Database Migrations and Index Management

Version 1.3.0-SNAPSHOT, 2026-02-13T14:32:21Z

Table of Contents

This guide explains Quantum's MongoDB migration subsystem (quantum-morphia-repos), how migrations are authored and executed, and how to manage indexes. It also documents the REST APIs that trigger migrations and index operations.

Chapter 1. Overview

Quantum uses a simple, versioned change-set mechanism to evolve MongoDB schemas and seed data safely across realms (databases). Key building blocks:

- ChangeSetBean: a CDI bean describing one migration step with metadata (from/to version, priority, etc.) and an execute method.
- ChangeSetBase: convenience base class you can extend; provides logging helpers and optional targeting controls.
- MigrationService: discovers pending change sets, applies them in order within a transaction, records execution, and bumps the DatabaseVersion.
- DatabaseVersion and ChangeSetRecord: stored in Mongo to track current schema version and previously executed change sets.

Chapter 2. Semantic Versioning

Semantic Versioning (SemVer) expresses versions in the form MAJOR.MINOR.PATCH (for example, 1.4.2):

- MAJOR: increment for incompatible/breaking schema changes.
- MINOR: increment for backward-compatible additions (new collections/fields that don't break existing code).
- PATCH: increment for backward-compatible fixes or small adjustments.

Why this matters for migrations: - Ordering: migrations must apply in a deterministic order that reflects real compatibility. SemVer provides a natural ordering and clear intent for authors and reviewers. - Compatibility checks: the application can assert that the current database is "new enough" to run the code safely.

How semver4j is used: - Parsing and validation: version strings are parsed into a SemVer object. Invalid strings fail fast during parsing, ensuring only compliant versions are stored and compared. - Introspection and comparison: the parsed object exposes major/minor/patch components and supports comparisons, enabling safe ordering and "greater than / less than" checks. - Consistent string form: the canonical string is retained for display, logs, and API responses.

How DatabaseVersion leverages SemVer: - Single source of truth: DatabaseVersion stores the canonical SemVer string alongside a parsed SemVer object for logic and comparisons. - Efficient ordering: for fast sorting and tie-breaking, DatabaseVersion also keeps a compact integer encoding of MAJOR.MINOR.PATCH as (major*100) + (minor*10) + patch (e.g., 1.0.3 → 103). This makes numeric comparisons straightforward while still recording the exact SemVer string. - Migration flow: when migrations run, successful execution records the new database version in DatabaseVersion. Startup checks compare the stored version to the required quantum.database.version to prevent the app from running against an older, incompatible schema.

Recommendations: - Always bump MAJOR for breaking data changes, MINOR for additive changes, and PATCH for backward-compatible fixes. - Keep change sets small and target a single to-version per change set to make intent clear. - Use SemVer consistently in getDbFromVersion/getDbToVersion across all change sets so ordering and compatibility checks remain reliable.

Chapter 3. Configuration

The following MicroProfile config properties influence migrations:

- quantum.database.version: target version the application requires (SemVer, e.g., 1.0.3). MigrationService.checkDataBaseVersion compares this to the stored version.
- quantum.database.migration.enabled: feature flag checked by resources/services when running migrations. Default: true.
- quantum.database.migration.changeset.package: package containing change sets (CDI still discovers beans via type, but this property documents the intended package).
- quantum.realmConfig.systemRealm, quantum.realmConfig.defaultRealm, quantum.realmConfig.testRealm: well-known realms used by MigrationResource when running migrations across environments. On startup, MigrationService checks whether the *system realm* database exists and is initialized; if missing or uninitialized, it runs all pending change sets and applies indexes (creating all entity collections, including credentials). Set `quantum.realmConfig.systemRealm` to your application's system database name (e.g. `system-psa-com`) in `application.properties` so the correct database is created and initialized when dropped or empty.
- quantum.defaultSystemPassword: initial password for the system user credential when created by the AddSystemUserCredential change set (default: `test123456`). Set in `application.properties`; the system user (quantum.realmConfig.systemUserId) is created in the credentials collection so login with that user works.
- quantum.database.version: target schema version (e.g. 1.0.4). Must be at least 1.0.4 for the system user credential to be created during migration.

