📚 Complete Testing & Testcontainers Guide

> **The ONLY guide you need for testing with Spring Boot, Testcontainers, and PostgreSQL**

Last Updated: December 31, 2025

📋 Table of Contents

Part 1: Quick Start

Part 2: Setup & Configuration

Part 3: Writing Tests

Part 4: Database Management

Part 5: Understanding Your Setup

Part 6: Troubleshooting

Part 1: Quick Start

TL;DR - Run Tests in 3 Steps

```
# 1. Start Docker
open -a Docker  # macOS

# 2. Run all tests
./gradlew test

# 3. Run test suite with data injection
./gradlew test --tests IntegrationTestSuite
```

That's it! PostgreSQL container starts automatically, schema generated from JPA entities, tests run.

What's in Your Test Setup

Your Test Stack

- ✅ **Spring Boot 3.2** with Kotlin
- ✅ **Testcontainers** for PostgreSQL
- ✅ **JPA/Hibernate** for schema generation
- ✅ **16 JPA Entities** with relationships
- ✅ **Container reuse** for fast tests
- ✅ **Test execution order** control

Production vs Tests

Aspect	Production	Tests
Database	SQLite	PostgreSQL (Testcontainers)
Schema	Managed by you	Auto-generated from entities
Port	Fixed	Dynamic (e.g., 52106)
Data	Persistent	Ephemeral (per test run)

Part 2: Setup & Configuration

Prerequisites

Required

1. **Docker Desktop** - Must be running

   ```
   open -a Docker  # Start Docker
   docker ps       # Verify it's running
   ```

2. **JDK 17** - Already configured

3. **Gradle** - Included (./gradlew)

Optional

- Node.js/npm (not needed for tests, only for Prisma in production)

Global Configuration Files

1. ~/.testcontainers.properties

Location: Your home directory (`~/.testcontainers.properties`)

Content:

```
testcontainers.reuse.enable=true
```

What it does:

- ✅ Enables container reuse across test runs
- ✅ Makes subsequent test runs much faster (seconds vs minutes)
- ✅ Container stays alive until Docker restart

Create it:

```
echo "testcontainers.reuse.enable=true" > ~/.testcontainers.properties
```

Or use the script:

```
./create-testcontainers-config.sh
```

---

2. src/test/resources/application-test.yml

Content:

```
spring:
  jpa:
    hibernate:
      ddl-auto: create  # Creates tables, keeps after tests
    show-sql: true
    properties:
      hibernate:
        format_sql: true
        dialect: org.hibernate.dialect.PostgreSQLDialect
```

```
logging:
  level:
    org.hibernate.SQL: DEBUG
    org.testcontainers: INFO
    com.scriptmanager: DEBUG
```

Key setting: `ddl-auto: create`

- `create` → Tables stay after test (can inspect with GUI)
- `create-drop` → Tables dropped after test (clean slate)

---

3. src/test/resources/junit-platform.properties

Content:

```
spring.test.constructor.autowire.mode=all
```

What it does:

- ✅ Enables constructor injection in tests
- ✅ No need for `@Autowired` on constructor
- ✅ Cleaner, more idiomatic Kotlin code

Test Configuration

TestcontainersConfiguration.kt

Location: `src/test/kotlin/com/scriptmanager/config/TestcontainersConfiguration.kt`

Key features:

```
@TestConfiguration
class TestcontainersConfiguration {
    @Bean
    @ServiceConnection  // ← Auto-configures datasource
    fun postgresContainer(): PostgreSQLContainer<*> {
        val container = PostgreSQLContainer(...)
        .withReuse(true)  // ← Container reuse

        container.start()
        printConnectionInfo(container)  // ← Shows connection details
        return container
    }
}
```

What it does:

1. ✅ Starts PostgreSQL container automatically
2. ✅ Prints connection info (host, port, credentials)
3. ✅ Configures Spring datasource via `@ServiceConnection`
4. ✅ Reuses container for fast subsequent runs

DatabaseConfig.kt (Production Only)

Important: This config is **excluded from tests**:

```
@Configuration
@Profile("!test")  // ← NOT active in test profile
class DatabaseConfig {
    @Bean
    fun dataSource(): DataSource {
        // SQLite for production
    }
}
```

Why: Tests use PostgreSQL from Testcontainers, not SQLite.

