Seed packs and declarative tenant seeding

Version 1.2.4-SNAPSHOT, 2026-01-31T06:27:21Z

Table of Contents

Quantum 1.2 introduces a seed-pack subsystem that lets applications publish versioned baseline content without hard-coding values in `ChangeSet` beans or maintaining a separate "seed" tenant.

Chapter 1. Introduction

1.1. The problem

In multi-tenant SaaS platforms, every new tenant must start with a known-good baseline of data: code lists, roles, default settings, reference values, and sometimes product- or region-specific content. Traditionally this baseline is scattered across ad-hoc SQL/Mongo scripts, hand-written bootstrap code, or a "template" tenant that is copied forward. These approaches are hard to version, review, test, and repeat reliably across environments.

Compounding the issue, tenants evolve over time. As modules are upgraded, their baseline content must be updated too. Without a disciplined mechanism, teams risk drift between environments and tenants, brittle migrations, and non-idempotent provisioning that causes duplicates or corruption.

1.2. Why this needs to be solved

- Operational consistency: Provisioning should be predictable, repeatable, and safe to re-run.
- Developer velocity: Changes to baseline data should be reviewed like code and travel with the module that owns them.
- Compliance and audit: You need to know exactly which version of seed content was applied to which tenant and when.
- Composability: Different product editions or SKUs need different combinations of baseline content without forked scripts.

1.3. How seed packs solve it

Seed packs provide a declarative, versioned, and composable way to describe tenant baseline data: - A manifest (manifest.yaml) declares datasets, natural keys, transforms, required indexes, and optional includes/archetypes. - Datasets point to JSON/NDJSON files that are upserted using natural-key filters, making runs idempotent. - Transforms inject tenant/realm identifiers and can rewrite references deterministically. - Includes compose other packs with exact versions or semantic version ranges, enabling dependency management. - Archetypes bundle a named set of packs to represent product tiers or verticals. - A registry records checksums per dataset so unchanged data is skipped on subsequent runs.

Together, these features make seeding safe, observable, and maintainable across development, test, and production.

The next section expands on why seed packs are beneficial and how to use them effectively.

Chapter 2. Why seed packs?

- **Versioned + reviewable**: seed packs are plain files (YAML + JSON/NDJSON) that live next to your module code. Pull requests show exactly which records changed.
- **Composable**: packs can depend on other packs and expose named *archetypes* for different product editions or verticals.
- **Pluggable sources**: load packs from the filesystem, object storage, or even a curated seed database by providing a custom `SeedSource`.
- **Tenant-aware**: transforms inject tenant identifiers and remap references before persisting.
- **Idempotent**: a `SeedRegistry` tracks checksums per dataset so provisioning can be re-run safely.

Chapter 3. High-level flow

1. `SeedLoader` discovers manifests via the configured `SeedSource` implementations (for example the provided `FileSeedSource`).
2. A manifest (`manifest.yaml`) declares datasets, required indexes, transforms, optional includes, and archetypes.
3. During provisioning a migration invokes `SeedLoader.apply(…)` with the packs (or archetype) that should be materialised for the tenant.
4. Records are parsed, transformed, and upserted through a `SeedRepository` implementation. The default `MongoSeedRepository` writes to the tenant realm using natural-key filters.
5. The `SeedRegistry` (backed by `_seed_registry` via `MongoSeedRegistry`) records the checksum so unchanged datasets are skipped on later runs.

Chapter 4. Manifest quick reference

```
seedPack: logistics-core
version: 1.4.2
includes:
  - accounting-base@^1.1

datasets:
  - collection: codeLists
    file: datasets/codelists.ndjson
    naturalKey: [codeListName, code]
    upsert: true
    requiredIndexes:
      - name: uk_codeLists_name_code
        unique: true
        keys:
          codeListName: 1
          code: 1
    transforms:
      - type: tenantSubstitution
        config:
          tenantField: tenantId
          orgField: orgRefName
          ownerField: ownerId
          realmField: realmId

archetypes:
  - name: FulfillmentPlus
    includes:
      - logistics-core@^1.4
      - shipping-defaults@~2
```

Chapter 5. Programmatic usage

```
SeedLoader loader = SeedLoader.builder()
        .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
        .seedRepository(new MongoSeedRepository(mongoClient))
        .seedRegistry(new MongoSeedRegistry(mongoClient))
        .build();

SeedContext ctx = SeedContext.builder(realmId)
        .tenantId(tenantId)
        .orgRefName(orgRef)
        .accountId(accountId)
        .ownerId(ownerId)
        .build();

loader.apply(List.of(
        SeedPackRef.range("logistics-core", "^1.4"),
        SeedPackRef.of("oms-defaults")
), ctx);
```

Callers can also use `loader.applyArchetype("FulfillmentPlus", ctx)` to resolve an archetype defined in any manifest.

Chapter 6. Extensibility hooks

- Implement `SeedSource` to load manifests from custom storage (S3, Git, curated seed DB…).
- Register additional `SeedTransformFactory` instances with the builder to support bespoke transformations (for example JMESPath projections or deterministic ObjectId mapping).
- Swap in a different `SeedRepository`/`SeedRegistry` to write to alternative datastores or change the idempotency policy.

Chapter 7. Operational tips

- Validate manifests in CI by running the loader against a disposable database.
- Keep seed pack versions aligned with module versions so upgrade paths are clear.
- Derive any ObjectIds deterministically from natural keys inside a transform so data can be re-applied without collisions.
- Use archetypes to model product tiers and optional modules: `TenantProvisioningService` can decide which archetype(s) to apply based on SKU.

