claim naming no mailbox here is refused, so create the mailbox first.

On the SASL surfaces the username travels alongside the token, and the two must name the **same account** — resolved through the directory, not compared as text. On the HTTP surfaces there is no second name, so the account is whatever the verified claim says.

Refusals are one answer. Wrong issuer, wrong audience, expired, bad signature, unknown key, unverified address, no such account — all of them answer the surface's ordinary authentication failure, with the cause in the server's log and nowhere else. Telling a caller which check their forged token failed is a tutorial in forging the next one. Note that the JMAP 401 names `Basic` in its `WWW-Authenticate` challenge whether or not a provider is configured; send the bearer token regardless. The self-service surface adds `Bearer` to its challenge only when a provider is configured, on the same principle that stops the mail protocols advertising a mechanism they cannot complete.

Re-authentication on sensitive self-service operations

Some `/account/*` operations demand the caller prove themselves a second time. **Five of them accept a bearer token's freshness as that proof**: change password, set a recovery address, mint an application password, enrol two-factor, disable two-factor. Revoking an application password is deliberately not one of them — cutting off access is always allowed.

Under Basic that second proof is `currentPassword` in the body: a second value, verified against the directory. **A bearer token has no second value.** It is the only credential in the request, so verifying it again would prove exactly what the request's own authentication already proved, and accepting it on that basis would silently downgrade "prove it is you" to "hold a session".

What substitutes is **freshness**. The token must carry an authentication that happened within the deployment's window (`ZEPHYR_OAUTH_REAUTH_MAX_AGE_SECS`, default **300 seconds**, plus the same 60-second clock-skew leeway the expiry check uses):

- `auth_time` (OIDC Core §2) is read first and is what your broker should set.
- `iat` is the fallback, because many providers omit `auth_time` unless `max_age` was requested. Prefer `auth_time`: a provider that refreshes silently mints a brand-new `iat` for a session somebody started days ago, and reading `iat` alone would call that a recent authentication.
- **A token carrying neither is refused.** The unknown case must not be the permissive one.

A token that is fine but too old answers **`401`** with the self-service two-field body:

```
{
  "error": "reauthenticationRequired",
  "description": "this action needs a recent sign-in. Return to your
identity provider and authenticate again (OpenID Connect
`max_age=300`), then retry."
}
```

Note the shape: `{error, description}`, the **self-service** convention of section 5 — not the admin API's RFC 9457 `{title, detail}`. Match on `error`.

This refusal is deliberately specific where every other refusal on this path collapses to one answer, and the reason is that whoever reads it has already authenticated: they are the

legitimate user, and this is the one thing they need in order to succeed. **The right client response is to send the person back to the provider with `max_age=<n>` and retry with the token that comes back** — not to retry the same token, which cannot start working, and not to fall back to asking for a password the user may not have.

KNOWN LIMIT — two groups of routes re-check a password and have no freshness path. The at-rest **encryption** routes (opt in or out, add a key, remove a key — section 5.7) and the **assistant** configuration routes (section 5.6) authenticate a bearer token perfectly well, but their second proof is a `currentPassword` / `password` field checked against the directory, and there is no bearer equivalent. So on an account that has **no ZephyrAB password at all** — which is the point of federating, and the shape section 5's own guidance recommends — those operations cannot be completed. Reading the state (`GET /account/encryption`, `GET /account/ai`) works normally; changing it does not. A client should not present those forms as available to a federated account without a password, and an operator who needs them reachable has to leave a password on the account. This is a gap in the implementation, not a policy: the five routes above show what the bearer path looks like.

3. Conventions

3.1 Bodies and fields

Request and response bodies are JSON with **camelCase** field names (`quotaBytes`, `displayName`, `nextCursor`). The one exception is the token endpoint, which is form-encoded with RFC 6749's snake_case parameter names. Unknown fields in requests are generally ignored; fields this manual marks as refused are refused by name.

3.2 Errors: RFC 9457 problems

Errors are `application/problem+json`. A real example (cancelling a queue entry without a reason):

```
{
  "type": "about:blank",
  "title": "invalid request",
  "status": 400,
  "detail": "reason is required: it reaches the sender inside the
bounce"
}
```

Members: `type` (URI, `about:blank` unless otherwise noted), `title` (short, stable, machine-matchable), `status`, `detail` (human-readable, actionable). OAuth token errors additionally carry the RFC 6749 error code in an `error` extension member. Match on `status` + `title`, not on `detail` text — detail wording can change.

Self-service routes (section 5) use a simpler two-field shape — `{"error": "codeName", "description": "..."}` — with camelCase error codes; each route's entry lists them.

3.3 Idempotency-Key

Mutating POSTs accept an `Idempotency-Key` header (max 128 characters). Retrying with the same key returns the original result rather than performing the action twice. Use one key per logical action:

```
curl -s -X POST https://admin.example.com/admin/api/tenants \
  -H "Authorization: Bearer TOKEN" \
  -H "Idempotency-Key: create-acme-2026-08-23-1" \
  -H "Content-Type: application/json" \
  -d '{"name": "Acme Corp", "planId": "PLAN_ID"}'
```

Where the header matters most: `restoreMailbox` binds the key to the job id, so a client retrying a request whose response it never saw gets the original job back (with `"replayed": true`) instead of starting a second restore.

3.4 Cursor pagination

Listings take `limit` (1–200, default 50) and `cursor` query parameters and answer with an `items` array plus `nextCursor` (a string, or `null` on the last page). Pass `nextCursor` back as `cursor` to continue.

Two honesty rules, and they matter to client code:

- **An endpoint that does not implement a cursor REFUSES one with 400 rather than ignoring it.** `GET /mailops/queue/messages` is the documented case. Silently ignoring a cursor would serve page one twice and call it page two.
- **A listing that hit its internal bound says so** — via a `truncated` extension field (queue search, mailbox ACL) or an explanatory note. Treat `truncated: true` as "narrow your filters", never as "that was everything". `GET /jobs` returns `"nextCursor": null` with no cursor support yet; the field is present so a last page and an unpaginated listing read the same.

3.5 Async jobs

Long-running operations answer `202 Accepted` with a job document when the node runs the job worker:

```
{ "jobId": "JOB_ID", "status": "queued" }
```

Poll `GET /jobs/{jobId}` for status, progress, and errors. Job statuses: `queued`, `running`, `succeeded`, `failed`, `partial`, `canceled`.

A `job.completed` webhook (section 7) will tell you when one reaches a terminal state, which spares you the polling loop — but it is an optimisation over this route, not a replacement for it. Keep the poll as your fallback: a receiver that was unreachable while the job finished catches up only by reading, and a delivery that exhausts its retries is abandoned rather than failing the job.

What to know before building on jobs:

- **Jobs must be safe to run twice.** A worker holds a lease, not a lock; if the worker dies, the job is reclaimed and re-run. The `attempts` field counts claims, and past `maxAttempts` (3) the job fails as poison rather than looping.
- **A node not configured to run jobs answers `501` (`"no job store"`)** to every job-backed operation, up front, rather than queueing work that will never run.
- Not every operation the OpenAPI document marks `202 + Job` is asynchronous in this build. `removeDomain` and `deprovisionTenant` are synchronous and answer `200` with what they did; `deleteAccount`, `restoreMailbox`, and `bulkProvisionAccounts` are real jobs. Each endpoint's entry in section 4 says which.

3.6 Dates

Timestamps are RFC 3339 in UTC (`2026-08-23T09:15:00Z`). Two delegation fields (`createdAt`, `expiresAt`) are Unix seconds, as the schema notes; everything else is RFC 3339.

4. Admin endpoint reference

All paths below are relative to the admin base URL, e.g. `https://admin.example.com/admin/api`. Every operation requires a bearer token except `POST /oauth/token` (section 2).

Per endpoint you get: purpose, the operation id (matching the OpenAPI document), the scope list (ANY-OF; `platform:admin` always satisfies), the subject binding (which resource your grant must cover), parameters, fields, response, and refusals. "Binding: platform" means only a platform-wide grant reaches it.

4.1 Tenants

GET /tenants

List tenants, paginated.

- **Operation:** `listTenants` · **Scopes:** `platform:admin`, `reseller:admin` · **Binding:** platform

Query parameter	Type	Meaning
`limit`	integer 1–200 (default 50)	Page size
`cursor`	string	From the previous page's `nextCursor`
`status`	`active` | `suspended` | `deprovisioning`	Filter
`cell`	string	Cell to list; absent means this cell. Listings are per-cell; a foreign cell is refused

(multi-cell only)

Response 200: `{ "items": [Tenant], "nextCursor": string|null }`. A Tenant carries `id`, `name`, `status`, `planId`, `resellerId`, `homeCell` (informational), `residencyRegion`, `externalRef`, `createdAt`.

POST /tenants

Create a tenant.

- **Operation:** `createTenant` · **Scopes:** `platform:admin`, `reseller:admin` · **Binding:** platform
- Accepts `Idempotency-Key`.

Field	Type	Required	Meaning
`name`	string	yes	Display name
`planId`	uuid	yes	Plan (CoS) the tenant subscribes to
`resellerId`	uuid	no	Owning reseller
`residencyRegion`	string	no	e.g. `eu-central`
`externalRef`	string	no	Caller's billing/CRM reference

Response 201: the Tenant.

Refusals worth knowing:

- Subscribing a tenant to a plan is gated on the `commerce` license entitlement — without it, `403` with title `license entitlement` naming the feature.

GET /tenants/{tenantId}

Get one tenant.

- **Operation:** `getTenant` · **Scopes:** `platform:admin`, `reseller:admin`, `tenant:admin`, `helpdesk:read` · **Binding:** the tenant in the path

Response 200: the Tenant. A tenant your grant does not cover answers the same 404 as one that does not exist.

PATCH /tenants/{tenantId}

Update a tenant's name, plan, or status.

- **Operation:** `updateTenant` · **Scopes:** `platform:admin`, `reseller:admin` · **Binding:** the tenant in the path

Field	Type	Required	Meaning
`name`	string	no	New display name

`planId`	uuid	no	Move to another plan (commerce entitlement applies)
`status`	TenantStatus	no	`active` / `suspended` / `deprovisioning`

Response 200: the updated Tenant.

DELETE /tenants/{tenantId}

Deprovision a tenant.

- **Operation:** `deprovisionTenant` · **Scopes:** `platform:admin` only · **Binding:** the tenant in the path

Deviation from the OpenAPI document: the spec says `202 + Job` with a retention grace; this build is **synchronous** and answers `200` with `{ "ok": true, "tenantId": ..., "note": ... }`. The note states plainly that no retention grace applies — the tenant row and its name index are gone immediately.

Refusals:

- **`409 tenant still has domains`** — remove the domains first. The response names them.
- **`409 tenant still has mailboxes`** — delete accounts first, one at a time, through `DELETE /accounts/{accountId}` (that path settles blob refcounts and quota; removing the tenant around live mailboxes would strand their addresses). The scan is bounded at 10,000 accounts; a tenant larger than the bound is refused for that reason rather than silently passing.
- This route does not cascade. That is deliberate.

4.2 Domains

Two door pairs exist for domain creation and listing, and the difference is the subject binding:

- **`/domains`** is the **global index** — spans tenants, `platform:admin` only, owning tenant named in the request **body** (a claim, not an identity).
- **`/tenants/{tenantId}/domains`** is the **tenant-scoped pair** — the tenant comes from the **path** and is checked against the caller's grant, so it is safe to delegate. Delegated tenant admins use this one.

GET /domains

List every domain on the platform.

- **Operation:** `listAllDomains` · **Scopes:** `platform:admin` only · **Binding:** platform

Response 200: `{ "items": [Domain], "nextCursor": null }`. No filters; this is the operator's whole-platform view.

POST /domains

Add a domain to any tenant, named in the body.

- **Operation:** `createAnyDomain` · **Scopes:** `platform:admin` only · **Binding:** platform
- Accepts `Idempotency-Key`.

Field	Type	Required	Meaning
`domain`	hostname	yes	The domain name
`tenant`	string	yes here	Owning tenant — **a tenant id is the contract**; a tenant name is accepted and normalised. Refused when empty on this global route; ignored on the tenant-scoped route
`selectors`	array of string	no	DKIM selectors to record. Defaults to `zab1`

Response 200: `{ "ok": true, "domain": "acme.example", "domainId": "..." }` — a short acknowledgement rather than the full Domain object; read it back with `GET /domains/{domainId}` if you need the rest. A new domain is minted a uuid `domainId`; re-adding an existing `(name, tenant)` preserves its id (an upsert), so re-running provisioning is safe.

The `tenant` field must name a tenant that exists, and what is stored is always its id. Pass the id: that is what the field means everywhere it is read. A name is accepted as a convenience — so an operator can type the tenant they can see in the console — and is resolved to the id **before anything is written**, rather than at every reader afterwards.

That difference is not cosmetic, and it is worth knowing which value ends up in your data. `DomainRecord.tenant` is consumed as a tenant **id** by the tenant-scoped DNS records, the shared SOA serial, the DNSSEC zone keys, the ownership check that guards creating an account by `domainId`, and — most visibly to an integrator — the **subject of a `domain.created` webhook** (section 7). Webhook subjects are matched exactly, so a domain stored against a tenant *name* is a domain whose events no `tenant:<uuid>` endpoint can ever match.

How the resolution behaves, in order:

1. The value is tried as a **tenant id** first: one lookup, which is what every well-behaved caller hits.
2. Failing that, tenant **names** are scanned, bounded at 500 tenants.
3. **An unresolvable value is refused and nothing is stored.** The error says which case it was — `"…" is neither a tenant id nor the name of a tenant on this platform — create the tenant first, or pass its id`, or, when the platform has more tenants than the name scan will walk, a refusal that says so explicitly and tells you to pass the id. Those are different facts: the second is not "no such tenant", and answering as though it were would send you looking for a tenant that exists.

Refusals:

- **403 `license limit`** — this would exceed the domain cap: the number the license names, or **one** on a deployment with no license file, where the Community edition hosts a single mail domain. The refusal names which of the two it is. Re-registering an existing domain (which is also what a DNS re-verification writes through) is never refused, in any license state.
- An empty `tenant`, or one that cannot be resolved, answers `{ "ok": false, "error": "..." }`. **Note the shape**: this route reports failure in its own body rather than as an RFC 9457 problem, and does so with a 200 status — the license refusal above is the exception and is a real `403` problem. **Branch on `ok`, not on the status code.**

GET /tenants/{tenantId}/domains

List one tenant's domains.

- **Operation:** `listDomains` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the tenant in the path
- Query: `limit`, `cursor`.

Response 200: `{ "items": [Domain], "nextCursor": string|null }`.

POST /tenants/{tenantId}/domains

Add a domain to this tenant. The tenant comes from the path, so a delegated tenant admin can use it.

- **Operation:** `createDomain` · **Scopes:** `tenant:admin` · **Binding:** the tenant in the path
- Accepts `Idempotency-Key`.
- Body: as `POST /domains`, except the `tenant` field is ignored — the path wins. A request naming one tenant in the path and another in the body has no correct interpretation, and preferring the path silently is less surprising than a 400 clients will hit by copying a payload.

Response 200: `{ "ok": true, "domain": ..., "domainId": ... }`, the same acknowledgement shape as the global route, and the same `ok`-not-status rule. The domain

starts in `pending_verification`. Same domain-cap refusal as the global route — the two doors share one check, not two opinions.

There is no tenant resolution to do here: the path parameter is already an id, and it is the one the RBAC layer checked against your grant. That is what makes this route safe to delegate and the global one platform-only.