Part 3: Writing Tests

Basic Test Structure

Simple Test

```
@SpringBootTest
@Import(TestcontainersConfiguration::class)
@ActiveProfiles("test")
class CommandInvokerIntegrationTest(
    private val commandInvoker: CommandInvoker
) {
    @Test
    fun `context loads successfully`() {
        assertNotNull(commandInvoker)
    }
}
```

Key annotations:

- `@SpringBootTest` → Full Spring context
- `@Import(TestcontainersConfiguration::class)` → PostgreSQL container
- `@ActiveProfiles("test")` → Uses test configuration
- Constructor parameters automatically injected (via junit-platform.properties)

Test with Repository

```
@SpringBootTest
@Import(TestcontainersConfiguration::class)
@ActiveProfiles("test")
class WorkspaceIntegrationTest(
    private val commandInvoker: CommandInvoker,
    private val workspaceRepository: WorkspaceRepository
) {
    @Test
    fun `create workspace persists to database`() {
        // Execute
        commandInvoker.invoke(CreateWorkspaceCommand("Test"))

        // Verify
        val workspaces = workspaceRepository.findAll()
        assertEquals(1, workspaces.size)
        assertEquals("Test", workspaces[0].name.value)
    }
}
```

Data Injection Tests

Overview

Data injection tests allow you to set up test data once and reuse it across multiple test classes. This is useful for:

- ✅ Setting up reference data (e.g., model configs, users)
- ✅ Creating complex data relationships
- ✅ Avoiding duplicate setup code across tests
- ✅ Faster test execution (setup once, use many times)

How Database Connection Sharing Works

Both `DataSetupTest` and other test classes share the **same PostgreSQL container** and **same database**:

```
@SpringBootTest
@Import(TestcontainersConfiguration::class)  // ← Same Testcontainers
config
@ActiveProfiles("test")                      // ← Same profile
```

Connection sharing mechanism:

- ✅ Container has `withReuse(true)` → Same container reused across tests
- ✅ Config has `ddl-auto: create` → Tables stay after first test

- ✅ Both use `@ServiceConnection` → Same database connection
- ✅ Data persists between test classes

Flow:

```
Test Suite Starts
    ↓
DataSetupTest runs
    ↓
Container starts (if not already running)
    ↓
Tables created (ddl-auto: create)
    ↓
Data injected to database
    ↓
DataSetupTest ends
    ↓
Tables STAY (ddl-auto: create, not create-drop)
    ↓
CommandInvokerIntegrationTest runs
    ↓
SAME container, SAME database
    ↓
Data from DataSetupTest is available ✅
```

Key Configuration:

```
# application-test.yml - MUST be 'create' not 'create-drop'
spring.jpa.hibernate.ddl-auto: create
```

Why:

- `create` → Tables stay after each test class (data persists)
- `create-drop` → Tables dropped after each test class (data lost!)

DataSetupTest.kt

Purpose: Inject test data that other tests can use.

File Location: `src/test/kotlin/com/scriptmanager/integration/DataSetupTest.kt`

Key Features:

- ✅ Runs first (controlled by test suite)
- ✅ Data persists to subsequent tests
- ✅ Controlled method execution order within class
- ✅ Shares instance across all methods

Annotations Explained:

- `@TestMethodOrder(MethodOrderer.OrderAnnotation::class)` → Methods run in order (1, 2, 3...)
- `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` → Share instance across all test methods
- `@Order(n)` → Specify execution order of methods

Complete Example:

```
@SpringBootTest
@Import(TestcontainersConfiguration::class)
@ActiveProfiles("test")
@TestMethodOrder(MethodOrderer.OrderAnnotation::class)  // ← Order methods
@TestInstance(TestInstance.Lifecycle.PER_CLASS)         // ← Share
instance
class DataSetupTest(
    private val commandInvoker: CommandInvoker,
    private val workspaceRepository: WorkspaceRepository  // Can inject
repositories too
) {
    private var workspaceId: Int? = null  // Share data between methods

    @Test
    @Order(1)
    fun `01 - setup workspaces`() {
        println("🔧 Setting up test workspaces...")
        commandInvoker.invoke(CreateWorkspaceCommand("Test Workspace 1"))
        commandInvoker.invoke(CreateWorkspaceCommand("Test Workspace 2"))

        // Save ID for use in later methods
        val workspace = workspaceRepository.findAll().first()
        workspaceId = workspace.id

        println("✅ Workspaces created (ID: $workspaceId)")
    }

    @Test
    @Order(2)
    fun `02 - setup folders`() {
        println("🔧 Setting up test folders...")
        // Use workspaceId from previous method
        commandInvoker.invoke(
            CreateFolderCommand(
                name = "Test Folder",
                workspaceId = workspaceId!!
            )
        )
        println("✅ Folders created")
    }

    @Test
    @Order(3)
    fun `03 - setup scripts`() {
        println("🔧 Setting up test scripts...")
        commandInvoker.invoke(
            CreateScriptCommand(
                name = "Test Script",
                content = "echo 'Hello from test'"
            )
        )
        println("✅ Scripts created")
    }

    @Test
    @Order(4)
    fun `04 - verify setup complete`() {
        println("✅ Verifying all test data...")
        val workspaces = workspaceRepository.findAll()
        assertTrue(workspaces.size >= 2, "Should have at least 2 workspaces")
        println("✅ Data setup verification complete!")
    }
}
```