Chapter 8. Primary scenarios

1. Initial tenant provisioning
    - Apply one or more seed packs to bootstrap a brand-new tenant (realm) with baseline code lists, roles, and default settings.
    - Use SeedPackRef.of("pack-name") or SeedPackRef.range("pack-name", "^1.4") to control versions.
2. Updating a module to a new version
    - Publish a new seed pack version (e.g., logistics-core 1.5.0) with incremental dataset changes.
    - Re-run loader.apply(…) for the same tenant; unchanged datasets are skipped via _seed_registry, modified datasets are re-applied.
3. Idempotent re-apply during deployments
    - Safe to invoke on every startup/migration. Upserts are driven by naturalKey and upsert: true.
    - Keep natural keys stable; derive surrogate IDs deterministically in a transform if needed.
4. Selecting product tiers with archetypes
    - Define archetypes in a manifest to bundle multiple seed packs under a named edition.
    - Call loader.applyArchetype("FulfillmentPlus", ctx) to materialize the predefined stack for a tenant.
5. Composing packs with includes
    - Use includes to depend on base packs (e.g., accounting-base@^1.1) and extend with your own datasets.
    - Includes support exact (=1.2.3) and range (e.g., ^1.4, ~2) selectors via SeedPackRef.parse("name@spec").
6. Partial refresh of specific datasets
    - You can split large packs into multiple datasets and re-apply only the packs you want by passing a smaller list to loader.apply(…).
7. Testing seed packs
    - Add an integration test similar to SeedLoaderIntegrationTest that seeds into an ephemeral MongoDB and asserts collection state and _seed_registry entries.

Chapter 9. Explicit examples

9.1. Example 1: Minimal manifest and NDJSON

```yaml
seedPack: demo-seed
version: 1.0.0

datasets:
  - collection: codeLists
    file: datasets/codeLists.ndjson
    naturalKey: [ code ]
    upsert: true
    requiredIndexes:
      - name: uk_codeLists_code
        unique: true
        keys:
          code: 1
    transforms:
      - type: tenantSubstitution
        config:
          tenantField: tenantId
          orgField: orgRefName
          accountField: accountId
          ownerField: ownerId
          realmField: realmId
```

Example NDJSON (datasets/codeLists.ndjson):

```
{"code": "NEW", "label": "New"}
{"code": "CLOSED", "label": "Closed"}
```

9.2. Example 2: Applying packs in code

```
SeedLoader loader = SeedLoader.builder()
        .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
        .seedRepository(new MongoSeedRepository(mongoClient))
        .seedRegistry(new MongoSeedRegistry(mongoClient))
        .build();

SeedContext ctx = SeedContext.builder("my-realm")
        .tenantId("tenant-123")
        .orgRefName("tenant-123")
        .accountId("acct-123")
        .ownerId("owner-123")
        .build();
```

```
loader.apply(List.of(
        SeedPackRef.of("demo-seed"),
        SeedPackRef.range("logistics-core", "^1.4")
), ctx);
```

9.3. Example 3: Using an archetype

```
archetypes:
  - name: FulfillmentPlus
    includes:
      - logistics-core@^1.4
      - shipping-defaults@~2
```

Apply programmatically:

```
loader.applyArchetype("FulfillmentPlus", ctx);
```

9.4. Example 4: Exact version and includes in a manifest

```
seedPack: shipping-defaults
version: 2.3.0
includes:
  - accounting-base@=1.1.2
  - logistics-core@^1.5

datasets:
  - collection: shippingMethods
    file: datasets/methods.json
    naturalKey: [ code ]
```

Chapter 10. Troubleshooting

- Manifest parsing errors: Confirm manifest.yaml keys match SeedPackManifest fields; boolean flags like upsert and unique must be proper booleans.
- Duplicate key or unique index violations: Check naturalKey and requiredIndexes; ensure transforms don’t change key fields inconsistently.
- Nothing changes on re-run: The _seed_registry may have recorded the same checksum; bump version or change dataset content.
- File resolution issues: Ensure FileSeedSource base path points to the correct seed-packs directory and file names match.

Chapter 11. How seeds are applied automatically at startup

The framework now applies seed packs via a dedicated SeedStartupRunner, independent of schema migrations. This runner discovers and applies the latest version of each seed pack for important realms (system/default/test) on application startup.

Key points: - Discovery: SeedStartupRunner constructs a `SeedLoader` with a `FileSeedSource` pointing at the configured seed root. Set `quantum.seed.root` (for tests we default to `src/main/resources/seed-packs`). The source walks the directory tree and locates every `manifest.yaml` file. - Selection: For each discovered seed pack name, the runner selects the latest semantic version and builds `SeedPackRef.exact(name, version)` for application. - Execution: The runner builds a `SeedContext` for the target realm and calls `loader.apply(refs, context)`. Indexes declared in the manifest are created before data is upserted. - Idempotency + repeatable: The `MongoSeedRegistry` stores a checksum per dataset in the realm's `_seed_registry` collection. If the checksum matches on a later run, the dataset is skipped; if it changes, the dataset is re-applied. - Concurrency safety: The runner uses a Sherlock distributed lock per realm to prevent concurrent execution across nodes.

Configuration snippet:

```
# test profile uses a local seed root
quantum.seed.root=src/test/resources/seed-packs
# control seed runner behavior
quantum.seeds.enabled=true
quantum.seeds.apply.on-startup=true
```

11.1. Configuring which realms receive seeds on startup

By default, `SeedStartupRunner` applies seeds to the system, default, and test realms. You can customize this behavior using a combination of:

1. **Configuration property** (`quantum.seed-pack.apply.realms`): A comma-separated list of realm names to seed on startup.
2. **Realm flag** (`applySeedsOnStartup`): A boolean field on the Realm model/entity that, when set to `true`, includes that realm for automatic seeding.

The startup runner unions both sources: realms from the config property plus realms with the `applySeedsOnStartup` flag enabled.

Configuration examples:

```
# Explicit list of realms to seed
quantum.seed-pack.apply.realms=system-com,production-com,staging-com
```

```
# Disable config-based realms (only flag-based realms will be seeded)
quantum.seed-pack.apply.realms=none
```

Setting the flag on a Realm entity:

```
Realm realm = Realm.builder()
    .emailDomain("acme.com")
    .databaseName("acme-com")
    .domainContext(domainContext)
    .applySeedsOnStartup(true)  // This realm will receive seeds on startup
    .build();
realmRepo.save(realm);
```

Or update an existing realm:

```
Optional<Realm> realmOpt = realmRepo.findByDatabaseName("acme-com", true, systemRealm
);
if (realmOpt.isPresent()) {
    Realm realm = realmOpt.get();
    realm.setApplySeedsOnStartup(true);
    realmRepo.save(realm);
}
```

Resolution logic:

- If `quantum.seed-pack.apply.realms` is configured with realm names (not "none" or empty), those realms are added to the list.
- The runner then queries the Realm collection in the system realm for all realms with `applySeedsOnStartup=true` and adds those to the list.
- The union of both sources (deduped) determines which realms receive seeds.
- If neither is configured, the default behavior applies: system, default, and test realms.

