PeeGeeQ Database Setup Guide

Comprehensive guide to database initialization across all services and environments

Version: 1.0
Date: December 1, 2025
Author: Mark Andrew Ray-Smith Cityline Ltd

Table of Contents

Overview

PeeGeeQ uses different database initialization strategies depending on the environment and service:

Environment	Method	Tool	Description
Production	Flyway Migrations	`peegeeq-migrations` module	Standalone JAR in CI/CD pipeline
Development	Flyway Migrations or Docker	Maven plugin or docker-compose	Automated or manual via scripts
Integration Tests	Programmatic Setup	`PeeGeeQDatabaseSetupService`	Template-based dynamic setup
Unit Tests	SQL Script Injection	`PeeGeeQTestSchemaInitializer`	Direct JDBC SQL execution

Key Principle: Production uses **migrations** (Flyway), tests use **programmatic setup** (templates).

Important: PeeGeeQ supports **custom PostgreSQL schemas** - the entire system can be deployed to any schema (e.g., `peegeeq`, `myapp_queue`, `tenant_a`) instead of the default `public` schema. This enables multi-tenant deployments, namespace isolation, and integration with existing databases.

Custom Schema Support

Overview

PeeGeeQ is designed to work with **any PostgreSQL schema**, not just the default `public` schema. This enables:

- **Multi-tenant deployments**: `tenant_a`, `tenant_b`, `tenant_c` schemas
- **Namespace isolation**: `myapp_queue`, `yourapp_queue` in same database
- **Integration with existing systems**: Deploy PeeGeeQ into existing database schemas
- **Security segregation**: Different schemas with different permissions
- **Schema-based sharding**: Horizontal scaling using schema partitioning

Default Schema Configuration

By default, PeeGeeQ uses **two schemas**:

Schema	Purpose	Contains
`peegeeq`	Main queue operations	`queue_messages`, `outbox`, `outbox_consumer_groups`, `dead_letter_queue`, `queue_template`, consumer group tables
`bitemporal`	Event sourcing	`bitemporal_event_log`, `event_store_template`, temporal views

Why Two Schemas?

1. **Logical Separation**: Queue operations vs event sourcing are distinct concerns
2. **Permission Management**: Grant queue access without event store access
3. **Backup Strategies**: Backup event store separately from queues
4. **Schema Evolution**: Evolve queue schema independently from event schema

Configuring Custom Schemas

Production (Flyway Migrations)

Method 1: Flyway Configuration File

```
# peegeeq-migrations/src/main/resources/application.properties

# Example 1: Custom schema names
flyway.schemas=myapp_messaging,myapp_events

# Example 2: Single schema for everything
flyway.schemas=myapp

# Example 3: Multi-tenant with tenant ID
flyway.schemas=tenant_${TENANT_ID}_queue,tenant_${TENANT_ID}_events

# Example 4: Namespace isolation
flyway.schemas=acme_peegeeq,acme_bitemporal
```

Method 2: Environment Variables

```
# Set custom schemas via environment
export FLYWAY_SCHEMAS="myapp_messaging,myapp_events"

# Run migrations
java -jar peegeeq-migrations.jar migrate
```

Method 3: Command Line

```
# Maven plugin
mvn flyway:migrate \
  -Dflyway.schemas=myapp_messaging,myapp_events

# Standalone JAR
java -jar peegeeq-migrations.jar migrate \
  -schemas=myapp_messaging,myapp_events
```

Application Configuration

Java Configuration

```
// Option 1: Using PeeGeeQConfiguration constructor
PeeGeeQConfiguration config = new PeeGeeQConfiguration(
    "production",              // profile
    "localhost",               // host
    5432,                      // port
    "myapp_db",               // database
    "dbuser",                 // username
    "password",               // password
    "myapp_messaging"         // CUSTOM SCHEMA
);

PeeGeeQManager manager = new PeeGeeQManager(config);
```

Properties File

```
# application.properties
peegeeq.db.schema=myapp_messaging
peegeeq.db.host=localhost
peegeeq.db.port=5432
peegeeq.db.database=myapp_db
peegeeq.db.username=dbuser
peegeeq.db.password=password
```

Environment Variables

```
export PEEGEEQ_DB_SCHEMA="myapp_messaging"
export PEEGEEQ_DB_HOST="localhost"
export PEEGEEQ_DB_PORT="5432"
export PEEGEEQ_DB_DATABASE="myapp_db"
export PEEGEEQ_DB_USERNAME="dbuser"
export PEEGEEQ_DB_PASSWORD="password"
```

Integration Tests

```
@BeforeAll
void setupDatabase() throws Exception {
    setupService = new PeeGeeQDatabaseSetupService();

    // Custom schema in test
    DatabaseConfig dbConfig = new DatabaseConfig.Builder()
        .host(postgres.getHost())
        .port(postgres.getFirstMappedPort())
        .databaseName("test_db_" + UUID.randomUUID())
        .username(postgres.getUsername())
        .password(postgres.getPassword())
        .schema("myapp_test")  // CUSTOM SCHEMA
        .build();

    DatabaseSetupRequest request = new DatabaseSetupRequest.Builder()
        .setupId("test_" + UUID.randomUUID())
        .databaseConfig(dbConfig)
        .queues(queueConfigs)
        .eventStores(eventStoreConfigs)
        .build();

    setupResult = setupService.createCompleteSetup(request).get();
}
```

Schema Layout Examples

Example 1: Single Schema for Everything

```
-- Use "myapp" for all PeeGeeQ tables
CREATE SCHEMA myapp;

-- All tables in myapp schema
myapp.queue_messages
myapp.outbox
myapp.outbox_consumer_groups
```

```
myapp.bitemporal_event_log  -- event store also in myapp
myapp.dead_letter_queue
myapp.queue_template
myapp.event_store_template
-- ... all other tables
```

Configuration:

```
flyway.schemas=myapp
peegeeq.db.schema=myapp
```

Example 2: Separate Schemas for Queue and Events

```
-- Queue operations in one schema
CREATE SCHEMA acme_queue;
acme_queue.queue_messages
acme_queue.outbox
acme_queue.outbox_consumer_groups
acme_queue.dead_letter_queue
acme_queue.queue_template

-- Event sourcing in another schema
CREATE SCHEMA acme_events;
acme_events.bitemporal_event_log
acme_events.event_store_template
```

Configuration:

```
flyway.schemas=acme_queue,acme_events
peegeeq.db.schema=acme_queue
# Event store uses acme_events automatically
```

Example 3: Multi-Tenant Architecture

```
-- Tenant A
CREATE SCHEMA tenant_a_queue;
CREATE SCHEMA tenant_a_events;

-- Tenant B
CREATE SCHEMA tenant_b_queue;
CREATE SCHEMA tenant_b_events;

-- Tenant C
```

```
CREATE SCHEMA tenant_c_queue;
CREATE SCHEMA tenant_c_events;
```

Configuration (per tenant):

```
// Tenant A configuration
PeeGeeQConfiguration tenantAConfig = new PeeGeeQConfiguration(
    "tenant_a",
    "localhost", 5432, "shared_db",
    "tenant_a_user", "password",
    "tenant_a_queue"  // Tenant-specific schema
);

// Tenant B configuration
PeeGeeQConfiguration tenantBConfig = new PeeGeeQConfiguration(
    "tenant_b",
    "localhost", 5432, "shared_db",
    "tenant_b_user", "password",
    "tenant_b_queue"  // Tenant-specific schema
);
```

Example 4: Integration with Existing Database

```
-- Existing application schemas
CREATE SCHEMA app_core;      -- Your existing tables
CREATE SCHEMA app_reporting; -- Your existing reports

-- Add PeeGeeQ to existing database
CREATE SCHEMA app_messaging; -- PeeGeeQ queue operations
CREATE SCHEMA app_events;    -- PeeGeeQ event sourcing

-- All schemas coexist in same database
app_core.users
app_core.orders
app_reporting.sales_summary
app_messaging.queue_messages     -- PeeGeeQ
app_messaging.outbox             -- PeeGeeQ
app_events.bitemporal_event_log  -- PeeGeeQ
```

Benefits:

- No separate database required
- Can use foreign keys between app and PeeGeeQ tables
- Shared connection pool
- Single backup/restore process

Schema Migration Script Updates

When using custom schemas, Flyway automatically adjusts the migration scripts:

Original Migration (uses `peegeeq` schema):

```
-- V001__Create_Base_Tables.sql
CREATE SCHEMA IF NOT EXISTS peegeeq;
CREATE TABLE peegeeq.queue_messages (
    id BIGSERIAL PRIMARY KEY,
    -- ...
);
```

How Flyway Handles Custom Schemas:

Flyway's `schemas` configuration tells it which schema(s) to manage, and it automatically:

1. Creates the schema(s) if they don't exist
2. Sets the PostgreSQL search_path for the migration
3. Executes all CREATE TABLE statements in the configured schema(s)

Note: The migration script itself still contains `peegeeq.` prefixes. If you need to use completely custom schema names in the SQL itself, you'll need to either:

1. **Use Flyway placeholders**:

```
-- V001__Create_Base_Tables.sql
CREATE SCHEMA IF NOT EXISTS ${queueSchema};
CREATE TABLE ${queueSchema}.queue_messages (
    id BIGSERIAL PRIMARY KEY,
    -- ...
);
```

```
# flyway.conf
flyway.placeholders.queueSchema=myapp_messaging
flyway.placeholders.eventSchema=myapp_events
```

2. **Create schema-specific migration files**: Duplicate V001 for different schema names (not recommended)

3. **Use dynamic SQL in migrations**: Use DO blocks to build table names dynamically (complex)

Recommended Approach: Use the standard `peegeeq` and `bitemporal` schema names in migration files, and let Flyway's `schemas` configuration control where they're created. This keeps migrations simple and portable.

Schema Permissions

When using custom schemas, ensure proper permissions:

```
-- Grant schema usage
GRANT USAGE ON SCHEMA myapp_messaging TO myapp_user;
GRANT USAGE ON SCHEMA myapp_events TO myapp_user;

-- Grant table permissions
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA myapp_messaging TO
myapp_user;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA myapp_events TO
myapp_user;

-- Grant sequence permissions (for BIGSERIAL columns)
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA myapp_messaging TO myapp_user;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA myapp_events TO myapp_user;

-- Grant future objects (for schema evolution)
ALTER DEFAULT PRIVILEGES IN SCHEMA myapp_messaging
    GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO myapp_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA myapp_messaging
    GRANT USAGE, SELECT ON SEQUENCES TO myapp_user;
```

Search Path Considerations

PostgreSQL Search Path:

By default, PostgreSQL uses `search_path = "$user", public`. When using custom schemas:

```
-- Option 1: Set search path for user
ALTER USER myapp_user SET search_path = myapp_messaging, myapp_events, public;

-- Option 2: Set search path for database
ALTER DATABASE myapp_db SET search_path = myapp_messaging, myapp_events, public;

-- Option 3: Set search path in session
SET search_path = myapp_messaging, myapp_events, public;
```

PeeGeeQ Recommendation: Always use **fully qualified table names** (schema.table) in application code to avoid search_path issues:

```
// ✅ GOOD: Fully qualified
String sql = "SELECT * FROM myapp_messaging.queue_messages WHERE topic = ?";

// ❌ BAD: Relies on search_path
String sql = "SELECT * FROM queue_messages WHERE topic = ?";
```

PeeGeeQ's internal code always uses fully qualified names, so search_path doesn't affect operations.

Schema Best Practices

1. Use Descriptive Schema Names

```
✅ GOOD:
  myapp_queue, myapp_events
  tenant_acme_messaging, tenant_acme_events
  prod_peegeeq, prod_bitemporal

❌ BAD:
  schema1, schema2
  s1, s2
  temp_schema
```

2. Consistent Naming Convention

```
✅ GOOD:
  All tenants: tenant_{id}_queue, tenant_{id}_events
  All apps: {appname}_queue, {appname}_events

❌ BAD:
  Mixed: tenant_a_queue, tenantB_messaging, tenant-c-events
```

3. Document Schema Purpose

```
-- Add comments to schemas
COMMENT ON SCHEMA myapp_queue IS 'PeeGeeQ queue operations for MyApp';
COMMENT ON SCHEMA myapp_events IS 'PeeGeeQ event sourcing for MyApp';
```

4. Plan for Growth

```
✅ GOOD: Leave room for expansion
  myapp_queue (can add myapp_analytics, myapp_cache later)

❌ BAD: Too generic
  myapp (what happens when you need another schema?)
```

5. Schema Naming Length Limit

```
PostgreSQL identifiers (including schema names) have a 63-character limit.

✅ GOOD:
  tenant_acme_corp_queue (23 chars)
```

```
❌ BAD:
  tenant_acme_corporation_very_long_name_queue_messaging_system (60+ chars)
```

Schema Definition Sources and Alignment

Critical Understanding: Three Sources of Truth

PeeGeeQ maintains schema definitions in **three distinct locations**, each serving a specific purpose:

```
┌─────────────────────────────────────────────────────────────────────┐
│                PeeGeeQ Schema Definition Sources                    │
└─────────────────────────────────────────────────────────────────────┘

1. FLYWAY MIGRATIONS (Production Foundation)
   Location: peegeeq-migrations/src/main/resources/db/migration/
   Files:
   ├── V001__Create_Base_Tables.sql (576 lines)
   │   └── Creates: outbox, queue_messages, dead_letter_queue,
   │               bitemporal_event_log, message_processing, etc.
   └── V010__Create_Consumer_Group_Fanout_Tables.sql (442 lines)
       └── Adds: outbox_topics, outbox_topic_subscriptions, fanout columns

2. SQL TEMPLATES (Dynamic Queue/EventStore Creation)
   Location: peegeeq-db/src/main/resources/db/templates/
   Structure:
   ├── base/        (36 files) - Schemas, extensions, template tables
   │   ├── 05-queue-template.sql        (queue_template table)
   │   └── 06-event-store-template.sql  (event_store_template table)
   ├── queue/       (8 files)  - Per-queue table creation
   │   ├── 01-table.sql         (LIKE queue_template INCLUDING ALL)
   │   ├── 02a-index-topic.sql
   │   ├── 02b-index-lock.sql
   │   └── ...
   └── eventstore/  (11 files) - Per-event-store table creation
       ├── 01-table.sql         (LIKE event_store_template INCLUDING ALL)
       ├── 02a-index-validtime.sql
       └── ...

3. JAVA EMBEDDED SQL (Unit Test Fast Path)
   Location: peegeeq-test-
support/src/main/java/.../PeeGeeQTestSchemaInitializer.java
   Purpose: Unit tests that don't need full integration test isolation
   Content: 747 lines of embedded CREATE TABLE statements
```

Why Three Sources?

