Per-Tenant Key Pairs — Design

Version 1.3.0-SNAPSHOT, 2026-02-13T23:56:09Z

Table of Contents

Implementation status: Phase 1 (configurable key paths) and Phase 2 (extractable default keys) are implemented. `TokenUtils` reads `quantum.jwt.private-key-location` and `quantum.jwt.public-key-location` from config, falling back to the classpath defaults (`privateKey.pem` / `publicKey.pem`). Supports both classpath (`classpath:`) and filesystem (`file:`) paths. The default keys are shipped in the optional `quantum-default-keys` JAR (included as an `<optional>true</optional>` dependency); remove this JAR in production and configure external key locations.

Chapter 1. Problem Statement

Today, all tenants share a single RSA key pair (`privateKey.pem` / `publicKey.pem`) that is:

1. **Hardcoded** — `TokenUtils` has the string `"privateKey.pem"` embedded in `generateUserToken()` and `generateRefreshToken()`.
2. **Classpath-only** — Keys are loaded via `Thread.currentThread().getContextClassLoader().getResourceAsStream()`.
3. **Bundled in each app** — Every application module (psa-app, oms-app, idp-app, b2bi-app) ships its own copy of the key files in `src/main/resources`.
4. **Globally cached** — `TokenUtils` uses a static volatile cache with double-checked locking. Once loaded, the key cannot change for the lifetime of the JVM.

This creates several pain points:

- Operators cannot rotate keys without rebuilding and redeploying the application.
- Multi-tenant deployments that require per-tenant signing isolation cannot achieve it.
- The embedded keys in the framework JAR are a convenience that becomes a liability in production.

Chapter 2. Phased Approach

2.1. Phase 1: Configurable Global Key Paths (Implemented)

Make the key file locations configurable via `application.properties` so operators can point to external key files without rebuilding:

```
# Override the private key location (default: classpath:privateKey.pem)
quantum.jwt.private-key-location=file:/opt/keys/signing.pem

# Override the public key location (default: classpath:publicKey.pem)
quantum.jwt.public-key-location=file:/opt/keys/verify.pem
```

Supported path prefixes:

- `classpath:` — Load from the classpath (default behavior, backward compatible)
- `file:` — Load from the filesystem
- No prefix — Treated as a classpath resource name (backward compatible with existing `privateKey.pem` convention)

Changes:

- `TokenUtils` gains a static `configure(String privateKeyLocation, String publicKeyLocation)` method.
- A new CDI bean `TokenUtilsConfigurer` (`@ApplicationScoped`, `@Startup`) reads the config properties and calls `TokenUtils.configure()` at application startup.
- `generateUserToken()` and `generateRefreshToken()` use the configured location instead of the hardcoded `"privateKey.pem"`.
- `readPrivateKey()` and `readPublicKey()` detect the `file:` prefix and use `FileInputStream` when present; otherwise fall back to classpath loading.
- Cache is invalidated when `configure()` is called with different paths.

2.2. Phase 2: Extractable Default Keys (Implemented)

The bundled default key pair (`privateKey.pem` / `publicKey.pem`) is now shipped in a separate, optional JAR module:

- **New module:** `quantum-default-keys` — contains the default PEM files plus a GraalVM `resource-config.json` so the keys are included in native builds.
- `quantum-jwt-provider` declares it as an `<optional>true</optional>` dependency, meaning it is available during framework development/testing but does **not** propagate transitively to consuming applications.

- Applications include `quantum-default-keys` as a runtime dependency for dev/testing convenience.
- In production, operators either:
  - Remove the `quantum-default-keys` dependency from their application POM, or
  - Override the key locations via `quantum.jwt.private-key-location` / `quantum.jwt.public-key-location` (Phase 1 config takes precedence over classpath defaults).

