

pse-edge-mcp — Developer Walkthrough

Audience: a developer or architect who has never seen this codebase and needs to understand it, debug it, or extend it.

Companion documents. This file explains *how the system works*. Two others explain *why it is the way it is*, and you should read them before changing anything structural:

Document	Contents
docs/plan.md	Every design decision: scope, caching policy, auth design, delivery history, risks
docs/endpoints.md	The verified PSE Edge endpoint map — request dialects, parameter names, response shapes
docs/deploy.md	Running it in production
CLAUDE.md	The short form of the invariants, kept next to the code

Version described: **0.18.0**. 36 modules, ~8,900 lines of source, ~6,800 lines of tests.

1. What this is, in one paragraph

An **MCP server** (Model Context Protocol — a standard way to expose tools to an LLM client) that serves **Philippine Stock Exchange** data taken from the PSE Edge disclosure portal. It exposes **13 read-only tools** — quotes, price history, disclosures, financial reports, dividends, index and market data — plus **one action tool**, `send_email`, on deployments with auth enabled.

It is **unofficial**. PSE Edge publishes no API, so this server speaks to the same internal endpoints the portal's own web pages call.

The one decision that explains the whole design

PSE Edge is a public utility with no rate-limit contract and no commercial relationship with this project. Hammering it during trading hours would be antisocial and would get the project blocked. So:

Prices are end-of-day: a cached price is never refetched while the market is open. Everything else hits PSE Edge at most once per unique query per day.

(Narrowed in 0.13.0 — before that, the freeze gated every domain.) Almost every structural choice follows from that sentence — the cache is a *freeze* rather than a TTL, every response carries freshness metadata, there is an opportunistic archive, and there is a politeness throttle independent of user quotas. If a change you are contemplating would increase load on PSE Edge, it is probably the wrong change.

2. Running it

Three modes, same tool implementations underneath.

MODE	COMMAND	AUTH	STORAGE
studio (local)	uvx pse-edge-mcp (Claude Desktop / Claude Code default)	never	in-memory
HTTP (dev)	pse-edge-mcp --http --port 8000	opt-in	in-memory or Postgres
HTTP (production)	docker compose -f compose.nas.yaml up -d + compose.tunnel.yaml	on	Postgres

studio never authenticates. It runs on the user's own machine as a subprocess of their client; there is no network boundary to guard and no second party to authenticate. Adding auth there would be pure friction.

Local development:

```
uv sync --all-extras      # Python 3.14, uv for everything
uv run pytest              # 323 tests, no network access
uv run ruff check .        # line length 100
uv run mypy src            # strict
```

3. Request lifecycle

This is the path every tool call takes. Read it once and most of the codebase becomes predictable.

MCP client (Claude Desktop, claude.ai connector, your own agent)

POST /mcp {"jsonrpc":"2.0","method":"tools/call", ...}

HealthApp	/health, /health/ready – answered here, never authenticated, never touch the DB
AuthApp	/oauth/*, /signup, /login, /account, /privacy, / – reachable WITHOUT a token, because you cannot present one before you have one
AuthMiddleware	bearer token → user; quota check; 401 + WWW-Authenticate, or 429
MCP app (SDK)	JSON-RPC framing, tool dispatch, DNS-rebinding Host check

server.py validate args → call repository → shape reply
(no domain logic here – see §5)

repositories.py build cache key → FreezeService.get(...) → parse → model

service.py
FreezeService

THE FREEZE DECISION – see §4 cache hit? market open? fetch or refuse

cache hit

Storage
(memory | Postgres)

cache miss + closed

client.py —HTTP→ edge.pse.com.ph
parsers.py ← HTML / JSON

archive.py (opportunistic, best-effort)

{"data": ..., "meta": {as_of, valid_until, from_cache, stale, data_policy}}

Two things worth noticing:

- **The composition is built exactly once**, in `asgi.py`. The CLI (`__main__.py`) and production both call `create_app()`, so they cannot drift apart.
- **/mcp is the only authenticated path**. Everything the browser needs during signup is deliberately in front of the auth gate.

4. The freeze policy — the core behaviour

This is the part to understand before anything else, because it is what makes the server *correct* rather than merely functional.

`FreezeService.get()` in `service.py` takes a per-read `policy` (chosen by the repository for its domain) and implements one decision table per policy. For **EOD-frozen** — price data, and deliberately also the default:

	MARKET CLOSED	MARKET OPEN
cache fresh	serve from cache	serve from cache
cache stale (expired)	FETCH upstream, store, serve	serve the stale value, flagged <code>meta.stale = true</code> ← never fetches
cache empty	FETCH upstream, store, serve	FETCH ONCE, serve flagged stale + <code>meta.note ("not a realtime value")</code>

For **daily-refresh** — every other domain — the market-open column disappears: a miss or expiry fetches at **any** hour, once, and repeats of the same query are served from storage until the next 15:00 close. **immutable** entries are fetched once ever.

Market hours are **09:30–15:00 Asia/Manila on trading days**; weekends and the `PSE_HOLIDAYS` table in `market_calendar.py` are non-trading.