Chapter 4. How change sets are discovered and executed

- Discovery: MigrationService#getAllChangeSetBeans locates all CDI beans implementing ChangeSetBean.
- Ordering: change sets are sorted by dbToVersionInt, then by priority (ascending). That ensures lower target versions apply before higher ones; priority resolves ties.
- Pending selection: For the target realm, MigrationService#getAllPendingChangeSetBeans considers a change set pending if either (a) it has never run, (b) the bean's changeSetVersion is greater than the last recorded one, or (c) the bean's checksum is non-null and differs from the last recorded checksum in ChangeSetRecord. It also compares each change set's dbToVersion against the stored DatabaseVersion.
- Locking: A distributed lock (Mongo-backed Sherlock) is acquired per realm before applying change sets to prevent concurrent execution.
- Transactions: Each change set runs within a MorphiaSession transaction; on success the change is recorded in ChangeSetRecord and DatabaseVersion is advanced (if higher). On failure the transaction is aborted and the error returned.
- Realms: Migrations run per realm (Mongo database). A change set can optionally be restricted to certain database names or even override the realm it executes against (see below).

Chapter 5. Authoring a change set

Implement ChangeSetBean; most change sets extend ChangeSetBase.

Required metadata methods:

- getId(): a string id for human tracking (e.g., 00003)
- getDbFromVersion() / getDbFromVersionInt(): previous version you are migrating from (SemVer and an int like 102 for 1.0.2)
- getDbToVersion() / getDbToVersionInt(): target version after running this change (SemVer and int)
- getPriority(): integer priority when multiple change sets have same toVersion
- getAuthor(), getName(), getDescription(), getScope(): informational fields recorded in ChangeSetRecord
- getChangeSetVersion() [optional]: an integer you bump when the definition/logic of this change set evolves but its target database version does not. Defaults to 1.
- getChecksum() [optional]: a stable checksum string representing the content/logic of the change set. If provided and it changes, the change set will re-run even if versions did not change.

Execution method:

- void execute(MorphiaSession session, MongoClient mongoClient, MultiEmitter<? super String> emitter)
- Perform your data/index changes using the provided session (transaction).
- Use emitter.emit("message") to stream log lines back to SSE clients.

Optional targeting controls (provided by ChangeSetBase):

- boolean isOverrideDatabase(): return true to execute against a specific database instead of the requested realm.
- String getOverrideDatabaseName(): the concrete database name to use when overriding.
- Set<String> getApplicableDatabases(): return a set of database names to which this change set should apply. Return null or an empty set to allow all.

Logging helper:

- ChangeSetBase.log(String, MultiEmitter) emits to both Quarkus log and the SSE stream.

Chapter 6. Example change sets in the framework

Package: com.e2eq.framework.model.persistent.morphia.changesets

- InitializeDatabase
- Seeds foundational data in a new realm: counters (e.g., accountNumber), system Organization and Account, initial Rule and Policy scaffolding, default user profiles and security model. Uses EnvConfigUtils and SecurityUtils to derive system DataDomain and defaults.
- AddAnonymousSecurityRules
- Adds a defaultAnonymousPolicy with an allow rule for unauthenticated actions such as registration and contact-us in the website area.
- AddRealms
- Creates the system and default Realm records based on configuration, if missing.

These are typical examples of idempotent change sets that can be safely re-evaluated.

Chapter 7. REST APIs to trigger migrations (MigrationResource)

Base path: /system/migration

Security: Most endpoints require admin role; dbversion is PermitAll for introspection.

- GET /system/migration/dbversion/{realm}
- Returns the current DatabaseVersion document for the given realm, or 404 if not found.
- Example: curl -s http://localhost:8080/system/migration/dbversion/system-com
- POST /system/migration/indexes/applyIndexes/{realm}
- Admin only. Calls MigrationService.applyIndexes(realm) which invokes Morphia Datastore.applyIndexes() for all mapped entities. Use this after adding @Indexed annotations.
- Example: curl -X POST -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/indexes/applyIndexes/system-com
- POST /system/migration/indexes/dropAllIndexes/{realm}
- Admin only. Drops all indexes on all mapped collections in the realm. Useful before re-creation or when changing index definitions.
- Example: curl -X POST -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/indexes/dropAllIndexes/system-com
- POST /system/migration/initialize/{realm}
- Admin only. Server-Sent Events (SSE) stream that executes all pending change sets for the specific realm.
- Example (note -N to keep connection open): curl -N -X POST -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/initialize/system-com
- GET /system/migration/start
- Admin only. SSE stream that runs pending change sets across test, system, and default realms from configuration.
- Example: curl -N -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/start
- GET /system/migration/start/{realm}
- Admin only. SSE for a specific realm.
- Example: curl -N -H "Authorization: Bearer $TOKEN" http://localhost:8080/system/migration/start/my-realm

SSE responses stream human-readable messages produced by MigrationService and your change sets. The connection ends with "Task completed" or an error message.