A `404` here does not mean the route is unimplemented. It is mounted, and it is the door to prefer. `404` is the single answer this API gives to two different situations — *that tenant does not exist* and *your grants do not cover it* — because distinguishing them would turn every tenant-scoped path into a way to discover which tenant ids are real. So when this route 404s, check both: that the path parameter is a tenant **id** rather than a tenant name (this route does no name resolution — that convenience exists only on the global door), and that your token's grants actually cover that tenant. The server-side log records which of the two it was; the response deliberately does not. Section 2.5 covers grants and subjects.

GET /domains/{domainId}

Get one domain.

- **Operation:** `getDomain` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the domain in the path

Response 200: the Domain — `id`, `tenantId`, `name`, `status` (`pending_verification` / `verified` / `failed` / `disabled`), `dkimSelectors`, `createdAt`, at-rest policy. The response also carries `statusSource`, saying whether the status you see is an operator override (`disabled`) or DNS-derived — `disabled` and `failed` both mean "mail is not flowing" and only one of them is deliberate.

Note: `catchAllAddress` is **absent** from responses on this build, not `null` — catch-all delivery does not exist in this server, and echoing `null` would read as "no catch-all configured" rather than "no such concept".

PATCH /domains/{domainId}

Set or clear the operator status override.

- **Operation:** `updateDomain` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path

Field	Type	Meaning
`status`	`"disabled"` or `null`	`disabled` stops the domain accepting mail at RCPT (senders get `550 5.2.1`) without deleting anything. `null` clears the override and returns the domain to its DNS-derived status.

		Omitting the field leaves status alone
`catchAllAddress`	—	**Refused** (see below)

Response 200: the updated Domain.

Refusals:

- **400 `unsupported field`** — `catchAllAddress` is declared by the API and this server does not implement catch-all delivery; it is refused by name rather than accepted-and-ignored, so nobody believes a fallback is configured when none exists.
- **400** — `status` values `verified`, `pending_verification`, `failed` are refused: they are observations derived from live DNS, not opinions. Re-run verification to change them.

DELETE /domains/{domainId}

Remove a domain.

- **Operation:** `removeDomain` · **Scopes:** `tenant:admin` · **Binding:** the domain in the path

Query parameter	Type	Meaning
`purgeKeys`	boolean	Also destroy the domain's DNSSEC and DKIM key material

Deviation from the OpenAPI document: the spec says `202 + Job`; this build is **synchronous** and answers `200` with what it destroyed.

Refusals:

- **409** while the domain still has mailboxes — delete the accounts first. The check is a safety net, separate from authorization: a caller entitled to remove the domain is still refused while accounts live on it. The count is bounded; a domain with more accounts than the scan cap is refused for that reason rather than under-reported.

GET /domains/{domainId}/dns-records

The DNS records the domain's owner must publish (MX, SPF, DKIM, DMARC, MTA-STS, TLS-RPT, TLSA, autodiscovery), each with live verification state.

- **Operation:** `getExpectedDnsRecords` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the domain in the path

Response 200: an array of records:

Field	Type	Meaning

`recordType`	`A` | `AAAA` | `MX` | `TXT` | `CNAME` | `TLSA` | `SRV`	Record type
`name`	string	Fully qualified owner name
`value`	string	Presentation form to publish
`purpose`	`mx` | `spf` | `dkim` | `dmarc` | `mta_sts` | `tls_rpt` | `tlsa` | `autodiscover` | `ownership`	What the record is for
`verified`	boolean	Whether live DNS currently serves it
`checked`	boolean	Whether anything looked. `verified: false` alone is two facts wearing one word — "probed and absent" and "nothing probed" — and only the first is actionable
`lastCheckedAt`	date-time	When

POST /domains/verify

Verify a domain's mail DNS ad hoc, against live public DNS.

- **Operation:** `verifyDomain` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** none (unscoped — it reads nothing stored; it only reports what public DNS answers anybody)

Deviation from the OpenAPI document: the spec places this at `POST /domains/{domainId}/verify` with `202 + Job`. The implementation serves it at **`POST /domains/verify`**, takes the domain in the body, and answers **synchronously**.

Field	Type	Required	Meaning
`domain`	hostname	yes	Domain to probe
`selectors`	array of string	no	DKIM selectors to check

Response 200: `{ "domain": ..., "status": ..., "auth": { per-mechanism results } }` — the overall verdict plus per-record findings.

POST /domains/{domainId}/dkim/rotate

Advance the domain's DKIM key rotation by one step.

- **Operation:** `rotateDkimKey` · **Scopes:** `tenant:admin`, `platform:admin` · **Binding:** the domain in the path. Deliberately **not** reachable by `domain:admin`: rotating a

signing key mid-flight is how outbound mail starts failing DKIM at every receiver, and the day-to-day zone-edit grant should not be able to do it.

Deviation: synchronous, not `202 + Job`. The response reports the step taken:

`step`	Meaning
`pre_publish`	The incoming selector is published and does **not** sign yet; it starts signing once resolvable for a full hold
`switch`	The new key signs; the old selector stays published through its hold
`retire`	The old selector left DNS
`nothing`	No step was due; the note says why (e.g. a phase already in flight is held to its clock)

Refusals:

- **`409 no dkim key`** — the domain has no key to rotate. "Rotate" is not "create": a first key is minted when the nameserver first serves an internal domain.
- **`501 no key material`** — this node holds no DKIM master key, so it cannot read or mint signing keys.

4.3 Domain DNS

For domains whose DNS ZephyrAB hosts (`internal` mode). Custom records are **additive**: they can never override the records ZephyrAB maintains for mail delivery.

GET /domains/{domainId}/dns

DNS mode, delegation targets, and custom records for a hosted zone.

- **Operation:** `getDomainDns` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path

Response 200:

Field	Type	Meaning
`domain`	string	The zone
`mode`	`internal` | `external`	`internal` = ZephyrAB is authoritative and generates the mail records; `external` = the customer keeps their own DNS and ZephyrAB only verifies it
`nameservers`	array of string	Delegate the domain to these to make ZephyrAB authoritative

`recordCount`	integer	Custom RRsets stored
`records`	array	The custom RRsets (`id`, `name`, `type`, `ttl`, `values[]` in presentation form)

PUT /domains/{domainId}/dns

Switch the domain between ZephyrAB-hosted and external DNS.

- **Operation:** `setDomainDns` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path
- Body: `{ "mode": "internal" | "external" }`.

Response 200: the updated DomainDns document.

Setting `internal` does **not** delegate the domain: the owner must still point their NS records at the returned nameservers. Until they do, the zone is served but nothing asks for it.

`mode` is the switch for the WHOLE zone, not for its mail records. A domain in `internal` mode has every one of its records answered by ZephyrAB's nameservers — the mail records the platform generates, every custom RRset from the endpoint below, and the address records for the nameservers themselves. Setting it to `external` stops all of that together. That is the intended behaviour when a customer is taking their DNS back, and it is worth stating plainly because the failure it produces is total rather than partial: if the domain is still delegated to ZephyrAB, its nameservers begin answering for nothing, and the whole zone disappears from the internet within about five minutes. The website goes with the mail.

So the order for handing DNS back is: publish the zone at its new provider, repoint the registrar's NS records, wait for the delegation to move, and only then set `external`. Doing it in the other order is an outage for as long as the two steps are apart.

There is no API to make a domain served *without* setting `internal`. The operator-side escape hatch is the `ZEPHYR_DNS_ALSO_SERVE` environment variable on the nameservers, which serves a named domain regardless of its mode. It is how a zone is exercised before anyone is delegated to it, and how a zone is restored in a hurry. See `docs/runbooks/dns-zone-serving.md`, which sets out exactly how much of a zone it brings back and covers recovering a domain whose mode was lost.

PUT /domains/{domainId}/dns/records

Create or replace one custom RRset in a hosted zone.

- **Operation:** `putDomainDnsRecord` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path

Field	Type	Required	Meaning
`name`	string	yes	**The fully qualified**

			owner name, inside the zone — see below
`type`	`A` | `AAAA` | `CNAME` | `TXT` | `MX` | `SRV` | `TLSA` | `CAA`	yes	Record type
`ttl`	integer 60–604800	yes	TTL in seconds
`values`	array of string, min 1	yes	Presentation form, e.g. `10 mail.example.com` for MX

`name` is the whole name, not a label relative to the zone. For the zone `acme.example`, a record on `www` is written `www.acme.example`; a record at the apex is written `acme.example`. A bare `www` is refused as out-of-zone (`www is not inside acme.example`) rather than being helpfully expanded — the same request would otherwise mean two different things depending on whether the server chose to expand it, and a zone file's owner names are absolute. A trailing dot is accepted and case is normalised.

One RRset per call. The `(name, type)` pair is the identity, so this is a replace, not an append: sending two `A` values for one name means one request carrying both in `values`, and sending them as two requests leaves you with only the second. To publish a whole zone, iterate.

Response 200: the stored RRset, including its `id` (derived from `(name, type)` — repeating a request **replaces** rather than duplicates).

Refusals — each a **409 naming the collision**, never a silent shadow:

- A record that would override one ZephyrAB maintains for mail delivery (its MX, SPF, DKIM, DMARC, MTA-STS…). The refusal names which record and why it matters — a customer-supplied MX at the apex is how "some of our mail bounces" happens days later with nothing visibly wrong.
- Names outside the zone.
- Server-owned types (SOA, NS, DNSSEC records).
- CNAMEs that break RFC 1034: at the apex, or alongside other data.

Validation runs against the **regenerated** zone, not a cached idea of it.

DELETE /domains/{domainId}/dns/records/{recordId}

Remove one custom RRset.

- **Operation:** `deleteDomainDnsRecord` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path
- `recordId` is the RRset handle from the listing (derived from `(name, type)`).

Response 204.

4.4 Domain encryption & settings

GET /domains/{domainId}/encryption

The domain-wide at-rest encryption policy, with an enrolment readiness survey.

- **Operation:** `getDomainAtRestPolicy` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path

The effective requirement for a mailbox is the **union** of this policy and the account holder's own opt-in: either may switch encryption on; neither may switch the other's off.

Response 200:

Field	Type	Meaning
`domainId`, `domain`	uuid, hostname	The domain
`policy`	`optional` | `required`	`optional` = each account decides (default). `required` = every account must have encrypted mail; an account with no usable public key has incoming mail **refused**, not stored in the clear. There is deliberately no "encrypt where possible" value
`readiness`	object	`examined` (accounts looked at), `complete` (false when the domain has more accounts than the survey bound — **a false here proves nothing about the rest and is never treated as a pass**), `withoutKeys`, `withoutKeysSample[]`, `ready`
`encryptionAvailable`	boolean	Whether this node runs at-rest encryption at all

PUT /domains/{domainId}/encryption

Set the policy.

- **Operation:** `setDomainAtRestPolicy` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path

Field	Type	Required	Meaning
`policy`	`optional` | `required`	yes	
`acknowledgeMailWillBeRefused`	boolean (default false)	no	Proceed with `required` even though the survey found accounts that will start having mail refused. Named for the consequence — setting it is a statement about what happens to real mail

Response 200 with the updated policy, or **409** (carrying the same document, readiness included) when:

- the node does not run at-rest encryption, or
- the readiness survey found accounts with no usable key (or could not survey completely) and `acknowledgeMailWillBeRefused` was not set.

Setting `optional` is always permitted: it removes only the requirement the domain imposed, never one an account chose for itself.

GET /domains/{domainId}/settings

Every declared setting, as this domain sees it.

- **Operation:** `getDomainSettings` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path

Response 200: `{ "domainId", "domain", "scope": "domain", "settings": [Setting] }`. Each Setting reports:

Field	Meaning
`key`	e.g. `links.ttl_days`, `av.enabled`, `smtp.subaddress`
`kind`	`protective` (resolves as a **union** — any scope may switch a protection on, none may switch another's off) or `operational` (most specific wins: account, then domain, then platform — unless the domain locked it)
`valueType`, `min`, `max`, `unit`	Type and bounds
`scopes`	Which of `platform` / `domain` / `account`

	may hold a value
`selfService`	Whether an account holder may set it through `/account/settings`
`managedBy`	Present when the setting is stored elsewhere (its own route) and is not written through this collection
`platform`, `domain`, `account`	What each layer has said (`{value, locked}`)
`effective`, `origin`	The value in force **at the domain**, and which layer supplied it (`fallback` / `platform` / `domain` / `domain (locked)` / `account` / `union`)
`locked`	The domain pinned this; narrower writes are refused
`ignoredInvalid`	A stored value was refused as invalid and is NOT in force — skipped, never clamped

The `effective` value here excludes the account layer: an account may still hold its own value for anything unlocked. The `scope` field names that so the answer cannot be misread as a mailbox's effective value.

PUT /domains/{domainId}/settings/{key}

Set one domain-scoped setting.

- **Operation:** `setDomainSetting` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path

Field	Type	Required	Meaning
`value`	boolean or integer (per the setting's `valueType`)	yes	
`locked`	boolean (default false)	no	Pin the value: refuse narrower overrides and ignore any already stored. Domain scope, **operational settings only** — a lock on a protective setting would forbid an account from turning a protection ON, so it is refused

Response 200: the Setting after the change. Refusals:

- **400** — out of range (**refused with the bound named, never clamped** — a silently clamped setting is one an administrator believes is in force and is not), wrong type, or a lock that does not apply.
- **403** — not settable at this scope, or stored elsewhere (see `managedBy` — e.g. the at-rest policy is only written through its own route).
- **404** — unknown domain, or a key the registry does not declare.

DELETE /domains/{domainId}/settings/{key}

Clear the domain's opinion of one setting.

- **Operation:** `clearDomainSetting` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the domain in the path

Response 200: the Setting after the change. Clearing is **not** the same as setting the default: the platform value applies again, and an account may override it again.

4.5 Accounts

GET /tenants/{tenantId}/accounts

List a tenant's accounts.

- **Operation:** `listAccounts` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the tenant in the path

Query parameter	Type	Meaning
`limit`	integer 1–200 (default 50)	Page size
`cursor`	string	Continue a listing
`domainId`	uuid	Filter to one domain
`status`	AccountStatus	Filter
`q`	string	Substring match on address / display name

Response 200: `{ "items": [Account], "nextCursor": string|null }`. An Account carries `id`, `tenantId`, `domainId`, `address`, `displayName`, `status` (`active` / `suspended` / `locked` / `maintenance` / `deleted`), `quotaBytes`, `usedBytes` (read-only), `aliases`, `createdAt`, `lastLoginAt`.

POST /tenants/{tenantId}/accounts

Create an account.

- **Operation:** `createAccount` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the tenant in the path
- Accepts `Idempotency-Key` — a replay returns the originally created account (200).

Field	Type	Required	Meaning
`domainId`	uuid	yes (preferred)	The owning domain.

			Resolved with a tenant-ownership check: a `domainId` belonging to another tenant answers a generic `400 "unknown domainId"` — no oracle
`domain`	hostname	back-compat only	A bare domain name, honoured only when `domainId` is absent. This legacy path skips the ownership check
`localPart`	string	yes	Left of the `@`
`displayName`	string	no	
`password`	string	no	Optional initial password (argon2id-hashed at rest, never echoed). Minimum 8 characters here; omit to leave the account without a credential until one is set
`quotaBytes`	integer	no	Defaults from the tenant's plan
`aliases`	array of email	no	Extra addresses delivering to this mailbox

Response 201: the Account.

Refusals:

- **400** with `"domainId is required (a bare domain name is accepted for back-compat)"` when both are absent.
- **409** — address already taken (the address index is global; an address is a mailbox, an alias, or a group, never two of them).

- **403 `license limit`** — the license caps mailboxes and this would be one past the cap. A deployment with no license file never meets this: the Community edition does not cap mailboxes.

