Testing Guide - Complete Reference

> **One guide for all your testing needs: Unit tests AND Integration tests**

🎉 Tests Now Work!

If you had issues: "I cannot find any test in gradle task" - **This is now fixed!**

What Was Fixed

1. Created Global Configuration File ✅

File: `src/test/resources/junit-platform.properties`

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

What this does:

- Enables constructor injection for ALL tests automatically
- No need for `@TestConstructor` annotation on each test
- Cleaner, more maintainable test code

2. Constructor Injection Now Works ✅

Before (Broken):

```
@SpringBootTest
class MyTest(
    private val commandInvoker: CommandInvoker
) {
    // ❌ Error: No ParameterResolver registered
}
```

After (Works!):

```
@SpringBootTest
class MyTest(
    private val commandInvoker: CommandInvoker
) {
    // ✅ Automatically injected!
}
```

3. Removed All JdbcTemplate References ✅

- This project uses **JPA repositories** exclusively
- No raw SQL queries
- All documentation updated to reflect repository-based testing

4. Consolidated All Documentation ✅

- Merged 7+ scattered markdown files into this ONE guide
- Includes both integration and unit testing
- Complete troubleshooting and examples

Quick Verification

Run these commands to verify everything works:

```
# 1. Check global config exists
cat src/test/resources/junit-platform.properties
# Output: spring.test.constructor.autowire.mode=all

# 2. Start Docker
open -a Docker

# 3. Run tests
./gradlew test
# ✅ Tests should pass!
```

📚 What's in This Guide

This is the **ONLY testing guide** you need to read. It covers:

1. **Integration Tests** (Testcontainers + PostgreSQL + Repositories)
2. **Unit Tests** (InMemoryEventQueue for command handlers)
3. When to use each approach
4. Complete examples for both

Table of Contents

Integration Testing (Recommended for Most Tests)

Unit Testing (For Fast Command Handler Tests)

Part 1: Integration Testing

> **Use this for:** End-to-end tests, database verification, full business flows

Quick Start - Integration Tests

TL;DR - Get Testing in 3 Steps

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

# 2. Run all tests
./gradlew test

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

What You Get

- ✅ Real PostgreSQL database in Docker
- ✅ Automatic Prisma schema application
- ✅ Production-like testing environment
- ✅ Fast test execution with container reuse
- ✅ Tests complete business flows

Prerequisites

Required for Integration Tests

⚠ IMPORTANT: Docker must be running!

1. **Docker Desktop** (macOS/Windows) or **Docker Engine** (Linux)
2. **2GB free disk space** (for PostgreSQL image)
3. **Internet connection** (first run only)

Optional

4. **Node.js and npm** (for Prisma schema application)
5. **Prisma CLI** (can use `npx prisma`)

Verification

```
# Check Docker
docker ps

# If fails, start Docker
open -a Docker  # macOS

# Check Node.js (optional)
node --version
npm --version
```

Setup & Configuration

Dependencies (Already Configured)

```
dependencies {
    testImplementation("org.springframework.boot:spring-boot-
testcontainers")
    testImplementation("org.testcontainers:postgresql:1.19.3")
    testImplementation("org.postgresql:postgresql:42.7.1")
}
```

TestcontainersConfiguration

```
@TestConfiguration
class TestcontainersConfiguration {
    @Bean
    @ServiceConnection
    fun postgresContainer(): PostgreSQLContainer<*> {
        val container =
PostgreSQLContainer(DockerImageName.parse("postgres:15-alpine"))
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test")
            .withReuse(true)  // Fast subsequent runs

        container.start()
        applyPrismaSchemaIfAvailable(container)
        return container
    }
}
```

Writing Integration Tests

Important: Constructor Injection Setup

For constructor injection to work in Spring Boot tests with Kotlin, we have a **global configuration** file:

File: `src/test/resources/junit-platform.properties`

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

This means you **don't need** `@TestConstructor` annotation on each test - it's already configured globally!

