Users & Security — Authentication and RBAC

MessageFoundry authenticates every operator and authorizes every action with **role-based access control (RBAC)**. It supports **local users** and **Active Directory** (LDAP bind, with optional Windows SSO), maps **AD security groups to roles**, and attributes every action to a unique user in a tamper-evident audit trail.

RBAC is built in — not a paid add-on. Password policy ships with secure defaults, AD-group-to-role mapping is automatic, and multi-factor authentication is required for local accounts.

> **Scope.** This document covers **identity, access control, and the audit of operator actions**. The protection of the *data* itself — at-rest storage and encryption, transport, logging and redaction, retention, and de-identification — is covered separately in the PHI handling guide. MessageFoundry is designed to run **inside the organization's private network**; the trust boundary and the management/data/inbound plane model are described in the PHI guide and the deployment guide.

Enforcement model

Authentication is **required** for the running service. The engine's `serve` command always attaches an auth layer, so every API route except `GET /health` demands a valid bearer token, and each route additionally demands a specific **permission**.

The in-process embedding factory is **fail-closed**: with no auth service attached it denies every protected route (HTTP 503) unless the caller explicitly opts out — the deliberate embedding/local-development escape hatch. The service refuses to serve with auth disabled on a non-loopback host.

Even with auth enabled, a non-loopback bind requires **TLS** — either in-process (an API TLS certificate) or terminated at a trusted upstream reverse proxy (with forwarded-header and trusted-proxy configuration). Without one of these, an explicit insecure-bind override is required; otherwise the bind is refused, so bearer tokens and PHI can never cross the network in cleartext by accident. There is no way to be silently served with unauthenticated full access, and a stray host edit cannot quietly void the loopback assumption.

> **Deployment guidance.** Exposing a listener off loopback always requires TLS in front of it. Plan to terminate TLS in-process or at a trusted reverse proxy before binding any interface beyond `127.0.0.1`.

First-run bootstrap admin

On first start against an empty store, the engine creates a single **bootstrap admin** (username `admin`, role `Administrator`) with a random one-time password **generated through the active password policy**. The password is **written to an owner-only file** next to the store — **never to the log** — and only the file's location is logged, so the credential is never captured in service stdout. Sign in with it, change the

password immediately (the account is flagged "must change password" and enforces this), and delete the file. Once any user exists, no further bootstrap occurs.

Auto-retirement. The bootstrap account exists only to seed the first real admin, so while still **unclaimed** (never password-changed) it self-retires: it is **disabled once a second administrator exists**, and — if left unclaimed — disabled a configurable number of hours after creation (default 72 h; a value of `0` disables the timer). Once you change its password it becomes a normal admin account and is never auto-disabled, so a single-admin deployment can't lock itself out. A retired bootstrap login is refused like any other invalid credential, and the retirement is audited.

Admin password reset

An administrator (holding `users:manage`) recovers a locked-out or compromised **local** account with `POST /users/{user_id}/reset-password`. The engine generates a one-time password through the active policy, flags the account "must change password", and **revokes the user's sessions**. The temporary credential is returned **once** in the response for the admin to convey out-of-band, and the affected user is also emailed a reset notice. The administrator therefore never sets a *lasting* password the user keeps — the one-time credential must be rotated on first login (ASVS 6.4.6).

AD users are refused (they authenticate against the directory); resetting your own account is refused (use self-service change-password). The action is audited. For the same reason, **admin-created accounts are flagged "must change password"**, so the operator's initial password is a one-time temporary that the user must rotate.

Anti-automation. A per-actor human-timing pacing floor on sensitive *authenticated* writes is deliberately not implemented for the default deployment: the API binds `127.0.0.1`, every sensitive write is RBAC-gated to an authenticated operator, and the unauthenticated brute-force surface is already bounded by the sliding-window rate limiter plus the per-actor PHI-read throttle. Machine-speed abuse of an authenticated admin endpoint is not material in the single-tenant, on-loopback console model; a pacing floor would be revisited only if a sensitive write is exposed off loopback.

Roles & permissions

Roles are a **fixed built-in set** (a custom-role builder is not yet available). Each maps to permissions from the catalog below; holding multiple roles grants the **union** of their permissions (deny-by-default otherwise).