Using Injected Data in Other Tests

After `DataSetupTest` runs, the data is available in all subsequent tests:

```
@SpringBootTest
@Import(TestcontainersConfiguration::class)
@ActiveProfiles("test")
class MyFeatureTest(
    private val workspaceRepository: WorkspaceRepository,
    private val scriptRepository: ScriptRepository
) {
    @Test
    fun `can query workspaces created by DataSetupTest`() {
        // Data from DataSetupTest is available!
        val workspaces = workspaceRepository.findAll()

        assertTrue(workspaces.size >= 2, "Should have workspaces from DataSetupTest")
        assertEquals("Test Workspace 1", workspaces[0].name.value)
    }

    @Test
    fun `can use existing scripts for testing new features`() {
        // Scripts created in DataSetupTest are available
        val scripts = scriptRepository.findAll()
        assertTrue(scripts.isNotEmpty(), "Should have scripts from DataSetupTest")
        // Test your new feature with existing data
        val result = myNewFeature.process(scripts.first())
        assertNotNull(result)
    }
}
```

Common Use Cases

Use Case 1: Setup Reference Data

```
@Test
@Order(1)
fun `setup model configurations`() {
    commandInvoker.invoke(
        CreateModelConfigCommand(
            name = "gpt-4",
            apiKey = "test-key",
            endpoint = "https://api.openai.com"
        )
    )
    commandInvoker.invoke(
        CreateModelConfigCommand(
            name = "claude-3",
            apiKey = "test-key",
            endpoint = "https://api.anthropic.com"
        )
    )
}
```

Use Case 2: Setup Test Users

```
@Test
@Order(1)
fun `setup test users`() {
    commandInvoker.invoke(
        CreateUserCommand(
            email = "admin@test.com",
            role = "ADMIN"
        )
    )
    commandInvoker.invoke(
        CreateUserCommand(
            email = "user@test.com",
            role = "USER"
        )
    )
}
```

Use Case 3: Complex Data Relationships

```
private var workspaceId: Int? = null
private var folderId: Int? = null

@Test
@Order(1)
fun `01 - setup workspace`() {
    commandInvoker.invoke(CreateWorkspaceCommand("Test Workspace"))
    workspaceId = workspaceRepository.findAll().first().id
}

@Test
@Order(2)
fun `02 - setup folder in workspace`() {
    // Uses workspaceId from previous test
    commandInvoker.invoke(
        CreateFolderCommand(
            name = "Test Folder",
            workspaceId = workspaceId!!
        )
    )
    folderId = folderRepository.findAll().first().id
}

@Test
@Order(3)
fun `03 - setup scripts in folder`() {
    // Uses folderId from previous test
    commandInvoker.invoke(
        CreateScriptCommand(
            name = "Script 1",
            folderId = folderId!!
        )
    )
    commandInvoker.invoke(
        CreateScriptCommand(
            name = "Script 2",
            folderId = folderId!!
        )
    )
}
```

Use Case 4: Bulk Data Creation

```
@Test
@Order(1)
fun `setup 100 test workspaces`() {
    repeat(100) { i ->
        commandInvoker.invoke(CreateWorkspaceCommand("Workspace $i"))
    }
    println("✅ Created 100 test workspaces for performance testing")
}
```

Best Practices

✅ DO:

- ✅ Make data setup idempotent (can run multiple times safely)
- ✅ Use meaningful names for test data (e.g., "Test Workspace 1")
- ✅ Document what data is created in comments
- ✅ Use `@Order` annotations to control method execution
- ✅ Store IDs in class variables for use across methods
- ✅ Add verification tests to ensure data setup succeeded

❌ DON'T:

- ❌ Create too much data (slows down tests)
- ❌ Use hard-coded IDs (use returned IDs from commands)
- ❌ Assume order without `@Order` annotations
- ❌ Create data with sensitive or production-like values
- ❌ Forget to verify data was actually created