POST /tenants/{tenantId}/accounts/bulk

Bulk create accounts (async job).

- **Operation:** `bulkProvisionAccounts` · **Scopes:** `tenant:admin` · **Binding:** the tenant in the path. Also gated on the `commerce` license entitlement.
- Accepts `Idempotency-Key`.

Field	Type	Required	Meaning
`format`	`inline`	yes	Only `inline` is implemented (see below)
`items`	array of account rows	yes for inline	Max 10,000 rows; each row is shaped like the `createAccount` body
`onConflict`	`skip` | `error` | `update` (default `skip`)	no	What to do about an existing address
`dryRun`	boolean (default false)	no	Validate without writing

Response 202: `{ "jobId", "status" }` — poll `GET /jobs/{jobId}`. Row-level failures land in the job's `errors` (capped at 100 stored strings; the `progress.failed` count stays exact).

Refusals:

- **501** for `format: jsonl` or `csv` — those read rows from a caller-supplied `sourceUrl`, and a server fetching caller-supplied URLs is a server-side request forgery surface this build does not open.
- **400** for any row carrying a `password`. Rows are stored in the job record, and ten thousand plaintext passwords at rest is a credential store nobody decided to build. Omit it — an account without a credential is completed through `PUT /accounts/{accountId}/password` or the user-facing flows.
- **501 `no job store`** when the node does not run the job worker.
- Every row's domain resolves through the same tenant-ownership check as `createAccount` — a row cannot reach across the tenant boundary the route was authorized against.

GET /accounts/{accountId}

Get one account.

- **Operation:** `getAccount` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the account in the path (via its stored tenant/domain — never one the request supplies)

Response 200: the Account.

PATCH /accounts/{accountId}

Update display name, quota, or the alias set.

- **Operation:** `updateAccount` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the account in the path. `helpdesk:write` is deliberately absent: an alias is a row in the global address index that SMTP reads at RCPT time — handing a mailbox a new address is not a password reset.

Field	Type	Meaning
`displayName`	string	
`quotaBytes`	integer	
`aliases`	array of email	The whole set, replacing what is stored
`planOverrideId`	—	**Refused** (`400`): declared by the spec, and there is no plan-override engine in this build. Refused by name so nobody believes an override was applied

Response 200: the updated Account.

DELETE /accounts/{accountId}

Schedule the account's destruction (async job).

- **Operation:** `deleteAccount` · **Scopes:** `tenant:admin` · **Binding:** the account in the path

Response 202: `{ "jobId", "status" }`. Purging is one storage transaction per message, so a large mailbox cannot be an HTTP handler holding a socket open. The account is **not** disabled by this handler: fencing is the first thing the purge job itself does, inside the one safe ordering (fence → revoke credentials → purge messages → unprovision last), so a queued-but-never-run job cannot leave an account half-dead.

Refusals: **501 `no job store`** when the node does not run jobs; 404 for an unknown account (or one your grant does not cover).

PUT /accounts/{accountId}/status

Set account status.

- **Operation:** `setAccountStatus` · **Scopes:** `tenant:admin`, `domain:admin`, `platform:admin` · **Binding:** the account in the path

Field	Type	Required	Meaning
`status`	`active` | `suspended` | `locked` | `maintenance` | `deleted`	yes	Non-active accounts never authenticate; RCPT behaviour follows status
`reason`	string	no	Audit
`ticketRef`	string	no	Audit

Response 200: the updated Account.

PUT /accounts/{accountId}/password

Admin password set / forced reset. Audit-logged; a ticket reference is mandatory.

- **Operation:** `setAccountPassword` · **Scopes:** `tenant:admin`, `helpdesk:write` · **Binding:** the account in the path. This is the route subject binding exists for: a helpdesk credential can only set passwords inside the tenant (or domain) it was granted on.

Field	Type	Required	Meaning
`ticketRef`	string	yes	Support ticket id — refused (`400`) when absent or empty
`password`	string	see note	The new password
`mustChange`	boolean (default true)	no	Declared by the spec
`revokeSessions`	boolean (default true)	no	Declared by the spec

Response 204.

Deviation: the spec says omitting `password` sends a reset link instead; this build answers **`501 reset flow not available`** for an omitted password — supply one. (Self-service reset links exist on the mail origin: `/recover`, section 5.)

GET /accounts/{accountId}/sieve

Read the account's Sieve filter script.

- **Operation:** `getAccountSieve` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the account in the path

Response 200: `{ "script": "..." }` — empty string when none is stored.

PUT /accounts/{accountId}/sieve

Replace the account's Sieve filter script.

- **Operation:** `setAccountSieve` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the account in the path
- Body: `{ "script": "..." }`. An empty script clears it.

Response 204.

The upload is **parse-gated**: the script must parse against the supported Sieve subset (section 6.6 lists the advertised extensions) or the request is refused with the parser's diagnosis. A filter that cannot be evaluated must never reach the delivery path — a stored-but-broken script would mean implicit keep and mail in INBOX instead of the user's folders.

4.6 Mailbox sharing (ACL)

Shared mailboxes ride RFC 4314 rights. The admin surface manages them account-by-account; over IMAP, shared mailboxes appear to the grantee under `Other Users/`.

GET /accounts/{accountId}/acl

Who can reach this account's mailboxes, and what it can reach — both directions.

- **Operation:** `getMailboxAcl` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the account in the path

Response 200:

Field	Meaning
`accountId`, `address`	The account
`grants`	What this account has shared OUT — one row per `(mailbox, identifier)`: `mailbox`, `identifier` (the grantee's canonical address), `rights` (RFC 4314 letters in canonical order, with the virtual `c`/`d` aliases appended when any member right is set — the same form a GETACL response carries), `grantedBy`, `grantedAt`
`sharedWithThisAccount`	Discovery pointers for mailboxes shared WITH it: `{owner, mailbox}`. Pointers only — the rights live on the owning account's record, so a stale pointer resolves to nothing rather than granting anything
`truncated`	The account has more mailboxes than one page reads. Reported, never inferred

An account always holds every right over its own mailboxes and is never listed as a grantee of itself.

PUT /accounts/{accountId}/acl

Grant, change, or remove one identifier's rights on one mailbox.

- **Operation:** `setMailboxAcl` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the account in the path. `helpdesk:write` is deliberately absent — this hands one person access to another person's mail, the same class of act as adding an alias.

Field	Type	Required	Meaning
`mailbox`	string	yes	e.g. `INBOX`, `Projects`
`identifier`	string	yes	Any address the grantee answers to, in the **same tenant**. Aliases resolve; the canonical address is stored
`rights`	string	yes	RFC 4314 `mod-rights`, byte for byte what SETACL accepts: bare letters replace, `+letters` add, `-letters` remove. Empty removes the entry

Response 200: `{ "mailbox", "identifier", "rights", "removed" }` — `removed: true` when the rights resolved to the empty set and the entry was deleted.

Refusals:

- An unrecognised right letter is refused **by name** rather than dropped (RFC 4314 §3.1).
- Unknown address, another tenant's account, a distribution group, a suspended or deleted mailbox — **one indistinguishable refusal** for all of them, so the route cannot be used to enumerate the directory.

DELETE /accounts/{accountId}/acl

Revoke one identifier's rights on one mailbox.

- **Operation:** `deleteMailboxAcl` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the account in the path

Query parameter	Required	Meaning
`mailbox`	yes	The mailbox
`identifier`	yes	The grantee

Response 200: `{ "mailbox", "identifier", "existed" }`. Idempotent: revoking a grant that is not there succeeds with `existed: false`, so a retry cannot fail — and an operator who named the wrong mailbox still learns nothing was removed.

4.7 Restores

POST /accounts/{accountId}/restores

Point-in-time mailbox restore from backup (async job).

- **Operation:** `restoreMailbox` · **Scopes:** `tenant:admin`, `compliance:write` · **Binding:** the account in the path. Presenting a copy of somebody's deleted mail where they can read it is a compliance action — hence the scope and the `ticketRef`.
- Accepts `Idempotency-Key` — a retry whose response was lost returns the original job with `"replayed": true` instead of starting a second restore.

Field	Type	Required	Meaning
`pointInTime`	date-time	yes	Restore point
`target`	`side_by_side`	yes	See the refusal below
`scope`	`mailbox` | `folders` (default `mailbox`)	no	Whole mailbox or named folders
`folders`	array of string	when `scope: folders`	
`sideBySideFolderName`	string (default `RESTORED`)	no	Prefix folder the restored copy lands under
`ticketRef`	string	yes in practice	Audit-logged with the actor

Response 202: `{ "jobId", "status", "target": "side_by_side", "prefix": ... }`.

Refusals:

- **400** — `target: "in_place"` is refused, and not as a to-do: in-place **overwrites live mail**, the one operation here that cannot be undone, and it is not something one HTTP request should be able to do. Use `side_by_side`; the user deletes the extra folder when done.
- **501 `no job store`** — the node does not run jobs.
- **501 `no restore procedure`** — the node has no restore procedure configured. Refused **before** queueing, so nobody polls a handle for work that cannot start.

4.8 Groups

Distribution groups: one address fanning out to many mailboxes, expanded at RCPT. A group's address is claimed in the same global index accounts use, so it can never collide with a mailbox. The `{address}` path parameter is a mail address — percent-encode the `@` (`team%40customer.example`).

GET /tenants/{tenantId}/groups

List groups.

- **Operation:** `listGroups` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the tenant in the path
- Query: `limit`, `cursor`.

Response 200: `{ "items": [Group], "nextCursor": string|null }`.

POST /tenants/{tenantId}/groups

Create a group.

- **Operation:** `createGroup` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the tenant in the path

Field	Type	Required	Meaning
`address`	email	yes	The group address, on a domain this tenant owns
`owner`	email	yes	A mailbox in this tenant. Bounces and anything arriving with a null reverse path go to the owner rather than being fanned out — and the owner is the envelope sender on copies to remote members. No group exists without a human who receives the consequences of it existing
`displayName`	string	no	
`members`	array of email	no	Deduplicated at write time
`posting`	`open` (default) |	no	`open` accepts mail

	`membersOnly`	from anyone; `membersOnly` accepts members and the owner. A posting refusal on the wire is `550 5.7.1` (policy), never `5.1.1` — the address exists

Response 201: the Group (`id`, `address`, `domain`, `displayName`, `owner`, `posting`, `members`, `memberCount`, `createdAt`).

Refusals: **409** when the address is already a mailbox, alias, or group.

GET /tenants/{tenantId}/groups/{address}

Get one group.

- **Operation:** `getGroup` · **Scopes:** `tenant:admin`, `domain:admin`, `helpdesk:read` · **Binding:** the tenant in the path

Response 200: the Group.

PATCH /tenants/{tenantId}/groups/{address}

Replace a group's members, posting policy, or display name.

- **Operation:** `updateGroup` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the tenant in the path. The write scopes are the narrower set on purpose: adding a member silently redirects a copy of everything sent to the group — the same authority as creating a forwarding rule.

Field	Type	Meaning
`members`	array of email	**The whole list, not a delta** — sending the list you believe is current lets the update transaction see what you saw
`posting`	GroupPosting	
`displayName`	string	

Response 200: the updated Group.

DELETE /tenants/{tenantId}/groups/{address}

Delete a group.

- **Operation:** `deleteGroup` · **Scopes:** `tenant:admin`, `domain:admin` · **Binding:** the tenant in the path

Response 204. The address answers `550 5.1.1` afterwards.

4.9 Delegations

Delegations are how subject-bound authority is handed out (section 2.5). A delegation is **immutable**: create and revoke, never update — changing one is revoke + create, which is the more honest audit trail.

GET /tenants/{tenantId}/delegations

List everything delegated inside this tenant.

- **Operation:** `listDelegations` · **Scopes:** `tenant:admin`, `platform:admin` · **Binding:** the tenant in the path

Response 200: `{ "items": [Delegation] }` (no cursor; a tenant's delegations are few and bounded). Includes grants bound to the tenant's domains, and grants that have **lapsed** — an operator asking "what has been handed out" needs to see an expired grant; `active` (computed, not stored) says which is which, and the authorization path honours nothing inactive.

A Delegation:

Field	Type	Meaning
`id`	uuid	
`principalClientId`	string	Who holds the authority. An OAuth client id, **or an account's mail address** for people signing in with the password grant. (The field name records the v1 reality: this control plane authenticates clients; there are no separate admin accounts)
`scope`	string	One scope from the taxonomy. `platform:admin` is refused here
`subjectKind`	`tenant` | `domain`	
`subjectId`	string	The tenant or domain uuid the scope is bound to
`tenantId`	uuid	The tenant the grant is filed

		under
`expiresAt`	Unix seconds or null	null = no expiry
`note`	string	
`createdAt`	Unix seconds	
`active`	boolean	In force right now

POST /tenants/{tenantId}/delegations

Grant a scope, bound to a subject, to a principal.

- **Operation:** `createDelegation` · **Scopes:** `tenant:admin`, `platform:admin` · **Binding:** the tenant in the path. Also gated on the `delegated-rbac` license entitlement.
- Accepts `Idempotency-Key`.

Field	Type	Required	Meaning
`principalClientId`	string	yes	The client id — or the mail address of the person — receiving the grant
`scope`	string	yes	From the taxonomy
`subjectKind`	`tenant` | `domain` (default `tenant`)	no	
`subjectId`	string	for a domain subject	Omitted = the tenant in the URL
`expiresAt`	Unix seconds or null	no	
`note`	string	no	

Response 201: the Delegation. Takes effect on the principal's **next request** — no new token needed.

Refusals worth knowing:

- **Platform subjects are refused outright.** Platform-wide authority is configured on the node (`ZEPHYR_ADMIN_PLATFORM_ACCOUNTS`, client configuration), never delegated at runtime by whoever already has it. Similarly `platform:admin` as a scope is refused: "every scope, but only inside one tenant" is a contradiction that reads stronger than it is.
- **The caller must already hold the scope it is handing out, on that subject** — this route cannot be used to climb. That refusal is a `403` (the caller is authenticated and the subject is one it may address).
- A subject outside the tenant in the URL, and one that does not exist, receive the **same 404**.

- **403 `license entitlement`** without `delegated-rbac`. Existing delegations keep being honoured in every license state — only creating new ones is gated, and revoking is never gated.

DELETE /delegations/{delegationId}

Revoke a grant.

- **Operation:** `revokeDelegation` · **Scopes:** `tenant:admin`, `platform:admin` · **Binding:** the delegation's own stored tenant (or domain) — a tenant admin revokes inside their tenant and nowhere else

Response 204. Effective on the principal's next request: grants are read per request, not snapshotted into tokens. An id that is already revoked, one that never existed, and one belonging to another tenant all answer **404 alike** — a delegation id must not be probeable. Revocation is never license-gated: taking access away must never require a license.

4.10 Plans

Plans (class-of-service bundles) are **versioned and immutable**. A patch publishes a new version and moves the plan's pointer; every version ever published stays readable, which is what makes a closed billing period explicable. Subscriptions pin the version in force at subscription time.

Plan **reads** are open to `tenant:admin/reseller:admin` scopes (a customer seeing what they could move to is reasonable) but the catalogue spans tenants, so on this implementation those scopes only reach it from a **platform-subject grant**. Plan **writes** are `platform:admin` and gated on the `commerce` license entitlement.

GET /plans

The plan catalogue.

- **Operation:** `listPlans` · **Scopes:** `platform:admin`, `reseller:admin`, `tenant:admin` · **Binding:** platform
- Query: `limit`, `cursor`, and `includeRetired=true` to include retired plans (excluded by default).

Response 200: `{ "items": [Plan], "nextCursor": string|null }`.