1. Flyway Migrations: Production Schema Foundation

Purpose: Version-controlled, auditable schema evolution for production databases

What It Creates:

- Base tables that all queues share: `outbox`, `queue_messages`, `dead_letter_queue`
- Bitemporal event log infrastructure: `bitemporal_event_log`
- Consumer group fanout tables: `outbox_topics`, `outbox_topic_subscriptions`
- Template tables: `queue_template`, `event_store_template` (for PostgreSQL LIKE)
- Extensions: `uuid-ossp`, `pg_stat_statements`
- Schemas: `peegeeq`, `bitemporal`

What It Does NOT Create:

- Individual queue tables (e.g., `orders_queue`, `payments_queue`)
- Individual event store tables (e.g., `order_events`, `payment_events`)

Used By:

- Production deployments (CI/CD pipelines)
- Development docker-compose setups
- Some integration tests that use Flyway directly

File Example:

```
-- V001__Create_Base_Tables.sql (excerpt)
CREATE TABLE IF NOT EXISTS outbox (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    payload JSONB NOT NULL,
    -- ... all columns
);

CREATE TABLE IF NOT EXISTS queue_messages (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    -- ... all columns
);

-- Template table for dynamic queue creation
CREATE TABLE peegeeq.queue_template (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    -- ... same columns as queue_messages
);
```

2. SQL Templates: Dynamic Table Creation Engine

Purpose: Parameterized SQL for creating tenant/feature-specific queue and event store tables at runtime

What It Creates:

- Per-queue tables: `tenant_acme_orders`, `tenant_acme_payments`, `feature_x_queue`

- Per-event-store tables: `tenant_acme_events`, `order_event_log`, `payment_event_log`
- Dedicated indexes for each queue/event store
- Notification triggers for each queue/event store

Key Feature: Uses PostgreSQL `LIKE ... INCLUDING ALL` to inherit from template tables

Used By:

- Production: `setupService.addQueue(setupId, config)` - creates new tenant queue
- Integration tests: `PeeGeeQDatabaseSetupService` - creates test-specific databases with multiple queues

Template Example:

```
-- queue/01-table.sql
CREATE TABLE IF NOT EXISTS {schema}.{queueName} (
    LIKE peegeeq.queue_template INCLUDING ALL
);

-- queue/02a-index-topic.sql
CREATE INDEX idx_{queueName}_topic
    ON {schema}.{queueName}(topic, visible_at, status);
```

Processing Flow:

```
// Runtime: Parameters injected
Map<String, String> params = Map.of(
    "queueName", "tenant_acme_orders",
    "schema", "peegeeq"
);

// SqlTemplateProcessor replaces placeholders
// Result: CREATE TABLE peegeeq.tenant_acme_orders (
//             LIKE peegeeq.queue_template INCLUDING ALL
//         );
```

3. Java Embedded SQL: Unit Test Fast Path

Purpose: Minimal schema setup for unit tests that need database but not full isolation

What It Creates:

- Same base tables as Flyway migrations
- Embedded directly in Java code (no external file loading)
- Uses JDBC `Statement.execute()` directly

Why Not Use Flyway or Templates?

- **Speed**: Faster than Flyway migration scanning

- **Simplicity**: No need for Vert.x or template processing
- **Isolation**: Unit tests don't need dynamic databases

Used By:

- `peegeeq-outbox` unit tests
- `peegeeq-native` unit tests
- Tests that don't require test-per-class database isolation

Code Example:

```
// PeeGeeQTestSchemaInitializer.java (excerpt)
private static void initializeOutboxSchema(Statement stmt) throws Exception {
    stmt.execute("""
        CREATE TABLE IF NOT EXISTS outbox (
            id BIGSERIAL PRIMARY KEY,
            topic VARCHAR(255) NOT NULL,
            payload JSONB NOT NULL,
            -- ... all columns (exact match to V001)
        )
        """);

    stmt.execute("CREATE INDEX IF NOT EXISTS idx_outbox_status_created " +
                 "ON outbox(status, created_at)");
}
```

Schema Alignment Strategy

The Alignment Challenge

With three sources of schema definitions, **how do we prevent drift?**

Example of Potential Drift:

```
-- Flyway V001: outbox table has 15 columns
ALTER TABLE outbox ADD COLUMN priority INT DEFAULT 5;

-- Template: queue_template not updated ❌
-- Unit Test SQL: PeeGeeQTestSchemaInitializer not updated ❌

Result: Production has priority column, tests don't → bugs!
```

Alignment Mechanisms

1. Template Inheritance from Flyway (Automatic)

Templates use `LIKE ... INCLUDING ALL` to inherit from template tables created by Flyway:

```
-- Flyway V001 creates template table
CREATE TABLE peegeeq.queue_template (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    -- ... 15 columns
);

-- Template inherits ALL columns automatically
CREATE TABLE {schema}.{queueName} (
    LIKE peegeeq.queue_template INCLUDING ALL  -- ✅ Automatic sync!
);
```

Result: When Flyway adds column to `queue_template`, ALL dynamically created queues get it automatically!

2. Contract Tests (Automated Validation)

Location: `peegeeq-migrations/src/test/java/dev/mars/peegeeq/migrations/SchemaContractTest.java`

Purpose: Validate Flyway schema matches expectations

```
@Test
void testOutboxTableStructure() {
    // Connects to database created by Flyway
    ResultSet rs = connection.getMetaData().getColumns(null, null, "outbox",
null);

    // Validates ALL columns present
    Set<String> columns = extractColumnNames(rs);
    assertThat(columns).contains(
        "id", "topic", "payload", "status", "retry_count",
        "required_consumer_groups",  // Added in V010
        "completed_consumer_groups", // Added in V010
        "completed_groups_bitmap"    // Added in V010
    );
}

@Test
void testQueueMessagesTableStructure() {
    // Validates queue_messages table columns
    // Ensures template table matches
}
```

3. Manual Synchronization (Unit Test SQL)

CRITICAL: When Flyway migrations change base tables, `PeeGeeQTestSchemaInitializer.java` must be updated manually.

Process:

1. Developer adds column to Flyway migration (e.g., V011__Add_Priority.sql)
2. Developer updates `PeeGeeQTestSchemaInitializer.java` to match
3. Contract tests validate both match

Verification:

```
# Run schema contract tests after migration changes
cd peegeeq-migrations
mvn test -Dtest=SchemaContractTest

# Run unit tests with updated schema
cd ../peegeeq-outbox
mvn test
```

4. Schema Drift Analysis (Manual Audit)

Location: `peegeeq-migrations/docs/old/SCHEMA_DRIFT_ANALYSIS.md`

Purpose: Periodic manual review of schema alignment

Results (as of November 2025):

- ✅ **NO CRITICAL DRIFT** detected
- ✅ All Flyway migrations match Java code
- ✅ All template tables match base tables
- ✅ Unit test SQL matches Flyway migrations

Process:

1. Quarterly schema audit
2. Compare Flyway SQL vs. Java embedded SQL
3. Compare template definitions vs. Flyway template tables
4. Document any intentional differences (future features)

Alignment Best Practices

When Adding a New Column

Step-by-Step Process:

```
1. Add to Flyway Migration
   File: peegeeq-
migrations/src/main/resources/db/migration/V011__Add_New_Column.sql

   ALTER TABLE outbox ADD COLUMN new_field VARCHAR(255);
   ALTER TABLE queue_messages ADD COLUMN new_field VARCHAR(255);

2. Update Template Table (if needed)
   File: peegeeq-
```

```
migrations/src/main/resources/db/migration/V011__Add_New_Column.sql

   ALTER TABLE peegeeq.queue_template ADD COLUMN new_field VARCHAR(255);

   (Dynamic queues will inherit automatically via LIKE)

3. Update Unit Test SQL
   File: peegeeq-test-support/src/main/java/.../PeeGeeQTestSchemaInitializer.java

   stmt.execute("""
       CREATE TABLE IF NOT EXISTS outbox (
           id BIGSERIAL PRIMARY KEY,
           -- ... existing columns
           new_field VARCHAR(255)  -- ✅ ADD HERE
       )
       """);

4. Update Java Code
   File: peegeeq-outbox/src/main/java/.../OutboxMessage.java

   public class OutboxMessage {
       private String newField;  // ✅ ADD HERE
       // getters/setters
   }

5. Run Contract Tests
   cd peegeeq-migrations
   mvn test -Dtest=SchemaContractTest

   (Tests will fail if schema doesn't match expectations)

6. Update All Tests
   cd ..
   mvn clean test

   (Ensure all tests pass with new schema)
```

When Adding a New Table

Critical: Tables that ALL queues use (like `outbox`, `dead_letter_queue`) go in Flyway migrations. Tables created per-queue (like `tenant_X_orders`) are created by templates.

```
If table is shared across all queues:
  → Add to Flyway migration (V012__Add_Audit_Log.sql)
  → Add to PeeGeeQTestSchemaInitializer.java
  → Add contract test to SchemaContractTest.java

If table is per-queue or per-tenant:
  → Add template to peegeeq-db/src/main/resources/db/templates/
  → Template inherits from existing template table
  → No unit test update needed (uses template system)
```

Schema Version Tracking

All three sources track schema versions differently:

```
1. Flyway Migrations
   Table: flyway_schema_history
   Tracks: V001, V010, V011, etc.
   Location: Automatically created by Flyway

2. SQL Templates
   Version: Embedded in .manifest files
   Tracks: Execution order (01-table.sql, 02a-index-topic.sql, etc.)
   Location: peegeeq-db/src/main/resources/db/templates/*/.manifest

3. Unit Test SQL
   Table: schema_version (manual)
   Tracks: Version string (optional)
   Location: Created by PeeGeeQTestSchemaInitializer
```

Documentation Files

File	Purpose	Update Frequency
`V001__Create_Base_Tables.sql`	Production schema foundation	On breaking changes
`V010__Create_Consumer_Group_Fanout_Tables.sql`	Consumer group fanout feature	Feature additions
`PEEGEEQ_COMPLETE_SCHEMA_SETUP.sql`	Standalone setup script (documentation)	After each migration
`PEEGEEQ_SCHEMA_QUICK_REFERENCE.sql`	Quick reference (documentation)	After each migration
`PeeGeeQTestSchemaInitializer.java`	Unit test schema	Every migration change
`SchemaContractTest.java`	Automated validation	Every migration change
`SCHEMA_DRIFT_ANALYSIS.md`	Manual audit report	Quarterly

Critical Takeaways

1. **Flyway Migrations = Foundation**: Creates schemas, extensions, base tables, and template tables
2. **SQL Templates = Dynamic Creation**: Creates tenant/feature-specific queues and event stores at runtime

3. **Unit Test SQL = Fast Path**: Minimal setup for unit tests without full integration test overhead
4. **Template Inheritance = Automatic Sync**: Dynamic queues inherit from template tables (no manual sync needed)
5. **Contract Tests = Validation**: Automated tests catch schema drift
6. **Manual Sync Required**: `PeeGeeQTestSchemaInitializer.java` must be updated when base tables change
7. **Quarterly Audits**: Manual schema drift analysis ensures long-term alignment

Common Questions

Q: Why not use Flyway everywhere?
A: Flyway is designed for schema evolution, not dynamic table creation. Templates enable runtime creation of tenant-specific queues.

Q: Why not use templates in unit tests?
A: Templates require Vert.x and async processing. Unit tests use synchronous JDBC for simplicity and speed.

Q: What if I forget to update PeeGeeQTestSchemaInitializer?
A: Unit tests will fail with "column does not exist" errors. Contract tests may also catch the issue.

Q: How do I know which source to update?
A: Use the decision matrix above: Shared tables → Flyway + Unit Test SQL. Per-queue tables → Templates only.

Q: Are template tables used in production?
A: Yes! Flyway creates `queue_template` and `event_store_template` tables. Runtime code uses `LIKE queue_template` to create tenant queues.

Why Templates? Understanding the Design

The Core Design: Dynamic Queue and Event Store Creation

PeeGeeQ's fundamental architecture is built around **runtime creation of queue and event store tables**. This is not a testing convenience - it's the core production feature that enables:

Primary Design Goal: Parameterized Schema Objects

PeeGeeQ creates **dedicated database tables for each queue and event store at runtime**:

- **Queue tables**: Each queue gets its own table with dedicated indexes and triggers
  - `peegeeq.orders_queue` (created for "orders" queue)
  - `peegeeq.payments_queue` (created for "payments" queue)
  - `peegeeq.tenant_123_orders` (created for tenant 123)
- **Event store tables**: Each event store gets its own table with temporal indexes
  - `bitemporal.order_event_log` (created for order events)
  - `bitemporal.payment_event_log` (created for payment events)
  - `bitemporal.tenant_456_events` (created for tenant 456)

Why This Design?

1. **Performance Isolation**: Each queue/event store has dedicated indexes and storage
2. **Multi-Tenancy**: Create tenant-specific queues without table bloat
3. **Feature Enablement**: Dynamically add queues when features are activated
4. **Scalability**: Partition data across multiple tables for horizontal scaling
5. **Maintenance**: Drop/archive old queues without affecting active ones

Production Example:

```
// Application creates queues dynamically at runtime
QueueConfig config = new QueueConfig.Builder()
    .queueName("tenant_acme_orders")  // Dynamic name!
    .maxRetries(3)
    .build();

// PeeGeeQ uses templates to create:
//   - peegeeq.tenant_acme_orders table
//   - 5 dedicated indexes
//   - notification trigger
setupService.addQueue(setupId, config).get();
```

Secondary Benefit: Test Validation

Integration tests need to **validate this core production feature** by:

- Creating multiple queues with different names
- Verifying each queue has correct schema structure
- Testing parallel queue creation (simulating multi-tenant onboarding)
- Validating dynamic database creation (for isolated test environments)

Test isolation is a side benefit, not the primary reason for templates. Tests use the same template-based creation that production uses.

The Solution: SQL Templates with Parameter Substitution

Templates enable PeeGeeQ's core architecture by providing reusable, parameterized SQL patterns:

```
Static Migration (Production)          Template (Integration Tests)
─────────────────────────────          ────────────────────────────

CREATE TABLE orders_queue (            CREATE TABLE {queueName} (
    id BIGSERIAL PRIMARY KEY,              id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255),                    topic VARCHAR(255),
    ...                                    ...
);                                     ) INHERITS ({schema}.queue_template);

CREATE INDEX idx_orders_topic          CREATE INDEX idx_{queueName}_topic
    ON orders_queue(topic);                ON {queueName}(topic);
```

At runtime (production or test), templates are processed:

```
// Production: Creating queue dynamically
Map<String, String> params = Map.of(
    "queueName", "tenant_acme_orders",
    "schema", "peegeeq"
);

// Template: CREATE TABLE {queueName} (...)
// Result:   CREATE TABLE tenant_acme_orders (...)
```

Why Not Just Use Flyway Migrations?

