Testing Strategy for Script Manager Backend

🎯 Overview

This document outlines the comprehensive testing strategy for the Script Manager backend, which uses:

- **Spring Boot + Kotlin**
- **Event-Driven Architecture** (Command/Event pattern)
- **PostgreSQL** (via Testcontainers in tests)
- **20 Domain Commands** across workspace, folder, and script management

---

📊 Test Categories

1. **Integration Tests** (PRIMARY STRATEGY)

Purpose: Test complete workflows end-to-end with real database interactions

What We Test:

- ✅ Command execution produces correct results
- ✅ Events are emitted with accurate payloads
- ✅ Data persists correctly in PostgreSQL
- ✅ Business logic and validation rules
- ✅ Parent-child relationships and cascades

Technology Stack:

- `@SpringBootTest` - Full Spring context
- **Testcontainers** - Real PostgreSQL database
- `BaseTest` - Automatic event cleanup between tests
- **No mocking** - Real repositories, real database

Example:

```
@SpringBootTest
class WorkspaceCreationTest(
    private val eventRepository: EventRepository,
    private val commandInvoker: CommandInvoker,
    private val objectMapper: ObjectMapper
) : BaseTest(eventRepository) {

    @Test
    fun `should create workspace and emit event`() {
        // Act
        val result =
commandInvoker.invoke(CreateWorkspaceCommand("MyWorkspace"))

        // Assert command result
```

```
        assertNotNull(result.id)

        // Assert event emitted
        val events = eventRepository.findAll()
            .filter { it.eventType == "WorkspaceCreatedEvent" }
        assertEquals(1, events.size)
    }
}
```

🗂️ Test Organization Structure

Organize tests by **domain aggregate** instead of one test per command:

```
src/test/kotlin/com/scriptmanager/integration/
├── BaseTest.kt                    # Shared base class with event cleanup
├── workspace/
│   ├── WorkspaceCreationTest.kt   # CreateWorkspace
│   ├── WorkspaceManagementTest.kt # Update, Delete, Reorder
│   └── WorkspaceFolderTest.kt     # AddFolder, RemoveFolder,
ReorderFolders
├── folder/
│   ├── FolderCreationTest.kt      # CreateFolder, CreateFolderInWorkspace
│   ├── FolderManagementTest.kt    # Update, Delete, Reorder, AddSubfolder
│   └── FolderHierarchyTest.kt     # Parent-child relationships
├── script/
│   ├── ScriptLifecycleTest.kt     # Create, Update, Delete
│   ├── ScriptHistoryTest.kt       # CreateScriptHistory
│   └── MarkdownTest.kt            # CreateMarkdown, UpdateMarkdown
└── appstate/
    └── AppStateTest.kt            # UpdateAppState
```

Benefits:

- Tests are grouped by feature area (easier to find)
- Related operations tested together (better coverage)
- Tests can share setup code (e.g., "create workspace" helper)

🧪 What to Test for Each Command

For each domain command, verify:

✅ Happy Path

- Command executes successfully
- Returns expected result
- Event is emitted with correct type
- Event payload matches expected data

- Data persists in database

✅ Business Logic

- Validation rules (e.g., name requirements, length limits)
- State transitions (e.g., can't delete workspace with folders)
- Uniqueness constraints
- Default values

✅ Edge Cases

- Empty inputs (if allowed)
- Very long inputs (boundary testing)
- Special characters in names/content
- Non-existent IDs (error handling)

✅ Relationships

- Parent-child associations (workspace → folder → script)
- Cascade behaviors (what happens when parent is deleted?)
- Ordering/reordering operations
- Multiple levels of nesting (if supported)

✅ Event Verification

- Event type is correct
- Event marked as `success = true`
- Event payload deserializes correctly
- Event contains all necessary data for consumers

🔧 Testing Infrastructure

BaseTest Class

```
@SpringBootTest
@ActiveProfiles("test")
@Import(TestcontainersConfiguration::class)
abstract class BaseTest(
    private val eventRepository: EventRepository
) {
    @BeforeEach
    fun truncateEventsBeforeEachTest() {
        eventRepository.deleteAll()  // Clean slate for each test
    }
}
```

Key Features:

- ✅ **Automatic event cleanup** - No test pollution
- ✅ **Fast** - Only truncates events table, not schema recreation
- ✅ **Persistent schema** - Tables stay, data is cleaned
- ✅ **No @DirtiesContext** - Much faster than recreating Spring context

Testcontainers Configuration

```
@TestConfiguration(proxyBeanMethods = false)
class TestcontainersConfiguration {
    @Bean
    fun postgresContainer(): PostgreSQLContainer<*> {
        return PostgreSQLContainer("postgres:17-alpine")
            .withDatabaseName("testdb")
            .apply {
                start()
                applySchemaFromFile()  // Run schema.sql once on startup
            }
    }
}
```

Key Features:

- ✅ Real PostgreSQL (not H2 or embedded DB)
- ✅ Schema applied once on container start
- ✅ Same database for all tests (fast, no recreation)
- ✅ Automatic cleanup via `@BeforeEach` in BaseTest

