Seed packs and declarative tenant seeding

Version 1.2.2-SNAPSHOT, 2025-10-14T21:03:56Z

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

```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
    requiredIndexes:
      - name: uk_codeLists_name_code
        unique: true
        keys:
          codeListName: 1
          code: 1
    transforms:
      - type: tenantSubstitution
        config:
          tenantField: tenantId
          orgField: orgId
          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 ApplySeedPacksChangeSet discovers and executes packs

The framework provides a built-in migration change set named "Apply Seed Packs" that automatically finds and applies seed packs to each realm during migrations.

Key points: - Discovery: `ApplySeedPacksChangeSet` constructs a `SeedLoader` with a `FileSeedSource` pointing at the configured seed root. Set `quantum.seed.root` (for tests we default to `src/test/resources/seed-packs`). The source walks the directory tree and locates every `manifest.yaml` file. - Selection: For each discovered seed pack name, the change set selects the latest semantic version and builds `SeedPackRef.exact(name, version)` for application. - Execution: The change set builds a `SeedContext` for the current Morphia session's database (realm) and calls `loader.apply(refs, context)`. Indexes declared in the manifest are created before data is upserted. - Idempotency: 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. - Automatic re-run on changes: `ApplySeedPacksChangeSet.getChangeSetVersion()` computes a fingerprint over all discovered manifests and dataset files. If any manifest or dataset content changes, the fingerprint changes; `MigrationService` sees a new change set version and re-executes the change set. This gives "rerun on content change" semantics without bumping a manual version.

Configuration snippet:

```
# test profile uses a local seed root
quantum.seed.root=src/test/resources/seed-packs
# ensure change set package includes the seed change set
quantum.database.migration.changeset.package="com.e2eq.framework.model.persistent.morp
hia.changesets"
```

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: accountId
          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 configured fields. Config keys in the manifest: - tenantField: field in the record to receive tenantId (if present in context) - orgField: field to receive orgRefName (if present) - ownerField: field to receive ownerId (if present) - accountField: field to receive accountId (if present) - realmField: field to receive realmId (always set to the current realm)

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

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).

Chapter 13. Test walkthrough: SeedLoaderIntegrationTest

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