Consequences a newcomer will trip over

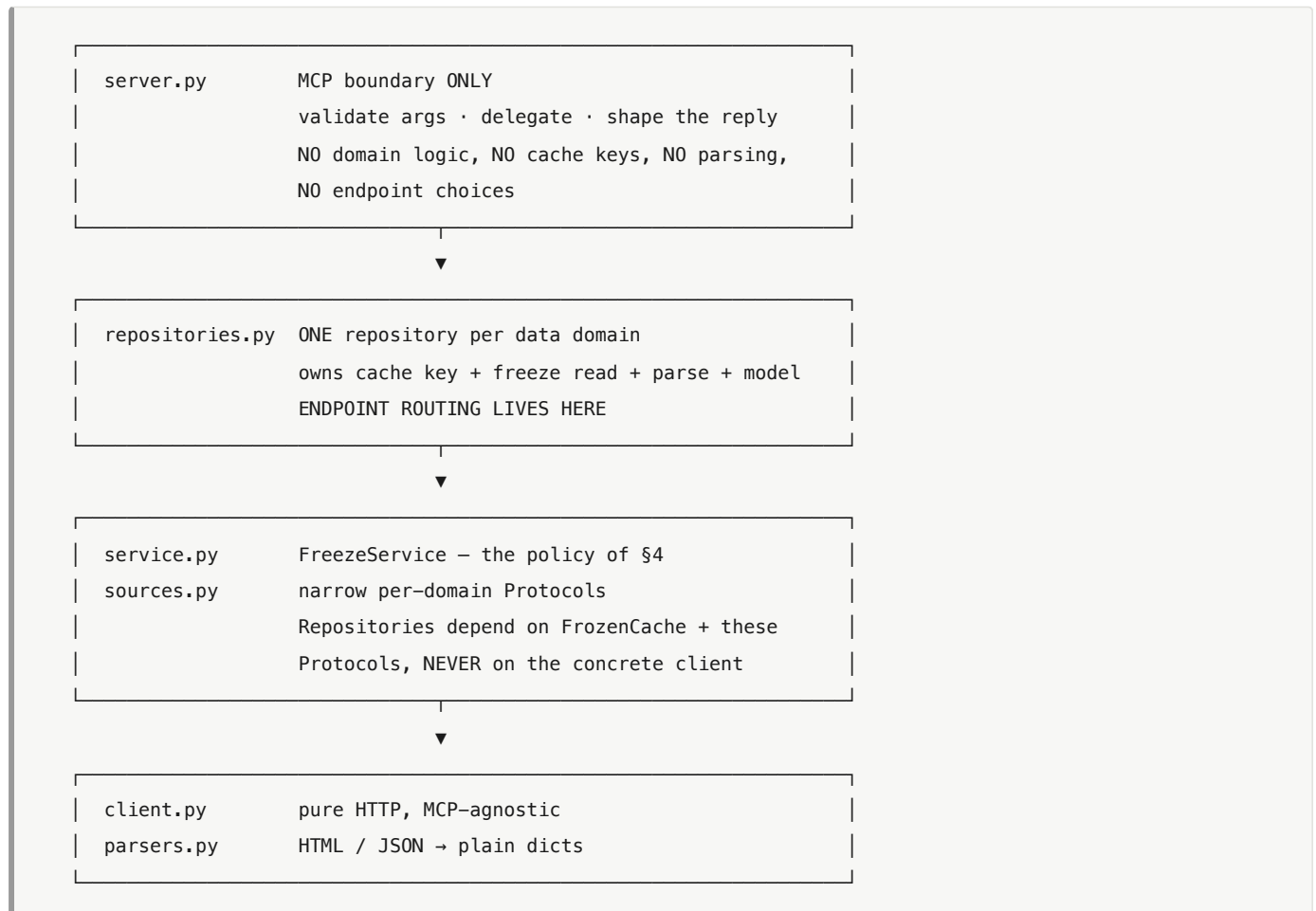
- **A session-time quote for a never-cached symbol serves `previous_close` only.** The one-time mid-session fetch must not present a delayed intraday number as a price, so `QuoteRepository.quote` trims the model to identity + `previous_close` (raw_fields emptied) and the `meta.note` says exactly that. The label derives from the entry's `fetch_at`, so every repeat that session carries it; the first ask after the close replaces the snapshot with the settled figures. (`MARKET_OPEN_NO_CACHE` is no longer raised — since 0.13.0 this fallback answers instead; the error class remains for clients that still handle the code.)
- **`meta.stale = true` means "not a settled end-of-day value"** — the market is open and you are seeing the last close, the value is a labelled mid-session snapshot (see `meta.note`), or PSE Edge was unreachable. It does not mean the data is wrong.
- **Concurrent cache misses collapse into one upstream request** (`SingleFlight` in `ratelimit.py`). Ten simultaneous first-time requests for the same symbol produce one HTTP call, not ten — and the flight body re-checks storage before fetching, so a miss that raced a concurrent store (same worker or a sibling via the shared Postgres cache) serves the stored entry instead of fetching again.
- **An upstream outage serves the last close, flagged `stale`.** If the fetch fails because PSE Edge is unreachable and an expired entry exists, it is served rather than discarded — holding real data and answering with an error instead is strictly worse. With nothing cached there is genuinely no answer, and `EDGE_UNAVAILABLE` is returned.
- **`policy="immutable"`** (passed by the disclosure-detail path) marks objects that never change upstream — a disclosure identified by `edge_no`. Those are cached forever with `valid_until: null` and `data_policy: "immutable"`; their first fetch may happen at any hour like any other non-price read.

The politeness throttle is separate from quotas

`throttle_rate_per_sec` (default 1.0/s, burst 2) limits **outbound** requests to PSE Edge and exists to protect *them*. Per-user quotas (60/min, 2000/day) limit **inbound** requests and exist to protect *you*. Do not conflate them; raising one does not justify raising the other.

5. Architecture and layering

The layering is enforced by convention and by tests. Violating it is the most likely way a well-meaning change gets rejected in review.



Supporting modules: `validation.py` (argument validators), `models.py` (Pydantic), `errors.py`, `cache.py` (Storage protocol), `archive.py`, `ratelimit.py`, `market_calendar.py`, `config.py`, `memo.py`.

Why repositories depend on Protocols

`sources.py` defines five narrow interfaces — `CompanySource`, `QuoteSource`, `DisclosureSource`, `CompanyInfoSource`, `MarketSource`. `PseEdgeClient` happens to satisfy all of them, but repositories only ever see the narrow one they need. The payoff is in `tests/test_repositories.py`: a repository can be tested with a five-line fake and **no HTTP mocking at all**.

Module map

Module	Lines	Responsibility
<code>auth_app.py</code>	1220	Browser- and client-facing routes: OAuth, signup, login, the tabbed account settings UI, privacy
<code>parsers.py</code>	752	HTML/JSON → dicts. The most PSE-specific code in the repo
<code>oauth.py</code>	748	OAuth 2.1 server: DCR, PKCE, code exchange, refresh rotation, <code>client_credentials</code>
<code>repositories.py</code>	526	The domain layer — five repositories
<code>admin.py</code>	485	<code>pse-edge-admin</code> CLI, including machine-client provisioning
<code>passkeys.py</code>	424	WebAuthn ceremonies and browser sessions
<code>server.py</code>	491	The 13 tool definitions
<code>models.py</code>	332	24 Pydantic models
<code>client.py</code>	268	HTTP to PSE Edge; both request dialects
<code>db.py</code>	255	SQLAlchemy tables, engine, schema check
<code>asgi.py</code>	240	The single composition point, and the DNS-rebinding allowlist
<code>auth.py</code>	190	<code>TokenService</code> , <code>QuotaTracker</code> . Must stay SQLAlchemy-free
<code>notifications.py</code>	143	<code>send_email</code> policy: self-only recipient, caps, escaping
<code>canary.py</code>	320	Nightly schema check: one endpoint per family, validated against its model

6. The tool surface

Thirteen tools are read-only: twelve return PSE Edge data in the same envelope (§7), and `get_server_version` (0.14.0) reports the server's own deployed release with **no meta** — meta is a data-freshness contract, and a version has no `as_of` . The fourteenth, `send_email` , is an **action** — see below. Every tool declares MCP `ToolAnnotations` (0.16.0): the read-onlys carry `readOnlyHint: true` so hosts can auto-approve them; `send_email` declares the opposite so no host ever auto-approves it.

Beyond tools, the server exposes one **MCP resource** (0.16.0): `pse-edge://attachment/{file_id}` serves a disclosure attachment's raw bytes via `resources/read` — most hosts cannot fetch arbitrary URLs, so `download_url` alone left files visible but unreadable. One PSE Edge fetch per file ever (immutable cache), 10 MB cap, `file_id` validated to digits. Each `get_disclosure` attachment advertises its `resource_uri` .

Two **prompts** (0.16.0) appear in host prompt-pickers — `market_recap` and `company_briefing(symbol)` — both baking in the meta-relay rules, and the briefing's `symbol` argument **autocomplete**s from PSE Edge's own autocomplete via `completion/complete` .