Requirement	Flyway Migrations	SQL Templates
Dynamic queue creation	❌ No	✅ Yes (core feature)
Parameterized table names	❌ No	✅ Yes (core feature)
Runtime queue creation	❌ No	✅ Yes (core feature)
Multi-tenant queue isolation	❌ Limited	✅ Yes (dedicated tables)
Feature flag queues	❌ Requires migration	✅ Instant creation
Version control	✅ Yes	⚠️ Template versions
Schema evolution	✅ Yes	⚠️ Via base template updates

Key Insight:

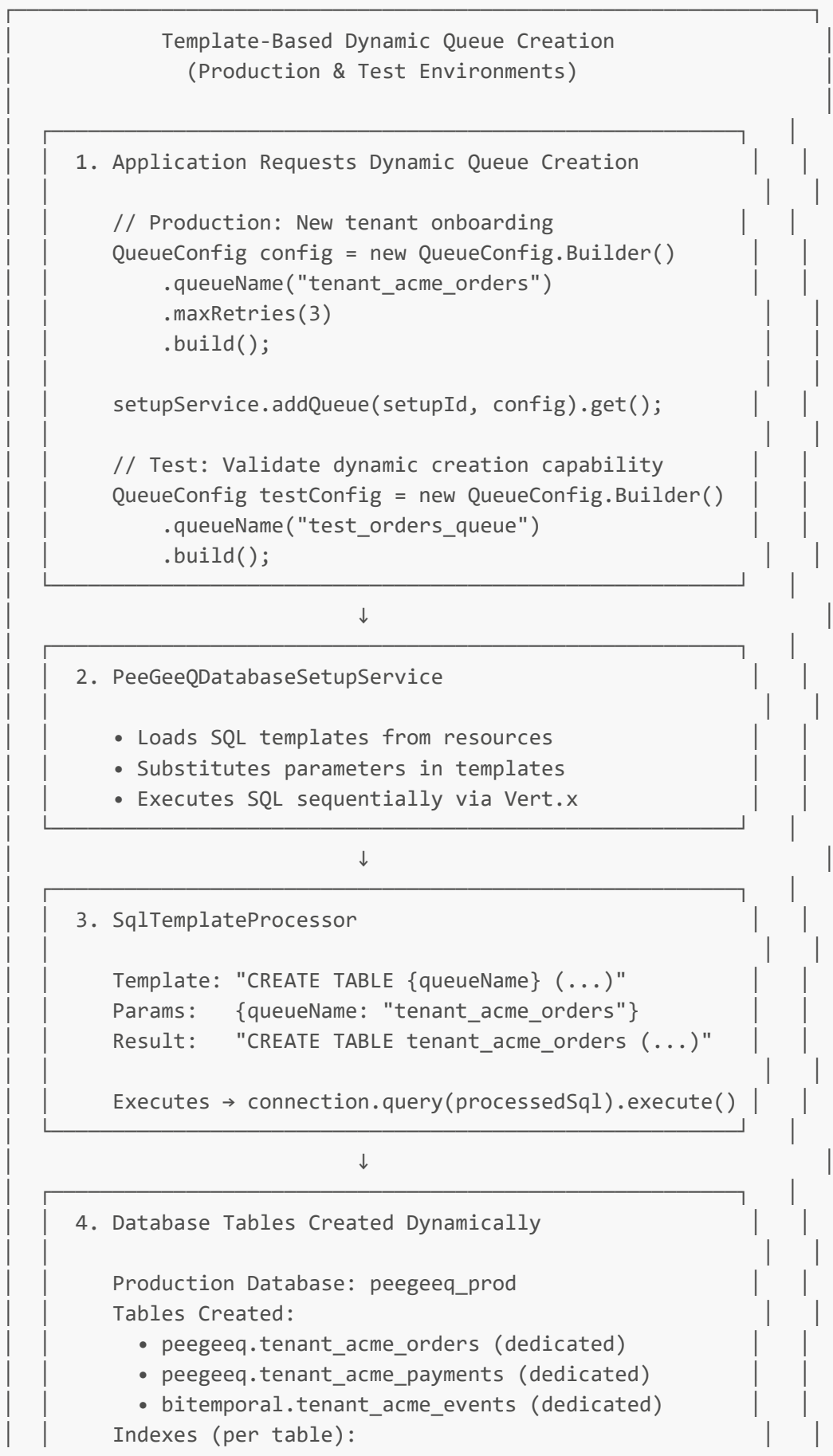
- **Flyway migrations** = Production schema **foundation** (extensions, schemas, base tables, template tables)
- **SQL templates** = Production **dynamic queue/event store creation** (per-queue tables, per-tenant tables, per-feature tables)
- **Both are used in production**: Migrations create the foundation once, templates create queues/event stores on-demand

Production Architecture:

```
1. Deployment: Run Flyway migrations
   ├─ Create schemas (peegeeq, bitemporal)
   ├─ Create extensions (uuid-ossp, pg_stat_statements)
   ├─ Create base tables (queue_messages, outbox, dead_letter_queue)
   └─ Create template tables (queue_template, event_store_template)

2. Runtime: Use templates for dynamic creation
   ├─ New tenant onboards → Create tenant_X_orders queue
   ├─ Feature enabled → Create feature_Y_queue
   └─ Workflow defined → Create workflow_Z_step_1_queue
```

Template Architecture Overview



```
│   │       • idx_tenant_acme_orders_topic                    │   │
│   │       • idx_tenant_acme_orders_visible                   │   │
│   │       • idx_tenant_acme_orders_priority                  │   │
│   │     Result: Tenant fully isolated at database level      │   │
│   └──────────────────────────────────────────────────────────┘   │
└──────────────────────────────────────────────────────────────────┘
```

Real-World Production Example: Multi-Tenant SaaS

Scenario: Multi-tenant order processing system where each tenant needs isolated queues.

Without Templates (limited approach):

```
// ❌ Problem: All tenants share same queue_messages table
// - No performance isolation
// - No data isolation
// - Difficult to scale per-tenant
// - Cannot drop tenant data independently

@RestController
public class OrderController {

    @PostMapping("/orders")
    public void createOrder(@RequestHeader("X-Tenant-Id") String tenantId,
                            @RequestBody Order order) {
        // All tenants use same table
        queue.send(order);  // queue_messages table

        // Problem: Millions of rows from all tenants in one table
        // Problem: Indexes cover all tenants (slower queries)
        // Problem: Cannot optimize per-tenant
    }
}
```

With Templates (PeeGeeQ's solution):

```
// ✅ Solution: Each tenant gets dedicated queue table
@RestController
public class OrderController {

    @Autowired
    private TenantQueueService tenantQueueService;

    @PostMapping("/orders")
    public CompletableFuture<OrderResponse> createOrder(
            @RequestHeader("X-Tenant-Id") String tenantId,
            @RequestBody Order order) {

        // Get or create tenant-specific queue (uses templates)
```

```
            return tenantQueueService.getOrCreateTenantQueue(tenantId)
                .thenCompose(queue -> queue.send(order))
                .thenApply(msg -> new OrderResponse(msg.getId()));
        }
    }

    @Service
    public class TenantQueueService {

        private final Map<String, Queue<Order>> tenantQueues = new ConcurrentHashMap<>
    ();

        public CompletableFuture<Queue<Order>> getOrCreateTenantQueue(String tenantId)
    {
            if (tenantQueues.containsKey(tenantId)) {
                return CompletableFuture.completedFuture(tenantQueues.get(tenantId));
            }

            // Use templates to create tenant-specific queue
            QueueConfig config = new QueueConfig.Builder()
                .queueName("tenant_" + tenantId + "_orders")  // Dynamic name!
                .maxRetries(3)
                .build();

            return setupService.addQueue(setupId, config)
                .thenApply(v -> {
                    // Templates created:
                    //   - peegeeq.tenant_acme_orders table
                    //   - 5 dedicated indexes (only for this tenant)
                    //   - notification trigger
                    Queue<Order> queue = queueFactory.createQueue(
                        "tenant_" + tenantId + "_orders",
                        Order.class
                    );
                    tenantQueues.put(tenantId, queue);
                    return queue;
                });
        }
    }
```

Benefits in Production:

- ✅ **Performance Isolation**: Each tenant's queue has dedicated indexes
- ✅ **Data Isolation**: Tenant data in separate tables (security, compliance)
- ✅ **Independent Scaling**: Scale storage per tenant
- ✅ **Tenant Lifecycle**: Drop tenant's table when they churn
- ✅ **Query Performance**: Indexes only cover one tenant's data
- ✅ **Maintenance**: Vacuum/analyze per tenant independently

How Tests Validate This Production Feature

Integration tests verify that dynamic queue creation works correctly:

```
@Test
void testDynamicQueueCreation_ValidatesProductionCapability() {
    // This test validates the production feature of creating queues dynamically

    // Create first queue (simulates first tenant onboarding)
    QueueConfig config1 = new QueueConfig.Builder()
        .queueName("tenant_1_orders")
        .build();
    setupService.addQueue(setupId, config1).get();

    // Create second queue (simulates second tenant onboarding)
    QueueConfig config2 = new QueueConfig.Builder()
        .queueName("tenant_2_orders")
        .build();
    setupService.addQueue(setupId, config2).get();

    // Verify both queues exist independently
    verifyTableExists("peegeeq", "tenant_1_orders");
    verifyTableExists("peegeeq", "tenant_2_orders");

    // Verify each has dedicated indexes
    verifyIndexExists("idx_tenant_1_orders_topic");
    verifyIndexExists("idx_tenant_2_orders_topic");

    // This proves the production capability works!
}
```

Template Inheritance and Table Templates

Advanced Feature: PostgreSQL Table Inheritance

Templates use PostgreSQL's table inheritance to ensure consistency:

```sql
-- Step 1: Create template table (base/03a-table-queue-template.sql)
CREATE TABLE peegeeq.queue_template (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    payload JSONB NOT NULL,
    status VARCHAR(50) DEFAULT 'AVAILABLE',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    -- ... all standard queue columns
);

-- Step 2: Create specific queue (queue/01-create-table.sql)
CREATE TABLE {schema}.{queueName} (
    -- Inherits ALL columns from queue_template
) INHERITS ({schema}.queue_template);
```

```
-- Result: orders_queue has exact same structure as queue_template
-- Benefit: Schema consistency enforced at database level
```

Why This Matters:

1. **Schema Consistency**: All queues have identical structure
2. **Centralized Updates**: Change template once, all queues benefit
3. **Type Safety**: Database enforces column types
4. **Performance**: Inherited indexes and constraints

How Templates Work

Template Directory Structure

Templates are organized as **directories** containing numbered SQL files and a `.manifest` file:

```
src/main/resources/db/templates/
│
├── base/                           # Base schema (extensions, schemas, templates)
│   ├── .manifest                   # Execution order manifest
│   ├── 01a-extension-uuid.sql      # One statement per file
│   ├── 01b-extension-pgstat.sql
│   ├── 02a-schema-peegeeq.sql
│   ├── 02b-schema-bitemporal.sql
│   ├── 03a-table-queue-template.sql
│   ├── 03b-table-eventstore-template.sql
│   ├── 04a-table-queue-messages.sql
│   ├── 04b-table-outbox.sql
│   ├── 04c-table-dlq.sql
│   ├── 05a-index-queue-topic.sql
│   ├── ... (31 files total)
│   └── 09e-consumer-index-topic.sql
│
├── queue/                          # Per-queue tables (parameterized)
│   ├── .manifest
│   ├── 01-create-table.sql         # CREATE TABLE {queueName}
│   ├── 02a-index-topic.sql         # CREATE INDEX idx_{queueName}_topic
│   ├── 02b-index-visible.sql
│   ├── 02c-index-status.sql
│   ├── 03-function-notify.sql
│   ├── 04-create-trigger.sql
│   └── ... (8 files total)
│
└── eventstore/                     # Per-event-store tables (parameterized)
    ├── .manifest
    ├── 01-create-table.sql         # CREATE TABLE {tableName}
    ├── 02a-index-event-id.sql      # CREATE INDEX idx_{tableName}_event_id
    ├── 02b-index-valid-time.sql
    ├── 02c-index-transaction-time.sql
```

```
    ├── ... (13 files total)
    └── 04-create-trigger.sql
```

Why Directories Instead of Single Files?

Historical Context - The Vert.x Bug:

Originally, templates were single SQL files with multiple statements:

```
-- OLD APPROACH (BROKEN): peegeeq-template.sql
CREATE EXTENSION "uuid-ossp";        -- Statement 1
CREATE SCHEMA peegeeq;               -- Statement 2
CREATE TABLE queue_template (...);   -- Statement 3
CREATE INDEX idx_queue_topic ON ...; -- Statement 4
-- ... 28 more statements

-- Problem: Vert.x PostgreSQL client's .query().execute()
-- only executes the FIRST statement, silently ignores the rest!
```

Tests passed for months because:

1. PostgreSQL base image pre-included extensions
2. Idempotent SQL (`CREATE SCHEMA IF NOT EXISTS`) masked failures
3. Test execution order caused accidental success
4. Only verified 4 of 30+ database objects

See: VERTX_MULTISTATEMENT_SQL_BUG_ANALYSIS.md for complete analysis.

Solution: Split into one statement per file, execute sequentially:

```
-- NEW APPROACH (WORKS): Multiple files
-- base/01a-extension-uuid.sql
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- base/01b-extension-pgstat.sql
CREATE EXTENSION IF NOT EXISTS "pg_stat_statements";

-- base/02a-schema-peegeeq.sql
CREATE SCHEMA IF NOT EXISTS peegeeq;

-- base/02b-schema-bitemporal.sql
CREATE SCHEMA IF NOT EXISTS bitemporal;

-- ... one statement per file
```

The Manifest File

Purpose: Defines execution order of SQL files in a directory.

Format: Plain text, one filename per line, comments start with #:

```
# base/.manifest - Execution order for base template

# Step 1: Extensions (required for UUID and monitoring)
01a-extension-uuid.sql
01b-extension-pgstat.sql

# Step 2: Schemas
02a-schema-peegeeq.sql
02b-schema-bitemporal.sql

# Step 3: Template tables (inheritance base)
03a-table-queue-template.sql
03b-table-eventstore-template.sql

# Step 4: Core operational tables
04a-table-queue-messages.sql
04b-table-outbox.sql
04c-table-dlq.sql

# Step 5: Indexes for performance
05a-index-queue-topic.sql
05b-index-queue-visible.sql
...
```

Loading Process:

```
// SqlTemplateProcessor.loadTemplateFiles()

1. Look for .manifest file in template directory
2. If found:
   - Read filenames line by line
   - Load each SQL file in order
   - Return List<String> of SQL content
3. If not found:
   - Try to load directory name as single file (backward compatibility)
```

Template Parameter Substitution

Syntax: Parameters use {parameterName} placeholder syntax.

Example 1: Queue Template

```
-- queue/01-create-table.sql (BEFORE substitution)
CREATE TABLE {schema}.{queueName} (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
```

```
    payload JSONB NOT NULL,
    visible_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    status VARCHAR(50) DEFAULT 'AVAILABLE',
    -- ... more columns
) INHERITS ({schema}.queue_template);
```

```
// Application code
Map<String, String> params = Map.of(
    "schema", "peegeeq",
    "queueName", "orders_queue"
);

templateProcessor.applyTemplateReactive(connection, "queue", params);
```

```
-- Result (AFTER substitution)
CREATE TABLE peegeeq.orders_queue (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    payload JSONB NOT NULL,
    visible_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    status VARCHAR(50) DEFAULT 'AVAILABLE',
    -- ... more columns
) INHERITS (peegeeq.queue_template);
```

Example 2: Queue Index Template

```
-- queue/02a-index-topic.sql (BEFORE)
CREATE INDEX idx_{queueName}_topic
    ON {schema}.{queueName}(topic, visible_at, status);
```

```
-- Result (AFTER substitution with params above)
CREATE INDEX idx_orders_queue_topic
    ON peegeeq.orders_queue(topic, visible_at, status);
```

Example 3: Event Store Template

```
-- eventstore/01-create-table.sql (BEFORE)
CREATE TABLE {schema}.{tableName} (
    id BIGSERIAL PRIMARY KEY,
    event_id VARCHAR(255) NOT NULL,
    event_type VARCHAR(255) NOT NULL,
```

```
    valid_time TIMESTAMP WITH TIME ZONE NOT NULL,
    transaction_time TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    payload JSONB NOT NULL,
    -- ... more columns
) INHERITS ({schema}.event_store_template);

-- Notification trigger (uses parameter)
CREATE TRIGGER {tableName}_notify_trigger
    AFTER INSERT ON {schema}.{tableName}
    FOR EACH ROW
    EXECUTE FUNCTION notify_event_insert('{notificationPrefix}');
```

```
// Application code
Map<String, String> params = Map.of(
    "schema", "bitemporal",
    "tableName", "order_event_log",
    "notificationPrefix", "order_event_"
);

templateProcessor.applyTemplateReactive(connection, "eventstore", params);
```

```
-- Result (AFTER substitution)
CREATE TABLE bitemporal.order_event_log (
    id BIGSERIAL PRIMARY KEY,
    event_id VARCHAR(255) NOT NULL,
    event_type VARCHAR(255) NOT NULL,
    valid_time TIMESTAMP WITH TIME ZONE NOT NULL,
    transaction_time TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    payload JSONB NOT NULL,
    -- ... more columns
) INHERITS (bitemporal.event_store_template);

CREATE TRIGGER order_event_log_notify_trigger
    AFTER INSERT ON bitemporal.order_event_log
    FOR EACH ROW
    EXECUTE FUNCTION notify_event_insert('order_event_');
```