Pattern 1: Context Loading Test

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

Pattern 2: Command Execution Test

```
@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 in database
        val workspaces = workspaceRepository.findAll()
        assertEquals(1, workspaces.size)
        assertEquals("Test", workspaces[0].name)
    }
}
```

Pattern 3: Business Logic Test

```
@SpringBootTest
@Import(TestcontainersConfiguration::class)
@ActiveProfiles("test")
class FolderHierarchyTest(
    private val commandInvoker: CommandInvoker,
    private val folderRepository: FolderRepository,
    private val scriptRepository: ScriptRepository
) {
    @Test
    fun `deleting parent folder cascades to children`() {
        // Setup
        val parentId = createFolder("Parent")
        val childId = createFolder("Child", parentId)

        // Execute
        commandInvoker.invoke(DeleteFolderCommand(parentId))

        // Verify cascade
        assertFalse(folderRepository.existsById(parentId))
        assertFalse(folderRepository.existsById(childId))
    }
}
```

Running Integration Tests

```
# Run all tests
./gradlew test

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

# With detailed output
./gradlew test --info
```

Troubleshooting

"Could not find Docker environment"

Solution:

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

"Prisma schema apply failed"

Solution: Prisma is optional. Tests will work with Hibernate DDL.

```
brew install node  # To enable Prisma
```

Tests are slow

Solution: Enable container reuse

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

Part 2: Unit Testing

> **Use this for:** Fast command handler tests without database

Quick Start - Unit Tests

TL;DR - Test Command Handlers Without Database

```
class CreateWorkspaceHandlerTest {
    private lateinit var eventQueue: InMemoryEventQueue
    private lateinit var handler: CreateWorkspaceHandler

    @BeforeEach
    fun setup() {
        eventQueue = InMemoryEventQueue()
        handler = CreateWorkspaceHandler()
    }

    @Test
    fun `emits workspace created event`() {
        // Execute
        handler.handle(eventQueue, CreateWorkspaceCommand("Test"))

        // Verify
        assertTrue(eventQueue.hasEventOfType<WorkspaceCreatedEvent>())

        val event = eventQueue.getFirstEventOfType<WorkspaceCreatedEvent>
()
        assertEquals("Test", event?.workspaceName)
    }
}
```

What You Get

- ✅ Fast execution (no database)
- ✅ No Docker required
- ✅ Easy event verification
- ✅ Test business logic in isolation

InMemoryEventQueue

What Is It?

An in-memory implementation of `EventQueue` for testing command handlers without database access.

Key Features

```
class InMemoryEventQueue : EventQueue {
    // Add events
    fun add(event: Any)
    fun addTransactional(event: Any)

    // Query events
    val allEvents: List<EventWrapper<Any>>
    val immediateEvents: List<EventWrapper<Any>>
    val postCommitEvents: List<EventWrapper<Any>>

    // Verify events (inline reified generics!)
    inline fun <reified T> hasEventOfType(): Boolean
    inline fun <reified T> getFirstEventOfType(): T?
    inline fun <reified T> getAllEventsOfType(): List<T>
}
```

Why Use It?

Aspect	InMemoryEventQueue	Database Tests
Speed	Milliseconds	Seconds
Setup	None	Docker required
Isolation	Pure unit test	Integration test
Focus	Business logic	Full flow

Writing Unit Tests

Example: Test Command Handler

```
class CreateScriptHandlerTest {
    private lateinit var eventQueue: InMemoryEventQueue
    private lateinit var scriptRepository: ScriptRepository  // Can mock
this
    private lateinit var handler: CreateScriptHandler

    @BeforeEach
    fun setup() {
        eventQueue = InMemoryEventQueue()
        scriptRepository = mockk()  // Or use real repo
        handler = CreateScriptHandler(scriptRepository)
    }

    @Test
    fun `creates script and emits event`() {
        // Setup
        val command = CreateScriptCommand(UUID.randomUUID(), "Test
Script")
        every { scriptRepository.save(any()) } returns script

        // Execute
        handler.handle(eventQueue, command)

        // Verify event
        assertTrue(eventQueue.hasEventOfType<ScriptCreatedEvent>())
        val event = eventQueue.getFirstEventOfType<ScriptCreatedEvent>()
        assertEquals("Test Script", event?.scriptName)

        // Verify repository called
        verify { scriptRepository.save(any()) }
    }
}
```

Example: Test Event Dispatch Timing

```
@Test
fun `immediate events vs transactional events`() {
    handler.handle(eventQueue, command)

    // Check immediate events
    val immediate = eventQueue.immediateEvents
    assertEquals(1, immediate.size)

    // Check transactional (post-commit) events
    val transactional = eventQueue.postCommitEvents
    assertEquals(1, transactional.size)
}
```