Tool	Arguments	Repository
<code>search_companies</code>	<code>query</code>	<code>CompanyRepository.search</code>
<code>validate_symbol</code>	<code>symbol</code>	<code>CompanyRepository.try_resolve</code>
<code>get_stock_quote</code>	<code>symbol</code>	<code>QuoteRepository.quote</code>
<code>get_price_history</code>	<code>symbol</code> , <code>start_date?</code> , <code>end_date?</code>	<code>QuoteRepository.history</code>
<code>search_disclosures</code>	<code>symbol?</code> , <code>start_date?</code> , <code>end_date?</code> , <code>template?</code> , <code>page</code>	<code>DisclosureRepository.search</code>
<code>search_disclosure_fulltext</code>	<code>keyword</code> , <code>start_date?</code> , <code>end_date?</code> , <code>symbol?</code> , <code>subject_title?</code> , <code>page</code>	<code>DisclosureRepository.fulltext</code>
<code>get_disclosure</code>	<code>edge_no</code> , <code>max_files?</code>	<code>DisclosureRepository.detail</code>
<code>get_company_profile</code>	<code>symbol</code>	<code>CompanyInfoRepository.profile</code>
<code>get_financial_highlights</code>	<code>symbol</code>	<code>CompanyInfoRepository.financials</code>
<code>get_dividends_and_rights</code>	<code>symbol</code>	<code>CompanyInfoRepository.dividends_and_rights</code>
<code>get_indices</code>	—	<code>MarketRepository.indices</code>
<code>get_market_summary</code>	—	<code>MarketRepository.summary</code>
<code>get_server_version</code>	—	none — reads the installed distribution's version in <code>server.py</code>
<code>send_email</code> (<i>auth only</i>)	<code>subject</code> , <code>body</code>	<code>NotificationService.send_to_self</code>

Dates are `YYYY-MM-DD` on the tool surface; the client converts to PSE Edge's `MM-dd-yyyy` wire format.

`search_disclosures` routes to two different upstream endpoints depending on whether `symbol` is supplied — market-wide with a date range, or one company's full history. That routing decision lives in the repository, which is exactly where the layering says it belongs.

`search_disclosure_fulltext` is deliberately a separate tool, not a better search. PSE Edge's keyword index only covers roughly 2023–2025, so it is not a superset of `search_disclosures`. The tool reports this coverage limit in its own results.

`send_email` — the one action tool

Policy lives in `notifications.py`, away from the MCP boundary, so every rule is testable without HTTP or a mail provider. Read that module's docstring before changing anything here.

The recipient is not a parameter. It is resolved from the ASGI scope the auth middleware populated (`server._caller`), i.e. from a validated bearer token. That single decision is what makes a mail tool safe to expose on a public server:

- **No open relay.** Signup is self-service, so a `to` argument would let anyone on the internet send mail from the operator's domain. The domain gets blocklisted, the provider suspends the account for abuse — and since the same provider sends verification email, *signup breaks with it*.
- **Nothing for prompt injection to steer.** This server returns disclosure text fetched from PSE Edge — third-party content the model reads and nobody here controls. "Email this to `attacker@example.com`" planted in a disclosure is an exfiltration path for any tool that accepts an address. Here there is no address argument to poison.

The rest is blast radius, since the direction is already fixed: registered **only when auth is enabled** (an auth-less deployment has no verified address, and a tool that is always listed and always fails just makes a model keep choosing it); body **escaped, never rendered as HTML**, because model-authored markup arriving from a trusted domain is what phishing looks like; 200-character subject, 20,000-character body, 20 messages per user per day.

It returns `{"data": ...}` with **no meta**, via `act()` rather than `reply().meta` answers "how fresh is this market data"; an action has no `as_of` and no `valid_until`, and inventing one would quietly make the freshness contract meaningless where it does matter.

7. The data contract

Every tool returns this envelope

```
{
  "data": { "...": "the payload" },
  "meta": {
    "as_of":      "2026-07-31T15:00:00+08:00",
    "valid_until": "2026-08-01T15:00:00+08:00",
    "from_cache": true,
    "stale":      false,
    "data_policy": "EOD-frozen",
    "note":       null
  }
}
```

`meta` is not decoration. It is the mechanism by which the server is honest about a freeze policy that would otherwise look like stale data. A client that ignores `meta` will eventually mislead someone about when a price was true.

- `data_policy` is "EOD-frozen" for price data, "daily-refresh" for every other domain, or "immutable" (with `valid_until: null`) for objects that never change upstream.
- `note`, when set, is a human-readable freshness caveat — e.g. a mid-session quote serving only `previous_close`.

Relay it to the user.

- **Clients should cache against `valid_until`**. Re-querying before it passes returns a byte-identical answer and only burns quota.

Error codes

Errors are returned as data — `{"error": "CODE", "message": ..., ...}` — not as exceptions, so an LLM can react to them rather than seeing a traceback.

Code	Meaning	Caller should
MARKET_OPEN_NO_CACHE	No longer raised since 0.13.0 — kept for old clients; the labelled previous_close fallback answers instead	Handle like any stale reply
SYMBOL_NOT_FOUND	No such ticker	Try search_companies
INVALID_ARGUMENT	Failed validation	Fix the arguments
ENDPOINT_CHANGED	PSE Edge's response shape changed	Alert a human — see §12
EDGE_UNAVAILABLE	Upstream unreachable and nothing cached	Retry later
RATE_LIMITED	An action's own budget is spent (e.g. daily email cap)	Honour retry_after_seconds
ACTION_UNAVAILABLE	The action cannot run here (no authenticated caller, or no mail provider)	Stop asking — do not retry
INTERNAL_ERROR	Anything else	File a bug