Role	Permissions
Administrator	everything (including `users:manage`, `audit:read`)
Operator	`monitoring:read`, `monitoring:diagnose`, `messages:read`, `messages:view_summary`, `messages:view_raw`, `messages:replay`, `messages:purge`, `connections:control`, `connections:test`
Deployment	`monitoring:read`, `config:deploy`, `config:validate`, `connections:test`
Coding	`monitoring:read`, `code:edit`, `config:validate`, `ai:assist`
Viewer	`monitoring:read`, `messages:read`
Auditor	`monitoring:read`, `audit:read`

Permission catalog: `monitoring:read`, `monitoring:diagnose`, `messages:read`, `messages:view_summary` (PHI), `messages:view_raw` (PHI), `messages:replay`, `messages:purge`, `connections:control`, `connections:test`, `config:deploy`, `config:validate`, `code:edit`, `service:configure`, `ai:assist`, `users:read`, `users:manage`, `audit:read`, `approvals:approve`.

> **AI coding assistance is RBAC-gated and centrally policy-governed.** `ai:assist` (held by **Coding** and **Administrator**) controls whether an identity may use the IDE AI assistant. The assistant is additionally bounded by an environment-clamped central **policy** (`mode` from off through PHI-safe, plus `data_scope` and `environment`) read via `GET /ai/policy`. That endpoint is intentionally unauthenticated — the install policy is non-sensitive operational config that a central "off" must be able to enforce on a tokenless client — and the identity-dependent decision rides in its `assist_permitted` field. See the AI assistance guide.

Route → permission map (engine API)

Endpoint(s)	Permission
`GET /health`	none (liveness)
`GET /channels`, `/connections`, `/status`, `/stats`, `ws /ws/stats`	`monitoring:read`
`POST /status/integrity-check`	`monitoring:diagnose`
`GET /messages`	`messages:read`; `messages:view_summary` unlocks the `summary` / `error` fields (per-property — see *Field-level authorization*)
`GET /messages/{id}`	`messages:view_raw` (the raw body); `messages:view_summary` unlocks `summary` / `error` and the nested `last_error` / event `detail` (per-property)
`GET /messages/{id}/responses`	`messages:read`; `messages:view_summary` unlocks the captured-reply `detail`, `messages:view_raw` the reply `body` (per-property)
`POST /messages/{id}/replay`	`messages:replay`
`GET /dead-letters`	`messages:read`; `messages:view_summary` unlocks the `summary` / `last_error` fields (per-property)
`POST /dead-letters/replay`	`messages:replay`
`POST /connections/{name}/{start,stop,restart}`	`connections:control`
`GET /connections/{name}/metadata`	`monitoring:read` (per-channel for inbound; a shared outbound is barred to scoped users)
`POST /connections/{name}/test`	`connections:test` (reachability probe — builds a fresh connector, honors egress policy, sends no real data, audited)
`POST /connections/{name}/purge`	`messages:purge`
`POST /config/reload`	`config:deploy`
`GET` / `PUT /users/{id}/channel-scope`	`users:manage` (per-channel RBAC)
`GET` / `PUT /ad-group-scope-map`	`users:manage` (AD-group → channel scope)
`GET /approvals`, `POST /approvals/{id}/approve`, `POST /approvals/{id}/reject`	`approvals:approve` (dual-control release/decline — see below)

> **Per-channel scoping.** Operational permissions can be confined to a set of connections per user (`PUT /users/{id}/channel-scope` ; `null` = all, the default). When a user is scoped, `messages:read` / `view_raw` / `replay` , dead-letter list/replay, and `connections:control` are restricted to their channels (out-of-scope message access returns 404 to avoid leaking existence; connection control returns 403; denials are audited). **Administrators are always all-channels.** Monitoring dashboards stay global. A channel-scoped user **cannot purge** a shared outbound (purge spans every inbound feeding it). **AD users** inherit their scope from the AD-group → scope map (channel `*` = all): on login the group-derived scope is persisted and stale sessions revoked. It is opt-in — with no matching mapped group, the user's existing scope is left untouched.
>
> `/config/reload` **executes Python** from the target directory in-process, so it is constrained beyond the `config:deploy` permission: the directory must resolve **within** an allowed root — the server's startup config directory or an entry in the allowed reload-roots list — otherwise it is rejected (403). Every reload (and every denial) is audited with the acting user; error responses are generic so a holder can't probe the filesystem via reload errors. Lock down the config/staging directories' access controls accordingly (see the service guide).

