# Seed packs and declarative tenant seeding

Version 1.2.2-SNAPSHOT, 2025-10-14T18:04:21Z

# Table of Contents


1. Introduction . . . 2
    1.1. The problem . . . 2
    1.2. Why this needs to be solved . . . 2
    1.3. How seed packs solve it . . . 2
2. Why seed packs? . . . 3
3. High-level flow . . . 4
4. Manifest quick reference . . . 5
5. Programmatic usage . . . 6
6. Extensibility hooks . . . 7
7. Operational tips . . . 8
8. Primary scenarios . . . 9
9. Explicit examples . . . 10
    9.1. Example 1: Minimal manifest and NDJSON . . . 10
    9.2. Example 2: Applying packs in code . . . 10
    9.3. Example 3: Using an archetype . . . 11
    9.4. Example 4: Exact version and includes in a manifest . . . 11
10. Troubleshooting . . . 12

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.