SqlTemplateProcessor Implementation

Core Logic:

```
public class SqlTemplateProcessor {

    /**
     * Apply SQL template directory with parameter substitution.
     *
     * CRITICAL: Vert.x PostgreSQL client's .query().execute() only executes
```

```
     * the FIRST statement in multi-statement SQL. Templates are organized as
     * directories with numbered files to work around this limitation.
     */
    public Future<Void> applyTemplateReactive(
            SqlConnection connection,
            String templateDir,
            Map<String, String> parameters) {

        // Step 1: Load all SQL files from directory (using .manifest)
        List<String> sqlFiles = loadTemplateFiles(templateDir);

        // Step 2: Execute files sequentially using Future composition
        Future<Void> chain = Future.succeededFuture();

        for (String sqlContent : sqlFiles) {
            // Step 3: Substitute parameters in SQL
            String processedSql = processTemplate(sqlContent, parameters);

            // Step 4: Chain execution (wait for previous to complete)
            chain = chain.compose(v ->
                connection.query(processedSql).execute()
                    .mapEmpty()
            );
        }

        return chain;
    }

    /**
     * Simple string replacement for parameters.
     */
    private String processTemplate(String template, Map<String, String>
parameters) {
        String result = template;
        for (Map.Entry<String, String> entry : parameters.entrySet()) {
            // Replace {paramName} with actual value
            result = result.replace("{" + entry.getKey() + "}", entry.getValue());
        }
        return result;
    }

    /**
     * Load SQL files from directory using .manifest file.
     */
    private List<String> loadTemplateFiles(String templateDir) throws IOException
{
        // 1. Try to load .manifest file
        String manifestPath = "/db/templates/" + templateDir + "/.manifest";
        InputStream manifestStream = getClass().getResourceAsStream(manifestPath);

        if (manifestStream != null) {
            // Read manifest and load files in order
            List<String> fileNames = readManifest(manifestStream);
            List<String> sqlContents = new ArrayList<>();
```

```
            for (String fileName : fileNames) {
                String filePath = "/db/templates/" + templateDir + "/" + fileName;
                String content = loadFile(filePath);
                sqlContents.add(content);
            }

            return sqlContents;
        } else {
            // No manifest - try single file (backward compatibility)
            String filePath = "/db/templates/" + templateDir;
            String content = loadFile(filePath);
            return List.of(content);
        }
    }
}
```

Template Execution Flow

Complete Example: Creating Orders Queue

```
// Step 1: Test defines what it needs
QueueConfig queueConfig = new QueueConfig.Builder()
    .queueName("orders_queue")
    .maxRetries(3)
    .build();

DatabaseSetupRequest request = new DatabaseSetupRequest.Builder()
    .setupId("test_12345")
    .databaseConfig(dbConfig)
    .queues(List.of(queueConfig))
    .build();

// Step 2: Setup service creates database and applies templates
setupService.createCompleteSetup(request).get();
```

Behind the Scenes:

```
┌─────────────────────────────────────────────────────────┐
│  PeeGeeQDatabaseSetupService.createCompleteSetup()      │
└─────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────┐
│  1. Create Database                                     │
│     CREATE DATABASE test_12345                          │
└─────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────┐
│  2. Apply Base Template (31 files)                      │
```

```
│                                                              │
│     SqlTemplateProcessor.applyTemplateReactive(             │
│         connection, "base", Map.of()                        │
│     )                                                        │
│                                                              │
│     Executes:                                                │
│     • 01a-extension-uuid.sql                                 │
│     • 01b-extension-pgstat.sql                               │
│     • 02a-schema-peegeeq.sql                                 │
│     • ... (31 files sequentially)                            │
└──────────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────────┐
│  3. Apply Queue Template (8 files, parameterized)            │
│                                                              │
│     SqlTemplateProcessor.applyTemplateReactive(              │
│         connection,                                          │
│         "queue",                                             │
│         Map.of(                                              │
│             "schema", "peegeeq",                             │
│             "queueName", "orders_queue"                      │
│         )                                                    │
│     )                                                        │
│                                                              │
│     Executes (after parameter substitution):                 │
│     • CREATE TABLE peegeeq.orders_queue (...)                │
│     • CREATE INDEX idx_orders_queue_topic (...)              │
│     • CREATE INDEX idx_orders_queue_visible (...)            │
│     • ... (8 files sequentially)                             │
└──────────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────────┐
│  4. Create PeeGeeQManager                                    │
│     • Connects to test_12345 database                        │
│     • Creates QueueFactory for orders_queue                  │
│     • Returns DatabaseSetupResult                            │
└──────────────────────────────────────────────────────────────┘
```

Template Best Practices

1. One Statement Per File

```
-- ✅ GOOD: 01-create-table.sql
CREATE TABLE {schema}.{queueName} (...);

-- ✅ GOOD: 02-create-index.sql
CREATE INDEX idx_{queueName}_topic ON {schema}.{queueName}(topic);

-- ❌ BAD: create-queue.sql
CREATE TABLE {schema}.{queueName} (...);
```

```
CREATE INDEX idx_{queueName}_topic ON {schema}.{queueName}(topic);
-- Only first statement executes due to Vert.x limitation!
```

2. Use Idempotent SQL

```
-- ✅ GOOD: Safe to run multiple times
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE SCHEMA IF NOT EXISTS peegeeq;
CREATE TABLE IF NOT EXISTS {schema}.queue_template (...);

-- ❌ BAD: Fails on second execution
CREATE EXTENSION "uuid-ossp";
CREATE SCHEMA peegeeq;
CREATE TABLE {schema}.queue_template (...);
```

3. Meaningful File Naming

```
✅ GOOD:
01a-extension-uuid.sql        # Clear: UUID extension
01b-extension-pgstat.sql      # Clear: pg_stat_statements
02a-schema-peegeeq.sql        # Clear: peegeeq schema
03a-table-queue-template.sql  # Clear: queue template table

❌ BAD:
file1.sql
file2.sql
setup.sql
```

4. Sequential Numbering with Grouping

```
✅ GOOD: Groups with sub-ordering
01a-extension-uuid.sql
01b-extension-pgstat.sql      # Same group (extensions)
02a-schema-peegeeq.sql
02b-schema-bitemporal.sql     # Same group (schemas)
03a-table-queue-template.sql
03b-table-eventstore-template.sql  # Same group (templates)

❌ BAD: No grouping visible
01-extension-uuid.sql
02-extension-pgstat.sql
03-schema-peegeeq.sql
04-schema-bitemporal.sql
```

5. Document Parameters in Comments

```
-- queue/01-create-table.sql
-- Parameters:
--   {schema} - Schema name (e.g., "peegeeq")
--   {queueName} - Queue table name (e.g., "orders_queue")

CREATE TABLE {schema}.{queueName} (
    ...
) INHERITS ({schema}.queue_template);
```

Architecture Summary

Three Database Setup Approaches

1. Flyway Migrations (Production & Development)

- **Module**: `peegeeq-migrations`
- **Method**: SQL migration scripts with version tracking
- **Files**: `V001__Create_Base_Tables.sql` (576 lines, complete schema)
- **Usage**: CI/CD pipelines, docker-compose, Maven commands
- **Advantages**:
  - Version control for schema changes
  - Rollback capability
  - Audit trail
  - Industry standard

2. Template-Based Setup (Integration Tests)

- **Class**: `PeeGeeQDatabaseSetupService` (825 lines)
- **Method**: SQL templates with parameter substitution
- **Files**: 52 SQL files in 3 directories (`base/`, `queue/`, `eventstore/`)
- **Usage**: Integration tests creating dynamic test databases
- **Advantages**:
  - Dynamic database creation
  - Parameterized table names
  - Isolated test environments
  - Per-test database cleanup

3. Direct SQL Injection (Unit Tests)

- **Class**: `PeeGeeQTestSchemaInitializer` (747 lines)
- **Method**: JDBC Statement execution of SQL scripts
- **Files**: Inline SQL strings for each schema component
- **Usage**: Unit tests requiring minimal schema setup
- **Advantages**:

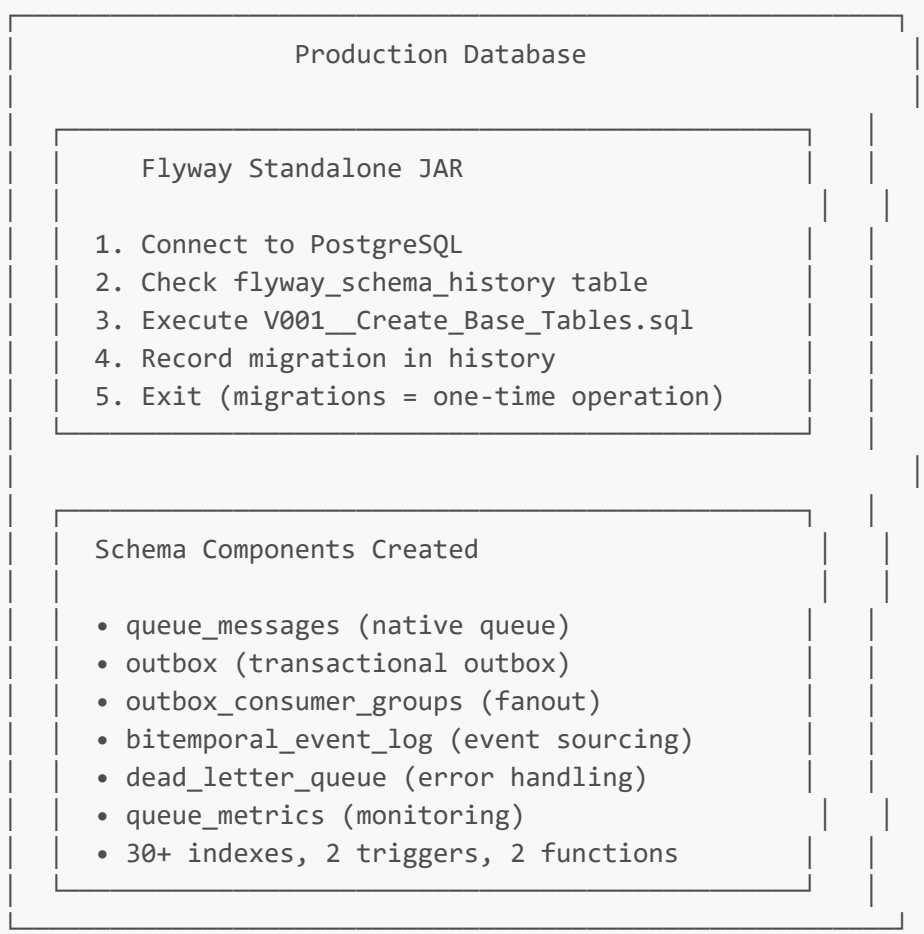
- Fast startup
- Component-level granularity
- No external dependencies
- Simple and predictable

Production Setup

Overview

Production databases are initialized using **Flyway migrations** from the `peegeeq-migrations` module. This approach provides version control, audit trails, and rollback capabilities.

Architecture



Deployment Process

Step 1: Build Migration JAR

```
cd peegeeq-migrations
mvn clean package -DskipTests
```

Output: `target/peegeeq-migrations.jar` (standalone executable)

Step 2: Run Migrations in CI/CD

GitLab CI Example:

```
# .gitlab-ci.yml
stages:
  - build
  - migrate
  - deploy

migrate-production:
  stage: migrate
  script:
    - export DB_JDBC_URL=$PROD_DB_URL
    - export DB_USER=$PROD_DB_USER
    - export DB_PASSWORD=$PROD_DB_PASSWORD
    - java -jar peegeeq-migrations/target/peegeeq-migrations.jar migrate
  only:
    - main

deploy-application:
  stage: deploy
  dependencies:
    - migrate-production
  script:
    - # Deploy PeeGeeQ application
  only:
    - main
```

GitHub Actions Example:

```
# .github/workflows/deploy.yml
name: Deploy Production

on:
  push:
    branches: [main]

jobs:
  migrate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Java
        uses: actions/setup-java@v3
        with:
          distribution: 'temurin'
          java-version: '23'
```

```
      - name: Build Migration JAR
        run: |
          cd peegeeq-migrations
          mvn package -DskipTests
      - name: Run Migrations
        env:
          DB_JDBC_URL: ${{ secrets.PROD_DB_URL }}
          DB_USER: ${{ secrets.PROD_DB_USER }}
          DB_PASSWORD: ${{ secrets.PROD_DB_PASSWORD }}
        run: |
          java -jar peegeeq-migrations/target/peegeeq-migrations.jar migrate

  deploy:
    needs: migrate
    runs-on: ubuntu-latest
    steps:
      - name: Deploy Application
        run: |
          # Application deployment commands
```

Step 3: Verify Migrations

```
# Check migration status
java -jar peegeeq-migrations.jar info \
  -url="jdbc:postgresql://prod-host:5432/peegeeq" \
  -user="produser" \
  -password="$PROD_PASSWORD"

# Expected output:
# +---------+-----------------------+---------------------+---------+
# | Version | Description           | Installed on        | State   |
# +---------+-----------------------+---------------------+---------+
# | 1       | Create Base Tables    | 2025-11-30 10:15:00 | Success |
# +---------+-----------------------+---------------------+---------+
```

Production Schema Components

When migrations complete successfully, the following components exist:

Core Tables (7 tables)

- `schema_version` - Schema version tracking
- `queue_messages` - Native high-performance queue (10k+ msg/sec)
- `outbox` - Transactional outbox pattern (5k+ msg/sec)
- `outbox_consumer_groups` - Consumer group fanout
- `message_processing` - INSERT-only lock-free processing
- `dead_letter_queue` - Failed message handling
- `bitemporal_event_log` - Bi-temporal event sourcing (3k+ msg/sec)

Monitoring Tables (2 tables)

- `queue_metrics` - Queue performance metrics
- `connection_pool_metrics` - Connection pool monitoring

Indexes (30+ indexes)

- Performance indexes for all tables
- Composite indexes for bi-temporal queries
- GIN indexes for JSONB payload/header queries
- Unique indexes for constraint enforcement

Functions & Triggers (4 objects)

- Trigger functions for automatic timestamp updates
- Notification triggers for real-time event streaming

Views (3 views)

- `bitemporal_current_state` - Current event state
- `bitemporal_latest_events` - Most recent events
- `bitemporal_event_stats` - Event statistics

Production Configuration

Database Requirements

- **PostgreSQL**: 15.13 or higher
- **Extensions Required**:
  - `uuid-ossp` - UUID generation
  - `pg_stat_statements` - Query performance monitoring
- **Minimum Permissions**:
  - `CREATE EXTENSION` (for uuid-ossp and pg_stat_statements)
  - `CREATE SCHEMA` (for peegeeq and bitemporal schemas)
  - `CREATE TABLE`, `CREATE INDEX`, `CREATE TRIGGER`
  - `SELECT`, `INSERT`, `UPDATE`, `DELETE` on all tables

Connection Configuration

```
# Production connection settings
flyway.url=jdbc:postgresql://prod-host:5432/peegeeq_prod
flyway.user=peegeeq_prod_user
flyway.password=${PROD_DB_PASSWORD}

# Schema Configuration (CUSTOMIZABLE)
# Default: peegeeq,bitemporal
# Custom examples:
#   - Single schema: myapp
```

```
#   - Tenant-specific: tenant_a_queue,tenant_a_events
#   - Namespace: acme_messaging,acme_events
flyway.schemas=peegeeq,bitemporal

flyway.locations=classpath:db/migration
flyway.baselineOnMigrate=true
flyway.validateOnMigrate=true
```