Notes:

- The Realm collection is always stored in the system realm (e.g., `system-com`).
- The `applySeedsOnStartup` flag defaults to `false` for all realms.
- This allows dynamic realm configuration without redeploying the application—just update the Realm entity in the database.

Chapter 12. Transforms in depth

Transforms are small, composable functions that shape each dataset record just before it is written to the database. They let you keep dataset files generic and inject environment/tenant specifics or perform repeatable rewrites at apply time.

What a transform gets and returns: - Input: the current record (a Map), the SeedContext, and the Dataset definition - Output: the next record (Map) to be passed to the rest of the pipeline; return null or an empty map to drop the record

Where transforms are declared (manifest):

```
datasets:
  - collection: codeLists
    file: datasets/codeLists.ndjson
    naturalKey: [ codeListName, code ]
    upsert: true
    transforms:
      - type: tenantSubstitution
        config:
          tenantField: tenantId
          orgField: orgRefName
          ownerField: ownerId
          accountField: accountNum
          realmField: realmId
      # Additional transforms can be added here and will execute in order
```

Execution semantics: - Ordering: transforms are executed top-to-bottom for each record - Short-circuit: if any transform returns null or an empty map, the record is skipped and no write occurs - Overwrite rules: a transform can set or overwrite fields on the record; when upsert=true, the final transformed record replaces the existing one matched by naturalKey - Interaction with naturalKey and indexes: transforms run before naturalKey validation and index creation; do not remove fields listed in naturalKey, otherwise an error will be thrown during write

Built-in transform types:

- tenantSubstitution: Injects identifiers from SeedContext into the record's dataDomain.* fields (tenant/org/owner/account) and sets realmId at the record root. Config keys in the manifest (defaults target dataDomain.*):
- tenantField: key inside dataDomain to receive tenantId (default: tenantId)
- orgField: key inside dataDomain to receive orgRefName (default: orgRefName)
- ownerField: key inside dataDomain to receive ownerId (default: ownerId)
- accountField: key inside dataDomain to receive account number (default: accountNum)
- realmField: root-level field to receive realmId (default: realmId)
- stringInterpolation: Performs variable substitution on string fields using `{variableName}` syntax. Walks through all string fields (including nested maps and lists) and replaces variable

references with resolved values. Config keys:

- fields: optional list of field names to interpolate; if omitted, all string fields are processed
- failOnMissing: if true, throws an exception when a variable cannot be resolved; if false (default), unresolved variables are left as-is

Built-in variables available from SeedContext: - `{realm}` or `{realmId}`: the realm/database identifier - `{tenantId}`: the tenant identifier - `{orgRefName}`: the organization reference name - `{accountId}`: the account identifier - `{ownerId}`: the owner identifier

Example manifest usage:

```
transforms:
  - type: stringInterpolation
    config:
      fields: ["runAsUserId", "description"]
      failOnMissing: false
```

Example dataset with variables:

```
{"refName":"adminRule","runAsUserId":"admin@{tenantId}","realm":"{realm}"}
{"refName":"systemRule","config":{"owner":"{ownerId}","account":"{accountId}"}}
```

After interpolation with tenantId="acme-corp" and realm="acme-realm":

```
{"refName":"adminRule","runAsUserId":"admin@acme-corp","realm":"acme-realm"}
{"refName":"systemRule","config":{"owner":"owner-123","account":"account-456"}}
```

Notes and guarantees: - Optional values missing from SeedContext are simply omitted; existing record values are preserved unless you target the same field - Transforms operate on in-memory maps and cannot perform I/O by default; keep them deterministic so re-runs are idempotent - Compose multiple transforms when needed (for example: first tenantSubstitution, then a custom id computation) - To add new transform types, implement SeedTransformFactory and register it during SeedLoader.builder() with registerTransformFactory("myType", new MyFactory()); then declare - type: myType in the manifest - To add custom variables for stringInterpolation, implement SeedVariableResolver and register it with the builder using addVariableResolver(resolver) or make it a CDI bean for automatic discovery

When to use transforms (and why): - Injecting tenant/realm identity: keep datasets source-controlled and generic; inject tenant IDs at apply time (tenantSubstitution) - Deterministic IDs: derive _id or other surrogate keys from naturalKey so upserts remain stable across environments - Normalization and defaults: add missing fields, convert formats, enforce enums before write - Reference remapping: translate human-readable codes in the dataset into datastore-specific identifiers (ObjectIds, UUIDs) in a repeatable way

Practical examples

1) Tenant identity injection (built-in) Manifest snippet:

```
transforms:
  - type: tenantSubstitution
    config:
      tenantField: tenantId
      orgField: orgRefName
      ownerField: ownerId
      accountField: accountId
      realmField: realmId
```

Why: keep codeLists.ndjson portable across tenants; provisioning injects the right IDs based on SeedContext.