Dual-control approval for high-value actions

High-value operations can require a **second approver** before they execute — a maker-checker control. It is **opt-in and deny-by-default** (off unless enabled): a single-operator deployment is never blocked, and existing behavior is unchanged until you turn it on.

When enabled for an operation, invoking it does **not** execute inline. The request (operation, its parameters, and the requester) is **persisted**, and the endpoint returns **202** with an `approval_id` ; the action is held until a **distinct** user holding `approvals:approve` releases it via `POST /approvals/{id}/approve` . The requester can **never approve their own request** (enforced server-side, not by a client confirmation). On release the captured operation is **re-executed** and **both identities** are written to the hash-chained audit log; `POST /approvals/{id}/reject` declines it, and a request older than the configured expiry can no longer be approved. Approvers see the open queue at `GET /approvals` .

The gated set is configurable; the first cut covers the two highest-PHI-impact flows — **bulk dead-letter replay** and **connection purge**. (Console "are you sure?" prompts are client-side only and bypassable via the raw API — they are *not* a second approver and do not satisfy this control.)

Step-up re-verification on sensitive operations

A highly sensitive operation requires the caller's session to have **re-proved its credential recently** — not merely to hold a valid token. A step-up dependency refuses with **403** (header `X-Step-Up-Required: 1`) unless the session re-verified within a short window (default **300 seconds**). The **initial login counts as the first verification** (the sudo-timestamp model): the session's re-auth timestamp is stamped at login and refreshed by `POST /me/reauth` , so a session only needs to re-verify once its window lapses. `POST /me/reauth` re-checks the **local** password (argon2id) or performs a **live Active Directory re-bind** for AD accounts, so AD operators are never locked out. It is rate-limited like the password change and audited.

Gated operations (all `users:manage` admin flows plus the replay/purge/config flows): create / delete user, set roles, set channel scope, admin session-revoke, admin reset-password, AD-group role/scope maps, dead-letter replay, single-message replay, connection purge, and config reload/deploy. Reads (listing users, maps, the audit log) are **not** gated.

Step-up re-proves the password as a secondary verification. The step-up gate **also** requires the session's second factor: an MFA-required caller is refused with `403` + `X-MFA-Required` until `POST /auth/mfa-verify` succeeds (a TOTP code or a single-use recovery code), so these routes carry both a recent password re-verify **and** the MFA factor. The step-up window composes with the dual-control approval above (the requester re-verifies; an independent approver still releases the action).

Multi-factor authentication

Multi-factor authentication is built in and required for local accounts. Local accounts enroll a native **RFC 6238 TOTP** second factor: `POST /me/mfa/enroll` returns a setup key and `otpauth://` URI for an authenticator app, `POST /me/mfa/confirm` activates it and returns the **single-use recovery codes** (shown once), and `POST /auth/mfa-verify` satisfies a session's second factor with a TOTP code or a recovery code. `DELETE /me/mfa` disables it; an administrator clears a lost authenticator via `POST /users/{id}/reset-mfa` (which also revokes the user's sessions).

Required MFA gates **local** accounts: a local administrator must satisfy MFA before any step-up operation (the gate returns `403` + `X-MFA-Required` until verified). **Enterprise/AD users get MFA through their own identity provider** — a directory login is satisfied by the directory's controls (for example Entra Conditional Access or an MFA proxy) and is never prompted for an engine TOTP. Because required MFA gates only local accounts, it is safe to enable even on an AD-only deployment.

The TOTP secret is stored **encrypted at rest** (the store cipher) and recovery codes are **argon2id-hashed**; verification uses the server clock and a constant-time compare with a ±1-step window. TOTP is a shared-secret factor; phishing-resistant **WebAuthn/FIDO2** at the same step-up boundary is a planned follow-on.

When the API is bound **off loopback**, the service makes the posture explicit at startup so an exposed PHI deployment can't silently run the administrator interface single-factor — it refuses to start a production PHI instance configured without required local-account MFA and warns on a non-production PHI instance (mirroring the keyless-store and open-egress startup gates). The remediation is simply to require MFA (or keep the bind on loopback).

Administrative-interface defense-in-depth

The administrative interface is defended by **multiple independent layers**, not network-location trust alone:

1. **Network-location / exposed gate** — the API binds `127.0.0.1` by default, and a non-loopback plaintext bind is refused at startup unless explicitly overridden. One layer, not the sole factor.
2. **Deny-by-default per-route RBAC** — every admin route asserts an explicit permission over an opaque bearer token; a denial is audited.
3. **Step-up re-verification** within a short window on every sensitive admin route (above).