Troubleshooting Data Injection

Problem: Tests Run in Wrong Order

Symptoms: DataSetupTest runs after other tests

Solution: Always run via test suite:

```
./gradlew test --tests IntegrationTestSuite  # ✅ Correct
./gradlew test                                # ❌ Order not guaranteed
```

Problem: Data Not Available in Second Test

Symptoms: CommandInvokerIntegrationTest can't find data from DataSetupTest

Possible causes:

1. ❌ Using `create-drop` instead of `create` → Tables dropped between tests

2. ❌ Not running via test suite → Order not guaranteed
3. ❌ Transaction rollback → Data not committed
4. ❌ Container restarted → New database

Solutions:

```
# Check application-test.yml
spring.jpa.hibernate.ddl-auto: create  # ← MUST be 'create' not 'create-drop'
```

```
# Always run via test suite
./gradlew test --tests IntegrationTestSuite
```

Problem: Container Not Shared

Symptoms: Each test starts a new container

Solution: Verify container reuse is enabled:

1. Check code:

```
// TestcontainersConfiguration.kt
.withReuse(true)  // ← Must be set
```

2. Check global config:

```
cat ~/.testcontainers.properties
# Should contain: testcontainers.reuse.enable=true
```

3. Create if missing:

```
echo "testcontainers.reuse.enable=true" > ~/.testcontainers.properties
```

Test Execution Order

Overview

Controlling test execution order is crucial when you have tests that depend on data created by other tests. JUnit 5 provides several ways to control test order, and we use the **Test Suite** approach for maximum control.

IntegrationTestSuite.kt

Purpose: Control which test classes run in which order.

File Location: `src/test/kotlin/com/scriptmanager/integration/IntegrationTestSuite.kt`

Key Features:

- ✅ Explicit order control - tests run in the order specified
- ✅ Clear and maintainable - easy to see test execution flow
- ✅ Reliable - guaranteed execution order
- ✅ Flexible - easy to add or reorder tests

Example:

```
@Suite
@SuiteDisplayName("Integration Tests with Data Setup")
@SelectClasses(
    DataSetupTest::class,                    // 1. Runs FIRST - injects data
    CommandInvokerIntegrationTest::class,    // 2. Runs SECOND - uses data
    WorkspaceFeatureTest::class,             // 3. Runs THIRD - more tests
    ScriptExecutionTest::class               // 4. Runs FOURTH - even more tests
    // Add more test classes here in desired order
)
class IntegrationTestSuite
```

Run the suite:

```
# Run all tests in correct order
./gradlew test --tests IntegrationTestSuite
```

What happens:

1. ✅ Container starts (or reuses existing)
2. ✅ Schema created from JPA entities (if not exists)
3. ✅ `DataSetupTest` runs → Injects data to database
4. ✅ `CommandInvokerIntegrationTest` runs → Uses injected data
5. ✅ `WorkspaceFeatureTest` runs → Uses same data
6. ✅ `ScriptExecutionTest` runs → Uses same data
7. ✅ All tests share same database connection and data

Execution Order Options

Option 1: Test Suite with @SelectClasses (Recommended ✅)

Use: `@Suite` with `@SelectClasses`

Example:

```
@Suite
@SelectClasses(
    DataSetupTest::class,
    OtherTest::class
)
class IntegrationTestSuite
```

Pros:

- ✅ Explicit order control
- ✅ Clear which tests run in which order
- ✅ Reliable and deterministic
- ✅ Easy to maintain

Cons:

- ⚠ Must maintain the list manually when adding new tests

When to use: When you need guaranteed execution order (recommended)

Option 2: Class Order Annotation

Use: `@TestClassOrder(ClassOrderer.OrderAnnotation)` with `@Order` on classes

Example:

```
@Order(1)
class DataSetupTest { ... }

@Order(2)
class CommandInvokerIntegrationTest { ... }
```

Pros:

- ✅ Annotation-based ordering
- ✅ Works with test discovery

Cons:

- ⚠ Requires JUnit 5.8+
- ⚠ Less explicit than test suite
- ⚠ Can be harder to see full test flow

When to use: When you prefer annotation-based configuration

Option 3: Alphabetical Naming

Use: Name classes like `Test01_DataSetup`, `Test02_Integration`

Example:

```
class Test01_DataSetupTest { ... }
class Test02_CommandInvokerIntegrationTest { ... }
```

Pros:

- ✅ Simple naming convention

Cons:

- ❌ Fragile - depends on naming
- ❌ Not reliable across all test runners
- ❌ Makes test names awkward

When to use: Never (use Test Suite instead)

Adding More Test Classes to Suite

To add a new test class to the execution order:

1. **Create your test class:**

```
@SpringBootTest
@Import(TestcontainersConfiguration::class)
@ActiveProfiles("test")
class MyNewFeatureTest(
    private val myService: MyService
) {
    @Test
    fun `test my new feature`() {
        // Test uses data from DataSetupTest
    }
}
```

2. **Add to IntegrationTestSuite:**

```
@Suite
@SelectClasses(
    DataSetupTest::class,                // 1. Data setup
    CommandInvokerIntegrationTest::class,  // 2. Existing tests
    MyNewFeatureTest::class              // 3. Your new test ← Add here
)
class IntegrationTestSuite
```

3. **Run the suite:**

```
./gradlew test --tests IntegrationTestSuite
```

Running Tests Individually vs Suite

Run Individual Test (Development)

```
# Run just data setup test
./gradlew test --tests DataSetupTest

# Run specific feature test
./gradlew test --tests CommandInvokerIntegrationTest
```

Use when:

- ✅ Developing a specific test
- ✅ Quick feedback loop
- ✅ Debugging single test

Note: Data from DataSetupTest may not be available if you run other tests individually!

Run Entire Suite (Recommended)

```
# Run all tests in correct order
./gradlew test --tests IntegrationTestSuite
```

Use when:

- ✅ Running full test suite
- ✅ CI/CD pipeline
- ✅ Before committing code
- ✅ Need guaranteed data availability

Run All Tests (No Order Guarantee)

```
# Runs all tests, but order NOT guaranteed
./gradlew test
```

Use when:

- ⚠ Tests are independent (don't rely on each other)
- ⚠ You understand the risks

Warning: Without test suite, execution order is not guaranteed! Tests may run in random order.

Dependencies Between Tests

Important: JUnit 5 Test Suites don't have built-in dependency management. Tests in the suite run sequentially but:

- ✅ Each test is independent from JUnit's perspective
- ✅ Data sharing happens through database (not in-memory)
- ✅ If one test fails, subsequent tests still run

Example flow:

```
DataSetupTest runs
  ↓ (data saved to database)
DataSetupTest passes/fails
  ↓ (subsequent tests run regardless)
CommandInvokerIntegrationTest runs
  ↓ (reads data from database)
CommandInvokerIntegrationTest passes/fails
  ↓ (and so on...)
```

If DataSetupTest fails:

- ❌ Data may not be created properly
- ⚠ Subsequent tests may fail due to missing data
- 💡 Check DataSetupTest first when debugging suite failures

Test Suite Best Practices

✅ DO:

- ✅ Put data setup tests first in the suite
- ✅ Order tests from simple to complex
- ✅ Group related tests together
- ✅ Use descriptive suite display names
- ✅ Document test dependencies in comments

Example with documentation:

```
@Suite
@SuiteDisplayName("Complete Integration Test Suite")
@SelectClasses(
    // Phase 1: Data Setup
    DataSetupTest::class,              // Creates workspaces, folders,
scripts

    // Phase 2: Core Features
    CommandInvokerIntegrationTest::class,  // Tests command invoker
    WorkspaceManagementTest::class,        // Tests workspace CRUD

    // Phase 3: Advanced Features
    ScriptExecutionTest::class,        // Tests script execution (needs
scripts)
    AiIntegrationTest::class           // Tests AI features (needs
configs)
)
class IntegrationTestSuite
```

❌ **DON'T:**

- ❌ Mix test types in same suite (unit tests + integration tests)
- ❌ Create circular dependencies between tests
- ❌ Rely on test execution order for independent tests
- ❌ Add too many tests to one suite (split into multiple suites if needed)

Multiple Test Suites

You can create multiple suites for different purposes:

```
// Fast tests - no data setup needed
@Suite
@SelectClasses(
    SimpleUnitTest::class,
    QuickIntegrationTest::class
)
class FastTestSuite

// Full integration - with data setup
@Suite
@SelectClasses(
    DataSetupTest::class,
    CompleteFeatureTest::class
)
class FullIntegrationTestSuite
```

Run specific suite:

```
./gradlew test --tests FastTestSuite
./gradlew test --tests FullIntegrationTestSuite
```

Part 4: Database Management

Connecting to Test Database

Get Connection Info

When you run tests, connection info is printed:

```
================================================================================
🔗 TESTCONTAINERS DATABASE CONNECTION INFO
================================================================================
📍 Host:     localhost
🔌 Port:     52106  ← USE THIS PORT!
💾 Database: testdb
👤 Username: test
🔑 Password: test
🔗 JDBC URL: jdbc:postgresql://localhost:52106/testdb
```

Or use the script:

```
./get-db-connection.sh
```

Connect with GUI Tools

DataGrip / IntelliJ Database

1. New Data Source → PostgreSQL
2. Host: `localhost` (NOT `http://localhost`)
3. Port: `52106` (from console output)
4. Database: `testdb`
5. User: `test`
6. Password: `test`

DBeaver / TablePlus / pgAdmin

Same settings as above.

