Secrets and Vault Configuration

Version 1.3.0-SNAPSHOT, 2026-02-17T20:12:18Z

Table of Contents

This section describes how secrets are used in Quantum applications and how to configure **global** (tenant-independent) vs **tenant-level** (tenant-specific) secrets, including integration with vaults such as HashiCorp Vault or AWS Secrets Manager.

Chapter 1. Two Levels of Secrets

Quantum distinguishes two levels of secrets. The JVM and framework need both; resolution and storage differ.

1.1. Global secrets (tenant-independent)

Global secrets are application-wide. They are resolved once at startup (or when the config source is first used) and are not scoped by realm.

- **Typical uses:** JWT signing key, MongoDB connection string, OIDC client secret, vault connection credentials, and (if you use a single shared key) a global LLM API key.
- **Who resolves them:** Quarkus config (e.g. `application.properties` with `${ENV_VAR}` placeholders) or a vault-backed config source that does **not** take a tenant/realm parameter.
- **Where to store:** Environment variables, `.env` (dev only, not committed), or a vault path that has no realm in the path (e.g. `secret/myapp/global/jwt-secret`).

Example (global JWT secret via env):

```
auth.jwt.secret=${JWT_SECRET:change-me-in-production}
```

Example (global secret from HashiCorp Vault KV path — see [vault-hashicorp]):

You configure a single KV path for the app; Quarkus or the Vault extension loads those keys at startup. No realm is involved.

1.2. Tenant-level secrets (tenant-specific)

Tenant-level secrets are per-realm. They are resolved at request time (or when tenant config is loaded) and must not be shared across tenants.

- **Typical uses:** Per-tenant LLM API keys (e.g. each tenant's own OpenAI/Claude/Gemini key), per-tenant OAuth client secrets, tenant-specific webhook signing keys, or credentials for tenant-specific external systems.
- **Who resolves them:** Tenant configuration (e.g. `TenantAgentConfig` or a tenant config store). The resolver receives the realm and returns the secret (or a reference to it) for that realm only.
- **Where to store:** Prefer a vault, with a path that includes the realm (e.g. `secret/myapp/tenant/<realm>/llm-api-key` or `secret/myapp/tenant/<realm>/oauth-client-secret`). Alternatively, a tenant config store (DB or repo) that holds either the secret value or a vault path/key reference for that tenant.

Example (tenant-level LLM API key):

- Config says: for realm `acme-corp`, use vault path `secret/myapp/tenant/acme-corp/llm`.
- At request time, when the agent runs for `acme-corp`, the framework resolves tenant config, sees

the vault path, and fetches the secret for that path only. Other realms never see `acme-corp`’s key.

Tenant-level secrets may require **runtime resolution**: the framework (or your custom `TenantAgentConfigResolver` / secret provider) must call the vault or config store per request (or per cached tenant). This is in contrast to global secrets, which are usually bound once at startup.

Chapter 2. Storing Secrets Without a Vault

For development or simple deployments, you can avoid a vault:

- **Global:** Set secrets in environment variables or in `application.properties` with placeholders (e.g. `auth.jwt.secret=${JWT_SECRET}`). Never commit real secrets; use `.env` locally and inject env in production.
- **Tenant-level:** Not recommended in properties files (realm-scoped keys do not fit the flat property model well). Prefer a vault with realm-scoped paths, or a tenant config store (e.g. MongoDB) that stores a vault path or secret reference per tenant rather than the raw secret.

See Configuration Reference for file locations and precedence.

Chapter 3. HashiCorp Vault (Quarkus)

Quarkus can integrate with HashiCorp Vault via the Quarkus Vault extension (Quarkiverse). The extension can expose Vault KV secrets as configuration and implements the Quarkus Credentials Provider for datasources and other consumers.

3.1. Global secrets with HashiCorp Vault

- Configure the Vault connection and a KV path that holds your global secrets (e.g. JWT secret, DB password).
- Map Vault keys to Quarkus config property names so that `auth.jwt.secret` (or similar) is supplied from Vault at startup.
- No realm is used in the path; the same values apply to the whole application.

Example (conceptual — exact properties depend on the extension version):

```
quarkus.vault.url=https://vault.example.com
quarkus.vault.authentication.client-token=${VAULT_TOKEN}
# KV path for global app secrets (e.g. key "jwt-secret" -> auth.jwt.secret)
quarkus.vault.kv-secret-engine.path=secret/myapp/global
```

Consult the Quarkus Vault documentation for current property names and Credentials Provider usage.

3.2. Tenant-level secrets with HashiCorp Vault

- Store each tenant's secrets under a path that includes the realm, e.g. `secret/myapp/tenant/<realm>/llm` or `secret/myapp/tenant/<realm>/oauth`.
- The Quarkus Vault extension is typically used for **global** config. For **tenant-level** secrets you usually need application code that:
  - Resolves the current realm (from request or context),
  - Builds the path, e.g. `secret/myapp/tenant/` + realm + `/llm`,
  - Calls Vault (e.g. via the extension's programmatic API or a Vault client) to read that path and returns the secret only for that tenant.
- Optionally, tenant configuration (e.g. in DB or properties) can store the vault path or key name per realm so that the path pattern is configurable.

This keeps tenant-level secrets out of a single global config and ensures they are resolved per tenant at runtime.

Chapter 4. AWS Secrets Manager

AWS Secrets Manager is not a built-in Quarkus ConfigSource. You can integrate it in two ways:

- **Custom ConfigSource:** Implement a Quarkus `ConfigSource` that fetches secrets from AWS Secrets Manager (e.g. by secret name or ARN) and exposes them as config properties. Use it for **global** secrets (e.g. one secret that holds JWT key or DB URL). Load at startup; no realm.
- **Application-level resolution:** For **tenant-level** secrets, use the AWS SDK in your tenant config resolver or secret provider: given a realm, look up a secret (e.g. by name `myapp/tenant/<realm>/llm`) and return the value. Cache per realm if needed.

Example (conceptual): Global secret name `myapp/global/jwt`; tenant secret name `myapp/tenant/acme-corp/llm`. Your code or custom ConfigSource fetches the appropriate secret by name; for tenant-level, the name includes the realm and is resolved when handling a request for that tenant.

Chapter 5. Summary

Level	Scope	When resolved	Typical storage
Global	Whole application	Startup (or first use)	Env, `application.properties` placeholders, or a single vault path (HashiCorp Vault / AWS Secrets Manager)
Tenant-level	Per realm	Request time (or when tenant config is loaded)	Vault path per realm (e.g. `secret/…/tenant/<realm>/…`) or tenant config store that references vault/key

For LLM API keys and other tenant-specific credentials, use **tenant-level** secrets and store them in a vault with realm-scoped paths (or a tenant store that references the vault). For JWT, DB, and vault connection credentials, use **global** secrets. See AI Agent Integration — Multi-Tenancy for how tenant agent config (including LLM) fits with per-tenant secrets and the two-level secret model.

Chapter 6. Future: Runtime secret API

The framework currently documents how to store and resolve secrets (env, vault paths, tenant config store) but does not expose a **runtime REST or Java API** for applications to read/write tenant-level secrets. Applications (e.g. psa-app) may implement their own tenant config store (e.g. MongoDB) and REST endpoints for secrets (e.g. LLM API key). A future framework addition could provide a generic tenant-level secret API (e.g. get/put by secret type and realm) backed by a vault or central store, so applications can use it instead of app-specific storage.