2) Deterministic _id from natural key (custom) - Goal: ensure stable MongoDB _id across re-applies and environments, derived from codeListName+code - Approach: implement a custom transform that computes a SHA-1/MD5 hash (or any deterministic function) and sets _id

Java registration:

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .registerTransformFactory("deterministicId", new DeterministicIdTransform.Factory
())
    .build();
```

Manifest usage:

```
transforms:
  - type: tenantSubstitution
  - type: deterministicId
    config:
      sourceFields: [ codeListName, code ]
      targetField: _id
      algorithm: sha1
```

Why: makes upserts resilient and allows cross-environment joins by a stable key.

3) Foreign key remapping by code (custom) - Goal: dataset uses human-readable statusCode; transform maps it to a canonical statusId - Approach: custom transform with an in-memory map or deterministic derivation

Manifest usage:

```
transforms:
```

```
  - type: mapCode
    config:
      field: statusCode
      target: statusId
      mapping:
        NEW: 100
        CLOSED: 900
```

Why: keeps datasets human-friendly while persisting efficient identifiers.

4) Defaulting and sanitization (custom) - Goal: ensure missing fields get defaults and strings are trimmed/lowercased - Approach: simple custom transform that fills defaults and cleans values

Manifest usage:

```
transforms:
  - type: defaults
    config:
      defaults:
        isActive: true
        locale: en_US
  - type: sanitize
    config:
      trim: [ label ]
      lowercase: [ email ]
```

Why: enforces consistency without editing large datasets.

Testing transforms: - Add integration tests that seed into an ephemeral DB and assert both record shape and _seed_registry entries - For custom transforms, add focused unit tests for edge cases (missing fields, nulls, unexpected types)

12.1. Creating your own transforms (example: DropIfTransform)

Custom transforms let you implement project-specific shaping logic. You implement two small interfaces and register the type on the SeedLoader builder. Below we walk through a simple "drop the record if a field equals a value" transform used in tests, called DropIfTransform.

Overview of the SPI: - SeedTransform: executes per-record and can return a new map (continue) or null/empty (drop this record). - SeedTransformFactory: builds a SeedTransform instance from the manifest's Transform definition (provides access to type and config map).

Minimal interfaces (simplified for clarity):

```
public interface SeedTransform {
    Map<String, Object> apply(Map<String, Object> record,
```

```
                              SeedContext context,
                              SeedPackManifest.Dataset dataset);
}

public interface SeedTransformFactory {
    SeedTransform create(SeedPackManifest.Transform transformDefinition);
}
```

Implementation: DropIfTransform - Behavior: if record[field] equals a configured value, return null to short-circuit the pipeline and skip writing the record; otherwise, pass the record through unchanged.

```
package com.example.seed.transforms;

import com.e2eq.framework.service.seed.*;
import java.util.Map;
import java.util.Objects;

public final class DropIfTransform implements SeedTransform {
  private final String field;
  private final String equalsValue;

  public DropIfTransform(String field, String equalsValue) {
    this.field = field;
    this.equalsValue = equalsValue;
  }

  @Override
  public Map<String, Object> apply(Map<String, Object> record, SeedContext context,
SeedPackManifest.Dataset dataset) {
    Object v = record.get(field);
    if (Objects.equals(Objects.toString(v, null), equalsValue)) {
      return null; // short-circuit: drop this record
    }
    return record;
  }

  public static final class Factory implements SeedTransformFactory {
    @Override
    public SeedTransform create(SeedPackManifest.Transform transformDefinition) {
      Map<String, Object> cfg = transformDefinition.getConfig();
      String field = Objects.toString(cfg.get("field"), null);
      String eq = Objects.toString(cfg.get("equals"), null);
      return new DropIfTransform(field, eq);
    }
  }
}
```

Registration on the SeedLoader builder:

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .registerTransformFactory("dropIf", new DropIfTransform.Factory())
    .build();
```

Manifest usage:

```
datasets:
  - collection: codeLists
    file: datasets/codeLists.ndjson
    naturalKey: [ code ]
    upsert: true
    transforms:
      - type: dropIf
        config:
          field: status
          equals: CLOSED
```

Notes and tips: - Validation: your factory should validate required config keys and fail fast with a clear error if missing/invalid. - Determinism: keep transforms pure and deterministic (no I/O) so seeding remains idempotent. - Short-circuit: returning null or an empty map drops the record; otherwise, the next transform in the list will receive the (possibly mutated) map. - Composition: you can chain several transforms; for example, first dropIf, then tenantSubstitution, then a custom deterministicId. - Packaging: test-only transforms can live under test sources; production transforms should be in main sources and registered where you construct the SeedLoader (for example, in a ChangeSet or a provisioning service).

12.2. Custom variable resolvers for stringInterpolation

The `stringInterpolation` transform supports custom variable resolution via the `SeedVariableResolver` SPI. This allows you to define application-specific variables that can be used in seed data alongside the built-in context variables.

SeedVariableResolver interface:

```
public interface SeedVariableResolver {
    /**
     * Attempts to resolve a variable by name.
     * @param variableName the name of the variable (without braces)
     * @param context the seed context
     * @return Optional containing the resolved value, or empty if not handled
     */
    Optional<String> resolve(String variableName, SeedContext context);

    /**
```

```
     * Returns the priority of this resolver. Higher priority = consulted first.
     * Default is 0. Built-in context resolver has priority -100 (lowest).
     */
    default int priority() {
        return 0;
    }
}
```

Implementation example (environment-based resolver):

```
package com.example.seed;

import com.e2eq.framework.service.seed.*;
import jakarta.enterprise.context.ApplicationScoped;
import java.util.Optional;

@ApplicationScoped  // CDI auto-discovery
public class EnvironmentVariableResolver implements SeedVariableResolver {

    @Override
    public Optional<String> resolve(String variableName, SeedContext context) {
        // Resolve variables prefixed with "env."
        if (variableName.startsWith("env.")) {
            String envName = variableName.substring(4);
            return Optional.ofNullable(System.getenv(envName));
        }
        return Optional.empty();
    }

    @Override
    public int priority() {
        return 10; // Higher than context resolver (-100)
    }
}
```

With the above resolver, you can use environment variables in seed data:

```
{"apiEndpoint": "{env.API_BASE_URL}", "adminEmail": "admin@{tenantId}"}
```

Manual registration (alternative to CDI):

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .addVariableResolver(new EnvironmentVariableResolver())
    .build();
```

Priority and override behavior: - Resolvers are consulted in priority order (highest first) - The first resolver that returns a non-empty Optional wins - Custom resolvers can override built-in context values by using a higher priority - The built-in `SeedContextVariableResolver` has priority -100, so any resolver with priority >= 0 will be consulted first

Example: overriding a context variable:

```
@ApplicationScoped
public class TenantOverrideResolver implements SeedVariableResolver {
    @Override
    public Optional<String> resolve(String variableName, SeedContext context) {
        if ("tenantId".equals(variableName)) {
            // Override tenantId with a computed value
            return Optional.of("custom-" + context.getTenantId().orElse("unknown"));
        }
        return Optional.empty();
    }

    @Override
    public int priority() {
        return 100; // Higher than context resolver
    }
}
```

12.3. Seed record listeners (reacting to seed data)

The `SeedRecordListener` SPI allows application components to react when seed records are applied to the database. This is useful for synchronizing application state with seeded data, such as registering scheduled jobs, updating caches, or triggering other side effects.

The framework provides only the SPI interface. Applications must implement their own listeners for collections they care about. Listeners run within the security context established by the seed operation, so they have access to the tenant/realm context.

SeedRecordListener interface:

```
public interface SeedRecordListener {
    /**
     * Determines whether this listener should receive callbacks for the given
collection.
     */
    boolean appliesTo(String collection, SeedContext context);

    /**
     * Called after seed records have been successfully applied to the database.
     */
    void onRecordsApplied(SeedRecordEvent event);
```

```
    /**
     * Returns the priority of this listener. Higher priority = invoked first.
     */
    default int priority() { return 0; }

    /**
     * If true, listener runs asynchronously in a separate thread.
     */
    default boolean async() { return false; }
}
```

12.3.1. Application-level implementation example

The following example shows how an application can implement a listener to synchronize job configurations when seeds are applied. This code would live in the application, not the framework:

```
@ApplicationScoped
public class JobRunnerSeedListener implements SeedRecordListener {

    @Inject
    JobScheduler jobScheduler;

    @Override
    public boolean appliesTo(String collection, SeedContext context) {
        return "autoPublishConfig".equals(collection) ||
               "scheduledJobs".equals(collection);
    }

    @Override
    public void onRecordsApplied(SeedRecordEvent event) {
        Log.infof("Syncing %d job config records from seed to JobRunner",
                event.getRecordCount());
        for (Map<String, Object> record : event.getRecords()) {
            jobScheduler.registerOrUpdate(record);
        }
    }

    @Override
    public int priority() {
        return 100; // High priority to run early
    }
}
```

With the above listener (implemented in your application), whenever the `autoPublishConfig` collection receives seed data, the JobRunner will automatically sync its internal state with the database.

12.3.2. Security context

Listeners run within the security context established by the seed operation (`SeedStartupRunner` or `SeedLoaderService`). This means:

- The `SecurityContext.getPrincipalContext()` is available with tenant/realm information
- Any repository operations or service calls will execute with appropriate permissions
- Listeners can access tenant-specific data and perform tenant-aware operations

Example accessing security context in a listener:

```
@Override
public void onRecordsApplied(SeedRecordEvent event) {
    Optional<PrincipalContext> principal = SecurityContext.getPrincipalContext();
    if (principal.isPresent()) {
        String tenantId = principal.get().getDataDomain().getTenantId();
        String userId = principal.get().getUserId();
        Log.infof("Processing seeds for tenant %s as user %s", tenantId, userId);
    }
    // Process records with proper tenant context...
}
```

12.3.3. SeedRecordEvent properties

- `getCollection()`: The collection/entity name
- `getSeedPack()`: The seed pack that provided the data
- `getVersion()`: The seed pack version
- `getContext()`: The SeedContext with realm/tenant info
- `getRealm()`: Shortcut for `getContext().getRealm()`
- `getRecords()`: List of applied records (transformed, final form)
- `isUpsert()`: Whether upsert (true) or insert-only (false)
- `getRecordCount()`: Number of records applied

12.3.4. Registration

CDI auto-discovery (recommended):

```
// Just annotate your listener with @ApplicationScoped
// It will be automatically discovered and registered
@ApplicationScoped
public class MyAppSeedListener implements SeedRecordListener { ... }
```

Manual registration (alternative):

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .addRecordListener(new MyAppSeedListener())
    .build();
```

12.3.5. Best practices

- Listeners are invoked after records have been persisted to the database
- Listener exceptions are caught and logged; they don't interrupt the seeding process
- Async listeners (`async() = true`) run in separate threads and don't block seeding
- CDI-discovered listeners are automatically registered when using `SeedLoaderService`
- Keep listener logic fast and idempotent since seeds may be re-applied
- Use the security context to ensure tenant-aware operations

Chapter 13. Test walkthrough: SeedLoaderIntegrationTest

The `SeedLoaderIntegrationTest` demonstrates end-to-end seeding using the demo seed pack at `src/main/resources/seed-packs/demo-seed`.

What the test does: - Creates a `SeedLoader` backed by `FileSeedSource`, `MongoSeedRepository`, and `MongoSeedRegistry`. - Builds a `SeedContext` populated with tenant/realm details to exercise the `tenantSubstitution` transform. - Applies the pack reference `SeedPackRef.of("demo-seed")`, which resolves the latest version of that pack (1.0.0 in tests). - Asserts that 2 records were inserted into the `codeLists` collection and that the `tenantSubstitution` fields were populated from the context. - Verifies an entry was written to `_seed_registry` with the dataset checksum and `records: 2`. - Re-applies the same pack and asserts the record count remains 2, demonstrating idempotency (thanks to upsert + registry checksum).

Why these design choices: - NDJSON for datasets: allows streaming large datasets and simple line-by-line diffs in code review; arrays are also supported for smaller payloads. - Natural-key upsert: manifests declare `naturalKey` to form the filter for `replaceOne(…, upsert=true)` ensuring idempotent writes and predictable overwrites. - Transform pipeline: keeps dataset files free of environment-specific values; all tenant/realm specifics are injected consistently at apply time. - Registry-based skip: checksums per dataset avoid unnecessary writes when content hasn’t changed—fast, safe re-runs during deployments. - Semantic-version selection: when multiple versions of a pack are available, the latest semver is used unless an exact version is requested.