A Plan:

Field	Type	Meaning
`id`, `name`	uuid, string	
`quotaBytes`	integer	Per-mailbox quota
`features`	object of boolean	Feature flags (`eas`, `archive`, `byok`, `legalHold`, …)

`rateLimits`	object	`outboundPerHour`, `recipientsPerMessage`
`retentionDays`, `coldTierAfterDays`, `hygieneProfile`		Policy knobs
`version`	integer	Which immutable version this document describes
`status`	`active` | `retired`	Whether it is in the catalogue
`included`	object	Metered allowances per period: `mailboxes`, `storageLogicalBytes` (tenant total, distinct from per-mailbox `quotaBytes`), `domains`, `inboundMessages`, `outboundMessages`, `outboundRecipients`. **Absent means "not metered", which is not the same as zero.** An allowance is a billing boundary, never an enforcement one — exceeding it is reported, not refused
`overage`	object	Unit rates beyond the allowance, in currency **micros** (integers, never floats): `currency` (ISO 4217, required once any rate is set), `perMailboxMicros`, `perGibStoredMicros`, `perInboundMessageMicros`, `perOutboundMessageMicros`, `perOutboundRecipientMicros`. ZephyrAB never multiplies quantity by rate — rounding, proration and tax are a billing system's decisions
`notes`	array of string	Which of this plan's fields

	the serving build **stores and does not act on**. A stored value a reader assumes is enforced is a promise nothing keeps, so each plan says which of its own fields are advisory

POST /plans

Publish a plan (version 1).

- **Operation:** `createPlan` · **Scopes:** `platform:admin` · **Binding:** platform · **Entitlement:** `commerce`
- Accepts `Idempotency-Key`.
- Body: `name` (required), `quotaBytes` (required), plus the optional `features`, `rateLimits`, `retentionDays`, `coldTierAfterDays`, `hygieneProfile`, `included`, `overage` as above.

Response 201: the Plan.

GET /plans/{planId}

One plan — current version, or any version ever published.

- **Operation:** `getPlan` · **Scopes:** `platform:admin`, `reseller:admin`, `tenant:admin` · **Binding:** platform

Query parameter	Meaning
`version`	Read a specific published version (`>= 1`). Absent = the current one. A non-numeric value is `400 invalid version`

Response 200: the Plan at that version.

PATCH /plans/{planId}

Publish a new version and point the plan at it.

- **Operation:** `updatePlan` · **Scopes:** `platform:admin` · **Binding:** platform · **Entitlement:** `commerce`

Field	Meaning
`name`, `quotaBytes`, `features`, `retentionDays`	Carried forward from the current version when absent
`status`	`retired` withdraws the plan from the catalogue; `active` returns it. Retiring changes nothing for tenants already on it, and a status-only patch publishes no

version

Response 200: the Plan (new version).

Refusals: a change to a **rate** or a rate limit is not patchable at all — that is a new plan. The other half of a bill (the metered period) is frozen once; editing rates in place would silently re-score every period already closed against it.

DELETE /plans/{planId}

Delete a plan.

- **Operation:** `deletePlan` · **Scopes:** `platform:admin` · **Binding:** platform · **Entitlement:** `commerce`

Response 204, or:

- **409 `plan in use`** — the plan has (or ever had) subscribers: a closed period was scored against it, and a bill nobody can explain is worse than a catalogue entry nobody uses. Move the tenants off it and retire it instead (`PATCH` with `status: "retired"`).

4.11 Jobs

GET /jobs

List async jobs the caller may see.

- **Operation:** `listJobs` · **Scopes:** `platform:admin`, `tenant:admin` · **Binding:** self-scoped — the listing reads only under subjects the caller holds a grant for, so a tenant admin's page simply does not contain another tenant's jobs

Query parameter	Meaning
`limit`	Page size
`type`	`bulk_accounts`, `tenant_deprovision`, `domain_remove`, `domain_verify`, `dkim_rotate`, `mailbox_restore`, `account_purge`, `tenant_migration`. An undefined value is `400 invalid filter`
`status`	`queued`, `running`, `succeeded`, `failed`, `partial`, `canceled`

Response 200: `{ "items": [Job], "nextCursor": null }` — no cursor yet; `null` is present so a last page and an unpaginated listing read the same.

A Job:

Field	Meaning
`id`, `type`, `status`	
`progress`	`{total, done, failed}`

`createdAt`, `finishedAt`	
`resultUrl`	Pre-signed report when there is one, else null
`errors`	Stored error strings, capped at 100 (the `progress.failed` count stays exact)
`attempts` / `maxAttempts`	How many times a worker has **claimed** the job / the poison bound (3). A worker holds a lease, not a lock, so `running` alone does not say whether this is the first attempt or the last
`leaseExpiresAt`	When the current worker's claim lapses and another may reclaim it; null when nothing holds it

GET /jobs/{jobId}

One job, if it is yours.

- **Operation:** `getJob` · **Scopes:** `platform:admin`, `tenant:admin`, `helpdesk:read` · **Binding:** self-scoped — the handler reads the job, takes the owner off the **record**, and checks the caller covers it, answering exactly as it does for a job that does not exist

Response 200: the Job. **404** for unknown-or-not-yours, indistinguishably.

Both routes answer **501 `no job store`** on a node not configured to run jobs.

4.12 Usage

GET /usage/tenants/{tenantId}

Metering snapshot for billing.

- **Operation:** `getTenantUsage` · **Scopes:** `platform:admin`, `reseller:admin`, `tenant:admin` · **Binding:** the tenant in the path

Query parameter	Meaning
`period`	ISO month, e.g. `2026-07`. Absent = the current (open) period. Malformed = `400`

Response 200. The spec's core fields plus the honesty fields this build adds:

Field	Meaning
`tenantId`, `period`	
`mailboxes`, `activeMailboxes`	**Stock** figures — sampled daily, aggregated as the period **peak** (never summed:

	summing daily counts would bill one mailbox thirty times)
`storageLogicalBytes`	Tenant total, from the same samples
`storagePhysicalBytes`	Always `null`, with the reason: blobs are content-addressed and deduplicated platform-wide, so a body shared between tenants has no defensible split. A fabricated number on an invoice is worse than an absent one
`inboundMessages`, `outboundMessages`	**Flow** figures — counted inside the transactions that made the events durable, aggregated as the period total
`final`	Whether the period is frozen. A frozen figure never changes; freezing happens once, after a grace window for in-flight events
`complete`	False when at least one daily stock sample did not see every account — the mailbox and storage figures are then a **floor**, not a measurement
`stockSamples`	On how many days of the period stock was actually sampled
`notes`	Plain-language caveats, e.g. `"not final: ... Do not raise an invoice from this."`

When the tenant's plan defines `included` allowances, the response also scores usage against them (quantity, allowance, excess, rate) — the arithmetic beyond that belongs to a billing system.

Do not bill from a response whose `final` is false or whose `complete` is false. The `notes` array says so in words.

4.13 Queue (mailops)

The outbound queue: messages accepted for remote delivery and not yet in a terminal state. Queue entries carry no tenant-resolvable subject, so all queue routes are platform-bound — which is exactly why `mailops:write` is a separate scope from `mailops:read`: an operator who should see the backlog is not automatically one who should cancel out of it.

GET /mailops/queue/messages

Search the outbound queue.

- **Operation:** `searchOutboundQueue` · **Scopes:** `mailops:read`, `platform:admin` · **Binding:** platform

Query parameter	Meaning
`limit`	Page size
`cursor`	**Refused with 400** — cursor pagination is not implemented here, and silently ignoring the parameter would serve page one twice and call it page two. Narrow with the filters
`tenantId`	Filter
`nextHopDomain`	Filter
`status`	`scheduled`, `in_flight`, `deferred` (terminal states never appear — a delivered or bounced entry leaves the queue)

Response 200: `{ "items": [QueueMessage], "truncated": bool }`. A QueueMessage: `queueId`, `messageId`, `tenantId`, `sender`, `recipients[]`, `nextHopDomain`, `status`, `attempts`, `nextRetryAt`, `lastError`, `ipPool`. `truncated: true` means the listing hit its bound — narrow, do not assume completeness.

GET /mailops/queue/messages/{queueId}

One queue entry.

- **Operation:** `getQueueMessage` · **Scopes:** `mailops:read`, `platform:admin` · **Binding:** platform

Response 200: the QueueMessage. A not-found on a truncated scan says "not found here", not "does not exist" — the detail text distinguishes them.

DELETE /mailops/queue/messages/{queueId}

Cancel a queued message. This **bounces** it: the entry settles exactly the way retry-exhaustion settles, so the acceptance-ledger flip and the RFC 3464 non-delivery report to the sender land in one transaction.

- **Operation:** `cancelQueueMessage` · **Scopes:** `mailops:write`, `platform:admin` · **Binding:** platform

Field	Type	Required	Meaning
`reason`	string, 1–200 chars	**yes**	Not only audit — **it reaches the sender inside the bounce they receive.** Write it for them

Response 204.

Refusals:

- **400** with no (or empty) `reason`.
- **409** while a worker holds a live lease on the entry — forcing a bounce under an in-progress SMTP conversation risks a message both delivered and reported failed. Wait for the lease to lapse or the attempt to finish.
- **501** on a node with no mailbox store: settling as bounced promises the sender a notification this node could not deliver.

POST /mailops/queue/messages/{queueId}/retry

Force an immediate retry of a deferred message.

- **Operation:** `retryQueueMessage` · **Scopes:** `mailops:write`, `platform:admin` · **Binding:** platform
- No body.

Response 202: `{ "queueId", "outcome" }` where `outcome` is one of:

`outcome`	Meaning
`rescheduled`	Brought forward; claimable now
`alreadyDue`	It was already claimable
`inFlight`	A worker holds it right now — unlike cancel, this is **reported rather than refused**, because the delivery you asked for is already happening

`attempts` is deliberately NOT reset. Resetting would grant the message a fresh retry schedule and move the moment the sender finally learns of a failure hours or days into the future. The bounce clock is the sender's, not ours.

GET /mailops/queue/status

Is outbound delivery paused, and why.

- **Operation:** `getQueueStatus` · **Scopes:** `mailops:read`, `platform:admin` · **Binding:** platform

Response 200: `{ "paused": bool, "by": string, "reason": string, "since": date-time }` (the last three present when paused).

PUT /mailops/queue/status

Pause or resume outbound delivery.

- **Operation:** `setQueueStatus` · **Scopes:** `mailops:write`, `platform:admin` · **Binding:** platform

Field	Type	Required	Meaning
`paused`	boolean	yes	
`reason`	string, max 200	required when pausing	Stored and reported by the status route

Response 200: `{ "paused": bool }`.

The pause is **fleet-wide, not node-local**: the flag is stored, and every queue runner on every frontend reads it at the top of its tick, so it takes hold everywhere within one tick. A pause held in one process's memory would stop that node while the others kept draining — which is exactly the wrong answer during an incident. Pausing stops **claiming** new deliveries; it deliberately does not stop delivering bounce notifications for failures that already happened.

4.14 Message trace

GET /mailops/trace/{messageId}

End-to-end trace of what became of one message.

- **Operation:** `traceMessage` · **Scopes:** `mailops:read`, `helpdesk:read`, `platform:admin` · **Binding:** platform (a queue id and an accept id name no tenant-owned resource, so a read grant here reaches every tenant's mail — this is a helpdesk surface, not a tenant-scoped one)

The path parameter is an **internal queue id or an accept id** — the identifier the SMTP session was told at acceptance. Either resolves; a queue entry carries its accept id, so holding one of the pair finds the other.

Response 200: ordered trace events, each `{ "ts", "stage", "detail", "node", "traceId" }`. Stages follow the message's actual path (acceptance → hygiene → delivery or queue-out → remote delivery → DSN). `traceId` is `null` on this build: the OTel context is deliberately not propagated across the async queue hop, and reporting a trace that stops half way would be worse than none.

Refusals:

- **501 `message-id lookup not implemented`** when the parameter looks like an RFC 5322 Message-ID (`<...@...>`): this server keeps no index from Message-ID to delivery. Trace by queue id (from the queue search) or accept id.
- **404** — nothing in the queue or the accept ledger for that id. When the queue scan was truncated, the detail says "not found here, not does-not-exist".
- **501** on a node without a store handle (no ledger view).

4.15 Ops (overview, alerts, deliverability)

Read-only windows for an admin console. All three are node-scoped snapshots, not a monitoring system — Prometheus is the monitoring system.

GET /ops/overview

Feature configuration and live process counters for this node.

- **Operation:** `getOpsOverview` · **Scopes:** `mailops:read` · **Binding:** platform

Response 200: `{ "features": {...}, "counters": {...} }` — what is switched on (RBL mode, AV, outbound caps, at-rest encryption, OTel export, FTS, ...) as the process parsed it at startup, plus this one process's own counters. It deliberately reports only what this process knows about itself.

GET /ops/alerts

Currently firing alerts, relayed from Alertmanager.

- **Operation:** `getOpsAlerts` · **Scopes:** `mailops:read` · **Binding:** platform

Response 200: `{ "alerts": [...] }`.

When Alertmanager is unconfigured this endpoint answers **501**, and when it is unreachable, **502** — it REFUSES rather than answering `[]`, because an empty list is indistinguishable from "nothing is wrong". Treat a refusal as "the instrument is dark", never as calm.

GET /ops/deliverability

Reputation feeds: DMARC trend, blocklist self-check, outbound gate.

- **Operation:** `getOpsDeliverability` · **Scopes:** `mailops:read` · **Binding:** platform

Response 200: three sections, each reporting its own absence honestly:

Section	Contents
`dmarc`	The aggregate-report trend from stored rua summaries (reports, message counts, failing sources), or "unconfigured" when no rua mailbox is set — never a guessed domain
`blocklist`	The last self-check recorded by the blocklist watcher, with its own timestamp. `"status": "neverRan"` is a distinct answer — **not the same as clean**
`outbound`	The outbound-cap gate counters (refusals, auto-locks)

4.16 Platform settings

The protocol switches: whether the blocklist is consulted, whether a DMARC failure refuses, whether malware scanning runs, recipient verification, sub-addressing, spam-filter

behaviour, and their numeric companions. Domain-scoped values for many of the same keys live under `/domains/{domainId}/settings` (section 4.4).

GET /settings

All platform-scope settings, stored and effective.

- **Operation:** `listPlatformSettings` · **Scopes:** `platform:admin` · **Binding:** platform

Response 200: each entry reports `stored` (what an operator set) and `fromEnvironment` **separately** — clearing a setting returns it to the environment value, and an operator needs to see what that is — plus the effective value and `refreshedAt`: when this node last re-read the record. Changes take effect on every frontend within one refresh interval (default 10 s), not instantly, and `refreshedAt` is how you observe that.

PUT /settings/{key}

Set one platform-scope setting.

- **Operation:** `setPlatformSetting` · **Scopes:** `platform:admin` · **Binding:** platform
- Body: `{ "value": boolean | integer }` per the setting's declared type.

Response 200: the new effective value. Out-of-range values are **refused, never clamped** — what you set is what is in force. A switch turned ON whose structural prerequisite is missing (a blocklist with no dedicated resolver, malware scanning with no scanner address) is stored and answered with a `warning` member, because the alternative is a cheerful 200 for a check that will never run.

DELETE /settings/{key}

Clear one platform-scope setting, returning it to the environment value.

- **Operation:** `clearPlatformSetting` · **Scopes:** `platform:admin` · **Binding:** platform

Response 200: the new effective value.

4.17 License

GET /license

License identity, computed state, usage, and entitlements for this deployment.