📏 Test Coverage Guidelines

Minimum Coverage per Command:

1. **1 Happy Path Test** - Basic success scenario
2. **1 Event Verification Test** - Event emitted correctly
3. **1 Persistence Test** - Data saved to database
4. **1+ Edge Case Tests** - Error handling, boundaries

Recommended Coverage per Aggregate:

- **5-10 tests per domain area** (workspace, folder, script)
- **Test workflows**, not just individual commands
- **Test relationships** between entities

🚫 What NOT to Test

❌ Don't test framework code

- Spring Boot internals

- JPA/Hibernate behavior
- Jackson JSON serialization (unless custom)

❌ **Don't over-mock**

- Use real repositories (you're already doing this ✅)
- Use real database via Testcontainers
- Only mock external services (e.g., AWS S3, external APIs)

❌ **Don't need test queues**

- Your `eventRepository` is the source of truth
- Query events directly from database in tests
- No need for in-memory event queue for testing

🎬 Test Execution

Run All Tests

```
./gradlew test
```

Run Specific Test Class

```
./gradlew test --tests "WorkspaceCreationTest"
```

Run Tests by Package

```
./gradlew test --tests "com.scriptmanager.integration.workspace.*"
```

Test Reports

After running tests, view HTML report at:

```
build/reports/tests/test/index.html
```

🏗️ Test Suite Organization (JUnit 5)

Define Test Order with @TestMethodOrder

```

```

```
@SpringBootTest
@TestMethodOrder(MethodOrderer.OrderAnnotation::class)
class OrderedWorkflowTest : BaseTest(eventRepository) {

    @Test
    @Order(1)
    fun `step 1 - create workspace`() {
        ...
    }

    @Test
    @Order(2)
    fun `step 2 - add folder to workspace`() {
        ...
    }

    @Test
    @Order(3)
    fun `step 3 - add script to folder`() {
        ...
    }
}
```

Group Tests with @Nested

```
@SpringBootTest
class WorkspaceTest : BaseTest(eventRepository) {

    @Nested
    inner class Creation {
        @Test
        fun `should create workspace`() {
            ...
        }
    }

    @Nested
    inner class Updates {
        @Test
        fun `should update workspace name`() {
            ...
        }
    }

    @Nested
    inner class Deletion {
        @Test
        fun `should delete workspace`() {
            ...
        }
```

```
    }
}
```

Create Test Suite (Run Multiple Test Classes)

```
@Suite
@SelectPackages("com.scriptmanager.integration.workspace")
class WorkspaceTestSuite
```

📝 Best Practices

1. **Use Descriptive Test Names**

```
// ✅ Good
@Test
fun `should emit WorkspaceCreatedEvent when creating workspace`()

// ❌ Bad
@Test
fun testCreate()
```

2. **Follow AAA Pattern**

```
@Test
fun `test name`() {
    // Arrange - Setup test data
    val input = CreateWorkspaceCommand("Test")

    // Act - Execute the command
    val result = commandInvoker.invoke(input)

    // Assert - Verify results
    assertEquals("Test", result.name)
}
```

3. **Use Unique Test Data**

```
// ✅ Prevents conflicts between parallel tests
val workspaceName = "TestWorkspace_${System.currentTimeMillis()}"
```

4. **Verify Events in Every Test**

```
// Always check events were emitted
val events = eventRepository.findAll()
    .filter { it.eventType == "WorkspaceCreatedEvent" }
assertEquals(1, events.size)
assertTrue(events.first().success)
```

5. **Test Event Payloads**

```
// Deserialize and verify event contents
val payload = objectMapper.readValue<WorkspaceCreatedEvent>(event.payload)
assertEquals(workspaceName, payload.workspace.name)
```

🎯 Success Criteria

Your testing strategy is successful when:

✅ **All API endpoints have test coverage** (workspace, folder, script, AI, appstate) ✅ **All domains tested** (scriptmanager domain + ai domain) ✅ **Each test runs independently** (no shared state issues) ✅ **Tests run fast** (<5 minutes for full suite) ✅ **High confidence in changes** (can refactor safely) ✅ **Event-driven behavior verified** (all events tested) ✅ **Real database tested** (no H2/in-memory surprises in prod)

🚀 Next Steps

1. ✅ **BaseTest created** - Event cleanup working
2. ✅ **Testcontainers configured** - PostgreSQL running
3. ✅ **First test written** - SimpleEventTest passes
4. 🔄 **Expand coverage** - Add tests for all 20 commands
5. 🔄 **Group by aggregate** - Organize into workspace/folder/script packages
6. 🔄 **Add relationship tests** - Test parent-child cascades
7. 🔄 **Add edge case tests** - Error handling, boundaries

📚 Additional Resources

- **JUnit 5 Documentation**: https://junit.org/junit5/docs/current/user-guide/
- **Testcontainers**: https://testcontainers.com/
- **Spring Boot Testing**: https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing
- **AssertJ** (better assertions): https://assertj.github.io/doc/

Last Updated: January 3, 2026