Alternatives considered: - Store seed state in an external table keyed only by version. Rejected in favor of per-dataset checksums to detect content drift without bumping versions. - Hardcode seeding logic inside ad-hoc migrations. Rejected for lack of composability and poor reviewability. - Use inserts only (no upsert). Rejected due to lack of idempotence and difficulty correcting baseline data.

Chapter 14. Archetypes explained

Archetypes are named compositions of seed packs that model a product edition, SKU, or vertical stack. Instead of listing several packs every time you provision a tenant, you define an archetype once in a manifest and then apply it by name.

What an archetype is in this context: - It lives inside a seed pack manifest under archetypes:. - It contains a list of includes (same syntax as top-level includes) referring to packs and version ranges. - When applied, the loader resolves those pack refs plus the hosting pack itself (the manifest that defines the archetype) so that local datasets are included as part of the archetype. - Resolution uses semantic version rules and deduplicates by pack name, respecting dependency order and preventing cycles.

When to use archetypes: - To represent product tiers (e.g., Community, Pro, Enterprise) that bundle different combinations of base packs and optional modules. - To group verticalized defaults (e.g., Logistics-Fulfillment, Healthcare-Core) without forcing consumers to know every underlying pack.

14.1. Example A: Define and apply an archetype in the same pack

Manifest (logistics-core/manifest.yaml):

```
seedPack: logistics-core
version: 1.4.2

includes:
  - accounting-base@^1.1

datasets:
  - collection: codeLists
    file: datasets/codelists.ndjson
    naturalKey: [ codeListName, code ]
    upsert: true

archetypes:
  - name: FulfillmentPlus
    includes:
      - logistics-core@^1.4     # self + constraints
      - shipping-defaults@~2
```

Applying it in code:

```
SeedLoader loader = SeedLoader.builder()
        .addSeedSource(new FileSeedSource("local", Paths.get("seed-packs")))
        .seedRepository(new MongoSeedRepository(mongoClient))
        .seedRegistry(new MongoSeedRegistry(mongoClient))
        .build();
```

```
SeedContext ctx = SeedContext.builder("my-realm").build();
loader.applyArchetype("FulfillmentPlus", ctx);
```

Notes: - applyArchetype looks up the latest manifest that defines an archetype named "FulfillmentPlus" across all discovered packs, resolves the include refs, and then applies the union. - If multiple manifests define the same archetype name, the latest semver manifest wins.

14.2. Example B: Cross-pack archetype in a dedicated "editions" pack

You can centralize product definitions into a thin pack that only defines archetypes and forward-references other packs:

Manifest (product-editions/manifest.yaml):

```
seedPack: product-editions
version: 1.0.0

archetypes:
  - name: Enterprise
    includes:
      - logistics-core@^1.5
      - shipping-defaults@~2
      - analytics-starter@^0.9
```

Apply in code:

```
loader.applyArchetype("Enterprise", ctx);
```

This keeps edition composition decoupled from individual module packs.

14.3. Resolution and ordering details

- Version matching: Each include can be exact (=1.2.3), a semver range (e.g., ^1.5, ~2), or omitted (latest). See SeedPackRef.parse("name@spec").
- Deduplication: If multiple includes select the same pack name (possibly different versions), the highest version that satisfies all constraints is chosen; duplicates are applied only once.
- Dependency order: Includes are recursively resolved depth-first, while the loader guards against cycles and applies datasets in a stable order per resolved pack.

14.4. Interaction with ApplySeedPacksChangeSet

- The Apply Seed Packs change set scans the seed root and applies the latest version of every

discovered pack to the realm. It does not automatically choose an archetype.

- Use applyArchetype programmatically (e.g., from a TenantProvisioningService) when you want to provision only the packs that belong to a specific edition.
- You can combine approaches: let migrations ensure baseline packs are present for all tenants; then, on tenant onboarding, call applyArchetype(...) to add edition-specific content.

14.5. Tenant provisioning with archetypes

The tenant provisioning API accepts an optional list of archetype names and will apply the corresponding seed packs during onboarding.

- Endpoint: POST /admin/tenants
- Request body fields (subset):
- tenantEmailDomain, orgRefName, accountId, adminUserId, adminUsername, adminPassword
- archetypes: optional array of strings (archetype names)
- Behavior:
- After running migrations and creating the admin user, each archetype is resolved across all manifests and applied using the same SeedLoader used elsewhere.
- If an archetype name is unknown, the request fails with 409/500 depending on context.

Example request:

```
{
  "tenantEmailDomain": "demo-archetype.example",
  "orgRefName": "demo-archetype.example",
  "accountId": "9999999999",
  "adminUserId": "admin@demo-archetype.example",
  "adminUsername": "admin@demo-archetype.example",
  "adminPassword": "secret",
  "archetypes": ["DemoArchetype"]
}
```

On success, the new realm (demo-archetype-example) will have the datasets from the selected archetypes applied and recorded in the _seed_registry.

Chapter 15. REST API for seed packs

The framework exposes admin-only endpoints to inspect and apply seed packs per realm (tenant DB). These are disabled to non-admin users via role checks.

Base path: /admin/seeds

Endpoints: - GET /admin/seeds/pending/{realm} - Lists pending seed packs for the realm. A pack is pending if any dataset checksum differs from the last applied or was never applied. - Optional query parameter: filter=pack1,pack2 to restrict by pack name. - Response example:

+

```
[
  {
    "seedId": "demo-seed@1.0.0",
    "seedPack": "demo-seed",
    "version": "1.0.0",
    "datasets": [
      {"collection": "codeLists", "file": "datasets/codelists.ndjson", "checksum":
"..."}
    ]
  }
]
```

- POST /admin/seeds/apply/{realm}
- Applies the latest version of all discovered packs (or only those matching filter).
- Query parameter: filter=pack1,pack2
- Response example:

  ```
  {"applied":["demo-seed"]}
  ```

- POST /admin/seeds/{realm}/{seedPack}/apply
- Applies the latest version of a single pack by name. Idempotent: unchanged datasets are skipped.
- Response example:

  ```
  {"applied":["demo-seed"]}
  ```

- GET /admin/seeds/history/{realm}
- Returns the per-dataset seeding history as recorded in the `_seed_registry` collection.