- **Operation:** `getLicense` · **Scopes:** `mailops:read`, `platform:admin` · **Binding:** platform

Response 200:

Field	Meaning
`state`	`community` / `valid` / `expiringSoon` / `grace` / `lapsed` / `invalid`. The same single evaluation the server boot-logs,

	exports as a metric, and hands the provisioning gate — one evaluator, never two opinions. An unusable file reads as `invalid`, never as `community`
`daysLeft`	Days to expiry (a large sentinel for community — there is nothing to renew)
`licensee`, `limits`, `features`	The signed file's public fields. Signature bytes never leave the file
`usage`	`{ "accounts", "domains", "cells" }` — **each `null` when it could not be read, never zero** (zero reads as headroom)
`entitlements`	The **resolved** booleans per feature — `multi-cell`, `delegated-rbac`, `commerce`, `sso` — not the raw list, so community and lapsed show all false
`ssoInUseWithoutEntitlement`	True while LDAP/OAuth is configured without the `sso` entitlement. Sign-on is never cut off; the flag is the enforcement
`enforced`	Whether this node runs the provisioning gate at all

This route itself refuses nothing. Enforcement lives at the provisioning choke points:

- Creating an account, domain, or cell past the tier's numbers — or while `lapsed` — answers **403** with title **`license limit`**, and the detail names the limit and whose license (e.g. "the license for X allows 7 domains and this would be the 8th — raise the limit or retire a domain").
- Entitlement-gated writes (`createDelegation`, plan writes, tenant plan subscription, `bulkProvisionAccounts`) answer **403** with title **`license entitlement`** — a deliberately distinct title, so logs can tell "over a number" from "not in the tier".
- **The delivery path never consults the license.** Mail flow, sign-in, and reads are never refused for license reasons, in any state.

4.18 Webhook endpoints

Registering and inspecting the endpoints ZephyrAB posts events to. **Section 7 is the integrator's half** — the event catalogue, the payload shapes, how to verify a signature, and what delivery does and does not promise. This section is the four management calls.

All four are **self-scoped**: the listing reads only endpoints whose subject your grants cover, and a fetch by id answers exactly as it does for an id that does not exist. There is no `Idempotency-Key` on registration — a duplicate endpoint is visible and deletable, where a mis-keyed one is not.

Every route answers **501 `webhooks are not enabled`** when the deployment has not switched the feature on.

GET /webhooks

The endpoints this caller administers.

- **Operation:** `listWebhookEndpoints` · **Scopes:** `platform:admin`, `tenant:admin`, `domain:admin` · **Binding:** self-scoped

Response 200: `{ "items": [WebhookEndpoint], "nextCursor": null }`. An endpoint carries `id`, `url`, `subject` (`{kind, id}`), `events`, `enabled`, `createdAt` (Unix seconds), `consecutiveFailures` and `lastError`.

The signing secret is never in this response, or any other. It exists in plaintext exactly once, in the answer to the registration that created it.

`consecutiveFailures` and `lastError` are the per-subscriber health view, and a success clears them — so a non-zero value means *now*, not "once had a bad day".

POST /webhooks

Register an endpoint.

- **Operation:** `createWebhookEndpoint` · **Scopes:** `platform:admin`, `tenant:admin`, `domain:admin` · **Binding:** self-scoped

Field	Type	Required	Meaning
`url`	uri	yes	HTTPS only, and must resolve entirely to public addresses. Section 7.4 has the full rule
`subject`	string	yes	`platform`, `tenant:<uuid>` or `domain:<uuid>` — the same spelling a delegation grant uses. **A claim, checked against your own grants**
`events`	array of string	yes, min 1	From the closed set in section 7.1. An endpoint subscribed to nothing would never be called, so an empty list is refused

Response 201: the endpoint, plus two fields present only here:

Field	Meaning
`secret`	The HMAC signing key, hex. **Store it now.** It is sealed at rest and there is no path that renders it again; rotating means registering a new endpoint and deleting the old one
`secretNote`	The same warning in words

and, when you subscribed to an event this build does not fire:

Field	Meaning
`notYetEmitted`	The subscribed event names that never arrive on this build — `domain.verified` today (section 7.1). The subscription is stored and will start working when they are emitted
`notYetEmittedNote`	Why, and that their absence is not a delivery failure

That field exists so the answer at the moment of subscribing tells you, rather than leaving you to discover it by waiting for an event that cannot come. Check for it; an empty-or-absent `notYetEmitted` means everything you asked for is live. (It is emitted by the server and is not declared in the OpenAPI schema, so a strict validator may not expect it.)

Refusals:

- **400 `unknown event`** — a name outside the closed set, **refused rather than dropped**. Silently registering an endpoint subscribed to less than it asked for looks fine until the missing event never arrives, and by then nobody remembers the typo. The response lists the whole set.
- **400 `invalid subject`** — not one of the three spellings.
- **400 `invalid webhook url`** — not HTTPS, unparseable, carries userinfo (`https://user@host/`), over 2000 characters, or resolves to a private address. The detail names which.
- **400** — the deployment already holds its 256-endpoint limit, or the event list is empty.
- **403 `subject not yours`** — you may only register for a subject you administer. This cannot be an enumeration oracle: the answer depends only on what you hold, never on whether the named tenant exists.
- **501 `no webhook key`** — the node holds no webhook master key, so it cannot store an endpoint secret. An HMAC secret cannot be hashed like other credentials (signing needs it back), so storing one in the clear is not an option and the node says so instead. A node in this state still **delivers** for endpoints registered elsewhere; it just cannot register new ones.

DELETE /webhooks/{webhookId}

Remove an endpoint **and everything queued for it**, in one transaction.

- **Operation:** `deleteWebhookEndpoint` · **Scopes:** `platform:admin`, `tenant:admin`, `domain:admin` · **Binding:** self-scoped

Response 204. Somebody else's endpoint answers the same 404 as one that never existed.

Deletion is exact even though the enqueue path works from a cached endpoint list: a worker re-reads the endpoint to get its secret, so a delivery for an endpoint that has gone is dropped rather than sent. The eventual half of that cache can cost a brand-new endpoint a missed event; it can never deliver to a removed subscriber.

GET /webhooks/{webhookId}/deliveries

What is **still queued** for this endpoint.

- **Operation:** `listWebhookDeliveries` · **Scopes:** `platform:admin`, `tenant:admin`, `domain:admin` · **Binding:** self-scoped
- Query: `limit` (default 50, max 200).

Response 200: `{ "items": [...], "truncated": bool, "nextCursor": null }`. Each item: `id` (the `X-Webhook-Id` a receiver will see), `event`, `attempts`, `maxAttempts`, `dueAt`, `occurredAt`, `leased`.

This is a backlog, not a history. A delivered, abandoned or dropped delivery leaves the queue, so an empty list is the healthy answer and says nothing about what was sent. If you need a record of what a receiver was given, keep it at the receiver.

`truncated: true` means the walk hit its bound before the end of the queue — the queue is ordered by due time and has no per-endpoint index, so this reads its head and filters. Treat it as "there may be more", never as a count.

5. Self-service API reference

These routes live on the **mail origin** (`https://mail.example.com`). They are what webmail itself uses; any client may use them.

Authentication. Unless marked unauthenticated, routes take **HTTP Basic**: username = the mail address (aliases resolve to the canonical account), password = the account password **or a session token** from `POST /account/session`. A session token is minted after both factors and is accepted anywhere a password is — it is how a client avoids resending the real password on every request, and how a two-factor account authenticates non-interactive things (including the admin password grant). **Application passwords are NOT accepted here**: the interactive surface is exactly the door a device credential must not open, so an app password cannot become a way around a second factor.

`Authorization: Bearer <jwt>` is accepted here too, where the deployment is pointed at an external identity provider — on every authenticated route, `POST /account/session` included. That covers the whole settings surface, so a federated user administers their own account without a ZephyrAB password existing at all. The scheme in the header decides once; a token is never tried as a password. Most of the routes below that ask for `currentPassword` have a different second proof under a bearer token — freshness rather than a second value — and answer `401 reauthenticationRequired` when the provider's authentication is too old; the encryption and assistant routes are the exception and still need a real password. Section 2.8 has the whole rule and names the exception.

Identity is the credential. No self-service path carries an account id — there is nothing for a caller to claim, so there is nothing to check that could be forgotten.

Errors are JSON `{ "error": "codeName", "description": "..." }` with camelCase codes. Authentication failures are rate-limited per source IP; over the limit everything answers 401 "too many attempts".

5.1 Signup and recovery (unauthenticated)

POST /signup

Create an account with an invite code. The only unauthenticated write path on the server, which is why almost everything in it is a refusal.

Field	Type	Required	Meaning
`inviteCode`	string	yes	Case-insensitive; dashes ignored
`localPart`	string	yes	The part before the `@`. **The domain, tenant and quota come from the invite**, never from the request. Charset is alnum/`.`/`-`/`_` — deliberately narrower than RFC 5321
`password`	string	yes	Minimum 12 characters — and that is the only rule
`displayName`	string	no	
`recoveryAddress`	string	no	Validated (MX + reachability probe) **before the invite is spent**, so a typo is

	fixed while the person is still at the form. Stored unverified; confirmed by the emailed link

Response 201: `{ "address": "chosen@invited-domain.example" }`.

Refusals, each a stable code:

- `invalidInvite` (403) — unknown, expired, and exhausted invites answer **identically**: saying "expired" confirms a code was once real, which is what makes a leaked list worth grinding. **The invite is checked before any probe runs**, so a caller without a valid code can never make this server open outbound connections.
- `addressTaken` / reserved local part — **the same 409 for both** (taken and reserved are indistinguishable, or the form becomes an address-enumeration oracle for any code holder). Reserved local parts (`postmaster`, `abuse`, …) are never grantable.
- `weakPassword` (400) — under 12 characters.
- `unreachableRecoveryAddress` (400) — the recovery domain publishes no MX or answers nothing on port 25. The invite is **not** spent by this refusal.
- `license limit` (403) — account creation is license-gated like the admin door.
- Rate-limited per source before any expensive work.

POST /recover

Request a password-reset link.

- Body: `{ "address": "user@example.com" }`.

Response 202, always the same: `{ "status": "If that address has an account with a verified recovery address, a reset link is on its way." }`

The answer is **identical** for an existing account, an unknown address, an account with no recovery address, and an unverified one — anything else makes an unauthenticated endpoint an address-existence oracle. The reset link is mailed to the **verified recovery address**; tokens are single-use, 30-minute, stored hashed.

POST /recover/confirm

Spend a reset token.

Field	Required	Meaning
`token`	yes	From the emailed link
`newPassword`	yes	Minimum 12 characters — checked **before** the token is spent, so a short password does not burn the link

Response 204 on success. Changing the password revokes every outstanding token and session.

GET /account/recovery/verify?token=…

The target of the recovery-address confirmation link (opened in a browser, not called by API clients). Marks the recovery address verified. A used, expired, or unknown token answers a plain-text 400: "That link is not valid, or it has already been used or expired."

5.2 Password, recovery address, sessions

POST /account/password

Change the password. Basic auth **and** the current password in the body (re-authentication, deliberately — a stolen session must not be enough).

Field	Required
`currentPassword`	yes
`newPassword`	yes (min 12)

Response 204. Revokes outstanding sessions and recovery tokens.

GET /account/recovery

Is this account recoverable?

Response 200: `{ "address", "verified", "recoverable", "configurable" }`. `recoverable` is computed server-side (address set **and** verified) so clients cannot derive it wrongly; `configurable` says whether this deployment can send recovery mail at all — when false, don't send users to set one up.

PUT /account/recovery

Set or change the recovery address.

Field	Required	Meaning
`currentPassword`	yes	Momentary access to a signed-in client must not be able to point recovery elsewhere
`address`	yes	Checked for reachability (MX required plus a port-25 banner probe — stricter than the RFC on purpose, because "there is a web server there" is not "there is a mail system there")

Response 204. A confirmation link is mailed; the address counts for nothing until the link is opened. Refusals: `unreachableRecoveryAddress` (with transient-vs-permanent wording), `noMailer` when the deployment cannot send.

POST /account/session

Exchange a credential for a session token. This is the sign-in path whether or not a second factor is configured.

Two ways in, and the header decides which:

- **`Authorization: Basic`** — address (aliases resolve) + the real account password, plus a TOTP code in the body when one is enrolled. Body: `{ "code": "123456" }`, empty or omitted when no second factor is enrolled, or on the first attempt before the client knows one is needed.
- **`Authorization: Bearer <jwt>`** — an identity provider's token, where one is configured. **No TOTP is asked for**, because the provider is the authentication authority and applies the organisation's own multi-factor policy; demanding ours as well would refuse every federated user who had enrolled, with no way to answer. This is the route that lets an application mint a mail session for somebody it has just authenticated, using that person's own token — see section 2.8 for why that removes the need for an impersonation endpoint.

Response 200: `{ "token": "...", "expiresInSecs": 43200 }` (12 hours). The token is a full credential everywhere — self-service, JMAP, IMAP, admin password grant.

Refusals: `totpRequired` (401) when a code is needed and none was sent — **not counted as a failed attempt**, the password was right; `badCode` (401) for a wrong code (counted); one indistinguishable 401 for wrong address/password, and one for a token that does not verify. A directory this node cannot read answers **503**, never 401 — an unreachable backend and a bad credential are different facts.

Note what follows for the **admin** control plane: its password grant spends a password or a session token, so an account with no ZephyrAB password can now reach it — by minting a session here with a bearer token first, then presenting that session token in the grant's `password` field.

5.3 Application passwords and two-factor

GET /account/app-passwords

Response 200: `{ "appPasswords": [ { "id", "label", "scopes", "createdAt", "lastUsedAt" } ], "max": ... }`. `lastUsedAt: 0` means never used — the one that is safe to revoke.

POST /account/app-passwords

Mint a device credential.

Field	Required	Meaning
`currentPassword`	yes	Re-authentication
`label`	no	e.g. "Tablet mail app"
`scopes`	no	Coarse on purpose: `mail` (IMAP/POP3/SMTP submission) and/or `dav` (accepted spellings: `dav`, `calendar`, `contacts`). Unknown names → `invalidScope`

Response 201: `{ "secret", "id", "label", "scopes" }`. **The `secret` is shown once and never again** — it is stored hashed. App passwords work on the protocol surfaces their scope names and are refused at every interactive surface and at the wrong-scope door (a real credential at the wrong door is refused without counting as a failed attempt).

DELETE /account/app-passwords/{id}

Revoke one. **Response 204.** Idempotent.

GET /account/totp

Response 200: `{ "enabled", "pending", "recoveryCodesRemaining" }`. `pending` is a third state, not a shade of the other two: an enrolment nobody confirmed protects nothing, and the user should be told to finish it.

POST /account/totp/enroll

Body: `{ "currentPassword" }`. **Response 200:** `{ "otpauthUri", "recoveryCodes": [...] }` — the recovery codes are shown **once**. Enrolment is pending until confirmed.

POST /account/totp/confirm

Body: `{ "code" }` from the authenticator app. Confirms the enrolment. A wrong code answers "that code is not right. Check your phone's clock is correct."

DELETE /account/totp

Disable the second factor. Body: `{ "code", "currentPassword" }` — the password **and** a working code, exactly like turning it on.

5.4 Settings, rules, spam, recently deleted

GET /account/settings · PUT /account/settings/{key} · DELETE /account/settings/{key}

The account layer of the same settings registry the admin API exposes (section 4.4). `GET` lists every setting with `selfService` visibility; `PUT` takes `{ "value": ... }` and obeys the same refuse-don't-clamp rule; `DELETE` clears the account's opinion. Only settings

whose registry entry says `selfService` are writable here — protective settings can be switched **on** by the account and can never switch a domain's protection off.