2.3. Phase 3: Per-Tenant Key Pairs in MongoDB (Future)

For deployments requiring per-tenant signing isolation:

2.3.1. Data Model

```
@Entity("tenantKeyPairs")
public class TenantKeyPair extends UnversionedBaseModel {
    /** Tenant realm this key pair belongs to */
    @Indexed
    private String realm;

    /** Key pair identifier (e.g., "primary", "rotation-2024-Q3") */
    @Indexed
    private String keyId;

    /** PEM-encoded RSA public key */
    private String publicKeyPem;

    /**
     * PEM-encoded RSA private key or vault reference.
     * In production, store a vault path (e.g., "vault:secret/data/tenant-a/signing-
key")
     * and resolve the actual key material at runtime via the vault SPI.
     */
    private String privateKeyPemOrVaultRef;

    /** Whether this is the active signing key for the realm */
    private boolean active;

    /** Key algorithm (e.g., "RS256") */
    private String algorithm;

    /** Creation and expiration timestamps for key rotation */
    private Date createdAt;
    private Date expiresAt;
}
```

2.3.2. Key Resolution Service

```
@ApplicationScoped
public class TenantKeyResolver {

    @Inject
    TenantKeyPairRepo keyPairRepo;

    /** Get the active private key for signing tokens in a realm */
    public PrivateKey getSigningKey(String realm) {
        TenantKeyPair kp = keyPairRepo.findActiveByRealm(realm);
        if (kp == null) {
            // Fallback to the global configured key
            return TokenUtils.readPrivateKey(TokenUtils.getPrivateKeyLocation());
        }
        return TokenUtils.decodePrivateKey(kp.getPrivateKeyPemOrVaultRef());
    }

    /** Get all public keys for a realm (supports key rotation overlap) */
    public List<PublicKey> getValidationKeys(String realm) {
        // Return all non-expired keys for the realm
        // During rotation, both old and new keys are valid
    }
}
```

2.3.3. Token Generation Changes

`TokenUtils.generateUserToken()` gains an overload accepting a `PrivateKey` directly, allowing `TenantKeyResolver` to supply the tenant-specific key:

```
public static String generateUserToken(String subject, Set<String> groups,
        long expiresAt, String issuer, PrivateKey signingKey, String keyId) { ... }
```

2.3.4. JWKS Endpoint (Optional)

Expose per-tenant JWKS so external systems can validate tenant-specific tokens:

```
GET /{realm}/.well-known/jwks.json
```

2.3.5. Key Rotation Workflow

1. **Generate** a new key pair for the tenant (via admin API or CLI).
2. **Store** it with `active=false` and a future `createdAt`.
3. **Activate** the new key (set `active=true`); the old key remains valid for verification until its `expiresAt`.

4. **Expire** the old key after a grace period (e.g., max token lifetime).

2.3.6. Security Considerations

- Private keys should be stored in a vault (HashiCorp Vault, AWS KMS) rather than directly in MongoDB. The `privateKeyPemOrVaultRef` field holds a vault reference; `TenantKeyResolver` resolves actual key material at runtime.
- Key pairs should be cached in memory with a TTL to avoid vault/database calls on every token operation.
- Access to the `tenantKeyPairs` collection should be restricted to system-level administrative operations.
- See Secrets and Vault Configuration for the vault integration model.

Chapter 3. Implementation Status

Feature	Status	Notes
Configurable global key paths	Implemented (Phase 1)	`quantum.jwt.private-key-location` / `quantum.jwt.public-key-location`; supports classpath + filesystem
Default keys in optional JAR	Implemented (Phase 2)	`quantum-default-keys` module with GraalVM `resource-config.json`; optional dependency in `quantum-jwt-provider`
Per-tenant key pairs in MongoDB	Design only (Phase 3)	`TenantKeyPair` entity, `TenantKeyResolver`, vault integration
Per-tenant JWKS endpoint	Design only (Phase 3)	Optional; needed when external systems validate tenant-specific tokens
Key rotation workflow	Design only (Phase 3)	Admin API or CLI for key lifecycle management

Chapter 4. Affected Files

4.1. Phase 1

File	Change
`TokenUtils.java`	Add `configure()` method, configurable key locations, `file:` path support, cache invalidation
`TokenUtilsConfigurer.java` (new)	CDI `@Startup` bean that reads config properties and configures `TokenUtils`
`TestTokenUtilsCache.java`	Updated to work with configurable paths

4.2. Phase 2

File	Change
`quantum-default-keys/pom.xml` (new module)	Resource-only JAR with `privateKey.pem` and `publicKey.pem`
`quantum-default-keys/src/main/resources/META-INF/native-image/…/resource-config.json` (new)	GraalVM native-image configuration to include PEM files in native builds
`quantum-jwt-provider/pom.xml`	Added `quantum-default-keys` as `<optional>true</optional>` dependency
`pom.xml` (parent)	Added `quantum-default-keys` to modules list and oss-release profile