4. **A genuine second authentication factor** at that step-up boundary — native TOTP MFA, so an MFA-enrolled/required admin presents a TOTP or recovery code, not a re-prompt of the same password.
5. **A contextual-risk signal** — when enabled, a sensitive admin action arriving from a **client IP the session has not verified from** emits an audit event and an out-of-band notice and **forces a fresh step-up**; a successful re-auth (or MFA verify) from that address re-anchors the session and clears the signal. The event and notice fire once per (session, new address), so a replayed token can't inflate the audit log. It is advisory and step-up-forcing only — it never changes an RBAC allow/deny and never blocks the non-admin request path. Default off, and byte-identical on a single-host loopback bind; recommended on for an off-loopback admin deployment.

Continuous identity verification underpins all of the above: every request re-resolves the user and roles from server-side state and re-checks idle/absolute timeout plus live disabled/role status, so a revoked privilege or disabled account takes effect immediately.

Device security-posture assessment is deployment-delegated, not built in-process: an attested/managed admin host and an **mTLS client certificate terminated at the reverse proxy** are the posture control, consistent with the on-prem console model. This is the documented residual for the device-posture clause.

Field-level (property) authorization

Beyond gating whole *endpoints*, the API gates individual **PHI-bearing properties** within a response, so a caller can see an object without seeing its patient-identifying fields. The policy is declared in one place and enforced by a single redaction helper applied to every returned row, rather than re-implemented inline per endpoint (where a new endpoint or field could silently leak PHI).

Response property	Carries	Unlocked by
`summary` (message / dead-letter / detail rows)	patient identifiers (MRN / name / order)	`messages:view_summary`
`error` / `last_error`, event `detail`, captured-response `detail`	exception / disposition text that can quote field values	`messages:view_summary`
`raw` (single-message body), captured-response `body`	the full message / reply	`messages:view_raw` (whole-body gate, at the endpoint)

A caller lacking `messages:view_summary` receives those properties as `null`; everything returned non-null is audited server-side (coalesced per actor per hour), so a scripted bulk read can't harvest the patient census unaudited. The body (`raw`) is the coarser whole-object gate and stays at the endpoint.

The same logical disposition text gates on `messages:view_summary` on **every** surface that returns it — the `GET /messages` and `GET /dead-letters` lists, the single-message detail (where the detail wrapper **and** each nested outbox/event record are redacted individually), and the captured replies. **`messages:view_raw` is not a superset of `messages:view_summary`** — they are independent permission flags. The built-in roles *happen* to grant them nested (a role-policy convention, not a permission-model guarantee), so the **Viewer** role sees metadata only; the disposition fields are gated on `view_summary` deliberately, so a future custom role

holding `view_raw` without `view_summary` still cannot reach exception text. Adding a new PHI-bearing response property means adding a row to the field map — a test asserts the map matches the expected set, so the gate can't be forgotten silently.

Write side (engine → store). Exception and disposition text is also scrubbed *before* it is stored. A router or handler is user code that can raise an exception interpolating a raw message fragment, so every value written to the error/disposition columns goes through a redaction chokepoint at the runner, the connectors, **and** the store write methods — so an HL7-shaped fragment can't land in those columns. HL7-shaped content (segment dumps, multi-delimiter field runs) is cut while the exception **type** and field **name** are kept; the residual control for free-text PHI a script invents (for example a bare `"DOE^JANE"`) remains the read-side gate above plus the "never put PHI in an exception" convention. Encrypting these columns at rest on **every** backend (SQLite and PostgreSQL already do) is a tracked defense-in-depth follow-up.

Write side — not applicable by design. The API exposes **no client-writable PHI properties**: every mutation is a coarse, separately permission-gated action (`messages:replay` / `messages:purge` / `config:deploy` / `connections:control`), not a per-field write. So there is no per-property *write* authorization surface today. The trigger to revisit is the first endpoint that lets a client write a PHI property (for example an edit/annotation API), at which point a writable-property-to-permission whitelist would be added alongside the read map.

Local vs Active Directory

Both kinds of user share one identity model (a user's auth provider is `local` or `ad`).