Authentication and roles: - All endpoints require role admin. In integration tests you can use @TestSecurity(user="admin", roles={"admin").

Configuration: - quantum.seed.root: filesystem path to the root folder where seed packs are discovered. In tests this defaults to src/test/resources/seed-packs. - quantum.seed.apply.filter: optional comma-separated list of pack names to limit automatic application by changeset.

Chapter 16. Using MorphiaSeedRepository (Morphia-backed seeding)

Note about collection vs modelClass: - You can now omit collection in a dataset when you specify modelClass. The framework will derive the collection name from the Morphia mapping of the model class. - Derivation rules: if the model class has @Entity with a non-empty value, that value is used as the collection name; otherwise the simple Java class name is used. - This derived name is used consistently for logging, in the _seed_registry entries, and for the Mongo fallback path. - Backwards compatibility: specifying collection is still supported, and will override the derived name.

In addition to the default Mongo-based persistence, you can instruct the seeding pipeline to persist a dataset via a Morphia repository mapped to a concrete UnversionedBaseModel. This enables: - Automatic class discriminator fields (for example, Morphia's _t) and proper collection mapping. - Population of framework-managed fields (dataDomain, audit info, etc.) by the repository layer. - Consistent security filtering and validations applied by Morphia repos in normal runtime.

How to enable Morphia for a dataset: - Add modelClass to the dataset in manifest.yaml with the fully-qualified class name that extends UnversionedBaseModel. - Keep naturalKey and transforms as usual. The loader will still compute checksums and idempotently upsert.

Example manifest snippet:

```
seedPack: morphia-demo
version: 1.0.0

datasets:
  - collection: CodeList
    file: datasets/codeList.ndjson
    naturalKey: [category, key]
    upsert: true
    modelClass: com.e2eq.framework.model.persistent.base.CodeList
    transforms:
      - type: tenantSubstitution
        config:
          tenantField: tenantId
          orgField: orgRefName
          ownerField: ownerId
          accountField: accountNum
          realmField: realmId
```

Walkthrough (based on the integration test MorphiaSeedRepositoryIntegrationTest): - The test builds a SeedLoader with MorphiaSeedRepository and MongoSeedRegistry pointing at src/test/resources/seed-packs. - A SeedContext is created for a test realm. For repository-layer behavior (dataDomain, permissions), a minimal SecurityContext is also established in the test. - The loader applies SeedPackRef.of("morphia-demo"), which reads datasets/codeList.ndjson with two CodeList records. - The dataset declares modelClass, so MorphiaSeedRepository resolves the

CodeList MorphiaRepo and attempts to save via Morphia. - If Morphia permission rules are not yet configured for the realm, MorphiaSeedRepository automatically falls back to a direct Mongo write, ensuring seeds still apply predictably but without Morphia-only fields. - The test then asserts that two documents exist in the CodeList collection and that _seed_registry has a matching history entry for the dataset checksum with records: 2.

Key behaviors and edge cases: - Indexes: When modelClass is present, ensureIndexes relies on Morphia mapping to enforce annotated indexes for the model. Any requiredIndexes declared in the manifest are still respected by the Mongo fallback. - Transforms: tenantSubstitution adds tenant context fields. When saving via Morphia, the repository adapts top-level tenant fields into dataDomain.* automatically for common models, and removes transient fields like realmId before mapping. - Permissions: If repository permissions prevent writes (for example, missing policies in a brand-new realm), MorphiaSeedRepository will log a warning and fall back to Mongo so seeding is not blocked. As you evolve policies, the Morphia path will be taken automatically on future runs. - Idempotency: Upsert semantics still honor naturalKey for both Morphia and Mongo paths, and _seed_registry records checksums so unchanged datasets are skipped.

See also: - Test source: quantum-framework/src/test/java/com/e2eq/framework/service/seed/MorphiaSeedRepositoryIntegrationTest.java - Manifest and dataset: quantum-framework/src/test/resources/seed-packs/morphia-demo - Admin APIs: /admin/seeds to list pending, apply packs, and inspect history per realm.

Chapter 17. Dataset URLs and routing (file:// and s3://)

Starting with Quantum 1.2.2, dataset files in a manifest can be specified either as:

- Relative paths (backward compatible): resolved relative to the manifest's own location as provided by its SeedSource (filesystem or S3).
- Absolute URLs with a scheme that identifies the source: currently supported schemes are file:// and s3://. The loader automatically routes these URLs to the appropriate SeedSource at runtime.

Notes: - If the dataset value contains "://", it is treated as a URL and routed by scheme. Otherwise it is treated as a relative path. - Using URLs allows you to mix sources inside a single manifest (e.g., some datasets from the local filesystem and others from S3). - This is extensible; additional schemes can be supported by adding optional modules in the future.

Examples:

```
seedPack: mixed-demo
version: 0.1.0

datasets:
  # Relative path (resolved relative to the manifest location)
  - collection: localDefaults
    file: datasets/defaults.ndjson
    naturalKey: [ key ]
    upsert: true

  # Absolute filesystem URL
  - collection: roles
    file: file:///opt/app/seed-packs/roles.ndjson
    naturalKey: [ code ]
    upsert: true

  # Absolute S3 URL
  - collection: carriers
    file: s3://my-seeds/mixed-demo/0.1.0/carriers.ndjson
    naturalKey: [ code ]
    upsert: true
```

Routing behavior: - Relative path: delegated to the SeedSource that loaded the manifest (unchanged behavior). - file:// URL: handled by FileSeedSource. - s3:// URL: handled by S3SeedSource (requires the optional quantum-seed-s3 module on the classpath).

Chapter 18. S3 Seed Source (optional module)

The framework provides an optional SeedSource backed by Amazon S3, packaged in a separate module so you can include it only when needed:

- Module: `quantum-seed-s3`
- Class: `com.end2endlogic.quantum.seed.s3.S3SeedSource`
- Purpose: Discover seed pack manifests and datasets stored in S3; optionally assume an IAM role via STS if cross-account access is required.