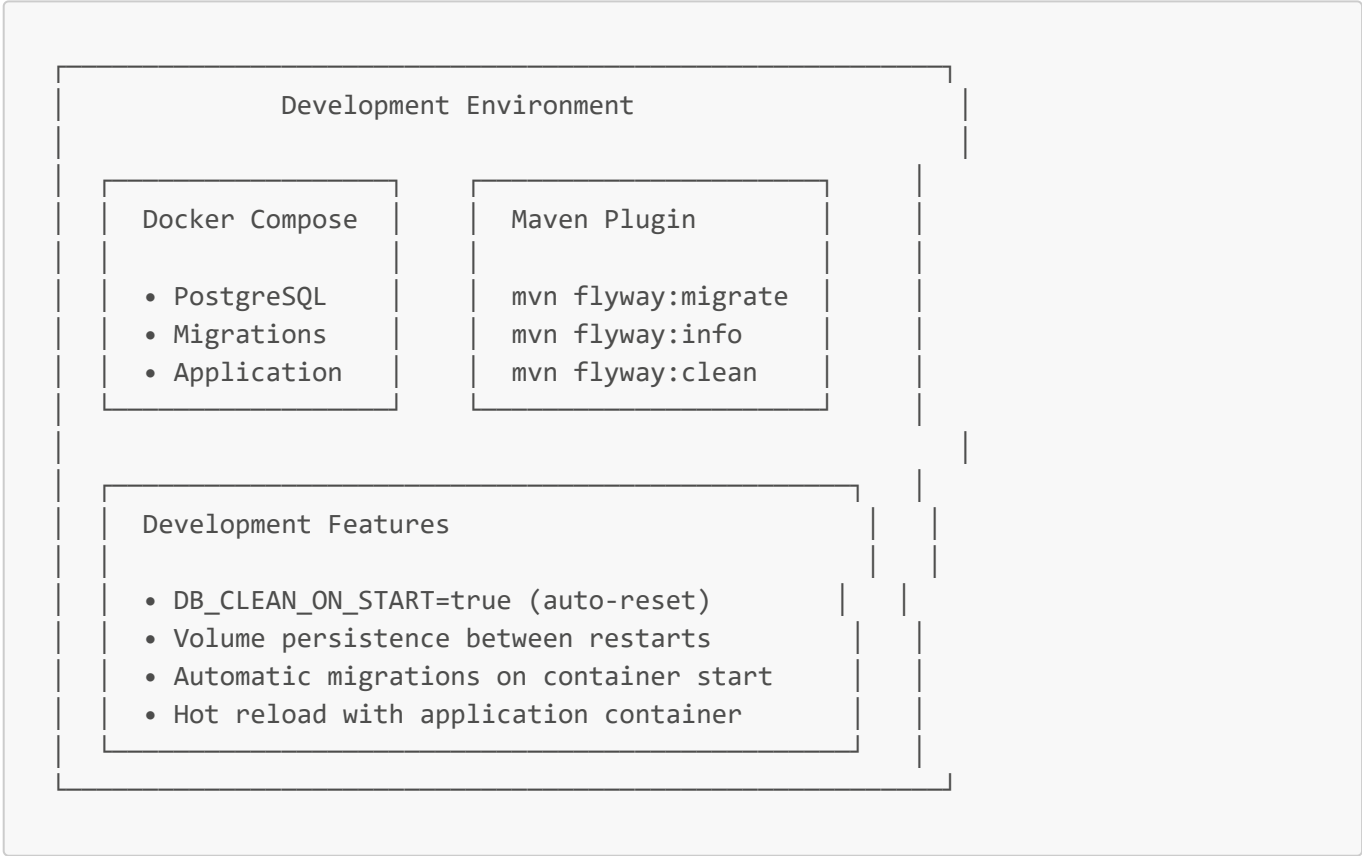
Development Setup

Overview

Development environments use **Flyway migrations** like production, but with additional tooling for rapid iteration:

- Docker Compose for automated local setup
- Maven plugin for manual migration control
- Development scripts for database reset

Architecture



Option 1: Docker Compose (Recommended)

File: `peegeeq-migrations/docker-compose.dev.yml`

Architecture

```yaml
services:
  postgres:
    image: postgres:15.13-alpine3.20
    ports: ["5432:5432"]
    environment:
      POSTGRES_DB: peegeeq_dev
      POSTGRES_USER: peegeeq_dev
      POSTGRES_PASSWORD: peegeeq_dev

  migrations:
    build: ./Dockerfile
    depends_on:
      postgres: {condition: service_healthy}
    environment:
      DB_JDBC_URL: jdbc:postgresql://postgres:5432/peegeeq_dev
      DB_USER: peegeeq_dev
      DB_PASSWORD: peegeeq_dev
      DB_CLEAN_ON_START: "true"  # Clean database before migrations
    command: migrate
    restart: "no"  # Run once and exit
```

Usage

```bash
# Start PostgreSQL + run migrations
cd peegeeq-migrations
docker-compose -f docker-compose.dev.yml up

# Migrations run automatically, then container exits
# PostgreSQL continues running with complete schema

# Reset database (clean + re-migrate)
docker-compose -f docker-compose.dev.yml down -v
docker-compose -f docker-compose.dev.yml up
```

Advantages

- Automated setup (PostgreSQL + migrations)
- Clean environment every time
- Matches production container behavior
- No local Java/Maven required

Option 2: Maven Plugin

File: `peegeeq-migrations/pom.xml`

Configuration

```
<plugin>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-maven-plugin</artifactId>
    <version>10.23.0</version>
    <configuration>
        <url>jdbc:postgresql://localhost:5432/peegeeq_dev</url>
        <user>peegeeq_dev</user>
        <password>peegeeq_dev</password>
        <schemas>
            <schema>peegeeq</schema>
            <schema>bitemporal</schema>
        </schemas>
        <locations>
            <location>classpath:db/migration</location>
        </locations>
    </configuration>
</plugin>
```

Usage

```
# Run migrations
cd peegeeq-migrations
mvn flyway:migrate

# Check migration status
mvn flyway:info

# Clean database (WARNING: deletes all data)
mvn flyway:clean

# Reset database (clean + migrate)
mvn flyway:clean flyway:migrate

# Validate migrations
mvn flyway:validate
```

Advantages

- Fine-grained control
- Manual migration timing
- Easy to integrate with IDE
- No Docker required

Option 3: Local Docker PostgreSQL

File: `docker-compose-local-dev.yml` (root directory)

Configuration

```
services:
  postgres:
    image: postgres:15.13-alpine3.20
    ports: ["5432:5432"]
    environment:
      POSTGRES_DB: peegeeq_dev
      POSTGRES_USER: peegeeq_dev
      POSTGRES_PASSWORD: peegeeq_dev
    volumes:
      - postgres_local_data:/var/lib/postgresql/data
```

Usage

```
# Start PostgreSQL only
docker-compose -f docker-compose-local-dev.yml up -d

# Run migrations manually
cd peegeeq-migrations
mvn flyway:migrate

# Application connects to localhost:5432
```

Advantages

- Persistent data between sessions
- Controlled migration timing
- Faster restarts (no re-migration)
- IDE-friendly development

Development Configuration

Local Settings

```
# peegeeq-migrations/src/main/resources/application-dev.properties
flyway.url=jdbc:postgresql://localhost:5432/peegeeq_dev
flyway.user=peegeeq_dev
flyway.password=peegeeq_dev
flyway.cleanDisabled=false  # Allow clean in dev (NEVER in prod)
flyway.baselineOnMigrate=true
flyway.outOfOrder=true
```

Environment Variables (Alternative)

```
export DB_JDBC_URL="jdbc:postgresql://localhost:5432/peegeeq_dev"
export DB_USER="peegeeq_dev"
export DB_PASSWORD="peegeeq_dev"
export DB_CLEAN_ON_START="true"  # Optional: clean before migrations
```

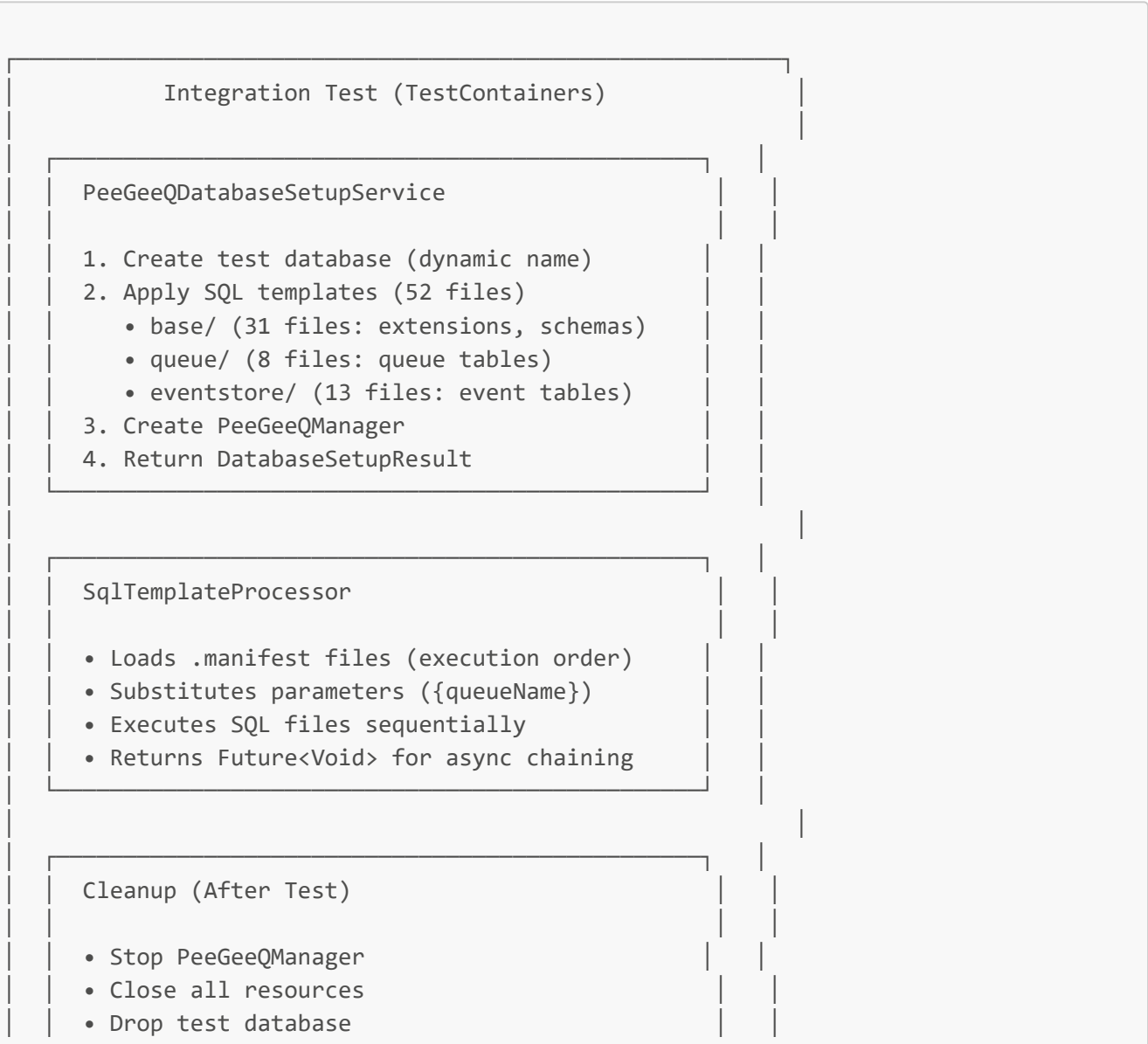
Test Setup

Overview

Test environments use **programmatic setup** instead of Flyway migrations. This allows:

- Dynamic database creation per test class
- Parameterized schema (e.g., unique table names)
- Fast cleanup and isolation
- No migration history pollution

Integration Tests Architecture



```
│  └──────────────────────────────────────────────┘  │
└──────────────────────────────────────────────────┘
```

Integration Test Pattern

Example Test Class

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class QueueIntegrationTest {

    @Container
    private static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>
("postgres:15.13-alpine3.20")
        .withDatabaseName("testdb")
        .withUsername("test")
        .withPassword("test");

    private PeeGeeQDatabaseSetupService setupService;
    private DatabaseSetupResult setupResult;

    @BeforeAll
    void setupDatabase() throws Exception {
        setupService = new PeeGeeQDatabaseSetupService();

        // Create dynamic test database
        DatabaseConfig dbConfig = new DatabaseConfig.Builder()
            .host(postgres.getHost())
            .port(postgres.getFirstMappedPort())
            .databaseName("test_db_" + UUID.randomUUID())  // Unique per test
            .username(postgres.getUsername())
            .password(postgres.getPassword())
            .schema("peegeeq")
            .build();

        // Define queues to create
        List<QueueConfig> queues = List.of(
            new QueueConfig.Builder()
                .queueName("test_queue_1")
                .maxRetries(3)
                .build()
        );

        // Define event stores to create
        List<EventStoreConfig> eventStores = List.of(
            new EventStoreConfig.Builder()
                .eventStoreName("test_events")
                .tableName("test_event_log")
                .notificationPrefix("test_event_")
                .build()
        );
```

```
        // Create complete setup
        DatabaseSetupRequest request = new DatabaseSetupRequest.Builder()
            .setupId("test_setup_" + UUID.randomUUID())
            .databaseConfig(dbConfig)
            .queues(queues)
            .eventStores(eventStores)
            .build();

        setupResult = setupService.createCompleteSetup(request).get(30,
TimeUnit.SECONDS);

        // Verify setup successful
        assertEquals(DatabaseSetupStatus.READY, setupResult.getStatus());
    }

    @AfterAll
    void cleanupDatabase() throws Exception {
        if (setupResult != null) {
            setupService.destroySetup(setupResult.getSetupId()).get(10,
TimeUnit.SECONDS);
        }
    }

    @Test
    void testQueueOperations() {
        // Use setupResult.getQueueFactories() to get queue instances
        // Test queue send/receive operations
    }
}
```

Template Structure

Templates are organized as directories with manifest files:

```
src/main/resources/db/templates/
├── base/
│   ├── .manifest                    # Lists files in order
│   ├── 01a-extension-uuid.sql       # CREATE EXTENSION "uuid-ossp"
│   ├── 01b-extension-pgstat.sql     # CREATE EXTENSION "pg_stat_statements"
│   ├── 02a-schema-peegeeq.sql       # CREATE SCHEMA peegeeq
│   ├── 02b-schema-bitemporal.sql    # CREATE SCHEMA bitemporal
│   ├── 03a-table-queue-template.sql # CREATE TABLE peegeeq.queue_template
│   ├── ... (31 files total)
│   └── 09e-consumer-index-topic.sql
│
├── queue/
│   ├── .manifest
│   ├── 01-create-table.sql          # CREATE TABLE {schema}.{queueName}
│   ├── 02a-index-topic.sql          # CREATE INDEX idx_{queueName}_topic
│   ├── ... (8 files total)
│   └── 04-create-trigger.sql
```

```
│
└── eventstore/
    ├── .manifest
    ├── 01-create-table.sql          # CREATE TABLE {schema}.{tableName}
    ├── 02a-index-event-id.sql       # CREATE INDEX idx_{tableName}_event_id
    ├── ... (13 files total)
    └── 04-create-trigger.sql
```

Template Parameters

Templates support parameter substitution:

```
-- queue/01-create-table.sql
CREATE TABLE {schema}.{queueName} (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    payload JSONB NOT NULL,
    -- ... columns
) INHERITS ({schema}.queue_template);