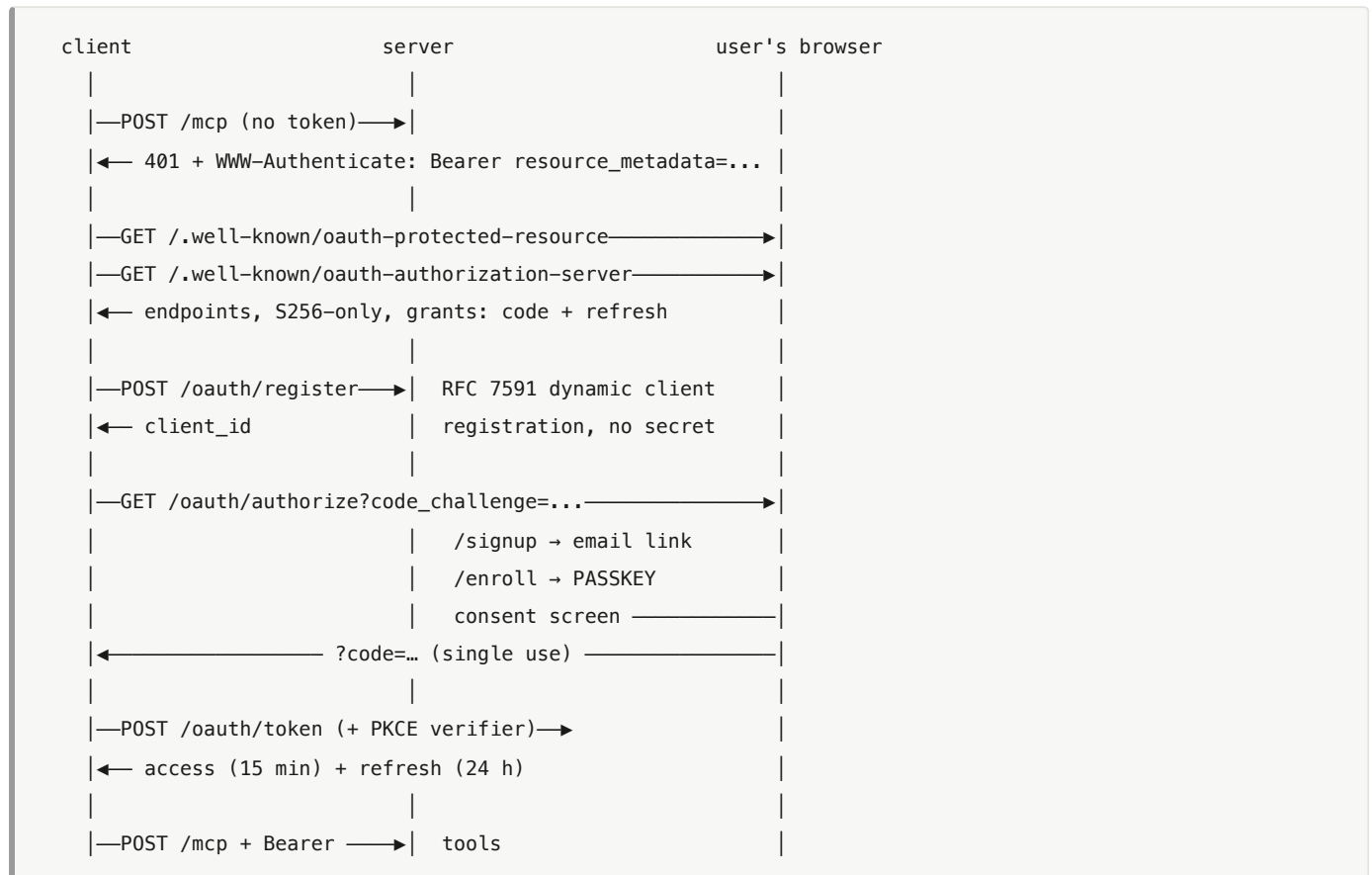
The mapping happens once, in `server.reply()` . Do **not** add per-tool `try/except` .

`reply()` is a helper that takes a callable, and deliberately not a decorator. The MCP SDK builds each tool's output schema from its return annotation, and a `functools.wraps` wrapper makes `inspect.signature` follow `__wrapped__` to the *inner* annotation, breaking schema generation. Tools must stay annotated `-> dict[str, Any]` . `tests/test_server_disclosures.py::test_tool_surface_is_stable` guards this.

8. Authentication

Auth is **opt-in** (`PSE_AUTH_REQUIRED=1` , which requires `DATABASE_URL`). `stdio` never authenticates.

The self-service flow



Exactly two steps involve a human: the passkey enrollment and the consent click. No passwords exist anywhere in the system.

Rules that must not regress

- Unknown client or unregistered redirect URI is **fatal and never redirects** — otherwise the server is an open redirector.
- Redirect URIs match **exactly**. No prefix matching.
- PKCE is mandatory, S256 only.**
- Authorization codes are single-use via one atomic `UPDATE ... WHERE consumed_at IS NULL RETURNING`.
- Refresh tokens rotate; **replaying a rotated one revokes the entire family** (RFC 9700 §4.14). A stolen refresh token gets one use before the theft is detected.
- Only `kind='access'` authenticates as a bearer.
- Tokens are opaque, `pse_`-prefixed, stored **only as SHA-256**.
- The validation-cache TTL (default 60 s) is **the revocation-latency budget and nothing else**. Refusals are never cached.
- `auth.py` must stay **SQLAlchemy-free** so the lean install path works; Postgres code lives in `auth_store.py` and imports lazily.

Headless and agentic access — `client_credentials`

A LangGraph app or the Anthropic Messages API MCP connector cannot open a browser, so it exchanges a client id and secret for a bearer token directly:

```

agent —POST /oauth/token————→ grant_type=client_credentials
|
|                                     HTTP Basic, or client_id/client_secret in the body
← access_token (1 h), token_type=Bearer, scope=mcp
|
|                                     ↑ NO refresh token
—POST /mcp + Bearer —————→ tools

```

Provisioned by an **operator**, never by a registrant. Two routes, one authority:

```

pse-edge-admin create-machine-client --name langgraph-app    # secret shown ONCE
pse-edge-admin revoke-machine-client <client_id>

```

Since **0.11.0** the same two operations also appear on the `/account` page, for accounts whose email is listed in `PSE_ADMIN_EMAILS`. That route exists because the deployment this server was built for is a NAS, where there is no shell to run the CLI in. It moved the authority from *has a shell* to *is a named operator* — it did not widen it. A non-operator account sees no panel, and both routes answer `404` rather than `403`, so there is no signal the surface exists.

The security property to understand before touching any of this: `/oauth/register` is open to the internet, so the right to use `client_credentials` must not be derivable from anything a registrant supplies — not a `grant_types` array it declares, not a requested auth method, not the presence of a secret. It is gated on `oauth_clients.client_type == 'machine'`, a column **only operator-authorized paths write** — the CLI and the `PSE_ADMIN_EMAILS`-gated web route, never DCR. A client that registers, declares the grant and sends a secret still gets `unauthorized_client`. If that check ever becomes conditional on request data, or if `admin_emails` is ever populated from something a user can set about themselves, anyone on the internet can mint tokens.

A worked client for the app-on-top-of-this case — one machine client, its token lifecycle, and the agent instructions to go with it — is in [examples/langgraph_client.py](#).

Two more decisions worth knowing before changing them:

- **A machine client is backed by a service user** (under the reserved, non-routable `machine.invalid` domain). That is what keeps the bearer path identical for both grants: `/mcp` validates by joining `auth_tokens` to `users`, so a token with no user behind it would need a special case in the middleware — and a second path through authentication is where a revocation check goes missing later. It also gives machine clients quotas, usage accounting and disablement for free.
- **No refresh token for this grant.** The client already holds a long-lived secret it can present again; a refresh token would be a second credential of equal power without the rotation benefit that justifies one.

`POST /oauth/token` is rate-limited (`FixedWindowLimiter` in `ratelimit.py`, 20/minute per `client_id` **and** per IP), because it is the one endpoint where a long-lived credential can be attacked online. Both keys are always counted, so tripping one cannot keep the other cold.

The older route still exists for clients that speak neither OAuth nor this grant, and is the only option on a plain-HTTP deployment since browsers restrict WebAuthn to secure contexts:

```

pse-edge-admin create-user agent@example.com
pse-edge-admin issue-token agent@example.com --note nightly-job    # shown once

```

Either way, give each agent **its own client or user**: quotas are per user, so a runaway job throttles itself, and revocation is surgical.

Privacy surface

`/account` shows a signed-in user everything held about them — since 0.12.0 as a tabbed settings page that also lets them sign any connected client out (`POST /account/sessions/revoke`, scoped to the caller's own token families);

`POST /account/delete` erases it all immediately, in one transaction, with no approval step. Usage is aggregated **per user-hour, never per request** — minimal collection *and* no write on the hot path — and purged after 90 days.