Common mistake: Using `http://localhost` instead of just `localhost`

Find Port Manually

```
# Check running containers
docker ps | grep postgres

# Output shows port mapping:
# 0.0.0.0:52106->5432/tcp
# The first number (52106) is your port
```

Container Lifecycle

How Container Reuse Works

```
First Test Run:
  1. Container doesn't exist
  2. Testcontainers creates PostgreSQL container
  3. Assigns random port (e.g., 52106)
  4. Container tagged for reuse
  5. Test runs
  6. Container stays alive ✅

Second Test Run:
  1. Testcontainers looks for reusable container
  2. Finds existing container
  3. Reuses same container (same port!)
  4. Test runs (much faster!)
  5. Container stays alive ✅
```

Key: Container port stays the same until you manually stop it or restart Docker.

Container Reuse Requirements

Both settings required:

1. In code:

```
.withReuse(true)  // TestcontainersConfiguration.kt
```

2. Global config:

```
testcontainers.reuse.enable=true  # ~/.testcontainers.properties
```

Without both, container is destroyed after each test run.

Stopping Containers

Option 1: Restart Docker (Simplest)

```
# Click Docker icon in menu bar → Restart
# Or:
osascript -e 'quit app "Docker"' && open -a Docker
```

Pros:

- ✅ Simplest method
- ✅ Cleans up everything
- ✅ No commands needed

Cons:

- ⚠ Stops ALL containers (not just test containers)

Option 2: Stop Test Container Only

```
# Use the script
./stop-test-db.sh

# Or manually
docker stop $(docker ps -q --filter ancestor=postgres:15-alpine)
docker rm $(docker ps -aq --filter ancestor=postgres:15-alpine)
```

Pros:

- ✅ Only stops test container
- ✅ Other containers keep running

Option 3: Automatic Stop (Change Config)

In `TestcontainersConfiguration.kt`:

```
.withReuse(false)  // Container stops after tests
```

Pros:

- ✅ Fully automatic
- ✅ No manual cleanup

Cons:

- ⚠ Slower test runs (container recreated each time)

When to Stop

Stop when:

- ✅ Done testing for the day
- ✅ Need to free the port
- ✅ Want clean slate

Keep running when:

- ✅ Still actively testing
- ✅ Want fast test runs
- ✅ Inspecting database with GUI

Part 5: Understanding Your Setup

Spring vs Prisma

Question: Is there Spring + Prisma integration?

Answer: NO ❌

Technology	Ecosystem	ORM
Prisma	Node.js/TypeScript	Prisma ORM
Spring	Java/Kotlin	JPA/Hibernate

Your setup:

- ✅ Spring Boot with JPA/Hibernate
- ✅ 16 JPA entities with `@Entity` annotations
- ✅ Hibernate generates schema from entities
- ❌ Prisma NOT used in tests (only in production if needed)

Why Not Use Prisma for Tests?

Problems:

- ❌ Requires Node.js/npm
- ❌ Prisma schema must be manually synced with JPA entities
- ❌ Two sources of truth (Prisma schema + JPA entities)
- ❌ Extra complexity

Solution (Current):

- ✅ JPA entities are single source of truth
- ✅ Hibernate auto-generates schema from entities
- ✅ No external dependencies
- ✅ Schema always matches entities

JPA Foreign Keys

Question: Can JPA automatically create foreign keys like Prisma?

Answer: YES! ✅

How it works:

```kotlin
// Your JPA entity
@Entity
class ScriptsFolder {
    @OneToMany
    @JoinTable(
        name = "rel_workspace_folder",
        joinColumns = [JoinColumn(name = "workspace_id")],
        inverseJoinColumns = [JoinColumn(name = "folder_id")]
    )
    var folders: MutableSet<ScriptsFolder>
}
```

Hibernate generates:

```sql
CREATE TABLE rel_workspace_folder
(
    workspace_id INTEGER NOT NULL,
    folder_id    INTEGER NOT NULL,
    CONSTRAINT fk_workspace
        FOREIGN KEY (workspace_id) REFERENCES workspace (id),
    CONSTRAINT fk_folder
        FOREIGN KEY (folder_id) REFERENCES scripts_folder (id)
);
```

All foreign key constraints are automatically created!