-- Usage in code:
Map<String, String> params = Map.of(
    "schema", "peegeeq",
    "queueName", "orders_queue"
);
templateProcessor.applyTemplateReactive(connection, "queue", params);

-- Results in:
CREATE TABLE peegeeq.orders_queue (...) INHERITS (peegeeq.queue_template);
```

Unit Tests Architecture

```
┌──────────────────────────────────────────────────────────┐
│              Unit Test (TestContainers)                  │
│                                                          │
│  ┌──────────────────────────────────────────────────┐    │
│  │  PeeGeeQTestSchemaInitializer                    │    │
│  │                                                  │    │
│  │  1. Connect via JDBC                             │    │
│  │  2. Execute inline SQL statements                │    │
│  │  3. Component-based initialization               │    │
│  │     • SchemaComponent.OUTBOX                     │    │
│  │     • SchemaComponent.NATIVE_QUEUE               │    │
│  │     • SchemaComponent.BITEMPORAL                 │    │
│  │  4. Return immediately (synchronous)             │    │
│  └──────────────────────────────────────────────────┘    │
│                                                          │
│  ┌──────────────────────────────────────────────────┐    │
│  │  Schema Components Available                     │    │
```

```
│   │                                                   │   │
│   │  • SCHEMA_VERSION                                 │   │
│   │  • OUTBOX                                         │   │
│   │  • NATIVE_QUEUE                                   │   │
│   │  • DEAD_LETTER_QUEUE                              │   │
│   │  • BITEMPORAL                                     │   │
│   │  • METRICS                                        │   │
│   │  • CONSUMER_GROUP_FANOUT                          │   │
│   │  • QUEUE_ALL (outbox + native + dlq)              │   │
│   │  • ALL (everything)                               │   │
│   └───────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────────┘
```

Example Unit Test

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class OutboxServiceTest {

    @Container
    private static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>
("postgres:15.13-alpine3.20");

    @BeforeAll
    void setupSchema() {
        // Initialize only what we need for this test
        PeeGeeQTestSchemaInitializer.initializeSchema(
            postgres,
            SchemaComponent.OUTBOX,
            SchemaComponent.DEAD_LETTER_QUEUE
        );
    }

    @AfterEach
    void cleanupTestData() {
        // Clean test data but keep schema
        PeeGeeQTestSchemaInitializer.cleanupTestData(
            postgres,
            SchemaComponent.OUTBOX
        );
    }

    @Test
    void testOutboxInsertion() {
        // Test outbox operations
    }
}
```

Available Schema Components

```
public enum SchemaComponent {
    SCHEMA_VERSION,           // Schema version tracking table
    OUTBOX,                   // Outbox pattern tables + indexes
    NATIVE_QUEUE,             // Native queue tables + indexes
    DEAD_LETTER_QUEUE,        // Dead letter queue table
    BITEMPORAL,               // Bi-temporal event log + views + triggers
    METRICS,                  // Metrics and monitoring tables
    CONSUMER_GROUP_FANOUT,    // Consumer group fanout tables
    QUEUE_ALL,                // All queue-related (OUTBOX + NATIVE + DLQ)
    ALL                       // Everything
}
```

Test Configuration Best Practices

1. Database Isolation

```
// Create unique database per test class
String dbName = "test_" + getClass().getSimpleName() + "_" + UUID.randomUUID();
```

2. Resource Cleanup

```
@AfterAll
void cleanup() {
    // Always cleanup in @AfterAll
    if (setupResult != null) {
        setupService.destroySetup(setupResult.getSetupId()).get();
    }
}
```

3. Parallel Test Execution

```
// Use unique setup IDs for parallel tests
String setupId = "test_" + Thread.currentThread().getId() + "_" +
UUID.randomUUID();
```

Dynamic Queue Creation at Runtime

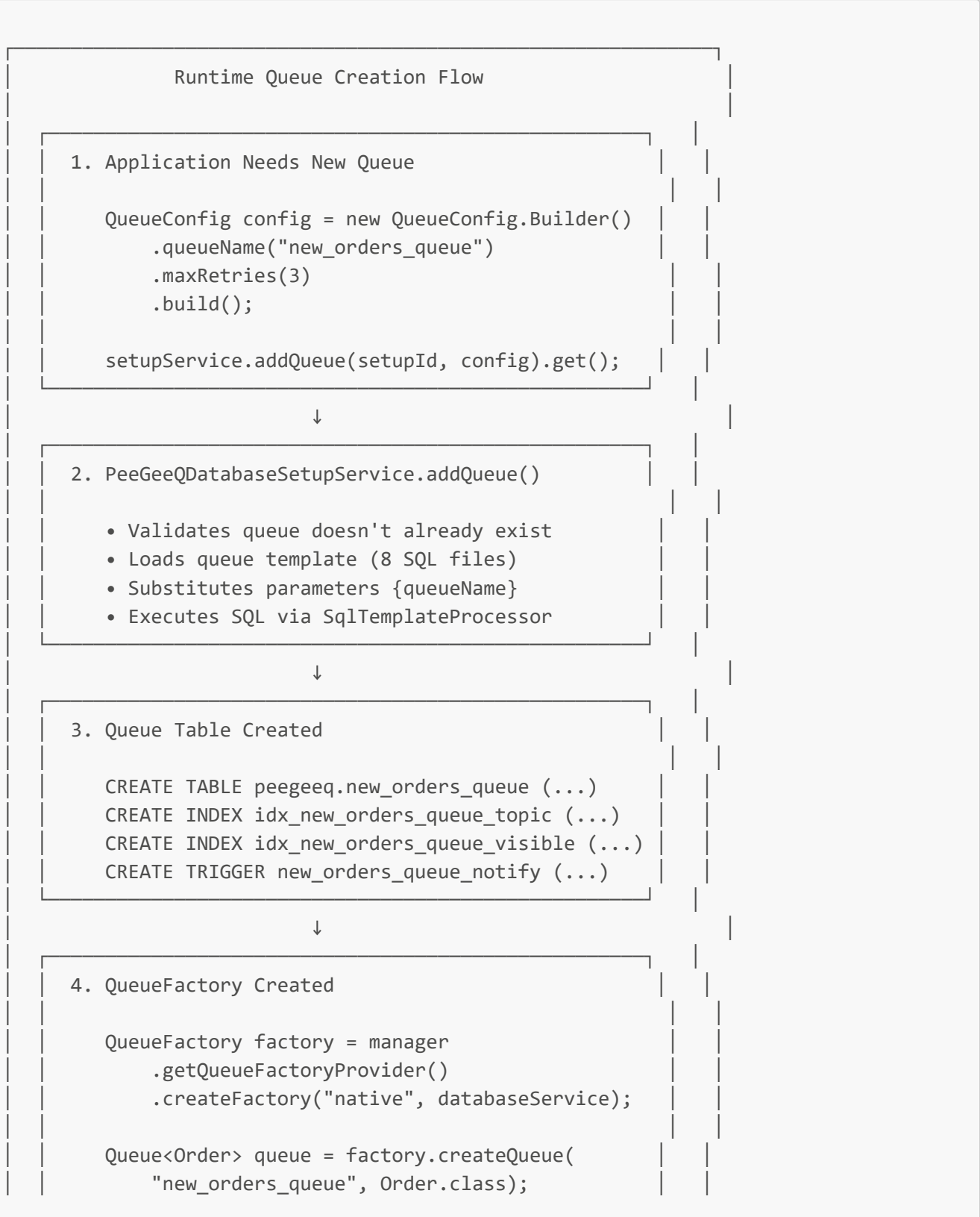
Overview

PeeGeeQ supports **dynamic queue creation at runtime** - the ability to create new queue tables on-demand without restarting the application or running migrations. This is critical for:

- **Multi-tenant systems**: Create queues dynamically when onboarding new tenants
- **Feature flags**: Enable new queues when features are activated
- **Dynamic workflows**: Create queues based on user-defined business processes
- **Auto-scaling**: Add queues dynamically as load increases
- **Plugin architectures**: Third-party plugins can create their own queues

How Dynamic Queue Creation Works

Architecture



```
│  └──────────────────────────────────────────────────────┘   │
│                          ↓                                 │
│  ┌──────────────────────────────────────────────────────┐   │
│  │  5. Queue Ready for Use                              │   │
│  │                                                      │   │
│  │     queue.send(new Order(...));                      │   │
│  │     queue.receive().thenAccept(msg -> ...);          │   │
│  └──────────────────────────────────────────────────────┘   │
└─────────────────────────────────────────────────────────────┘
```

Production Usage

Example 1: Multi-Tenant Queue Creation

```
/**
 * Service that creates tenant-specific queues on demand
 */
public class TenantQueueService {

    private final PeeGeeQDatabaseSetupService setupService;
    private final String setupId;
    private final Map<String, QueueFactory> tenantQueues = new ConcurrentHashMap<>
();

    public CompletableFuture<Queue<Order>> getOrCreateTenantQueue(String tenantId)
{
        String queueName = "tenant_" + tenantId + "_orders";

        // Check if queue already exists
        if (tenantQueues.containsKey(queueName)) {
            return CompletableFuture.completedFuture(
                tenantQueues.get(queueName).createQueue(queueName, Order.class)
            );
        }

        // Create new queue dynamically
        QueueConfig config = new QueueConfig.Builder()
            .queueName(queueName)
            .maxRetries(3)
            .retryDelayMillis(5000)
            .build();

        return setupService.addQueue(setupId, config)
            .thenApply(v -> {
                // Create queue factory
                QueueFactory factory = createQueueFactory();
                tenantQueues.put(queueName, factory);

                logger.info("Created dynamic queue for tenant: {}", tenantId);
                return factory.createQueue(queueName, Order.class);
            });
```

```
    }
}

// Usage in application
@RestController
@RequestMapping("/api/tenants")
public class TenantController {

    @Autowired
    private TenantQueueService tenantQueueService;

    @PostMapping("/{tenantId}/orders")
    public CompletableFuture<OrderResponse> createOrder(
            @PathVariable String tenantId,
            @RequestBody OrderRequest request) {

        // Get or create tenant-specific queue
        return tenantQueueService.getOrCreateTenantQueue(tenantId)
            .thenCompose(queue -> {
                Order order = new Order(tenantId, request);
                return queue.send(order);
            })
            .thenApply(msg -> new OrderResponse(msg.getId()));
    }
}
```

Example 2: Feature Flag Activated Queue

```
/**
 * Creates queues dynamically when features are enabled
 */
public class FeatureQueueManager {

    private final PeeGeeQDatabaseSetupService setupService;
    private final String setupId;
    private final Set<String> activeQueues = ConcurrentHashMap.newKeySet();

    @EventListener
    public void onFeatureEnabled(FeatureEnabledEvent event) {
        String featureName = event.getFeatureName();
        String queueName = featureName.toLowerCase() + "_queue";

        if (activeQueues.contains(queueName)) {
            logger.info("Queue {} already exists", queueName);
            return;
        }

        logger.info("Creating queue for newly enabled feature: {}", featureName);

        QueueConfig config = new QueueConfig.Builder()
            .queueName(queueName)
```

```
            .maxRetries(5)
            .build();

        setupService.addQueue(setupId, config)
            .thenRun(() -> {
                activeQueues.add(queueName);
                logger.info("Queue {} created and ready", queueName);
            })
            .exceptionally(ex -> {
                logger.error("Failed to create queue for feature: {}",
featureName, ex);
                return null;
            });
    }
}
```

Example 3: Dynamic Workflow Queues

```
/**
 * Creates queues for user-defined workflows
 */
public class WorkflowQueueService {

    public CompletableFuture<Void> createWorkflowQueues(Workflow workflow) {
        List<CompletableFuture<Void>> queueCreations = new ArrayList<>();

        // Create a queue for each step in the workflow
        for (WorkflowStep step : workflow.getSteps()) {
            String queueName = String.format("workflow_%s_step_%s",
                workflow.getId(), step.getName());

            QueueConfig config = new QueueConfig.Builder()
                .queueName(queueName)
                .maxRetries(step.getMaxRetries())
                .build();

            CompletableFuture<Void> future = setupService.addQueue(setupId,
config);
            queueCreations.add(future);
        }

        // Wait for all queues to be created
        return CompletableFuture.allOf(
            queueCreations.toArray(new CompletableFuture[0])
        ).thenRun(() -> {
            logger.info("Created {} queues for workflow: {}",
                queueCreations.size(), workflow.getId());
        });
    }
}
```

Development Usage

Dynamic queue creation is also used during development for:

Hot Reload Support

```
/**
 * Development mode: Create queues on-the-fly without restart
 */
@Profile("development")
@Component
public class DevQueueManager {

    @PostMapping("/dev/queues/create")
    public CompletableFuture<String> createDevQueue(@RequestParam String name) {
        QueueConfig config = new QueueConfig.Builder()
            .queueName("dev_" + name)
            .maxRetries(1)
            .build();

        return setupService.addQueue(setupId, config)
            .thenApply(v -> "Queue created: dev_" + name);
    }
}
```

Integration Test Usage

Dynamic queue creation is extensively used in integration tests:

```
@Test
void testDynamicQueueCreation() throws Exception {
    // Start with no queues
    DatabaseSetupRequest request = new DatabaseSetupRequest.Builder()
        .setupId("test_dynamic")
        .databaseConfig(dbConfig)
        .queues(List.of())  // No initial queues
        .build();

    DatabaseSetupResult result = setupService.createCompleteSetup(request).get();

    // Later, add queue dynamically
    QueueConfig queueConfig = new QueueConfig.Builder()
        .queueName("dynamic_test_queue")
        .maxRetries(3)
        .build();

    setupService.addQueue("test_dynamic", queueConfig).get();

    // Verify queue exists and works
    DatabaseConfig dbConfig = result.getDatabaseConfig();
```

```
    try (Connection conn = DriverManager.getConnection(
            dbConfig.getJdbcUrl(),
            dbConfig.getUsername(),
            dbConfig.getPassword())) {

        ResultSet rs = conn.createStatement().executeQuery(
            "SELECT COUNT(*) FROM information_schema.tables " +
            "WHERE table_schema = 'peegeeq' " +
            "AND table_name = 'dynamic_test_queue'"
        );

        rs.next();
        assertEquals(1, rs.getInt(1), "Queue table should exist");
    }
}
```

What Gets Created

When a queue is created dynamically, the following database objects are created:

Queue Table

```
CREATE TABLE {schema}.{queueName} (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    payload JSONB NOT NULL,
    visible_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    lock_id BIGINT,
    lock_until TIMESTAMP WITH TIME ZONE,
    retry_count INT DEFAULT 0,
    max_retries INT DEFAULT 3,
    status VARCHAR(50) DEFAULT 'AVAILABLE',
    headers JSONB DEFAULT '{}',
    error_message TEXT,
    correlation_id VARCHAR(255),
    message_group VARCHAR(255),
    priority INT DEFAULT 5
) INHERITS ({schema}.queue_template);
```

Performance Indexes (5 indexes)

```
CREATE INDEX idx_{queueName}_topic
    ON {schema}.{queueName}(topic, visible_at, status);

CREATE INDEX idx_{queueName}_visible
    ON {schema}.{queueName}(visible_at) WHERE status = 'AVAILABLE';
```

```
CREATE INDEX idx_{queueName}_lock
    ON {schema}.{queueName}(lock_id) WHERE lock_id IS NOT NULL;

CREATE INDEX idx_{queueName}_status
    ON {schema}.{queueName}(status, created_at);

CREATE INDEX idx_{queueName}_priority
    ON {schema}.{queueName}(priority, created_at);
```