Part 3: Best Practices

Constructor Injection Best Practices

✅ Always Use Constructor Injection

```
// ✅ GOOD - Constructor injection
@SpringBootTest
class MyTest(
    private val commandInvoker: CommandInvoker,
    private val repository: MyRepository
) {
    @Test
    fun myTest() {
        // Dependencies guaranteed initialized
    }
}

// ❌ BAD - Field injection
@SpringBootTest
class MyTest {
    @Autowired
    lateinit var commandInvoker: CommandInvoker

    @Test
    fun myTest() {
        // Risk of accessing before initialization
    }
}
```

Why Constructor Injection?

1. **No `lateinit`** - Guaranteed initialization
2. **Immutable** - Use `val` instead of `var`
3. **Explicit** - Dependencies visible in signature
4. **Kotlin idiomatic** - Recommended best practice
5. **Easier to test** - Can create with mocks easily

Only Inject What You Use

```
// ❌ Over-injection
class MyTest(
    private val service1: Service1,
    private val service2: Service2,
    private val repository1: Repository1,
    private val repository2: Repository2
) {
    @Test
```

```
    fun `simple test`() {
        service1.doSomething()  // Only using service1!
    }
}

// ✅ Minimal injection
class MyTest(
    private val service1: Service1
) {
    @Test
    fun `simple test`() {
        service1.doSomething()
    }
}
```

When to Use Which

Use Integration Tests When:

✅ Testing complete business flows ✅ Verifying database persistence ✅ Testing repository queries ✅ Verifying transactions ✅ Testing cascade operations ✅ End-to-end scenarios

Example: "Does creating a workspace actually save to database?"

Use Unit Tests When:

✅ Testing command handler logic in isolation ✅ Verifying event emission ✅ Testing validation logic ✅ Fast feedback loops ✅ No database needed ✅ Testing edge cases

Example: "Does the handler emit the correct event type?"

Recommended Test Pyramid

```
        Unit Tests (70%)
       /              \
      /  InMemory      \
     /   EventQueue     \
    /____________________\
   Integration Tests (30%)
  /                      \
 /   Testcontainers       \
/__________________________ \
    PostgreSQL Database
```

Strategy:

- Write **many unit tests** with `InMemoryEventQueue` (fast, focused)
- Write **fewer integration tests** with Testcontainers (comprehensive, slow)

FAQ

Q: Should I use integration or unit tests?

A: Both! Use unit tests for fast feedback on business logic. Use integration tests to verify everything works together.

Q: Do I need Docker for all tests?

A: No! Unit tests with `InMemoryEventQueue` don't need Docker. Only integration tests need Docker.

Q: Can I test without Prisma schema?

A: Yes! If Prisma schema isn't found, Hibernate DDL will create tables automatically.

Q: What if Docker isn't available in CI?

A: Most CI providers have Docker built-in (GitHub Actions, GitLab CI). For those that don't, focus on unit tests.

Q: How do I speed up tests?

A:

1. Write more unit tests (milliseconds)
2. Enable container reuse for integration tests
3. Use `@Transactional` for auto-rollback

Q: Can I use @Autowired instead of constructor injection?

A: Technically yes, but constructor injection is Kotlin best practice (safer, immutable, more idiomatic).

Summary

Documentation Files

File	Purpose	Read This?
This file	Complete testing guide	✅ YES
`README.md`	Project overview	For context
`CommandInvokerIntegrationTest.kt`	Example integration test	Reference
`InMemoryEventQueueTest.kt`	Example unit test	Reference

Testing Approaches

Type	Tool	Speed	Use For
Unit	InMemoryEventQueue	⚡ ⚡ ⚡ Fast	Business logic

Type	Tool	Speed	Use For
Integration	Testcontainers	⚡ Slower	Full flows

Quick Commands

```
# Integration tests
open -a Docker && ./gradlew test

# Unit tests only
./gradlew test --tests *HandlerTest

# Check setup
./check-test-prerequisites.sh
```

🎉 **You now have everything you need for both unit AND integration testing!**

Next Steps:

1. Write unit tests with `InMemoryEventQueue` for fast feedback
2. Write integration tests with Testcontainers for confidence
3. Follow constructor injection pattern
4. Enjoy fast, comprehensive testing! 🚀