- **Local users** authenticate with an argon2id-hashed password and are assigned roles explicitly (`PUT /users/{id}/roles` or the console Users page).
- **AD users** authenticate by LDAP simple-bind over **LDAPS**. The engine binds with a service account to find the user, binds as the user to verify the password, then resolves group membership (including **nested** groups). Their roles are **re-synced from AD groups on every login** through the AD-group-to-role map, so manual role assignment does not apply to AD users.
- **Windows SSO (Kerberos)** — optional and experimental. `POST /auth/negotiate` completes a SPNEGO exchange for passwordless login on a domain-joined client; the resulting principal's groups are resolved the same way. It requires a server keytab/SPN and is single-leg only (no mutual authentication and no NTLM-fallback / multi-leg exchange). Every Kerberos reject path is audited.

AD-group → role mapping

An admin sets which AD groups govern which role via `GET/PUT /ad-group-map` (or the console). Group identifiers are matched case-insensitively and may be either the group **DN** or its **sAMAccountName**. A user in multiple mapped groups gets the union of those roles.

```
CN=MF-Admins,OU=Groups,DC=example,DC=com  ->  administrator
CN=MF-Ops,OU=Groups,DC=example,DC=com     ->  operator
```

Sessions

Sessions are **opaque server-side tokens** (not JWTs): the client holds the token, the store keeps only its SHA-256, so logout, expiry, and role changes take effect immediately. Each request enforces an **idle timeout** (default 30 min) and an **absolute lifetime** (default 12 h); changing a password, disabling a user, or an AD-group/role change on re-login revokes that user's sessions. Session validation **fails closed on a backward wall-clock step** (an NTP step-back or VM snapshot revert) rather than reviving an expired token, and the idle clock is only refreshed by **user-driven** requests — a background keepalive does not keep a session alive. A configurable cap limits concurrent sessions per user (default **5**; a login beyond the cap revokes the user's oldest; `0` = unlimited).

The console stores the token in the OS keyring (Windows Credential Manager) and sends it as `Authorization: Bearer <token>` (the WebSocket prefers the header; the legacy query-parameter form is deprecated because it leaks into proxy/access logs). The keyring item is a PHI-scoped credential (the user's full RBAC for the session lifetime); the console re-validates it against `/auth/me` on startup (discarding a stale or revoked one) and **refuses to send credentials over plaintext HTTP to a non-loopback host** unless explicitly run in an insecure trusted-network development mode.

Session inventory & targeted revocation

Users and admins can see and revoke individual sessions:

- `GET /me/sessions` — your active sessions (created / last-used / expiry / client; the current one is flagged). The session `id` is a one-way hash of the opaque token, safe to expose.
- `DELETE /me/sessions/{id}` — revoke one of **your own** sessions (ownership-checked: another user's id returns 404, never revealing or touching it).
- `DELETE /me/sessions` — "sign out everywhere else": revoke all your sessions except the current.
- `DELETE /users/{id}/sessions` (`users:manage`) — admin force-sign-out of a user (offboarding or suspected compromise).

Every targeted revoke is audited (with scope and actor). The console surfaces this: an **Active sessions** dialog in the account menu lists your sessions and offers per-session revoke plus "sign out everywhere else", and the **Users** page has a **Revoke sessions** action for admin force-sign-out.

Security-event notifications

Users are notified of security-relevant changes to their account through **two** channels:

- **Out-of-band email to the affected user** (default on; it reuses the alert SMTP transport and is sent to each user's **own** address — not the operator alert distribution list). Fired on account **lockout** and the **first successful login after repeated failed attempts** (suspicious-login signals); and on **password change**, **email change**, **role change**, and **account disable** (credential changes). An email-change notice goes to the **old** address so the legitimate owner is alerted even if the change was hostile. With no SMTP configured (or for accounts with no email on file), the email is simply skipped. Emission is **best-effort** — a notification failure is logged and never blocks a login or an admin action.

- `GET /me/security-events` — a pull-based feed of the caller's own audited `auth.*` events (sign-ins, lockouts, password changes), most-recent-first, for accounts without a deliverable mailbox. It is a read-only view over the tamper-evident audit log (no new store of record). Admin-initiated changes (whose audit actor is the admin) are delivered by the email channel, not shown in this self view.