`tests/test_privacy.py::test_eraser_leaves_nothing_behind` walks `metadata.tables`, so a new user-keyed table **fails that test** rather than silently retaining personal data. If you add a table with a user foreign key, that test is your reminder to wire it into erasure.

9. Credentials and the token lifecycle

§8 covers *which* flow a caller uses. This section covers what is actually generated, what is stored, and how a credential lives and dies. Two things surprise most readers: the **passkey private key is never generated here** (§9.3), and there are **three different ways to end up holding a bearer token** (§9.5) that all converge on one row shape.

9.1 Every credential in one table

Everything random comes from `secrets` — the CSPRNG — never `random`.

Credential	Generated by	Shape	At rest	Lifetime
Bearer token (access)	"pse_" + <code>secrets.token_urlsafe(32)</code>	pse_ + 43 chars	SHA-256 only	15 min (<code>PSE_ACCESS_TTL_MIN</code>)
Bearer token (refresh)	same generator	pse_ + 43 chars	SHA-256 only	24 h (<code>PSE_REFRESH_TTL_HOURS</code>), single-use — every rotation mints a new one and revokes the old pair; <code>PSE_REFRESH_UNUSED_TTL_HOURS</code> (24) caps a family that never rotated
Machine access token	same generator	pse_ + 43 chars	SHA-256 only	1 h , no refresh
CLI-issued token	same generator	pse_ + 43 chars	SHA-256 only	30 days
DCR <code>client_id</code>	<code>secrets.token_urlsafe(16)</code>	22 chars	plaintext — a public identifier	until revoked
DCR <code>client_secret</code>	—	none is issued	—	—
Machine <code>client_id</code>	"mcp-" + <code>secrets.token_urlsafe(12)</code>	mcp- + 16 chars	plaintext	until revoked
Machine <code>client_secret</code>	<code>secrets.token_urlsafe(48)</code>	64 chars	SHA-256 only	until revoked
Authorization code	<code>secrets.token_urlsafe(32)</code>	43 chars	SHA-256 only	300 s , single-use
Web session id	<code>secrets.token_urlsafe(32)</code>	43 chars	SHA-256 only	20 min
OAuth flow id	<code>secrets.token_urlsafe(16)</code>	22 chars	plaintext — see below	15 min
Email verification token	<code>secrets.token_urlsafe(32)</code>	43 chars	SHA-256 only	30 min
<code>user_id</code> , <code>family_id</code>	<code>uuid.uuid4().hex</code>	32 hex chars	plaintext — identifiers, not secrets	—
Passkey private key	the authenticator, not this server	—	never leaves the device	until the user deletes it

The `pse_` prefix is not structural — it exists so a leaked token is **greppable by secret scanners**. The `mcp-` prefix on a machine `client_id` is likewise cosmetic: it is *not* what authorizes `client_credentials`. That is the `client_type` column, and nothing in a request can influence it (§8).

9.2 What is stored, and what is not

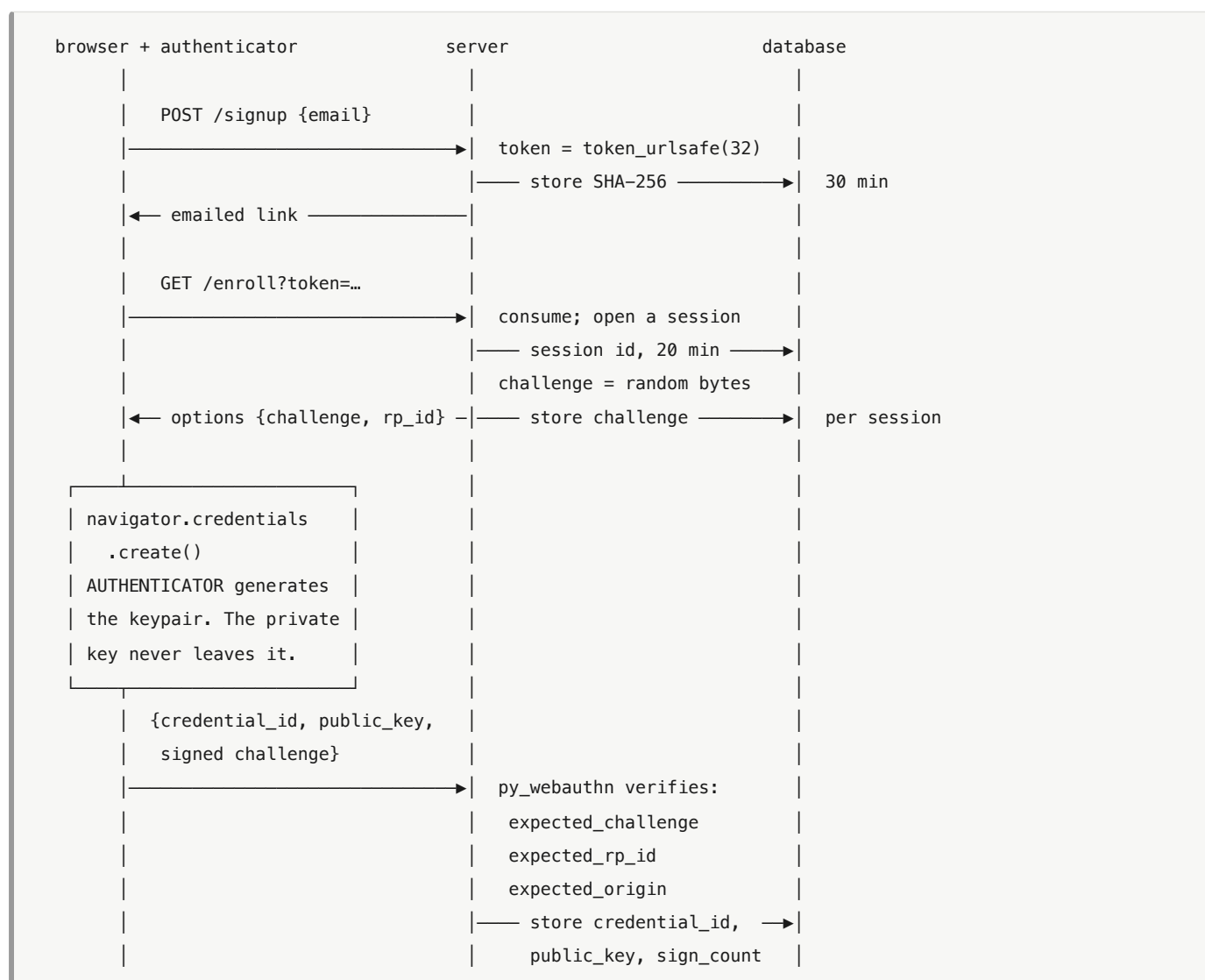
One rule covers everything above: **anything that authenticates is stored as SHA-256 and is never recoverable**. The plaintext exists exactly once, in the response that issues it — which is why the CLI prints *"Store the secret now — it is not recoverable, only revocable"* and the created-machine-client page is `cache-control: no-store`. The session cookie is covered by that rule too — `web_sessions` stores `sid_hash`, not the `sid`.