Notification Trigger

```
CREATE TRIGGER {queueName}_notify_trigger
    AFTER INSERT ON {schema}.{queueName}
    FOR EACH ROW
    EXECUTE FUNCTION notify_queue_insert();
```

Function (shared across all queues)

```
CREATE OR REPLACE FUNCTION notify_queue_insert()
RETURNS TRIGGER AS $$
BEGIN
    PERFORM pg_notify('queue_insert', NEW.topic || ':' || NEW.id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Dynamic Event Store Creation

Similarly, event stores can be created dynamically:

```
/**
 * Create event store dynamically at runtime
 */
public CompletableFuture<EventStore<OrderEvent>> getOrCreateEventStore(String
storeName) {
    EventStoreConfig config = new EventStoreConfig.Builder()
        .eventStoreName(storeName)
        .tableName(storeName + "_log")
        .notificationPrefix(storeName + "_event_")
        .build();

    return setupService.addEventStore(setupId, config)
        .thenApply(v -> {
            // Create event store factory
            EventStoreFactory factory = new PeeGeeQEventStoreFactory(manager);
            return factory.createEventStore(
```

```
                OrderEvent.class,
                "bitemporal." + storeName + "_log"
            );
        });
}
```

Event Store Objects Created:

- Event log table with bi-temporal columns
- 10+ indexes for temporal queries
- GIN indexes for JSONB payload/headers
- Notification trigger for event streaming
- Inherited from `event_store_template`

Performance Considerations

1. Creation Time

```
Dynamic queue creation takes approximately:
- Queue table: ~50ms
- 5 indexes: ~100ms
- Trigger: ~20ms
- Total: ~170ms per queue
```

2. Connection Pooling

```
// Use connection pool for dynamic creation
@Configuration
public class DynamicQueueConfig {

    @Bean
    public PeeGeeQDatabaseSetupService setupService() {
        // Setup service uses connection pooling internally
        return new PeeGeeQDatabaseSetupService();
    }
}
```

3. Concurrent Creation

```
// Safe for concurrent queue creation
CompletableFuture<Void> future1 = setupService.addQueue(setupId, config1);
CompletableFuture<Void> future2 = setupService.addQueue(setupId, config2);
CompletableFuture<Void> future3 = setupService.addQueue(setupId, config3);
```

```
// All three queues created in parallel (if different names)
CompletableFuture.allOf(future1, future2, future3).get();
```

4. Idempotent Creation

```
// Safe to call multiple times with same queue name
// Uses CREATE TABLE IF NOT EXISTS internally
setupService.addQueue(setupId, config).get();  // Creates table
setupService.addQueue(setupId, config).get();  // No-op (already exists)
```

Limitations and Considerations

1. Schema Must Exist

```
// ❌ BAD: Schema doesn't exist
QueueConfig config = new QueueConfig.Builder()
    .queueName("orders_queue")
    .build();

// Schema "peegeeq" must be created first (via base template or migrations)
```

2. Template Table Required

```
// ❌ BAD: queue_template doesn't exist
// Dynamic queue creation requires queue_template to exist

// ✅ GOOD: Apply base template first
setupService.applySchemaTemplatesAsync(request).get();  // Creates templates
setupService.addQueue(setupId, queueConfig).get();      // Now works
```

3. Naming Constraints

```
// ❌ BAD: Invalid table names
"orders-queue"     // Hyphens not allowed
"123_orders"       // Can't start with number
"order queue"      // Spaces not allowed

// ✅ GOOD: Valid table names
"orders_queue"
"orders_v2"
"tenant_123_orders"
```

4. PostgreSQL Identifier Limit

```
Maximum table name length: 63 characters

❌ BAD:
"tenant_acme_corporation_very_long_company_name_orders_queue_v2"  // 64 chars

✅ GOOD:
"tenant_acme_corp_orders_queue"  // 31 chars
```

Best Practices

1. Queue Naming Convention

```
// ✅ GOOD: Consistent naming
"tenant_{tenantId}_orders"           // Multi-tenant
"workflow_{workflowId}_step_{name}" // Workflow steps
"feature_{featureName}_events"       // Feature-based

// ❌ BAD: Inconsistent naming
"orders_queue_for_tenant_123"
"tenant_456_queue_orders"
"payments_queue_tenant789"
```

2. Cache Queue Factories

```
// ✅ GOOD: Cache factories to avoid recreation
private final Map<String, QueueFactory> queueFactories = new ConcurrentHashMap<>
();

public QueueFactory getQueueFactory(String queueName) {
    return queueFactories.computeIfAbsent(queueName, name ->
        manager.getQueueFactoryProvider()
            .createFactory("native", manager.getDatabaseService())
    );
}

// ❌ BAD: Create factory every time
public QueueFactory getQueueFactory(String queueName) {
    return manager.getQueueFactoryProvider()
        .createFactory("native", manager.getDatabaseService());  // Expensive!
}
```

3. Track Created Queues

```
// ✅ GOOD: Maintain registry of dynamically created queues
@Component
public class QueueRegistry {

    private final Set<String> createdQueues = ConcurrentHashMap.newKeySet();

    public CompletableFuture<Void> createQueueIfNotExists(String queueName) {
        if (createdQueues.contains(queueName)) {
            return CompletableFuture.completedFuture(null);
        }

        QueueConfig config = new QueueConfig.Builder()
            .queueName(queueName)
            .build();

        return setupService.addQueue(setupId, config)
            .thenRun(() -> {
                createdQueues.add(queueName);
                logger.info("Registered new queue: {}", queueName);
            });
    }

    public boolean queueExists(String queueName) {
        return createdQueues.contains(queueName);
    }

    public Set<String> getAllQueues() {
        return Collections.unmodifiableSet(createdQueues);
    }
}
```

4. Handle Creation Failures Gracefully

```
// ✅ GOOD: Proper error handling
public CompletableFuture<Queue<Order>> getOrCreateQueue(String queueName) {
    return createQueueIfNeeded(queueName)
        .thenApply(v -> queueFactory.createQueue(queueName, Order.class))
        .exceptionally(ex -> {
            logger.error("Failed to create queue: {}", queueName, ex);

            // Check if queue already exists (race condition)
            if (queueExists(queueName)) {
                return queueFactory.createQueue(queueName, Order.class);
            }

            throw new QueueCreationException("Unable to create queue: " +
queueName, ex);
        });
}
```

5. Monitor Queue Creation

```
// ✅ GOOD: Emit metrics for queue creation
@Component
public class MonitoredQueueService {

    @Autowired
    private MeterRegistry meterRegistry;

    public CompletableFuture<Void> createQueue(QueueConfig config) {
        Timer.Sample sample = Timer.start(meterRegistry);

        return setupService.addQueue(setupId, config)
            .whenComplete((result, error) -> {
                sample.stop(Timer.builder("peegeeq.queue.creation")
                    .tag("queue", config.getQueueName())
                    .tag("success", error == null ? "true" : "false")
                    .register(meterRegistry));

                if (error == null) {
                    meterRegistry.counter("peegeeq.queue.created").increment();
                } else {

meterRegistry.counter("peegeeq.queue.creation.failed").increment();
                }
            });
    }
}
```

Comparison: Static vs Dynamic Queue Creation

Aspect	Static (Migrations)	Dynamic (Runtime)
When	Application startup	On-demand
Speed	All queues created upfront	Only when needed
Flexibility	Fixed set of queues	Create as needed
Migration	Version controlled	No migration needed
Restart Required	Yes (for new queues)	No
Multi-tenant	All tenants upfront	Per-tenant on-demand
Use Case	Known queue set	Dynamic queue needs
Production	✅ Recommended for fixed queues	✅ Recommended for dynamic needs

When to Use Each Approach

Use Static Migration when:

- ✅ Queue set is known at design time
- ✅ All applications need same queues
- ✅ Version control of schema is important
- ✅ Production stability is critical
- ✅ Schema changes need approval

Use Dynamic Creation when:

- ✅ Multi-tenant with tenant-specific queues
- ✅ Feature flags enable new queues
- ✅ User-defined workflows need queues
- ✅ Plugin architecture with plugin-specific queues
- ✅ A/B testing with experimental queues

Use Both (hybrid approach):

```
// Static: Core queues in migrations
// V001__Create_Base_Tables.sql creates:
//   - orders_queue
//   - payments_queue
//   - notifications_queue

// Dynamic: Tenant-specific queues at runtime
public CompletableFuture<Queue<Order>> getTenantQueue(String tenantId) {
    String queueName = "tenant_" + tenantId + "_orders";
    return getOrCreateQueue(queueName);
}
```

Service-Specific Patterns

peegeeq-db (Core Database Module)

Purpose: Core database access and management

Setup Method: Uses all three approaches depending on context

- **Production**: Flyway migrations via `peegeeq-migrations`
- **Integration Tests**: `PeeGeeQDatabaseSetupService`
- **Unit Tests**: `PeeGeeQTestSchemaInitializer`

Key Classes:

- `PeeGeeQDatabaseSetupService` (825 lines)
  - Creates test databases dynamically
  - Applies SQL templates from `src/main/resources/db/templates/`
  - Manages PeeGeeQManager lifecycle
  - Handles cleanup and resource disposal

- `SqlTemplateProcessor` (153 lines)
    - Loads SQL files from template directories
    - Processes `.manifest` files for execution order
    - Substitutes parameters in SQL ({queueName}, {tableName})
    - Executes SQL using Vert.x reactive PostgreSQL client

Database Objects Created:

- Core operational tables (queue_messages, outbox, dead_letter_queue)
- Template tables (queue_template, event_store_template)
- Consumer group tables (5 tables for fanout pattern)
- Indexes (30+ for performance)
- Functions and triggers (2 each)

Usage Example:

```
// Integration test
PeeGeeQDatabaseSetupService setupService = new PeeGeeQDatabaseSetupService();
DatabaseSetupRequest request = new DatabaseSetupRequest.Builder()
    .setupId("test_123")
    .databaseConfig(dbConfig)
    .queues(List.of(queueConfig))
    .eventStores(List.of(eventStoreConfig))
    .build();
DatabaseSetupResult result = setupService.createCompleteSetup(request).get();
```

peegeeq-native (Native Queue Implementation)

Purpose: High-performance native PostgreSQL queue (10k+ msg/sec)

Database Requirements:

- `queue_messages` table (from Flyway migrations or setup service)
- `message_processing` table (INSERT-only lock-free processing)
- `dead_letter_queue` table (failed message handling)
- Indexes on topic, visible_at, status, priority

Setup Patterns:

Production

```
// Database already initialized via Flyway
// Native queue connects to existing schema
PeeGeeQConfiguration config = new PeeGeeQConfiguration();
PeeGeeQManager manager = new PeeGeeQManager(config);

// Get native queue factory
QueueFactory factory = manager.getQueueFactoryProvider()
```

```
    .createFactory("native", manager.getDatabaseService());

// Use queue
Queue<Message> queue = factory.createQueue("orders_queue", Message.class);
```

Integration Tests

```
// Setup creates queue tables dynamically
DatabaseSetupRequest request = new DatabaseSetupRequest.Builder()
    .queues(List.of(
        new QueueConfig.Builder()
            .queueName("test_native_queue")
            .maxRetries(3)
            .build()
    ))
    .build();

DatabaseSetupResult result = setupService.createCompleteSetup(request).get();

// Native queue factory available in result
QueueFactory factory = result.getQueueFactories().get("test_native_queue");
```

Unit Tests

```
// Initialize only native queue schema
PeeGeeQTestSchemaInitializer.initializeSchema(
    postgres,
    SchemaComponent.NATIVE_QUEUE,
    SchemaComponent.DEAD_LETTER_QUEUE
);

// Direct JDBC operations for testing
try (Connection conn = DriverManager.getConnection(jdbcUrl, user, password)) {
    PreparedStatement stmt = conn.prepareStatement(
        "INSERT INTO peegeeq.queue_messages (topic, payload) VALUES (?, ?)"
    );
    // ... test operations
}
```

Key Features:

- Lock-free message processing using `message_processing` table
- Priority-based message ordering (1-10 scale)
- Automatic retry with exponential backoff
- Dead letter queue for failed messages
- Real-time visibility control

peegeeq-outbox (Transactional Outbox Pattern)

Purpose: Transactional message publishing with ACID guarantees (5k+ msg/sec)

Database Requirements:

- `outbox` table (from Flyway migrations or setup service)
- `outbox_consumer_groups` table (for fanout to multiple consumers)
- `dead_letter_queue` table
- Indexes on status, topic, consumer_group_name

Setup Patterns:

Production

```
// Database already initialized via Flyway
PeeGeeQConfiguration config = new PeeGeeQConfiguration();
PeeGeeQManager manager = new PeeGeeQManager(config);

// Get outbox factory
QueueFactory factory = manager.getQueueFactoryProvider()
    .createFactory("outbox", manager.getDatabaseService());

// Outbox pattern: insert into outbox within business transaction
try (Connection conn = dataSource.getConnection()) {
    conn.setAutoCommit(false);

    // Business operation
    stmt.executeUpdate("INSERT INTO orders (id, customer, amount) VALUES (...)");

    // Outbox message (same transaction)
    stmt.executeUpdate("INSERT INTO peegeeq.outbox (topic, payload) VALUES
(...)");

    conn.commit();  // Both committed atomically
}
```

Integration Tests

```
// Setup service creates outbox tables
DatabaseSetupRequest request = new DatabaseSetupRequest.Builder()
    .queues(List.of())  // No native queues needed
    .eventStores(List.of())
    .build();

DatabaseSetupResult result = setupService.createCompleteSetup(request).get();

// Outbox tables available via PeeGeeQManager
PeeGeeQManager manager = result.getManager();
```

Unit Tests

```
// Initialize outbox schema
PeeGeeQTestSchemaInitializer.initializeSchema(
    postgres,
    SchemaComponent.OUTBOX,
    SchemaComponent.DEAD_LETTER_QUEUE
);

// Test outbox operations
try (Connection conn = DriverManager.getConnection(jdbcUrl, user, password)) {
    conn.setAutoCommit(false);

    PreparedStatement stmt = conn.prepareStatement(
        "INSERT INTO peegeeq.outbox (topic, payload, status) VALUES (?, ?,
'PENDING')"
    );
    stmt.setString(1, "test_topic");
    stmt.setString(2, "{\"data\":\"test\"}");
    stmt.executeUpdate();

    conn.commit();
}
```

Key Features:

- ACID transaction guarantees with business data
- Consumer group fanout (one message → multiple consumers)
- Status tracking per consumer group
- Automatic retry with configurable backoff
- Dead letter queue for permanent failures

peegeeq-bitemporal (Bi-temporal Event Store)

Purpose: Event sourcing with temporal queries and corrections (3k+ msg/sec)

Database Requirements:

- `bitemporal_event_log` table (from Flyway migrations or setup service)
- Event store tables created dynamically per event store
- Indexes for temporal queries (valid_time, transaction_time)
- Triggers for notification streaming
- Views for current state and statistics

Setup Patterns:

Production

```
// Database initialized via Flyway
PeeGeeQConfiguration config = new PeeGeeQConfiguration();
PeeGeeQManager manager = new PeeGeeQManager(config);

// Create event store factory
EventStoreFactory factory = new PeeGeeQEventStoreFactory(manager);

// Create event store for specific entity
EventStore<OrderEvent> orderEvents = factory.createEventStore(
    OrderEvent.class,
    "bitemporal.order_events"  // Table name
);

// Append events
BiTemporalEvent<OrderEvent> event = orderEvents.append(
    "order-123",          // event ID
    "OrderCreated",       // event type
    Instant.now(),        // valid time
    orderCreatedPayload   // event data
).get();
```

Integration Tests