Password policy

Local passwords follow an **ASVS 5.0-aligned** policy: **minimum length 15**, **no mandatory character-class composition** (the per-class requirement flags are opt-in and default off, because ASVS forbids mandatory composition), plus **offline breached/common-password screening** (a bundled top-10k list, no live network call) and a small **context-word deny-list** (application, vendor, and HL7 terms). The policy is enforced identically on create-user and change-password and is tunable in configuration. AD passwords are governed by Active Directory.

Two further screens, both on by default and fully offline:

- **Username-in-password rejection** — a password that *contains* the user's own username (case-insensitive, for usernames of at least 4 characters) is rejected, catching common choices that a corpus can't.
- **Larger operator breach corpus** — point this at an offline list to augment the bundled top-10k: a **plaintext** file *or* an **HIBP-style SHA-1-hash export** (auto-detected), checked locally with no network call. Use a curated subset (it is loaded into memory), not the full multi-gigabyte set; a configured-but-unreadable path is warned at startup and falls back to the bundled list.

Authentication pathways — comparative strength

Pathway	Factor	Brute-force defense	Notes
Local (argon2id)	password + required MFA (TOTP)	**per-account lockout** (5 attempts / 15 min) + breach/context policy + global rate-limit	the only pathway the engine itself can lock out
AD (LDAPS simple-bind)	password (MFA via the directory)	the **directory's** lockout/complexity policy (engine has the global rate-limit only)	password strength + lockout are the AD domain's responsibility
Kerberos / SPNEGO	domain ticket (often MFA-backed)	the **domain's** controls; passwordless on a joined client	experimental, single-leg; no engine-side password

Lockout asymmetry: the engine's per-account lockout protects **local** accounts only. AD/Kerberos brute-force resistance is the directory's job — so for AD-backed deployments, set the domain lockout/complexity policy accordingly; the engine's global sliding-window rate-limit is the only engine-side throttle that also covers the AD login path. Native TOTP MFA is required for local accounts; AD/Kerberos MFA is delegated to the directory.

Brute-force & abuse protection

Beyond per-account lockout, the unauthenticated auth surface (`/auth/login` , `/auth/negotiate` , `/me/password`) is **rate-limited** by an in-process sliding window — per client IP and globally — so password-spraying across many usernames (which never trips a single account's lockout) is bounded, and concurrent argon2id work is **capped** so a login flood can't exhaust the executor. Request bodies are capped (1 MiB) and auth request fields have length limits.

The **authenticated PHI-read endpoints** (`/messages` , `/messages/{id}` , `/dead-letters`) carry a **per-actor anti-automation throttle** — a sliding window keyed on the acting user, on by default at a generous cap (120 reads/min) that clears normal console and human use while bounding scripted PHI harvesting. It complements pagination and the per-access audit trail on those routes; a throttled read is logged (never silent) and returns `429` .

These are all **in-process** protections; an exposed or multi-host deployment must additionally front the API with a proxy/WAF limiter and TLS.

Audit

Every authentication and authorization event is written to the durable audit log with the acting user: `auth.login_success` / `auth.login_failed` / `auth.login_locked` / `auth.logout` / `auth.permission_denied` / `auth.channel_denied` , plus `user.created` / `user.roles_changed` / `user.channel_scope_changed` / `user.deleted` , the AD-group map and scope-map updates, and the AD-scope-resync event. PHI access (viewing a raw message or displaying patient summaries) is recorded with the viewer. Read the trail via `GET /audit` (`audit:read`). **Credentials, tokens, and PHI bodies are never logged** (only ids and counts land in event detail).

Tamper-evidence. Each audit row carries a `row_hash` that chains the previous row's hash with this row's content (SHA-256), so deleting, editing, or reordering any row is detectable. Verify the chain with `messagefoundry audit-verify` (exit 0 = intact). Rows written before the feature are chained on first start. This is in-database tamper-*evidence*, not prevention — restrict the store/file access controls (and run least-privilege; see the service guide) so the log can't be rewritten in the first place.

Off-box forwarding. The hash chain detects on-host tampering but lives on the same host as the data it protects; if that host is compromised, local evidence can be tampered with. The **general log** can therefore be shipped **off-box** to a syslog/SIEM collector (structured JSON supported), so an independent copy survives a host compromise. The same PHI-redaction and control-character-scrub filters apply to the forwarded stream as to stdout; the syslog transport is plaintext, so terminate it at a local TLS-forwarding agent or keep it on a trusted management network. The **audit rows themselves** are **also** forwarded off-box: every committed audit row ships as PHI-redacted metadata to the same forwarder, across all three store backends — so both the operational log and the tamper-evident audit trail survive a host or database compromise.