JPA Relationship Annotations

Annotation	Creates	Use For
`@ManyToOne`	FK column	Many-to-one relationship

Annotation	Creates	Use For
`@OneToMany`	FK in other table	One-to-many relationship
`@JoinColumn`	Direct FK	Single foreign key
`@JoinTable`	Join table + 2 FKs	Many-to-many relationship

Example from your entities:

- ✅ Workspace → Folders (join table)
- ✅ Folder → Scripts (join table)
- ✅ Folder → Subfolder (self-referencing)
- ✅ All cascade operations work

SQLite vs PostgreSQL in Tests

Question: Why SQLite error in tests?

Problem: Test was using SQLite instead of PostgreSQL.

Root cause: `DatabaseConfig` was active in test profile, creating SQLite datasource.

Solution: Added `@Profile("!test")` to exclude it from tests:

```
@Configuration
@Profile("!test")  // ← Only active when NOT test
class DatabaseConfig {
    @Bean
    fun dataSource(): DataSource {
        // SQLite for production
    }
}
```

Result:

- ✅ Production: Uses SQLite
- ✅ Tests: Uses PostgreSQL (from Testcontainers)

Why Different Databases?

Benefits:

1. ✅ PostgreSQL has full SQL support (better for tests)
2. ✅ SQLite is lightweight (good for desktop app)
3. ✅ Tests catch SQL compatibility issues
4. ✅ JPA abstracts away database differences

Your setup:

- Production: SQLite (configured in DatabaseConfig)
- Tests: PostgreSQL (configured by Testcontainers)

Tables Dropping After Tests

Question: Why do tables drop after tests?

Answer: This is expected behavior based on `ddl-auto` setting.

Options

```
# Option 1: create-drop (Clean slate)
spring.jpa.hibernate.ddl-auto: create-drop
```

- ✅ Creates tables on start
- ✅ Drops tables on end
- ✅ Clean every run

```
# Option 2: create (Keep for inspection) - CURRENT
spring.jpa.hibernate.ddl-auto: create
```

- ✅ Creates tables on start
- ✅ Keeps tables on end
- ✅ Can inspect with GUI

Current setting: `create` (tables stay after tests)

What Happens

With `create-drop`:

```
Test starts → Creates tables → Tests run → Drops tables → Clean
```

With `create` (current):

```
Test starts → Creates tables → Tests run → Keeps tables → Can inspect
```

On next run:

```
Drops old tables → Creates new tables → Fresh data
```

Part 6: Troubleshooting

Common Issues

Issue 1: "Docker not running"

Symptoms:

```
Error: Could not find Docker environment
```

Solution:

```
open -a Docker  # Start Docker
docker ps       # Verify
```

Issue 2: "Container not reused"

Symptoms: New container created every test run (slow)

Solution:

1. Check `~/.testcontainers.properties` exists:

   ```
   cat ~/.testcontainers.properties
   # Should show: testcontainers.reuse.enable=true
   ```

2. Create if missing:

   ```
   echo "testcontainers.reuse.enable=true" > ~/.testcontainers.properties
   ```

Issue 3: "Constructor injection not working"

Symptoms:

```
No ParameterResolver registered for parameter [...]
```

Solution: Check `src/test/resources/junit-platform.properties` exists:

```
spring.test.constructor.autowire.mode=all
```

Issue 4: "SQLite error in tests"

Symptoms:

```
org.sqlite.SQLiteException: [SQLITE_ERROR] SQL error
```

Solution: Verify `DatabaseConfig` has `@Profile("!test")`:

```
@Configuration
@Profile("!test")  // ← Must be present
class DatabaseConfig { ... }
```

Issue 5: "Data not available in second test"

Symptoms: CommandInvokerIntegrationTest can't find data from DataSetupTest

Possible causes:

1. ❌ Using `create-drop` instead of `create`
2. ❌ Not running via test suite
3. ❌ Wrong execution order

Solution:

1. Check `application-test.yml`:

   ```
   spring.jpa.hibernate.ddl-auto: create  # NOT create-drop
   ```

2. Run via test suite:

   ```
   ./gradlew test --tests IntegrationTestSuite
   ```

Issue 6: "Can't connect with GUI tool"

Symptoms: Connection refused

Common mistakes:

- ❌ Using `http://localhost` → Should be just `localhost`
- ❌ Using port 5432 → Should be mapped port (e.g., 52106)
- ❌ Container stopped → Check with `docker ps | grep postgres`

Solution:

1. Get correct port:

   ```
   ./get-db-connection.sh
   ```

2. Use settings from output:

   - Host: `localhost` (no http://)
   - Port: from script output
   - Database: `testdb`
   - User: `test`
   - Password: `test`

FAQ

Q: How do I run tests?

```
./gradlew test
```

Q: How do I run tests with data injection?

```
./gradlew test --tests IntegrationTestSuite
```

Q: How do I connect to the test database?

```
./get-db-connection.sh
```

Then use the connection info in your GUI tool.