```
// Setup creates event store tables dynamically
DatabaseSetupRequest request = new DatabaseSetupRequest.Builder()
    .eventStores(List.of(
        new EventStoreConfig.Builder()
            .eventStoreName("test_order_events")
            .tableName("test_order_event_log")
            .notificationPrefix("order_event_")
            .build()
    ))
    .build();

DatabaseSetupResult result = setupService.createCompleteSetup(request).get();

// Event stores available in result
EventStore<?> eventStore = result.getEventStores().get("test_order_events");
```

Unit Tests

```
// Initialize bitemporal schema
PeeGeeQTestSchemaInitializer.initializeSchema(
    postgres,
    SchemaComponent.BITEMPORAL
);

// Test direct SQL operations
```

```
try (Connection conn = DriverManager.getConnection(jdbcUrl, user, password)) {
    PreparedStatement stmt = conn.prepareStatement(
        "INSERT INTO bitemporal.bitemporal_event_log " +
        "(event_id, event_type, valid_time, transaction_time, payload) " +
        "VALUES (?, ?, ?, ?, ?::jsonb)"
    );
    stmt.setString(1, "evt-123");
    stmt.setString(2, "TestEvent");
    stmt.setTimestamp(3, Timestamp.from(Instant.now()));
    stmt.setTimestamp(4, Timestamp.from(Instant.now()));
    stmt.setString(5, "{\"data\":\"test\"}");
    stmt.executeUpdate();
}
```

Key Features:

- Bi-temporal tracking (valid_time + transaction_time)
- Event corrections with audit trail
- Point-in-time queries (as of any timestamp)
- Aggregate versioning and history
- JSONB payload for flexible event data
- Notification triggers for event streaming

Temporal Queries:

```
-- Current state (as of now)
SELECT * FROM bitemporal_current_state WHERE event_id = 'order-123';

-- Historical state (as of specific time)
SELECT * FROM bitemporal_event_log
WHERE event_id = 'order-123'
  AND transaction_time <= '2025-11-01'
ORDER BY transaction_time DESC
LIMIT 1;

-- All corrections for an event
SELECT * FROM bitemporal_event_log
WHERE event_id = 'order-123'
ORDER BY version;
```

peegeeq-migrations (Flyway Migration Module)

Purpose: Production database initialization and version control

Setup Method: Flyway migrations only (this IS the migration module)

Key Files:

- `src/main/resources/db/migration/V001__Create_Base_Tables.sql` (576 lines)
  - Complete schema definition

- All tables, indexes, functions, triggers
- Idempotent (safe to re-run)

Usage:

Standalone JAR

```
# Build
mvn clean package -DskipTests

# Run migrations
java -jar target/peegeeq-migrations.jar migrate

# Check status
java -jar target/peegeeq-migrations.jar info

# Validate
java -jar target/peegeeq-migrations.jar validate
```

Maven Plugin

```
cd peegeeq-migrations

# Run migrations
mvn flyway:migrate

# Check status
mvn flyway:info

# Clean database (dev only!)
mvn flyway:clean

# Validate migrations
mvn flyway:validate
```

Docker

```
# Build image
docker build -t peegeeq-migrations .

# Run migrations
docker run --rm \
  -e DB_JDBC_URL=jdbc:postgresql://host:5432/db \
  -e DB_USER=user \
  -e DB_PASSWORD=pass \
  peegeeq-migrations migrate
```

Migration Script Structure:

```
-- V001__Create_Base_Tables.sql

-- Header (metadata)
-- executeInTransaction=false

-- Schema version tracking
CREATE TABLE IF NOT EXISTS schema_version (...);

-- Core tables
CREATE TABLE IF NOT EXISTS outbox (...);
CREATE TABLE IF NOT EXISTS queue_messages (...);
CREATE TABLE IF NOT EXISTS bitemporal_event_log (...);
-- ... all tables

-- Indexes
CREATE INDEX idx_outbox_status_created ON outbox(status, created_at);
CREATE INDEX idx_queue_messages_topic_visible ON queue_messages(topic,
visible_at);
-- ... 30+ indexes

-- Views
CREATE OR REPLACE VIEW bitemporal_current_state AS ...;
-- ... 3 views

-- Functions and Triggers
-- (defined in migration script)
```

peegeeq-test-support (Test Utilities)

Purpose: Shared test utilities for all modules

Key Classes:

- `PeeGeeQTestSchemaInitializer` (747 lines)
  - Component-based schema initialization
  - Fast JDBC-based setup for unit tests
  - Granular schema component control

Usage Patterns:

Full Schema

```
PeeGeeQTestSchemaInitializer.initializeSchema(
    postgres,
    SchemaComponent.ALL
);
```

Selective Components

```
// Only what you need for specific test
PeeGeeQTestSchemaInitializer.initializeSchema(
    postgres,
    SchemaComponent.OUTBOX,
    SchemaComponent.NATIVE_QUEUE,
    SchemaComponent.DEAD_LETTER_QUEUE
);
```

Queue-Related Only

```
// Convenience enum for all queue components
PeeGeeQTestSchemaInitializer.initializeSchema(
    postgres,
    SchemaComponent.QUEUE_ALL  // Expands to OUTBOX + NATIVE + DLQ
);
```

Cleanup Between Tests

```
@AfterEach
void cleanup() {
    // Remove test data but keep schema
    PeeGeeQTestSchemaInitializer.cleanupTestData(
        postgres,
        SchemaComponent.OUTBOX,
        SchemaComponent.NATIVE_QUEUE
    );
}
```

Advantages Over PeeGeeQDatabaseSetupService:

- **Faster**: Direct JDBC, no Vert.x startup
- **Simpler**: No database creation, uses TestContainer database
- **Granular**: Initialize only what you need
- **Synchronous**: No CompletableFuture complexity
- **Ideal for unit tests**: Testing specific components

Database Schema Components

Complete Schema Overview

Note: PeeGeeQ uses **two separate PostgreSQL schemas** by default (`peegeeq` and `bitemporal`), but both can be customized to any schema name(s) you prefer. You can even use the same schema for both (e.g.,

`myapp` for everything) or integrate into existing schemas.

```
│  │  • Timestamp update triggers                    │     │
│  │  • Notification triggers                        │     │
│  └─────────────────────────────────────────────────┘     │
│                                                          │
│  ┌─────────────────────────────────────────────────┐     │
│  │  Views (3)                                      │     │
│  │  • bitemporal_current_state                     │     │
│  │  • bitemporal_latest_events                     │     │
│  │  • bitemporal_event_stats                       │     │
│  └─────────────────────────────────────────────────┘     │
└──────────────────────────────────────────────────────────┘
```

Table Relationships

```
queue_messages ──┐
                 ├──> message_processing (1:N)
outbox ──────────┤
                 └──> dead_letter_queue (1:N)

outbox ──────────> outbox_consumer_groups (1:N)

queue_template <─── queue tables (inheritance)

event_store_template <─── event store tables (inheritance)

bitemporal_event_log ──> bitemporal_current_state (view)
                     ├──> bitemporal_latest_events (view)
                     └──> bitemporal_event_stats (view)
```

Schema Component Dependencies

```
Extensions (uuid-ossp, pg_stat_statements)
    ↓
Schemas (peegeeq, bitemporal)
    ↓
Template Tables (queue_template, event_store_template)
    ↓
Core Tables (queue_messages, outbox, bitemporal_event_log)
    ↓
Supporting Tables (dead_letter_queue, message_processing)
    ↓
Consumer Group Tables (5 tables)
    ↓
Indexes (30+)
    ↓
Functions & Triggers (4)
    ↓
Views (3)
```

Setup Methods Comparison

Feature Matrix

Feature	Flyway Migrations	PeeGeeQDatabaseSetupService	PeeGeeQTestSchemaInitializer
Environment	Production, Dev	Integration Tests	Unit Tests
Speed	Medium (1-2s)	Slow (5-10s)	Fast (<1s)
Complexity	Low	High	Low
Flexibility	Low	High	Medium
Version Control	✅ Yes	❌ No	❌ No
Rollback	✅ Yes	❌ No	❌ No
Parameterization	❌ No	✅ Yes	❌ No
Dynamic DB Creation	❌ No	✅ Yes	❌ No
Async/Reactive	❌ No	✅ Yes	❌ No
Component Selection	❌ No	❌ No	✅ Yes
Cleanup	Manual	✅ Automatic	✅ Automatic

When to Use Each Method

Use Flyway Migrations When:

- ✅ Setting up production databases
- ✅ Need version control and audit trail
- ✅ Schema changes need to be tracked
- ✅ Multiple environments need same schema
- ✅ Rollback capability is required
- ✅ CI/CD pipeline deployment

Use PeeGeeQDatabaseSetupService When:

- ✅ Integration tests need isolated databases
- ✅ Dynamic database creation per test
- ✅ Parameterized table names required
- ✅ Testing full PeeGeeQManager lifecycle
- ✅ Need QueueFactory and EventStore instances
- ✅ Async/reactive patterns being tested

Use PeeGeeQTestSchemaInitializer When:

- ✅ Unit tests need minimal schema setup
- ✅ Speed is critical (fast test execution)
- ✅ Only specific components needed
- ✅ Testing individual components in isolation
- ✅ No PeeGeeQManager instance needed
- ✅ Simple JDBC operations being tested

Troubleshooting

Common Issues

Issue 1: "Table does not exist" Errors

Symptoms:

```
FATAL: dead_letter_queue table does not exist - schema not initialized properly
FATAL: queue_messages table does not exist - schema not initialized properly
```

Cause: Database not initialized before application startup

Solution:

```
# Production: Run migrations first
cd peegeeq-migrations
mvn flyway:migrate

# Development: Use docker-compose
docker-compose -f docker-compose.dev.yml up

# Integration Tests: Use PeeGeeQDatabaseSetupService
setupService.createCompleteSetup(request).get();

# Unit Tests: Use PeeGeeQTestSchemaInitializer
PeeGeeQTestSchemaInitializer.initializeSchema(postgres, SchemaComponent.ALL);
```

Issue 2: "Templates not found" in Tests

Symptoms:

```
Templates not found in database testdb: queue_template=false,
event_store_template=false
```

Cause: Extensions not created (missing uuid-ossp or pg_stat_statements)

Solution:

```
// Ensure extensions are available in PostgreSQL image
@Container
private static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>
("postgres:15.13-alpine3.20")
    .withCommand("postgres -c shared_preload_libraries=pg_stat_statements");

// OR use base template which handles extension failures gracefully
```

Issue 3: Multi-Statement SQL Not Executing

Symptoms:

- Only first SQL statement in file executes
- Missing indexes, triggers, or tables

Cause: Vert.x PostgreSQL client limitation (only executes first statement)

Solution: SQL templates are now organized as directories with numbered files

```
base/
├── .manifest              # Lists files in order
├── 01a-extension.sql      # One statement per file
├── 01b-schema.sql
└── ...
```

See Also: docs/devtest/VERTX_MULTISTATEMENT_SQL_BUG_ANALYSIS.md

Issue 4: Parallel Test Failures

Symptoms:

```
Database "test_db_123" is being accessed by other users
Test 4's manager is connecting to Test 5's database
```

Cause: Shared database name or non-unique setup IDs

Solution:

```
// Use unique database name per test class
String dbName = "test_" + getClass().getSimpleName() + "_" + UUID.randomUUID();

// Use unique setup ID per test
```

```
String setupId = "test_" + Thread.currentThread().getId() + "_" +
System.nanoTime();

// Use programmatic configuration (not System.setProperty)
PeeGeeQConfiguration config = new PeeGeeQConfiguration(
    setupId,
    host,
    port,
    dbName,
    username,
    password,
    schema
);
```

Issue 5: Flyway Migration Conflicts

Symptoms:

```
Validate failed: Migrations have failed validation
Detected resolved migration not applied to database: 1
```

Cause: Database state doesn't match migration history

Solution:

```
# Check migration status
mvn flyway:info

# Repair migration history
mvn flyway:repair

# Clean and re-migrate (DEV ONLY!)
mvn flyway:clean flyway:migrate

# Production: Never use clean, investigate and fix manually
```

Issue 6: Permission Denied on CREATE EXTENSION

Symptoms:

```
ERROR: permission denied to create extension "uuid-ossp"
ERROR: must be superuser to create extension "pg_stat_statements"
```

Cause: Database user lacks CREATE EXTENSION permission

Solution:

```
-- Grant extension creation permission
ALTER USER peegeeq_user WITH SUPERUSER;

-- OR create extensions as superuser first
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "pg_stat_statements";

-- Then run migrations as regular user
```

Alternative: Production code has fallback to minimal schema without extensions

Issue 7: Test Cleanup Failures

Symptoms:

```
Failed to drop test database: test_db_123 - database is being accessed by other
users
```

Cause: Open connections still active

Solution:

```
@AfterAll
void cleanup() {
    // Stop PeeGeeQManager first (closes all connections)
    if (setupResult != null && setupResult.getManager() != null) {
        setupResult.getManager().stop();
    }

    // Then destroy setup
    if (setupService != null && setupResult != null) {
        setupService.destroySetup(setupResult.getSetupId()).get();
    }
}
```

Debug Logging

Enable debug logging to troubleshoot setup issues:

```
# logback.xml or application.properties
logging.level.dev.mars.peegeeq.db.setup=DEBUG
logging.level.org.flywaydb=DEBUG
```

Key log messages:

```
🔧 Applying base template: base
🔧✅ Base template SQL executed
🔧🔍 Verifying templates exist in bitemporal schema...
✅ FINAL: All schema templates applied successfully
```

Summary

Quick Reference

Task	Method	Command/Code
Setup Production DB	Flyway Migrations	`java -jar peegeeq-migrations.jar migrate`
Setup Dev DB	Docker Compose	`docker-compose -f docker-compose.dev.yml up`
Setup Integration Test	PeeGeeQDatabaseSetupService	`setupService.createCompleteSetup(request).get()`
Setup Unit Test	PeeGeeQTestSchemaInitializer	`initializeSchema(postgres, SchemaComponent.ALL)`
Set Custom Schema (Flyway)	Configuration	`flyway.schemas=myapp_queue,myapp_events`
Set Custom Schema (App)	Environment Variable	`export PEEGEEQ_DB_SCHEMA="myapp_queue"`
Verify Schema	Flyway Info	`mvn flyway:info`
Reset Dev DB	Flyway Clean	`mvn flyway:clean flyway:migrate`
Check Health	PeeGeeQManager	`manager.getHealthService().checkHealth()`

Key Takeaways

1. **Production = Flyway Migrations**: Always use `peegeeq-migrations` module
2. **Tests = Programmatic Setup**: Use setup service or test initializer
3. **Custom Schemas Supported**: Deploy to any PostgreSQL schema, not just `public`

4. **Multi-Tenant Ready**: Use schema-per-tenant for isolation and scalability
5. **One Statement Per File**: Vert.x limitation requires single-statement SQL files
6. **Unique Database Names**: Critical for parallel test execution
7. **Cleanup After Tests**: Always stop manager before destroying setup
8. **Component-Based Testing**: Only initialize what you need in unit tests
9. **Template-Based Integration**: Use templates for dynamic test databases
10. **Fully Qualified Names**: Always use schema.table in queries for clarity

---

Related Documentation

- **VERTX_MULTISTATEMENT_SQL_BUG_ANALYSIS.md** - Deep dive into Vert.x SQL execution limitation
- **PEEGEEQ_MIGRATIONS_DEPLOYMENT_GUIDE.md** - Production deployment patterns
- **PEEGEEQ_MIGRATIONS_README.md** - Migration module overview
- **TESTING-GUIDE.md** - Comprehensive testing guide
- **PEEGEEQ_COMPLETE_GUIDE.md** - Full system architecture