HIPAA §164.312 alignment

MessageFoundry **supports a HIPAA-compliant deployment** by providing the technical safeguards described here. Compliance is a property of the overall deployment and operating environment, not of the software alone.

- **Unique user identification** (required) — every user is a distinct account; no shared logins.
- **Person/entity authentication** (required) — local argon2id and/or AD bind; lockout on brute force; MFA for local accounts.
- **Audit controls** (required) — durable, user-attributed audit trail (append-only via the store API).
- **Automatic logoff** (addressable) — idle and absolute session timeouts.
- **Emergency access** (required) — the bootstrap admin provides break-glass; treat its credential as a sealed secret.

Console sign-in

`python -m messagefoundry.console` shows a sign-in dialog (Local / Active Directory) when the engine requires auth, caches the token in the OS keyring, gates UI actions by permission, exposes a **Users** admin page to `users:manage` holders, and offers **Sign out** (clears the token).

Interface authentication (machine-to-machine)

Beyond operator sign-in, interface and API clients authenticate by the mechanism appropriate to the integration:

- **Mutual TLS (mTLS)** — a client certificate terminated at the reverse proxy authenticates the calling system, the recommended posture for an exposed admin or API plane.
- **OAuth 2.0 client-credentials** — for the outbound FHIR/REST connector, MessageFoundry mints a per-request bearer token via the `client_credentials` grant and re-mints on a `401`. The token endpoint is gated by egress policy.
- **SMART-on-FHIR Backend Services** — the FHIR connector authenticates to a SMART Backend Services endpoint with a signed-JWT (RS384/ES384) client assertion, opted in per connection. It is **client-only** — no App Launch flow and no authorization-server facade.

MLLP-over-TLS is available per connection for HL7 v2 transport, and the DICOM C-STORE SCP inbound supports DICOM-over-TLS with allowlisted calling AE titles and peer IPs. A non-loopback plaintext bind on any of these listeners is refused at startup unless explicitly overridden.

Security posture & self-assessment

MessageFoundry has completed an **internal self-assessment against OWASP ASVS 5.0 at Level 3**: **212 requirements met, 0 failed, 0 partial, 133 not applicable**, of 345 total. This is a **self-assessment, not an**

external audit. At Level 3, an independent code review and penetration test are *recommended but not required*; both are planned. MessageFoundry is not "certified" against any standard, and no compliance outcome is guaranteed.

Supply-chain & CI security

Automated security scanning runs in continuous integration:

- **pip-audit** — audits the committed lockfile for known-CVE dependencies, so the audit is reproducible rather than auditing a fresh latest-resolve.
- **bandit** — Python static analysis over the engine source.
- **Dependabot** — scheduled dependency-update PRs for `pip` and GitHub Actions.
- **CodeQL** code scanning and **secret scanning** with push protection.
- A private vulnerability-disclosure policy is published with the project.

Planned additions include a generated CycloneDX **SBOM** kept as a build artifact and a full-history **secret-history scan** to complement secret scanning.

Dependency lockfile

The project's dependency manifest carries lower-bound ranges; the **pinned, hashed** resolution lives in a committed lockfile and its cross-platform exported view (with per-package hashes). CI verifies the two are in sync and audits the export. For a fully reproducible, tamper-resistant install, use `pip install --require-hashes` against the exported lockfile (the SQL Server extra also needs the OS-level Microsoft ODBC Driver 18, which is not pip-installable).

Before installing the engine wheel itself, **verify its release provenance** — MessageFoundry publishes signed, attested wheels (Sigstore identity + SLSA provenance), so `gh attestation verify` confirms who built the artifact. Hash-pinning proves the bytes match the lockfile; provenance verification proves the build's origin. See the install guide.

Roadmap (not yet built)

Entra ID / OIDC federation, custom roles, and the remaining `code:edit` / `config:validate` / `service:configure` API endpoints are deliberate follow-ups, along with phishing-resistant WebAuthn/FIDO2 as an additional second factor at the step-up boundary.

MessageFoundry is an independent product and is not affiliated with or endorsed by Mirth, Corepoint, Cloverleaf, Rhapsody, or Ensemble; those names are trademarks of their respective owners. Comparative statements are offered in good faith and should be verified against each product's current documentation.