The one deliberate exception is the **OAuth flow id**, stored in the clear because it is an identifier rather than a credential: holding it authenticates nothing. The secrets in that flow are the authorization code (hashed, single-use, 300 s) and the PKCE verifier, which the server never sees at all — it only ever stores the S256 challenge and recomputes.

A fast hash rather than bcrypt/argon2 is deliberate, and recorded in `plan.md` §6: these are **32–48 bytes of CSPRNG output, not passwords**. There is no dictionary to attack and no low-entropy guess to slow down, so a KDF would buy nothing while adding latency to every single authenticated request. That reasoning holds *only* because the values are full-entropy — if a human-chosen secret is ever accepted anywhere, it does not transfer.

9.3 The passkey — the one credential this server never generates

This is the part most people get backwards. The server generates a **challenge**; the *authenticator* (Touch ID, a phone, a security key) generates the keypair, keeps the private half forever, and hands back only the public half.



Four consequences worth internalising:

- **The database holds no passkey secret.** A full dump of `webauthn_credentials` lets an attacker *verify* signatures, never *produce* them.

- **The challenge is stored server-side per session and cleared on use**, so a replayed assertion has nothing to match.
- **rp_id is the hostname of PSE_PUBLIC_URL**, and WebAuthn scopes credentials to it. Change that value and every enrolled passkey stops working — they cannot be migrated, only re-enrolled. This is why a wrong PSE_PUBLIC_URL presents as "passkeys are broken" rather than as a configuration error (§16).
- **sign_count must advance**. py_webauthn rejects a counter that fails to increase — the standard cloned-authenticator signal.

9.4 client_id — two kinds, and only one of them can be a machine



A DCR client is **public and holds no secret at all** — PKCE binds the code instead, which is what makes registration safe to leave open. The client_type value is written explicitly at the point untrusted input creates a client, rather than left to a schema default three files away, precisely because it is the value that denies client_credentials .

9.5 Three ways to mint a bearer token

Path	Who uses it	Returns	Refresh?
authorization_code + PKCE	a human in a browser, via a DCR client	access + refresh	yes
refresh_token	that same client, silently	a new access + refresh pair	rotates
client_credentials	a headless agent, via a machine client	access only	no
pse-edge-admin issue-token	a client that speaks no OAuth	access (30 days)	n/a

All four write **the same auth_tokens row shape** — same kind='access' , same hashing. That sameness is load-bearing, and _mint_machine 's docstring says why: /mcp validation is one indexed lookup joining auth_tokens to users , and a token of any other shape would need a special case there — "exactly the kind of second code path that later turns out to skip a revocation check."

A machine token gets **no refresh token** on purpose: the client already holds a long-lived secret it can present again, so a refresh token would be a second credential of equal power without the rotation benefit that justifies one.

9.6 The lifecycle, end to end



Refusals are never cached. Only successful validations are, and the cached entry can never outlive the token itself.

9.7 The 60-second cache is a revocation budget, not a performance knob

`PSE_TOKEN_CACHE_TTL` (default 60 s) means exactly one thing: **a revoked or disabled token keeps working for at most that long.** Nothing else in the system depends on the number.

`plan.md` §6 records the arithmetic that set it, and it is worth knowing before anyone "tunes" it: $\text{auth lookups/s} \approx \min(\text{request rate}, \text{active tokens} \div \text{cache TTL})$. On an EOD service where few users make more than one request every few seconds, a 5-second cache saves almost nothing, while 60 seconds cuts auth reads roughly 6× and caps revocation lag at one minute. Lower it if you want faster revocation; do not lower it expecting a speedup.

9.8 Rotation, reuse, and revocation

Every refresh mints a **new** access+refresh pair and revokes the pair it replaces — the presented refresh token *and* the access token minted alongside it, which nothing legitimate still holds. All tokens descended from one authorization share a `family_id`:

```
auth code → [access1 refresh1] family=F
           ↳ refresh → [access2 refresh2] family=F (access1 refresh1 revoked)
                       ↳ refresh → [access3 refresh3] family=F

refresh1 presented again → it was already revoked, so it leaked
                        → REVOKE THE ENTIRE FAMILY F (RFC 9700 §4.14)
                        → invalid_grant + a WARNING log line
```

Reuse of an already-rotated refresh token is treated as **theft, not a mistake** — the legitimate holder gets logged out too, which is the intended trade.

Revocation marks rows; deletion is separate. **Every mint opportunistically purges rows past their expiry**, so `auth_tokens` holds only rows that still matter: live credentials, plus revoked ones kept for reuse detection until they

would have expired anyway.

What revokes what:

Action	Effect
revoke-token <plaintext>	that one token
Revoke on /account (Sessions & tokens)	that token family — self-service sign-out of one client
disable-user	the account and all its tokens
revoke-machine-client	the client, its outstanding tokens, and its service account
refresh reuse detected	every token in that family_id
delete-user	hard-deletes everything, in one transaction

All of these land within PSE_TOKEN_CACHE_TTL at the /mcp boundary — but **immediately** at /oauth/token , which reads the client row directly. A revoked machine client cannot mint even one more token, while a token it minted a moment earlier may survive up to 60 seconds.

10. Storage and the archive

Postgres is **optional**. DATABASE_URL unset gives an in-memory cache and a NullArchive — the zero-config stdio path, which does not even need the postgres extra installed.

DATABASE_URL unset	DATABASE_URL set
InMemoryStorage	PostgresStorage → shared cache, so
NullArchive	N replicas still cause exactly one
(per-process, ephemeral)	upstream fetch per boundary
	PostgresArchive → EOD bars and
	disclosures accumulate over time

Tables: cache_entries , eod_bars , disclosures , users , auth_tokens , webauthn_credentials , web_sessions , oauth_clients , oauth_flows , email_verifications , usage_events .

Two columns carry more weight than their size suggests. oauth_clients.client_type ('dcr' | 'machine') is the entire authorization boundary for client_credentials — see §8. oauth_clients.service_user_id links a machine client to the account its tokens authenticate as, deliberately without a foreign key, matching the oauth_flows.user_id precedent so erasure ordering stays simple.

The archive is opportunistic. Nothing crawls. It fills only from fetches a user already caused, so it deepens over time at zero additional cost to PSE Edge — which matters because PSE Edge itself serves only limited history.

Rules:

- **Schema is applied by Alembic only**, never create_all at runtime. Compose runs a one-shot migrate service and check_schema() fails loudly at startup if migrations were skipped.
- **A failed archive write must never fail a read.** It is caught broadly — a dead database raises OSError , not merely SQLAlchemyError .
- Postgres imports are **lazy**, inside build_storage() . A test asserts SQLAlchemy does not leak into sys.modules on import pse_edge_mcp.server .

11. Extending the server

Adding a tool to an existing domain

1. Add a method to the relevant repository in `repositories.py`. It owns the cache key, the `FreezeService.get()` call, the parse, and the model.
2. Add a thin tool in `server.py`: validate arguments, call the repository, `return await reply(run)`. Keep the `-> dict[str, Any]` annotation.
3. Add a Pydantic model in `models.py` if the shape is new.
4. Record a fixture in `tests/fixtures/` and test the repository with a fake source.