GET /account/rules · PUT /account/rules

Vacation auto-reply and forwarding, as a form. The form **generates the account's Sieve script** rather than storing a parallel setting — what the panel shows is what will run at delivery. `GET` reads the stored script back through the real interpreter; a hand-written script it could not have generated comes back `custom: true`, and `PUT` refuses to overwrite it without `replaceCustom: true` (a filter replaced by a checkbox has no undo). Vacation replies obey RFC 3834 (one per correspondent per interval; suppressed for bulk/list/auto-submitted mail); forwarding is capped at 5 addresses, with "keep a copy" explicit.

POST /account/spam/train

Tell the filter about one message.

Field	Required	Meaning
`messageId`	yes	`{mailbox}:{uid}` — the same id shape JMAP `Email/get` hands out
`class`	yes	`spam` or `ham`

Response 200: `{ "trained": bool, "retracted": "spam"|"ham" (when this replaced the opposite verdict), "reason": "..." (when nothing was written) }`. `trained: false` is ordinary — the message was already marked this way. The same training happens implicitly when a client moves mail into or out of Junk over IMAP or flips `$junk`/`$notjunk` over JMAP; all doors share one implementation, so marking from two clients cannot double-count.

GET /account/deleted

Messages recoverable after a permanent delete.

Response 200: `{ "enabled", "retainSecs", "items": [ { "handle", "mailbox", "deletedAt", "size" } ], "truncated" }`. `enabled: false` with an empty list is a different fact from an empty list on a deployment that keeps deletes — a client must be able to say "your mail was never recoverable here" rather than "nothing to recover".

POST /account/deleted/{handle}/restore

Put one back. **Response 200:** `{ "mailbox", "uid" }` — the mailbox it returned to and its **new** uid (never the old one, which may have been reissued). If the folder it lived in no longer exists, it is re-created. A second restore of the same handle, a malformed handle, and a neighbour's handle all answer the same 404.

5.5 Link attachments

GET /account/links

The bounds, before you upload. **Response 200:** `{ "enabled", "thresholdBytes", "maxBytes", "maxTtlDays", "maxDownloads", "scanning" }`. When the feature is off: `enabled: false` and zeros. Exists so a sender learns the maximum is 30 days **before** typing 365 and waiting out a 40 MB upload.

POST /account/links

Create a share link. **`multipart/form-data`**, not JSON:

Part	Meaning
`file`	The bytes; filename and content type ride the part headers
`expiresDays`	Omit entirely when blank — do not send empty strings
`maxDownloads`	Omit when blank. `0` means **unlimited** on this platform, so a blank read as 0 would be wrong
`password`	Omit when blank; minimum length applies (`weakPassword`)

Response 201: `{ "url", "expiresAt", "expiresDays", "maxDownloads", "passwordSet", ... }` — **the applied values**, read back from the stored record. The response reports what was applied precisely so asked-for and in-force can never silently differ. Options **outside the configured bounds are refused** (`optionOutOfBounds`, 400), never quietly replaced.

Other refusals: `malformedUpload` (400), `overHardCap` (400), `scannerUnavailable` (the file could not be scanned and this deployment fails closed), infected files refused at upload — the sender is told at submission, not the recipient at download. `linksNotConfigured` (501) when the feature is off.

DELETE /account/links/{id}

Revoke a link. The `{id}` is the **token hash** (from your sent copy's metadata), deliberately not the token itself — if the URL were the handle, anyone holding the link (including the recipient) could revoke as well as download. Revocation is immediate: nothing is cached. Someone else's link answers exactly like one that never existed. Re-revoking succeeds (idempotent; case-insensitive hex).

5.6 AI assistance

GET /account/ai

What assistance is available to this account: whether the operator runs a backend, whether the account registered its own, and which takes precedence. When neither exists the client should say so rather than offering a dead button.

PUT /account/ai

Register (or decline) a backend of your own.

Field	Meaning
`password`	Re-authentication
`delivery`	`serverProxied` (the server calls your endpoint) or `clientDirect` (the browser calls it; **the server refuses to relay** — `409 clientDirect` — and never stores a key for it, structurally: there is no field to store one in)
`endpoint`, `model`, `external`, `apiKey`	The backend. An `apiKey` offered for a client-direct config is refused, not dropped
`declineOperator`	Refuse the operator's backend independently — "no assistant at all" is expressible

POST /account/ai/remove

Remove the account's own configuration. A POST rather than DELETE because it carries `password`, and request bodies on DELETE are something proxies disagree about.

POST /account/ai/draft

Server-proxied drafting. Body: `{ "task", "instruction", "original", "draft", "attempt", "sendPlaintextToServer" }`. For an at-rest-encrypted account, the server-proxied path is refused unless `sendPlaintextToServer: true` — a fresh decision each request, not a setting. Refusals are 409s with actionable codes ("there is no backend", "yours is client-direct, call it yourself"). The model drafts; **it never sends** — the result is text for a human to act on.

5.7 Encryption (at-rest, per-user)

GET /account/encryption

Response 200: `{ "optedIn", "required", "requiredBy": "none"|"account"|"domain"|"domain+account", "domainRequires", "mailIsBeingRefused", "format": "openpgp"|"smime"|null, "keys":`

`[ { "fingerprint", "addedMs", "active", "format", "label", "notAfterMs" } ] }`.

`mailIsBeingRefused` is the field to show in red: encryption is required and there is nothing to encrypt to, so incoming mail is bouncing right now. Key material is never echoed in listings. `notAfterMs` (S/MIME) is shown, never enforced — expiry warns, it does not stop mail.

PUT /account/encryption

Opt in or out. Body: `{ "currentPassword", "enabled" }`. **Opting in requires a usable public key already uploaded** — under fail-closed delivery, an opted-in account with no key bounces everything it is sent, and that state must not be one click away. Opting out of the account layer never overrides a domain mandate (the union rule).

POST /account/encryption/keys

Upload a public key. Body: `{ "currentPassword", "publicKey" }` (`certificate` is accepted as an alias for S/MIME). OpenPGP armored keys and S/MIME certificates are both accepted, but **one format per account** — the other format is refused while any active credential of the first exists. A block that parses as a **private** key is refused whatever its armor header claims. Encryption uses the encryption **subkey** on modern OpenPGP certificates.

POST /account/encryption/keys/remove

Remove a key. Body: `{ "currentPassword", "fingerprint" }`. (Also served as `POST /account/encryption/keys/{fingerprint}/remove` — same handler, password still from the body.) Removing the last key while opted in flips the account into the mail-refused state, and the panel says so.

5.8 Client settings

GET /account/clients

What a mail app should be configured with, served from the same configuration that decides which listeners actually run — a port that is not served cannot be advertised.

Response 200: `{ "host", "imap", "pop3", "submissions", "submission", "davUrl", "davAutodiscover", "autoconfig" }`. Fields are `null`/absent for services this deployment does not run. **503 `notConfigured`** when the deployment advertises nothing.

5.9 The download origin

Served from a **separate origin** (`https://files.example.com`). No account, no cookie — the URL is the credential.

GET /d/{token}

Fetch a shared file. Also `GET /d/{cell}/{token}` in multi-cell deployments (the extra segment routes to the owning cell).

- **200** — the file, always `Content-Disposition: attachment`, content type collapsed to safe values (HTML, SVG, anything script-ish becomes `application/octet-stream`), `nosniff`, `default-src 'none'; sandbox` CSP, no referrer. If the link has a password, 200 is instead a minimal HTML form.
- **404** — unknown token, or a valid token presented on the wrong origin (origin isolation is enforced in the handler, not just the proxy).
- **410** — expired, revoked, or download-cap exhausted. **Byte-identical for all three**: a distinguishable answer would tell whoever holds a leaked URL that someone else already fetched it. The sender is notified of expiry exactly once, however many times a dead link is hammered.
- **403** — the stored scan verdict is infected. Permanent, identical under both scanner-failure policies.
- **503** — the file could not be scanned right now and this deployment fails closed; retry later.

POST /d/{token}

Submit the link password: an ordinary form POST with a `password` field (never a query string — URLs end up in logs). Wrong password says so — a valid 128-bit token already proves the holder has the link, so hiding "wrong password" would cost a real user their only clue while telling an attacker nothing. The password is checked **before** the download cap is claimed, so guessing cannot burn a limited link, and before the bytes are read, so guessing cannot buy the server work.

6. Mail protocols

All protocol surfaces authenticate against the same account store as the HTTP APIs: address (or alias) + password, session token, or an application password on the surfaces its scope covers. Where an external identity provider is configured, the OAuth mechanisms appear alongside — and only then: a mechanism that cannot succeed is never advertised. IMAP, POP3, submission and ManageSieve take the token through SASL (`XOAUTH2`, `OAUTHBEARER`); JMAP takes it as `Authorization: Bearer`; CalDAV/CardDAV do not take one at all. Section 2.8 is the full rule, including the claim mapping and the re-authentication window.

6.1 JMAP

Discovery. The Session resource is served at `https://mail.example.com/.well-known/jmap` (and at `/jmap/session`), authenticated with **`Authorization: Basic`** — or **`Authorization: Bearer <jwt>`** where an identity provider is configured (RFC 8620 §8.2; see section 2.8). Every JMAP route takes either, including upload, download,

EventSource and the WebSocket upgrade. An unauthenticated request gets a 401 challenge; that challenge names `Basic` whatever the deployment's identity configuration, so do not read it as "bearer is unsupported here".

Capabilities advertised (and enforced — a `using` entry outside this set is `urn:ietf:params:jmap:error:unknownCapability`):

Capability	Notes
`urn:ietf:params:jmap:core`	`maxSizeUpload` 76,578,816 (derived from the 100 MiB message ceiling minus MIME overhead — the same constant the upload route enforces), `maxConcurrentUpload` 4, `maxConcurrentRequests`, `maxCallsInRequest` 16, `maxObjectsInGet` 500, `maxObjectsInSet` 500, collations `i;ascii-casemap`, `i;octet`
`urn:ietf:params:jmap:mail`	Per-account: `maxMailboxesPerEmail: 1` (the single-mailbox storage model, declared the spec's own way), `maxSizeAttachmentsPerEmail`, `emailQuerySortOptions`: `receivedAt`, `size`, `subject`, `from`, `to`, `sentAt`, `hasKeyword`, `relevance` (ZephyrAB's own ranked sort — valid only with a `body` or `text` filter condition, refused by name without one)
`urn:ietf:params:jmap:submission`	`maxDelayedSend: 0` and `undoStatus` always `final` — no delayed send is advertised because none is implemented; a sent message is not recallable
`urn:ietf:params:jmap:contacts`	RFC 9610
`urn:ietf:params:jmap:calendars`	Calendars draft
`urn:ietf:params:jmap:quota`	RFC 9425; `Quota/get` returns real figures
`urn:ietf:params:jmap:websocket`	RFC 8887 — present only when the deployment enables it (see below), absent otherwise

URL templates in the Session (RFC 8620 templates, absolute):

- `apiUrl`: `/jmap` (POST)
- `downloadUrl`: `/jmap/download/{accountId}/{blobId}/{name}?accept={type}`
- `uploadUrl`: `/jmap/upload/{accountId}/`

- `eventSourceUrl`: `/jmap/eventsource?types={types}&closeafter={closeafter}&ping={ping}`

Blobs. A blobId is `{mailbox}:{uid}` for a whole message and `{mailbox}:{uid}#{part}` for one part — account-scoped handles, never content hashes. Upload is `POST` to the upload URL (registered with and without the trailing slash, so either substitution works) and returns `{ "accountId", "blobId", "type", "size" }`; uploaded blobs are staged with a bounded per-account budget and expire if never referenced by an `Email/set`. Downloads are always `Content-Disposition: attachment` with script-ish content types collapsed to `application/octet-stream` — the bytes are a stranger's mail on the server's own origin.

Push.

- **EventSource** at `eventSourceUrl`: an initial `state` event, then a `StateChange` per change. `accountId` is **optional** — RFC 8620 §7.3's template has no slot for one; it is still checked when sent. Browsers cannot attach an `Authorization` header to EventSource, so an unauthenticated stream gets a bare 401 (no Basic challenge — no password popup) and well-behaved clients fall back to polling.
- **WebSocket** (RFC 8887), when enabled: connect to `/jmap/ws` with subprotocol `jmap` and ordinary `Authorization` on the upgrade request. Credentials in the subprotocol list are **refused** — that value is echoed back and appears in proxy logs. A browser instead mints a **single-use, 30-second ticket** with an authenticated `POST /jmap/ws/ticket` (`{ "ticket", "expiresInSeconds": 30 }`) and spends it on the handshake query string — acceptable only because it is single-use and dead in half a minute. The server re-checks the credential periodically and closes (1008) on revocation or max session age; a `Response` that does not fit the socket closes it, while an advisory `StateChange` that does not fit is dropped and counted.

Result references (back-references). RFC 8620 §3.7 is supported, so a client can find messages and fetch them in **one** round trip instead of two. Prefix the argument with `#` and point it at an earlier call's result:

```
{
  "using": ["urn:ietf:params:jmap:core", "urn:ietf:params:jmap:mail"],
  "methodCalls": [
    ["Email/query", { "accountId": "u", "filter": { "inMailbox":
"INBOX" } }, "0"],
    ["Email/get", {
      "accountId": "u",
      "#ids": { "resultOf": "0", "name": "Email/query", "path": "/ids"
},
      "properties": ["id", "subject", "from", "receivedAt"]
    }, "1"]
  ]
}
```

`path` is a JSON Pointer (RFC 6901). A `*` segment applies the rest of the pointer to every item of an array and flattens one level, so `/list/*/id` collects the ids from a list of objects.

Four behaviours worth knowing before you build against it:

- **`resultOf` must name an EARLIER call**, matched on both its id and the response's `name`. A forward reference resolves against nothing.
- **A reference to a call that FAILED does not resolve.** An errored call is recorded under the name `error`, so a reference asking for `Email/query` finds no match and gets `invalidResultReference` — rather than quietly operating on the output of a call that did not happen.
- **A bad reference fails only that method call.** The rest of the batch still runs, so a client that sent six calls does not lose the five that were fine. The failing response keeps its own call id.
- **Giving both `ids` and `#ids` is `invalidArguments`**, not a silent preference for one.

A reference supplies **values, never authority**. Even used for `accountId`, the method still derives its coordinates from the authenticated identity, so the only thing a back-referenced account id can do is fail that comparison and return `accountNotFound`. Expansion is bounded; a reference that would assemble an unreasonable number of values is refused rather than truncated, because a truncated list is indistinguishable from a complete one.

Email/set patch pointers. `Email/set{update}` accepts RFC 8620 §5.3 patch pointers — `{ "keywords/$seen": true }` — resolved against the message's current value, so setting one keyword cannot clear another. Mixing whole-property and pointer forms for the same property is `invalidPatch`.

EmailSubmission. `EmailSubmission/set` sends a draft through the same signing/queue/retry path as SMTP submission. `onSuccessUpdateEmail` is honoured (issued as a real `Email/set`, e.g. to clear `$draft` and file to Sent) and only when the submission succeeded. The envelope is derived from To/Cc/Bcc when not supplied; Bcc is stripped from the wire bytes; a `mailFrom` the account does not own is `forbiddenFrom`.

Identity is read-only. `Identity/get` derives identities from the account's canonical address plus its aliases — the same directory record SMTP submission checks. There is no `Identity/set`, and every identity reports `mayDelete: false`: two sources of truth for "which addresses may this account send as" is a spoofing bug waiting to happen.