18.1. When to use S3 vs. filesystem

Use S3 when any of the following apply: - You want a single canonical location for seed content shared across services and environments. - You need to distribute seed packs across many runtimes (containers, serverless) without baking files into images. - You want to leverage S3 features: versioning, object immutability, access logging, and cross-account sharing. - CI/CD pipelines or content teams publish seed packs independently from application deployments.

Filesystem may be simpler when: - Developing locally with small packs that live inside your project. - You prefer packs bundled within the application JAR/image and don't need central distribution.

Advantages of S3 over filesystem: - Centralized distribution and caching via S3/CloudFront. - Cross-account access with STS AssumeRole. - Object versioning and retention policies for auditability. - Decouples seed content delivery from app build artifacts.

18.2. S3 layout conventions

S3SeedSource expects the same on-disk structure as FileSeedSource, just hosted under a bucket + optional prefix. Each seed pack version is a folder containing a manifest file and any referenced datasets. For example:

- s3://my-seeds/seed-packs/customer/1.0.0/manifest.yaml
- s3://my-seeds/seed-packs/customer/1.0.0/customers.ndjson
- s3://my-seeds/seed-packs/logistics-core/1.4.2/manifest.yaml

Notes: - The manifest file name defaults to `manifest.yaml` but can be overridden. - Dataset file paths in the manifest are resolved relative to the manifest's key (folder).

18.3. Configuration and credentials

Constructor parameters for S3SeedSource: - id: Human-friendly source id (e.g., "s3-main"). - bucket: S3 bucket name. - prefix: Optional prefix under which seed packs live (e.g., "seed-packs/"). May be empty. - manifestFileName: Optional. Defaults to "manifest.yaml". - region: Optional AWS region. If omitted, AWS SDK v2 region resolution applies. - roleArn: Optional ARN of a role to assume via STS.

If omitted/blank, the default credentials chain is used. - roleSessionName: Optional session name when assuming a role. Defaults to "quantum-seed". - externalId: Optional ExternalId for AssumeRole if the target role trust policy requires it. - roleDuration: Optional session duration for AssumeRole; defaults to 30 minutes.

Credential modes: - Default credentials chain (no roleArn): The AWS SDK v2 uses environment variables, profile, ECS/EC2/Lambda task role, etc. - AssumeRole (roleArn provided): The source uses STS to assume the specified role before calling S3.

IAM considerations: - The credentials (base or assumed) must allow `s3:ListBucket` on the bucket (scoped to the prefix) and `s3:GetObject` for objects under the prefix. - For cross-account usage, grant the calling account sts:AssumeRole on the target role, and in the role's policy grant the S3 permissions above.

Minimal target role policy example:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:ListBucket"],
      "Resource": "arn:aws:s3:::my-seeds",
      "Condition": {"StringLike": {"s3:prefix": ["seed-packs/*"]}}
    },
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject"],
      "Resource": "arn:aws:s3:::my-seeds/seed-packs/*"
    }
  ]
}
```

18.4. Usage examples

Create a loader that discovers packs from S3 with the runtime IAM role or local credentials:

```
import com.e2eq.framework.service.seed.*;
import com.end2endlogic.quantum.seed.s3.S3SeedSource;
import software.amazon.awssdk.regions.Region;

SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new S3SeedSource(
        "s3-main",
        "my-seeds",
        "seed-packs/",
        "manifest.yaml",
        Region.US_EAST_1,
```

```
        null, // roleArn null => use default creds / runtime role
        null,
        null,
        null))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .build();
```

Assuming a cross-account role:

```
SeedLoader loader = SeedLoader.builder()
    .addSeedSource(new S3SeedSource(
        "s3-cross-account",
        "partner-seeds",
        "prod/",
        "manifest.yaml",
        Region.US_WEST_2,
        "arn:aws:iam::123456789012:role/SeedReadOnly",
        "quantum-seed-session",
        "external-id-abc123",
        java.time.Duration.ofMinutes(45)))
    .seedRepository(new MongoSeedRepository(mongoClient))
    .seedRegistry(new MongoSeedRegistry(mongoClient))
    .build();
```

Using the builder helper:

```
import com.end2endlogic.quantum.seed.s3.S3SeedSourceBuilder;

S3SeedSource s3 = new S3SeedSourceBuilder()
    .id("s3-main")
    .bucket("my-seeds")
    .prefix("seed-packs/")
    .region(Region.US_EAST_1)
    .build();
```

18.5. Defining datasets in the manifest (S3)

Manifests can reference S3 in two ways:

- Relative paths (backward compatible): when a manifest itself is loaded from S3, dataset files are resolved relative to the manifest's S3 key (folder), just like the filesystem source.
- Absolute URLs: you can specify s3://bucket/key URLs directly in any manifest. These are routed to S3 regardless of where the manifest was discovered from.

Example S3 manifest and dataset layout:

```
s3://my-seeds/seed-packs/customer/1.0.0/manifest.yaml
s3://my-seeds/seed-packs/customer/1.0.0/customers.ndjson
s3://my-seeds/seed-packs/customer/1.0.0/roles.json
```

Manifest file (manifest.yaml):

```yaml
seedPack: customer
version: 1.0.0

datasets:
  - collection: customers
    file: customers.ndjson     # resolved relative to the manifest's folder in S3
    naturalKey: [ accountNumber ]
    upsert: true

  - collection: roles
    file: roles.json           # also relative to the manifest's folder
    naturalKey: [ code ]
    upsert: true

archetypes:
  - name: CustomerBase
    includes:
      - customer@=1.0.0
```

No additional configuration is required in the manifest to indicate S3; the SeedSource you configure at runtime determines where manifests are discovered and how dataset paths are resolved.

18.6. Troubleshooting

- Access denied: Verify credentials or the assumed role's permissions for ListBucket/GetObject on the bucket/prefix.
- Region mismatch: Provide an explicit region if your runtime's default region doesn't match the bucket's region.
- Object not found: Confirm the prefix and that the manifest file name matches (default is manifest.yaml).
- Slow discovery: Use a tighter prefix to avoid scanning a large bucket namespace; consider S3 inventory or object tagging strategies if your repository grows significantly.