Adding a whole data domain

A new domain is a **new repository plus thin tools** — never fetch/parse orchestration in `server.py`. Concretely:

1. Add the narrow Protocol to `sources.py`.
2. Add the fetch method to `client.py`, using the correct dialect (§12).
3. Add the parser to `parsers.py`. Parse by **<thead> label, never column position**.
4. Add the repository, the models, and the tools.
5. Capture a real fixture; tests never touch the network.

Checklist before opening a PR

- `uv run ruff check .` · `uv run mypy src` · `uv run pytest` all clean
- New endpoint → new fixture, and `docs/endpoints.md` updated
- Shape drift raises `EndpointChangedError` rather than returning partial data
- All reads go through `FreezeService.get()` — **never call `PseEdgeClient` from a tool**
- Bump `version` in `pyproject.toml` and roll `CHANGELOG.md`'s Unreleased section **in the same PR** as the work being released

Branching

All work lands on `develop`. `develop` reaches `main` only by pull request; `main` is protected. **Merging to `main` publishes a multi-arch image**, and a merge that changes `version` also cuts a GitHub Release and an immutable `<version>` tag. Never commit to `main` directly.

12. Debugging guide

Symptom → cause

Symptom	Almost certainly
<code>POST /mcp</code> → 421 Invalid Host header	The SDK's DNS-rebinding guard. <code>PSE_PUBLIC_URL</code> must match the host clients actually use — <code>asgi.transport_security_for()</code> derives the allowlist from it
OAuth completes, then the first tool call fails	Same as above. Everything about auth is fine; look at the transport layer
A quote with only <code>previous_close</code> populated, <code>stale: true</code> + <code>meta.note</code>	Working as designed: session-time ask for a never-cached symbol. Settled figures arrive after the close
<code>ENDPOINT_CHANGED</code>	PSE Edge changed shape. Re-capture the fixture, compare against <code>docs/endpoints.md</code>, fix the parser. Never paper over it. The nightly canary should have mailed you first — check why it did not
Canary mail: "PSE Edge canary FAILED"	Exactly the above, caught before a user hit it. Same fix
HTTP 415 from an <code>.ax</code> endpoint	Wrong dialect — that endpoint needs a JSON body, not form encoding
Disclosure results in the wrong order	<code>sortType</code> must be the literal string <code>"date"</code>
Signup → 503	The mail provider refused. The log names the sender address; ZeptoMail verifies exact domains, so a verified <code>example.com</code> does not cover <code>sub.example.com</code>
Passkeys "just don't work"	<code>PSE_PUBLIC_URL</code> does not match the browser's origin. Enrolled credentials cannot be migrated, only re-enrolled
App container stopped, empty log	A start-time failure, not a crash — a crash always leaves logs. Usually a taken host port, or a bind-mount whose source does not exist
NAS UI shows the project as "Error"	<code>migrate</code> is a one-shot that exits 0. That is success. Judge health by <code>curl /health</code>
<code>ImportError</code> on sqlalchemy in a lean install	Something imported Postgres code eagerly. It must be lazy, inside <code>build_storage()</code>
<code>POST /oauth/token</code> → unauthorized_client	The client is not a machine client. Only <code>pse-edge-admin create-machine-client</code> can authorize this grant — a self-registered client never can, by design
<code>POST /oauth/token</code> → invalid_client (401)	Wrong secret, unknown client, or a revoked one. All answered identically on purpose, so the endpoint is not an oracle for which client ids exist
<code>POST /oauth/token</code> → slow_down (429)	Rate limit: 20/minute per <code>client_id</code> and per IP. Honour <code>Retry-After</code>
A machine client's token stops working early	<code>revoke-machine-client</code> revokes outstanding tokens too, and revocation lands within <code>PSE_TOKEN_CACHE_TTL</code> (60 s)

Useful commands

```
curl -s https://<host>/health # liveness
curl -s https://<host>/health/ready # readiness (checks DB)
curl -s https://<host>/well-known/oauth-protected-resource # is PSE_PUBLIC_URL right?
curl -sS -o /dev/null -w '%{http_code}\n' -X POST https://<host>/mcp # expect 401

docker compose -f compose.nas.yaml logs -f app
docker compose -f compose.nas.yaml exec app pse-edge-admin list-users
docker compose -f compose.nas.yaml exec app pse-edge-admin list-machine-clients
```

Reading the log

Every line is timestamped (ISO-8601 with an offset) in both formats; `PSE_LOG_JSON=1` gives one JSON object per line for a log shipper. Secrets are redacted in both formats as a backstop — nothing here logs a credential deliberately.

The lines worth knowing by sight:

Line	Means
starting pse-edge-mcp <version> ...	the config the process actually resolved to. First thing to read in an incident
upstream: fetching from PSE Edge key=... policy=...	a real request left for PSE Edge. <code>policy=E0D-frozen</code> during market hours is legitimate only once per never-cached key — repeats of one price key within a session mean the freeze is broken
upstream: fetched and cached key=... duration_ms=...	it came back, and how slow it was
freeze: uncached price read during the session – fetching once, serving as non-realtime	the 0.13.0 fallback working, not a fault
auth: rejected a bearer token / quota: refused	refusals only — successes are the access log
oauth: issued a client_credentials access token	a machine client minted a token
oauth: REFRESH TOKEN REUSE detected (WARNING)	a rotated token was presented twice, so it leaked
oauth: DENIED client_credentials to a non-machine client (WARNING)	a misconfigured integration, or someone probing the gate

upstream: fetching during market hours means the freeze invariant is broken. Those lines should be a trickle after each 15:00 Manila close and absent between 09:30 and 15:00 on a trading day. That is the single most useful thing to grep this log for.

Two PSE Edge dialects

Getting this wrong produces an HTTP 415 that looks like a server fault:

```

JSON dialect      POST /common/DisclosureCht.ax
                  Content-Type: application/json
                  → JSON body. Form encoding gives 415.

Form dialect      POST /announcements/search.ax
                  POST /companyDisclosures/search.ax
                  POST /keyword/search.ax
                  Content-Type: application/x-www-form-urlencoded
                  → an HTML FRAGMENT, not JSON.

Wire dates are MM-dd-yyyy. PSE Edge's parameter names sometimes
differ from its own HTML form field names – trust docs/endpoints.md
over the page source.

```

13. Testing

Tests never touch PSE Edge. All HTTP is mocked with `respx` against 15 fixtures in `tests/fixtures/`, recorded from real captures. A new endpoint requires a new fixture.

Layer	How it is tested
Repositories	Five-line fake sources — no HTTP mocking (<code>test_repositories.py</code>)
Client / parsers	<code>respx</code> + recorded fixtures
Freeze policy	Injected calendar pinning the clock (<code>test_service.py</code>)
Postgres	testcontainers, marked <code>@pytest.mark.postgres</code>
Auth	<code>test_auth_journey.py</code> — soft-webauthn, real signatures , real Postgres
Deployment	<code>test_deploy.py</code> — health probes, logging, the real ASGI factory

A lesson worth internalizing

250 passing tests once missed a bug that made the server unusable in production: the transport-security setting. The journey test passed `transport_security=...` **explicitly**, so it exercised a configuration production never built — the fixture worked around the exact defect it should have caught.