`Email/import` — the way to store a message you built yourself. `Email/set{create}` assembles a message from properties (addresses, subject, body, attachments) and refuses an explicit `bodyStructure`, so it cannot express an arbitrary MIME structure. When you hold the bytes already, upload them and import them:

```
{
  "using": ["urn:ietf:params:jmap:core", "urn:ietf:params:jmap:mail"],
  "methodCalls": [
    ["Email/import", {
      "accountId": "you@example.com",
      "emails": {
        "i1": {
          "blobId": "upload-…",
          "mailboxIds": { "INBOX": true },
```

```
          "keywords": { "$seen": true },
          "receivedAt": "2019-03-04T05:06:07Z"
        }
      }
    }, "c0"]
  ]
}
```

The stored octets are the octets you gave. Nothing parses and re-emits them. That is what makes this the path for **end-to-end encrypted mail** — an RFC 3156 `multipart/encrypted` or S/MIME `application/pkcs7-mime` message has every byte covered by a signature or an envelope, so a message rebuilt from parsed properties is not the same message, it is a broken one. It is equally the path for a migration, where fidelity is the point. Everything derived from those octets — threading, search indexing, the metadata `Email/get` answers with — is derived by the same code every delivered message goes through, so an imported message is an ordinary one afterwards.

`blobId` takes either handle shape: an `upload-…` blob you staged, or the `{mailbox}:{uid}#{part}` handle `Email/get` gives for a part of a message you already hold — which is how you import a `message/rfc822` attachment as a message of its own. Both resolve against your own account; a blob belonging to another account, one that never existed and one whose bytes have been reclaimed are deliberately one answer.

Per-entry refusals arrive in `notCreated`, and one bad entry never takes its neighbours down: `blobNotFound`, `invalidEmail` (the octets do not begin with an RFC 5322 header field), `tooLarge`, `invalidProperties`. `ifInState` is enforced — a stale value refuses the whole call with `stateMismatch` and imports nothing.

Two deviations. There is **no `alreadyExists`**: importing the same blob twice stores it twice, exactly as an IMAP `APPEND` does. And **`mailboxIds` takes one mailbox**, the same deviation `Email/set{create}` carries. At most 64 messages per call.

Multi-cell. A Session requested on a frontend that does not own the account answers with the **owning cell's** URLs (apiUrl, uploadUrl, downloadUrl, eventSourceUrl) — RFC 8620's own redirection mechanism; just use the URLs you are handed and re-read the Session there. Routing runs before authentication (credentials are cell-local), so this works even before a password check is possible; the stub Session carries state `"0"` and no WebSocket capability (that is the owning cell's configuration to advertise).

6.2 IMAP

Port 993, implicit TLS. The plaintext listener is loopback/dev only, and there is no IMAP STARTTLS — TLS is implicit; clients use 993. Log in with the address (aliases work), and the password, a session token, or a `mail`-scoped application password.

Capability line as served:

```
IMAP4rev1 IDLE UIDPLUS MOVE CONDSTORE QRESYNC ENABLE ID NAMESPACE
SPECIAL-USE CREATE-SPECIAL-USE SORT THREAD=ORDEREDSUBJECT ACL
```

```
RIGHTS=texk
AUTH=PLAIN UNSELECT ESEARCH LIST-EXTENDED LIST-STATUS STATUS=SIZE
```

Conditionally appended:

- `THREAD=REFERENCES` — only when the store is actually reading its conversation index; with threading off, a REFERENCES response would group nothing while claiming to.
- `AUTH=XOAUTH2 AUTH=OAUTHBEARER` — only when an identity provider is configured.
- `UTF8=ACCEPT` (RFC 6855) — only when the deployment enables it (off by default). When advertised, an APPEND carrying 8-bit headers from a session that has not ENABLEd it is refused per the RFC's MUST.

Sent/Drafts/Trash/Junk/Archive auto-provision with SPECIAL-USE attributes on first LIST. IMAP4rev2 is deliberately **not** advertised: this server implements most of its fold-ins but not the full rev2 session contract, and half-advertising a revision is worse than staying rev1.

Documented deviations and behaviours a client author should know:

- **`SEARCH BODY` and `SEARCH TEXT` match analyzed WORDS, not raw substrings.** Terms are tokenized, case-folded, and stemmed exactly as the index is, so `invoicing` finds "invoice" — and `nvoice` finds nothing. A multi-word needle requires the words **adjacent and in order** (a phrase), not merely co-present. RFC 3501 defines BODY as a substring match; this is the same deviation most index-backed servers make. Address and subject search keys remain substring matches.
- **Date SEARCH keys are supported** (`BEFORE/ON/SINCE` on INTERNALDATE; `SENTBEFORE/SENTON/SENTSINCE` on the Date header, read in the timezone it declares; calendar days, not instants). RFC 5032 `OLDER/YOUNGER` are **refused by name**.
- **Shared mailboxes appear under `Other Users/`** (`NAMESPACE` answers `(("Other Users/" "/"))` for them). Rights are RFC 4314; they are re-read per command, not cached at SELECT, and `EXAMINE` genuinely means read-only.
- APPEND honours its optional date-time argument (with the zone applied) and its flags, in one transaction.
- A command line that is not valid UTF-8 gets a tagged `BAD [CLIENTBUG]` with the session intact — never a dropped connection.

6.3 POP3

Port 995, implicit TLS. RFC 1939 command set plus `CAPA` (RFC 2449), `UIDL`, `TOP`, and `AUTH PLAIN` (RFC 5034) alongside `USER/PASS`.

The session is an **INBOX snapshot taken at login**: messages arriving mid-session appear next time. There is no maildrop lock; concurrent sessions are safe because deletion is idempotent. `DELE` takes effect at `QUIT` (a dropped connection deletes nothing), and `RETR` marks the message `\Seen` so IMAP and JMAP views stay coherent.

6.4 SMTP submission

Port 587 (STARTTLS, required before AUTH) and port 465 (implicit TLS). Both advertise the same extensions: `PIPELINING`, `SIZE`, `8BITMIME`, `ENHANCEDSTATUSCODES`, `SMTPUTF8`, `CHUNKING`, and `AUTH PLAIN LOGIN` — plus `AUTH XOAUTH2 OAUTHBEARER` when an identity provider is configured. On 465 the session is TLS from the first byte and STARTTLS is correctly absent.

Rules a submitting client will meet:

- **The authenticated account must own the sending address** (canonical or alias). A spoofed MAIL FROM is refused; over JMAP the same rule surfaces as `forbiddenFrom`.
- **Message size**: the `SIZE` value advertised in EHLO is the deployment's message ceiling (`ZEPHYR_MAX_MESSAGE_BYTES`, default 104,857,600 = 100 MiB). Note most external receivers cap near 25 MB — a 100 MiB message to a remote recipient will bounce at the far end.
- **Oversized-attachment auto-conversion**, when the deployment enables link attachments: attachments over the threshold (and messages over the total-size bound) are converted to download links at submission, largest parts first, until the message fits. **Signed or encrypted mail is never rewritten** — `multipart/signed`, `multipart/encrypted`, S/MIME and PGP messages pass through untouched below the hard cap, and above it the submission is **refused with a reason naming the signature** rather than corrupted. A refusal here fails the submission with a readable SMTP error while the sender's composer is still open — deliberately the opposite of the ingress rule, where accepted mail is never bounced by policy.
- **Per-account outbound caps** (messages/hour, recipients/day) answer readable 451/554 refusals; refused sends never consume allowance.
- Malware scanning (when on) refuses with 554 naming the signature, in both directions.

6.5 ManageSieve

RFC 5804. Implicit TLS on port 4190 in the reference deployment (the plaintext listener is loopback/dev). SASL `PLAIN`, plus `XOAUTH2`/`OAUTHBEARER` when an identity provider is configured.

The capability greeting identifies `"IMPLEMENTATION" "ZephyrAB"`, `"VERSION" "1.0"`, the SASL list, and the `SIEVE` extension string — the same list `require` enforces:

```
fileinto vacation envelope imap4flags date body relational subaddress
comparator-i;octet comparator-i;ascii-casemap comparator-i;ascii-
numeric
variables include
```

(`redirect` is a base RFC 5228 action, so `require "redirect"` is tolerated but not advertised.)

Behaviours: uploads are parse-gated (a script that does not parse is refused with the diagnosis); `SETACTIVE` atomically mirrors the chosen script into the delivery slot; `DELETESCRIPT` refuses the active script. Runtime errors at delivery resolve to the implicit keep — a broken filter files to INBOX rather than losing mail.

6.6 CalDAV / CardDAV

Served under `/dav/` on the mail origin: calendars at `/dav/calendars/{account}/{collection}/`, address books at `/dav/addressbooks/{account}/{collection}/`. Basic auth (password, session token, or a `dav`-scoped app password) — **bearer tokens are not accepted on this surface**, so a federated user reaches calendar and contacts with a `dav`-scoped application password. A default calendar (`personal`) and address book (`contacts`) auto-provision, because iOS Contacts never creates one.

Discovery (RFC 6764). From a bare address: SRV lookups (`_caldavs._tcp`, `_carddavs._tcp` — published by the operator), then `/.well-known/caldav` and `/.well-known/carddav`, which redirect into `/dav/` (trailing-slash variants included). The apex domain also redirects the well-known paths, so a client told only `user@example.com` finds the mail host.

Scheduling (RFC 6638) is server-side. The server advertises `calendar-auto-schedule`; each account has `schedule-inbox`/`schedule-outbox` collections (reserved names — a calendar cannot be created with them):

- A PUT of an event with ATTENDEEs sends the iMIP invitations (DKIM-signed, queued, retried like user mail).
- An inbound REQUEST is filed in the schedule-inbox **and** materialised into the default calendar as NEEDS-ACTION; a verified REPLY updates the organiser's stored PARTSTAT; a verified CANCEL withdraws. An unverified stranger's message is filed, never applied.
- An attendee RSVPs by PUTting the updated object (inbox item or calendar copy) or POSTing a REPLY to the outbox.
- **Free/busy** is a `VFREEBUSY` REQUEST POSTed to the schedule-outbox, answered per recipient with merged periods (`BUSY` vs `BUSY-TENTATIVE` kept distinct). Same-tenant recipients only; anyone else answers `5.3;No scheduling support for user`, identically for other tenants and unknown addresses.

Sync. `sync-collection` REPORTs with tokens; deletions are reported explicitly (the removed href with a 404 status) rather than silently omitted. ETags are content-derived and conditional writes (`If-Match`, `If-None-Match`) are atomic. `calendar-query`/`addressbook-query` filtering is implemented, including time ranges with VTIMEZONE resolution; an unsupported filter element is refused with the RFC's precondition rather than silently matching everything.

6.7 Autodiscovery

What a well-behaved client can find on its own, given only an address — provided the operator publishes the records:

Mechanism	Records / URLs
RFC 6186 / 8314 SRV	`_imaps._tcp`, `_pop3s._tcp`, `_submission._tcp`, `_submissions._tcp`, `_jmap._tcp` → host and port
RFC 6764 (DAV)	`_caldavs._tcp`, `_carddavs._tcp` SRV, plus the well-known redirects above
Thunderbird autoconfig	`https://autoconfig.example.com/mail/config-v1.1.xml`
JMAP	`/.well-known/jmap` on the mail host

`GET /account/clients` (section 5.8) returns the same facts as JSON for a signed-in user. `GET /domains/{domainId}/dns-records` (section 4.2) tells an administrator exactly which records to publish, with live verification state.

7. Webhooks

ZephyrAB will POST an event to a URL you register, so an external provisioning system does not have to poll for the things it cares about. Register endpoints with the four calls in section 4.18; this section is what arrives, how to check it is really from us, and what delivery does and does not promise.

A webhook is an optimisation over polling, never a replacement for it. `GET /jobs/{jobId}` and the ordinary read routes remain the authority, and they have to: a receiver that was down for an hour catches up by reading, and the only thing that makes that possible is the state still being readable. A delivery that exhausts its retries is abandoned — it never fails the job it was announcing, and nothing waits for it.

7.1 The event catalogue

Seven names, and the set is **closed**. Subscribing to `"*"` or to an unrecognised name is refused rather than ignored, so an endpoint can never be quietly signed up to a category of event added later.

Event	Fires when	`data` carries
`job.completed`	An async job reaches a terminal state	*(no envelope — see below)*
`account.created`	An account is provisioned, by any door: signup, admin API, bulk, LDAP auto-	`accountId`, `address`, `domain`, `domainId`, `tenantId`, `status`

	provision	
`account.deleted`	An account is unprovisioned	`accountId`, `address`, `domain`, `domainId`, `tenantId`
`account.status_changed`	An account's status is set	`accountId`, `address`, `tenantId`, `from`, `to`
`domain.created`	A domain is **registered**	`domainId`, `domain`, `tenantId`
`domain.deleted`	A domain is removed	`domainId`, `domain`, `tenantId`
`domain.verified`	**Never, on this build** — see below	—

Two of those rows carry more than they look like:

- **`domain.created` fires on registration only, not on re-verification.** The same storage write backs both, and announcing every "verify DNS" as a creation would tell a subscriber a domain was registered each time somebody pressed a button.
- **`account.status_changed` reports `from` equal to `to`** when a status was re-asserted rather than changed. The write path reads outside its own transaction and cannot honestly claim to have observed a transition, so it says what it did rather than suppressing the event.

`domain.verified` is subscribable and never arrives. A domain's verification status is derived live from public DNS on every read and is never stored — the control plane refuses to let an operator assert `verified` for exactly that reason — so there is no transition to fire on. It stays in the set, parseable and documented as dormant, so a stored subscription keeps working on the day the transition is stored. Registration tells you at the time (`notYetEmitted`, section 4.18); **do not read its absence as a delivery failure.**

7.2 The envelope

Every event except `job.completed` arrives in this shape:

```
{
  "id": "5f1c…",
  "event": "account.created",
  "occurredAt": "2026-08-24T09:15:02Z",
  "subject": { "kind": "tenant", "id": "0db2dd38-…" },
  "data": { "accountId": "…", "address": "alice@example.com" }
}
```

`id` is the **event's** id — one value across every endpoint that received it, so two subscribers can correlate. It is **not** the deduplication key; that is the `X-Webhook-Id` header (section 7.3).

`job.completed` is the odd one out and carries the bare `Job` object, with no envelope. The OpenAPI document has declared it that way since v1, an integrator may already have written against it, and it is honoured verbatim rather than tidied. The consequence for your code is a rule worth following anyway: **dispatch on the `X-Webhook-Event` header, never on the body's shape.**

7.3 Verifying a delivery

Three headers, plus `Content-Type: application/json`:

Header	Meaning
`X-Signature`	`t=<unix-seconds>,v1=<hex>`
`X-Webhook-Event`	The event name — what to dispatch on
`X-Webhook-Id`	The **delivery** id: stable across every retry, and the deduplication key

`v1` is `HMAC-SHA256(secret, "{t}.{raw body}")`. **The timestamp is inside the signed material**, which is what stops a captured delivery being replayable for ever: a receiver can refuse anything older than its own tolerance and cannot be talked out of it by an attacker editing `t`, because editing `t` breaks the signature. The scheme is deliberately the one Stripe and several others use, so a library you already have will verify it.

```
import hashlib, hmac, time

def verify(secret_hex: str, sig_header: str, body: bytes, tolerance:
int = 300) -> bool:
    parts = dict(p.split("=", 1) for p in sig_header.split(","))
    t, v1 = parts["t"], parts["v1"]
    if abs(time.time() - int(t)) > tolerance:          # replay window
        return False
    want = hmac.new(secret_hex.encode(), f"{t}.".encode() + body,
                    hashlib.sha256).hexdigest()
    return hmac.compare_digest(want, v1)               # constant time
```

Three things receivers get wrong:

- **Sign the RAW body**, before any parse-and-re-serialise. A round-tripped body is not the bytes that were signed.
- **The secret is the hex STRING as issued**, used as ASCII key material — not the 32 bytes it decodes to.
- **Enforce a freshness window.** The timestamp is only worth signing if somebody checks it.