Chapter 8. Per-entity index management (BaseResource)

Every entity resource that extends BaseResource<T, R extends BaseMorphiaRepo<T>> exposes a convenience endpoint to (re)create indexes for a single collection in a realm.

- POST <entity-resource-base-path>/indexes/ensureIndexes/{realm}?collectionName=<collection>
- Admin only. Invokes R.ensureIndexes(realm, collectionName).
- Use this when you want to reapply indexes for one collection without touching others.
- Example (assuming a ProductResource at /products): curl -X POST -H "Authorization: Bearer $TOKEN" \ "http://localhost:8080/products/indexes/ensureIndexes/system-com?collectionName=product"

Chapter 9. Global index management (MigrationService)

MigrationService also exposes programmatic index utilities used by the MigrationResource endpoints:

- applyIndexes(realm): calls Morphia Datastore.applyIndexes() for the realm.
- dropAllIndexes(realm): iterates mapped entities and drops indexes on each underlying collection.

Chapter 10. Validating versions at startup

- MigrationService.checkDataBaseVersion() compares the stored DatabaseVersion in each well-known realm to quantum.database.version and throws a DatabaseMigrationException when lower than required. This prevents the app from running against an incompatible schema.
- MigrationService.checkInitialized(realm) is a convenience that asserts DatabaseVersion exists and is >= required version; helpful for preflight checks.

Chapter 11. Notes and best practices

- Make change sets idempotent: Always check for existing records before creating/updating indexes or documents.
- Use SemVer consistently for from/to versions. The framework computes an integer form (e.g., 1.0.3 → 103) for ordering.
- Prefer small, focused change sets with clear descriptions and authorship.
- Use the MultiEmitter in execute(…) to provide progress to operators consuming the SSE endpoint.
- Apply new indexes with applyIndexes after deploying models with new @Indexed annotations; optionally dropAllIndexes then applyIndexes when changing index definitions across the board.
- Limit scope: use getApplicableDatabases() to constrain execution to specific databases, or isOverrideDatabase/getOverrideDatabaseName to target a different database when appropriate.

11.1. Checksum vs. from/to versions

ChangeSetBean introduces two optional mechanisms that influence whether a change set runs again after it has already executed:

- changeSetVersion (int): A manual counter you increment when you intentionally want the same change set (same to-version) to run again because its logic changed or needs to reapply. Defaults to 1.
- checksum (String): A content-derived fingerprint of the change set's implementation. If provided, the framework will re-run the change set whenever the checksum differs from the last recorded value, even if versions and changeSetVersion are unchanged.

How they complement getDbFromVersion/getDbToVersion:

- getDbFromVersion/getDbToVersion define the database version boundary the change set moves the realm across. They control ordering and whether the database version should advance on success.
- changeSetVersion/checksum control repeatability of a change set at a fixed to-version. They do not change ordering; they only indicate that the change set should be applied again.

Typical usage patterns:

- Minor logic refactor that should re-apply: bump changeSetVersion, or update checksum if you compute it from code/resources.
- Data or index definition tweaked without a version bump: provide getChecksum() that reflects the relevant definitions (e.g., hash of DDL/index specs or embedded dataset), so the system auto-detects changes and re-runs.
- Actual schema version change: bump getDbToVersion (and possibly from) as usual; you may leave changeSetVersion and checksum alone.

Recording and behavior in ChangeSetRecord:

- On each execution, MigrationService stores changeSetVersion and checksum alongside from/to versions.
- A change set is considered pending if no record exists, or the bean's changeSetVersion is greater than the recorded one, or the bean's checksum is non-null and differs from the recorded checksum.
- After a successful run, DatabaseVersion is updated to dbToVersion if it is higher than current.

Caveats:

- If you return null from getChecksum(), only first-run and changeSetVersion increases can trigger re-execution.
- Ensure your checksum is stable across processes and deterministic for the same logic; do not include timestamps or environment-specific data.
- When migrating large datasets, prefer idempotent logic so repeated runs are safe even if checksum forces a re-run.