Q: How do I stop the test container?

Option 1 (Simplest): Restart Docker **Option 2:** Run `./stop-test-db.sh` **Option 3:** Change config to `withReuse(false)`

Q: Do I need Docker for tests?

Yes, Docker must be running for integration tests with Testcontainers.

Q: Can I use SQLite for tests instead?

Not recommended. SQLite has limited SQL features. PostgreSQL provides:

- ✅ Full SQL support
- ✅ Foreign key constraints
- ✅ Production-like behavior

Q: Why does the port change?

The port is **dynamically assigned** by Docker. But with `withReuse(true)`, the same container (and same port) is reused until you stop it.

Q: How do I add more test data?

Edit `DataSetupTest.kt` and add your commands:

```kotlin
@Test
@Order(1)
fun `01 - setup workspaces`() {
    commandInvoker.invoke(CreateWorkspaceCommand("My Workspace"))
}
```

Q: How do I verify foreign keys were created?

1. Run test
2. Connect with GUI tool (DataGrip, DBeaver)
3. View table structure
4. Check "Foreign Keys" tab

Or run SQL:

```sql
SELECT *
FROM information_schema.table_constraints
WHERE constraint_type = 'FOREIGN KEY';
```

Q: What's the difference between `create` and `create-drop`?

Setting	Tables After Test	Use For

Setting	Tables After Test	Use For
`create`	Stay	Development (can inspect)
`create-drop`	Dropped	CI/CD (clean slate)

Q: Why 16 entities? What are they?

You have 16 JPA entities in your project:

- Workspace, ScriptsFolder, ShellScript
- AiProfile, ModelConfig, ScriptAiConfig
- Event, HistoricalShellScript
- And 8 more...

All automatically get tables + foreign keys created by Hibernate!

Quick Commands Reference

```
# Start Docker
open -a Docker

# Run all tests
./gradlew test

# Run test suite with data injection
./gradlew test --tests IntegrationTestSuite

# Run specific test
./gradlew test --tests CommandInvokerIntegrationTest

# Get database connection info
./get-db-connection.sh

# Stop test container
./stop-test-db.sh

# Create testcontainers config
./create-testcontainers-config.sh

# Check container is running
docker ps | grep postgres

# Restart Docker (stops all containers)
# Click Docker icon → Restart
```

File Reference

Configuration Files

- `src/test/resources/application-test.yml` - Test config
- `src/test/resources/junit-platform.properties` - Constructor injection
- `~/.testcontainers.properties` - Container reuse

Test Files

- `src/test/kotlin/com/scriptmanager/integration/`
    - `DataSetupTest.kt` - Data injection
    - `CommandInvokerIntegrationTest.kt` - Example test
    - `IntegrationTestSuite.kt` - Test execution order

Config Classes

- `src/test/kotlin/com/scriptmanager/config/`
    - `TestcontainersConfiguration.kt` - Container setup

Helper Scripts

- `get-db-connection.sh` - Get connection info
- `stop-test-db.sh` - Stop container
- `create-testcontainers-config.sh` - Setup config

Summary

What You Have

✅ **Spring Boot 3.2** with Kotlin ✅ **PostgreSQL** via Testcontainers ✅ **16 JPA entities** with automatic schema generation ✅ **Container reuse** for fast tests ✅ **Data injection** test pattern ✅ **Test execution order** control ✅ **Foreign keys** automatically created ✅ **GUI connection** support

Key Features

- 🚀 **Fast:** Container reuse makes subsequent runs seconds
- 🔒 **Production-like:** Real PostgreSQL, not in-memory
- 🧪 **Isolated:** Clean database for each test suite run
- 📊 **Inspectable:** Connect with GUI tools
- 🎯 **Controlled:** Test execution order guaranteed

Next Steps

1. **Run tests:** `./gradlew test`
2. **Add test data:** Edit `DataSetupTest.kt`
3. **Inspect database:** Use `./get-db-connection.sh` + GUI tool
4. **Write more tests:** Follow the patterns in the guide

🎉 **You're all set! Everything is configured and ready to use!**

Questions? Refer to the Troubleshooting and FAQ sections.

This guide consolidates all testing and Testcontainers documentation. No other testing guides needed.