A fixture that configures something production leaves at its default is a blind spot, not a test. Where practical, test through `create_app()` — the real factory — rather than hand-assembling the stack.

14. Configuration reference

All settings live in `config.py` as a frozen dataclass; environment variables override.

Variable	Default	Notes
PSE_EDGE_BASE_URL	https://edge.pse.com.ph	
PSE_MARKET_OPEN / PSE_MARKET_CLOSE	09:30 / 15:00	Asia/Manila; drives the freeze
PSE_THROTTLE_RPS / PSE_THROTTLE_BURST	1.0 / 2	Outbound politeness
PSE_TIMEOUT_SEC / PSE_RETRY_ATTEMPTS	20.0 / 3	
DATABASE_URL	unset	Unset ⇒ in-memory + no archive
PSE_DB_POOL_SIZE / PSE_DB_MAX_OVERFLOW	5 / 10	
PSE_AUTH_REQUIRED	false	Requires DATABASE_URL
PSE_TOKEN_CACHE_TTL	60	The revocation-latency budget
PSE_QUOTA_PER_MIN / PSE_QUOTA_PER_DAY	60 / 2000	Per user, in-process
PSE_PUBLIC_URL	http://localhost:8000	The highest-risk setting — see below
PSE_ACCESS_TTL_MIN / PSE_REFRESH_TTL_HOURS	15 / 24	A client idle longer than the refresh TTL re-runs the browser flow
PSE_REFRESH_UNUSED_TTL_HOURS	24	Refresh lifetime until a family first rotates; clamped to the full TTL, so it only bites if you raise PSE_REFRESH_TTL_HOURS
ZEPTOMAIL_API_KEY	unset	Unset ⇒ emails logged to console
PSE_EMAIL_FROM	no-reply@localhost	Must be a verified sender domain
PSE_USAGE_RETENTION_DAYS	90	The privacy page states 90
PSE_STATEFUL / PSE_SSE	false / false	See below
PSE_LOG_JSON / PSE_LOG_LEVEL	false / INFO	

PSE_PUBLIC_URL must be the real external HTTPS URL. It simultaneously drives the WebAuthn `rp_id`, the links in verification emails, the OAuth issuer in discovery documents, and the DNS-rebinding host allowlist. A wrong value breaks passkeys in a way that looks like a browser bug, and enrolled credentials cannot be recovered — only re-enrolled. Verify it after every deploy: `curl -s https://<host>/.well-known/oauth-protected-resource`

Why HTTP is stateless with plain JSON by default

The server is read-only tools over data the freeze holds still. It uses **none** of the features MCP sessions exist to enable — no notifications, no resource subscriptions, no sampling, no elicitation, no progress. Every request is self-contained.

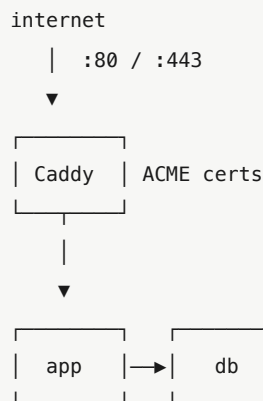
That makes horizontal scaling ordinary: any replica serves any request behind plain round-robin — no sticky routing, no per-session memory, no event store. Without SSE, idle clients hold no connection, so N users stop meaning N concurrent connections.

`--stateful` and `--sse` restore session mode for anyone who needs resumability or server-initiated messages. They are independent flags: statelessness is the scaling property, plain JSON is what keeps ordinary proxies out of the way.

Workers are processes. All in-memory state — quota windows, the parse memo, the token cache, the usage buffer — is **per worker**, so N workers means a user's effective quota ceiling is up to N× nominal. Scale workers for CPU, and scale limits with them.

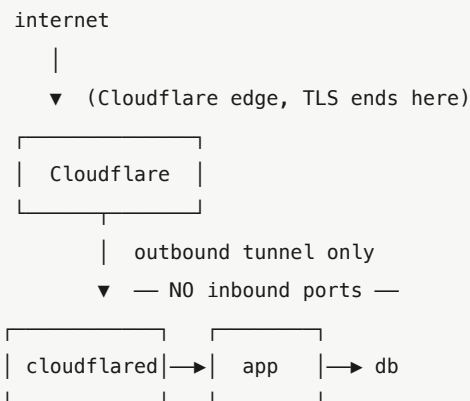
15. Deployment topologies

A. VPS that owns ports 80/443
compose.prod.yaml



Requires the router to forward
80→8280 and 443→8243: ACME
validates on 80/443 only.

B. NAS behind Cloudflare Tunnel
compose.nas.yaml [+ compose.tunnel.yaml]



Needs no inbound ports at all.
Tunnel route is HTTP → app:8000
(container port, not the host port)

Both pull the published, CI-gated multi-arch image; neither builds from source. Pin `PSE_IMAGE_TAG` — a moving `:latest` that lags the compose file reads as a broken deployment.

The runtime image contains only what the app needs to run: multi-stage, non-root, no build tools, no dev dependencies, no source tree, no credentials in any layer. `scripts/check_image.py` asserts installed distributions **equal** the resolved runtime closure from `uv export`, so any stray package fails the build. Size is reported as information only — **the gate is necessity, not a number.**

16. Gotchas

Findings that cost real debugging time. Most are recorded in `docs/endpoints.md` with the evidence.

- **Parse disclosure tables by `<thead>` label, never by column position.** The two search endpoints order their columns differently.
- **Financial-report units labels contradict each other** between the annual and quarterly sections. Values are passed through **verbatim and never rescaled**; each period reports its own `currency_units` for the caller to check.
- **Index changes are printed unsigned.** PSE Edge shows direction only as a colour and a ▲/▼ glyph, so signs are derived from the glyph.
- **selectolax traps:** `iter()` walks direct children only, and `css("a, b")` returns matches **grouped by selector, not in document order** — which once filed all four financial statements under the last heading. Use `root.traverse()`.
- **Accounting negatives** `(1,234)` parse to `-1234`.

- `dividends_and_rights_form.do` is an empty shell. It posts to `dividends_and_rights_list.ax` once per tab, with `DividendsOrRights` in the query string and `cmpy_id` in the body.
- Homepage feeds are keyed by PSE Edge's own group labels, not invented buckets.
- `market_calendar.PSE_HOLIDAYS` is a yearly-maintained seed. Verify it against the PSE holiday circular each December and whenever touching calendar logic.
- `/health` must never touch the database. A DB-dependent liveness probe restarts every replica during a blip, turning a recoverable outage into an outage plus a restart storm. `/health/ready` is where the DB check belongs.

17. Where to look next

I want to...	Read
Understand why a decision was made	<code>docs/plan.md</code>
Add or fix an upstream call	<code>docs/endpoints.md</code> , then <code>client.py</code> + <code>parsers.py</code>
Change what a tool returns	<code>models.py</code> , then the repository
Change caching behaviour	<code>service.py</code> — and re-read §4 first
Deploy or operate it	<code>docs/deploy.md</code>
Know the invariants quickly	<code>CLAUDE.md</code>