The signature protects integrity and authenticity, not privacy — which is why the URL must be HTTPS.

7.4 What delivery promises

At-least-once, and unordered. Both halves need handling in your receiver:

- A receiver that answers `200` after its network drops the response **will be sent the same delivery again**. Deduplicate on `X-Webhook-Id`.
- Retries mean a later event can arrive before an earlier one. **Order by `occurredAt`**, never by arrival.
- One event to several endpoints produces several deliveries with different `X-Webhook-Ids` and the same envelope `id`.

Retry policy: exponential from 2 seconds, capped at 1 hour, **12 attempts** — roughly a day — then abandoned.

Answer	What happens
`2xx`	Success. The only success
`3xx`	**Not followed, and not a success.** A redirect is "ask somewhere else", which is precisely what must not be honoured. Counts as a failure and retries
`408`, `429`	Retried
Other `4xx`	**Abandoned immediately.** Retrying an unauthorised or malformed delivery twelve times changes nothing and looks like an attack from your side
`5xx`, timeout, connect or TLS failure	Retried until the attempt limit

So: answer `2xx` once you have durably accepted the event, and answer `4xx` only when you mean "never send this again".

Constraints on the URL, all enforced at registration **and again before every delivery**, because a hostname can be re-pointed at any time afterwards:

- **HTTPS only**, verified against the **system trust roots**. A private-CA endpoint will fail and there is no way to add a CA — terminate on a publicly-trusted certificate.
- **Must resolve entirely to public addresses.** Loopback, RFC 1918, link-local (`169.254.169.254` included), CGNAT and unique-local are refused. If *any* address in the DNS answer is non-public the delivery is refused: a mixed answer is the rebinding attack, not a coincidence.
- **Exactly one request, no redirects.** A `302` to a cloud metadata service does not work, which is the whole reason the rule exists.
- **No userinfo** (`https://user@host/`) — it means different things to different parsers, and a URL whose host depends on which library reads it cannot be checked.

Payloads are capped at 64 KiB; anything larger is dropped and counted rather than truncated, since a receiver cannot tell a truncated body from a malicious one.

7.5 Two timing behaviours worth designing around

A newly-registered endpoint can miss events for a moment. Events are staged inside the transaction that commits the thing being announced, and that transaction must not read — so the enqueue path works from an endpoint list refreshed on a timer (30 s), which registration republishes immediately. The window is small and one-directional: it can cost a brand-new endpoint an event, never deliver one to a removed subscriber.

A delivery may arrive twice after a worker dies. Deliveries are leased for 60 seconds; a worker that dies holding one leaves it to be re-claimed. This is the same at-least-once property as above and needs nothing done about it beyond deduplicating.

7.6 Known defect — a tenant-scoped endpoint and `domain.created`

If you want domain events and you can register at platform scope, do that.

Webhook subjects are matched **exactly**, so a `tenant:<uuid>` endpoint receives an event whose subject is that same uuid. Account events carry the tenant's uuid and match correctly. Domain events carry the domain record's stored tenant field — and on deployments where a domain was registered through the global `POST /domains` with a tenant **name**, that field held the name, so the event's `subject.id` was a name, no grant could name it, and a correctly-registered tenant endpoint silently received nothing. The payload's `tenantId` was inconsistent with `account.created` for the same reason.

`POST /domains` now resolves a name to the tenant id before storing it (section 4.2), so domains registered from here on carry an id and their events match. **A domain record written before that change still holds whatever was stored then**, so on an existing deployment the symptom can persist for those domains. Two practical rules: **correlate account and domain events on `domainId`, not on `tenantId`**, and prefer a platform-scoped endpoint for domain events — `wants()` answers true for platform whatever the subject says, so those are unaffected either way.

`docs/runbooks/webhooks.md` is the operator-side account of this and is the place to check whether a given deployment is affected.

8. Errors appendix

The refusals a client will actually meet, and what they mean. Admin problems are `application/problem+json`; match on `status` + `title`.

Status	Title / code	Where	Meaning and what to do
400	`invalid_scope`	Token endpoint	The `scope` parameter named an unknown scope or one the client does not hold. Only

			reachable after a valid credential
400	`unsupported_grant_type`	Token endpoint	Use `client_credentials` or `password`
400	`invalid request`	Everywhere	Malformed or refused field; the detail names it (e.g. `catchAllAddress`, a row `password` in bulk, `in_place`)
401	`invalid_client` / `invalid_grant`	Token endpoint	Bad credential — **or a rate-limited source; the two are byte-identical by design**, with no `Retry-After`. Do not retry in a loop
401	(bearer)	Admin routes	Missing/ malformed/ unknown/expired token, one indistinguishable answer, `WWW-Authenticate: Bearer`
401	`reauthentication Required`	Self-service, bearer only	The token is good; the identity provider authenticated the person longer ago than the operation allows. Send them back to the provider with `max_age` and retry with the token that comes back — retrying this one cannot work (section 2.8)
400	`unknown event`, `invalid subject`, `invalid webhook url`	`POST /webhooks`	The event name is outside the closed set (the response lists it), the subject

			is not one of the three spellings, or the URL is not HTTPS / carries userinfo / resolves to a private address
403	`subject not yours`	`POST /webhooks`	You may only register an endpoint for a subject you administer. Depends only on what you hold, never on whether the named subject exists
403	`insufficient scope`	Admin routes	Authenticated, wrong role; the response names the required scopes. Also the answer when a tenant-bound grant reaches a cross-tenant surface ("requires a platform-wide grant")
403	`license limit`	Create account / domain / cell	Over the tier's numbers, or lapsed. The detail names the limit and whose license
403	`license entitlement`	Delegations, plan writes, plan subscription, bulk	The feature is not in the tier. Deliberately a different title from `license limit`, so logs can tell "over a number" from "not included"
404	`not found`	Admin routes	Does not exist — **or exists and your grant's subject does not cover it.** Byte-identical on purpose; there is no way to tell, and that

			is the point
409	`conflict`	Various	Named condition: address taken, tenant still has domains/mailboxes, plan in use, DNS record collides with a server-managed record, live lease on a queue entry, DKIM key absent, domain not ready for required encryption
410	—	Download origin	Link expired, revoked, or exhausted — indistinguishably
422	—	JMAP	**Body validation runs before authentication**: a malformed unauthenticated `POST /jmap` answers 422, not 401. Test authN with a well-formed body, or the check passes for the wrong reason
429	—	Token endpoint	**Absent by design.** Throttled token requests answer 401 like a bad credential; a 429 would tell a guesser when its guesses stopped being evaluated. (Self-service auth rate limits answer 401 "too many attempts")
501	`no job store, reset flow not`	Various	The capability is not available on this

	`available`, `format not implemented`, `message-id lookup not implemented`, `no restore procedure`, `linksNotConfigured`, `webhooks are not enabled`, `no webhook key`		node/build; the detail says what to do instead. `no webhook key` is the narrower case: the node delivers webhooks but cannot register new endpoints
502 / 501	—	`/ops/alerts`	Alertmanager unreachable / unconfigured — refused rather than an empty list
503	`admin authentication not configured`	Everything	Fail-closed node: no admin credential configured
5xx	`internal error`	Anywhere	The request could not be processed; nothing about your input is implied

Self-service errors use `{ "error", "description" }` with camelCase codes; the notable ones are listed per route in section 5 (`invalidInvite`, `addressTaken`, `weakPassword`, `unreachableRecoveryAddress`, `totpRequired`, `badCode`, `reauthenticationRequired`, `optionOutOfBounds`, `scannerUnavailable`, `noMailer`, `notConfigured`).

9. Worked examples

End to end with curl. Substitute your admin origin, ids, and credentials; responses are abbreviated.

1. Get a platform token:

```
TOKEN=$(curl -s https://admin.example.com/admin/api/oauth/token \
  -d grant_type=client_credentials \
  -d client_id="CLIENT_ID" -d client_secret="CLIENT_SECRET" | jq
-r .access_token)
```

2. Create a tenant:

```
curl -s -X POST https://admin.example.com/admin/api/tenants \
  -H "Authorization: Bearer $TOKEN" \
  -H "Idempotency-Key: acme-create-1" \
```

```
  -H "Content-Type: application/json" \
  -d '{"name": "Acme Corp", "planId": "PLAN_ID"}'
# 201 → {"id": "TENANT_ID", "name": "Acme Corp", "status":
"active", ...}
```

3. Create a domain in that tenant (the tenant-scoped door, so it also works for delegated admins):

```
curl -s -X POST
https://admin.example.com/admin/api/tenants/TENANT_ID/domains \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type:
application/json" \
  -d '{"domain": "acme.example"}'
# 201 → {"id": "DOMAIN_ID", "name": "acme.example", "status":
"pending_verification", ...}
```

4. Read the DNS records the owner must publish:

```
curl -s https://admin.example.com/admin/api/domains/DOMAIN_ID/dns-
records \
  -H "Authorization: Bearer $TOKEN"
# 200 → [{"recordType":"MX","name":"acme.example","value":"10
mail.example.com",
"purpose":"mx","verified":false,"checked":true}, ...]
```

5. Create an account:

```
curl -s -X POST
https://admin.example.com/admin/api/tenants/TENANT_ID/accounts \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type:
application/json" \
  -d '{"domainId": "DOMAIN_ID", "localPart": "pat", "displayName":
"Pat Lee"}'
# 201 → {"id": "ACCOUNT_ID", "address": "pat@acme.example", ...}
```

6. Set its password — `ticketRef` is mandatory:

```
curl -s -X PUT
https://admin.example.com/admin/api/accounts/ACCOUNT_ID/password \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type:
application/json" \
  -d '{"password": "A-Long-Initial-Passw0rd", "ticketRef":
"ONBOARD-1234"}'
# 204
```

7. Delegate tenant administration to a person (their mail address is the principal):

```
curl -s -X POST
https://admin.example.com/admin/api/tenants/TENANT_ID/delegations \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type:
application/json" \
  -d '{"principalClientId": "pat@acme.example", "scope":
```

```
"tenant:admin"}'
# 201 → {"id": "DELEGATION_ID", "subjectKind": "tenant", "active":
true, ...}
```

8. The delegated admin signs in and lists their domains — password grant, then the tenant-scoped listing (the global `/domains` would be a 403 for them):

```
PTOKEN=$(curl -s https://admin.example.com/admin/api/oauth/token \
  -d grant_type=password \
  -d username="pat@acme.example" -d password="A-Long-Initial-Passw0rd"
| jq -r .access_token)

curl -s https://admin.example.com/admin/api/tenants/TENANT_ID/domains
\
  -H "Authorization: Bearer $PTOKEN"
# 200 → {"items": [{"name": "acme.example", ...}], "nextCursor": null}
```

(If Pat has TOTP enrolled, first mint a session token on the mail origin — `POST /account/session` with the code — and use that token as the `password` value above.)

9. Pause outbound delivery with a reason, then resume:

```
curl -s -X PUT
https://admin.example.com/admin/api/mailops/queue/status \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type:
application/json" \
  -d '{"paused": true, "reason": "upstream provider incident
INC-421"}'
# 200 → {"paused": true}   (fleet-wide within one queue tick)

curl -s https://admin.example.com/admin/api/mailops/queue/status \
  -H "Authorization: Bearer $TOKEN"
# 200 → {"paused": true, "by": "...", "reason": "upstream provider
incident INC-421", "since": "..."}

curl -s -X PUT
https://admin.example.com/admin/api/mailops/queue/status \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type:
application/json" \
  -d '{"paused": false}'
# 200 → {"paused": false}
```

If a script can pause production, make the resume a `trap EXIT`, not the last line.

10. Register a webhook endpoint and keep the secret — it is in this response and nowhere else:

```
curl -s -X POST https://admin.example.com/admin/api/webhooks \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type:
application/json" \
  -d '{"url": "https://hooks.example.com/zephyr",
```

```
       "subject": "tenant:TENANT_ID",
       "events": ["account.created", "account.deleted"]}'
# 201 → {"id": "WEBHOOK_ID", "url": "...", "subject":
{"kind":"tenant","id":"TENANT_ID"},
#        "events": ["account.created","account.deleted"], "enabled":
true,
#        "secret": "SIGNING_SECRET_HEX", "secretNote": "store this now
— ..."}
```

Check the response for `notYetEmitted` before you rely on anything you subscribed to, then watch the backlog and the subscriber's health:

```
curl -s
https://admin.example.com/admin/api/webhooks/WEBHOOK_ID/deliveries \
  -H "Authorization: Bearer $TOKEN"
# 200 → {"items": [...], "truncated": false, "nextCursor": null}
#        an empty list is the healthy answer — this is a backlog, not
a history
```

11. Check the license:

```
curl -s https://admin.example.com/admin/api/license -H "Authorization:
Bearer $TOKEN"
# 200 → {"state": "valid", "daysLeft": 365,
#        "usage": {"accounts": 14, "domains": 7, "cells": 3},
#        "limits": {...}, "entitlements": {"multi-cell": true, ...},
"enforced": true}
```

10. Glossary

Term	Meaning
Accept id	The identifier a message is given at SMTP acceptance; written into the acceptance ledger and echoed to the sender. One of the two handles `traceMessage` resolves
Account	A mailbox: canonical address, aliases, credential, quota, settings. Identified by uuid on the admin API; by its address everywhere a user signs in
Alias	An additional address delivering to an account's mailbox. A row in the same global address index as accounts and groups
At-rest encryption (P11)	Per-user encryption of stored mail to a key the user uploaded; the server cannot read it back. `required` mail with no usable key is refused, never stored in the clear
Cell	A data-plane unit (storage + frontends)

	hosting a set of domains. Clients are routed by DNS and by the JMAP Session; callers never address a cell directly
Delegation	A stored `(scope, subject)` grant to a principal (client id or account address). Immutable; read per request; revocation is immediate
Grant / subject	Scope says WHAT; subject (platform, tenant, or domain) says ON WHOSE DATA. Both halves must be covered
Group	A distribution list: one address fanning out to member mailboxes at RCPT, with a required owner who receives its bounces
Idempotency-Key	Header making a mutating POST safe to retry: the replay returns the original result
Invite	A single- or multi-use code that authorizes `/signup`; carries the domain, tenant, and quota. A bearer credential — stored hashed
Job	An async admin operation. Lease-based claims; safe to run twice; poll `GET /jobs/{jobId}`
Link attachment	A large file stored once and shared as a capability URL on the download origin, with expiry, optional password, and an optional download cap
Plan (CoS)	A versioned, immutable policy bundle tenants subscribe to: quota, features, rate limits, metered allowances, overage rates
Problem	An RFC 9457 `application/problem+json` error body
Queue entry	One message awaiting remote delivery, with retry schedule and lease. Terminal states (delivered/bounced/canceled) leave the queue
Scope	A role name from the OAuth taxonomy (section 2.4)
Session token	A credential minted by `POST /account/session` after password + second factor; valid 12 h; accepted anywhere a password is
Setting (protective / operational)	A declared key in the layered registry. Protective resolves as a union (protections

	can only be added); operational resolves most-specific-wins unless domain-locked
Tenant	The isolation unit. Every stored object carries a tenant id; no API path reads across tenants without a platform-wide grant
ticketRef	A support-ticket reference required on audit-sensitive operations (password set, restores)
Webhook endpoint	A registered HTTPS URL, a subject, and a subscription to some of the seven event names. Holds a signing secret returned exactly once at registration
Webhook delivery	One attempt to hand one event to one endpoint. Identified by `X-Webhook-Id`, which is stable across retries and is the deduplication key — distinct from the envelope's `id`, which identifies the event across endpoints