Testing Spring Boot Applications Demystified

Workshop Slides

About Me

- [Your introduction here]
- Contact: [Your contact info]
- GitHub: [Your GitHub info]

Workshop Agenda

- Slot 1: Spring Boot Testing Basics (105 min)
- Slot 2: Sliced Testing (115 min)
- Slot 3: Integration Testing (90 min)
- Slot 4: Best Practices & Pitfalls (70 min)

Workshop Repository

- GitHub: [URL]
- Materials: 4 lab projects
- Each lab focuses on different testing techniques
- Domain: Library Management System

SECTION 1

Spring Boot Testing Basics

Development Environment Setup

- Java 21
- IDE (IntelliJ, VS Code, Eclipse)
- Maven
- Git

Maven Basics

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Maven Lifecycle

1. **validate**: validate project structure
2. **compile**: compile source code
3. **test**: run tests
4. **package**: package compiled code
5. **verify**: run integration tests
6. **install**: install in local repository
7. **deploy**: deploy to remote repository

Spring Boot Testing Basics

- Built on JUnit 5 (Jupiter)
- Mockito integration
- Spring Test Context Framework
- Auto-configuration for tests
- Sliced testing support

JUnit 5 & Mockito - The Foundation

```
@Test
void testBookService() {
    // Given
    Book book = new Book("123", "Test Book", "Test Author");
    when(bookRepository.findById("123")).thenReturn(Optional.of(book));

    // When
    Optional<Book> result = bookService.getBookById("123");

    // Then
    assertTrue(result.isPresent());
    assertEquals("Test Book", result.get().getTitle());
    verify(bookRepository).findById("123");
}
```

Design for Testability

Prefer Constructor Injection

```
// Hard to test
public class BookService {
    private final BookRepository bookRepository = new BookRepositoryImpl();
}

// Testable
public class BookService {
    private final BookRepository bookRepository;

    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }
}
```

Avoiding Static and Direct Instantiation

```
// Hard to test
public LocalDate getDueDate() {
    return LocalDate.now().plusDays(14);
}

// Testable
public LocalDate getDueDate(Clock clock) {
    return LocalDate.now(clock).plusDays(14);
}
```

Don't Overuse Mockito

- Mocks can make tests brittle
- Consider using real objects when:
  - They have no external dependencies
  - They're simple value objects
  - They're collections or other standard library classes

JUnit 5 Extensions

```
@ExtendWith(MockitoExtension.class)
class BookServiceTest {
    @Mock
    private BookRepository bookRepository;

    @InjectMocks
    private BookService bookService;
}
```

Exercise: Write a JUnit Jupiter Extension

Create a custom extension for timing tests:

```
public class TimingExtension implements BeforeTestExecutionCallback,
                                        AfterTestExecutionCallback {
    private static final Logger logger = LoggerFactory.getLogger(TimingExtension.class);

    @Override
    public void beforeTestExecution(ExtensionContext context) {
        getStore(context).put("start", System.currentTimeMillis());
    }

    @Override
    public void afterTestExecution(ExtensionContext context) {
        long start = getStore(context).remove("start", Long.class);
        long duration = System.currentTimeMillis() - start;
        logger.info("Test {} took {} ms", context.getDisplayName(), duration);
    }

    private Store getStore(ExtensionContext context) {
        return context.getStore(Namespace.create(getClass(), context.getRequiredTestMethod()));
    }
}
```

Refactoring Time-Based Code

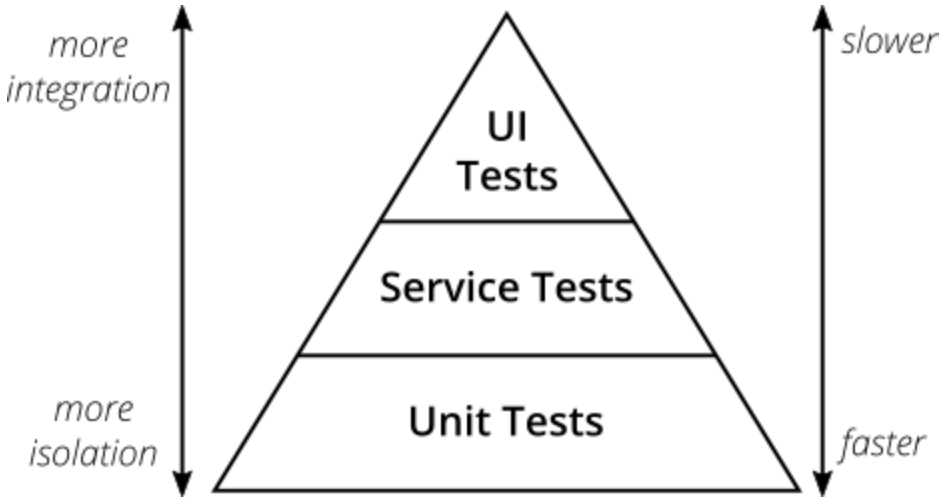
Before:

```
public boolean isOverdue(BookLoan loan) {
    return LocalDate.now().isAfter(loan.getDueDate());
}
```

After:

```
public boolean isOverdue(BookLoan loan, Clock clock) {
    return LocalDate.now(clock).isAfter(loan.getDueDate());
}
```

Spring's Testing Pyramid



- Unit Tests: Fast, focused, isolated
- Integration Tests: Verify components work together
- End-to-End Tests: Full application testing

Lab 1: Exercises

1. Write unit tests for the Book entity
2. Create a custom JUnit 5 extension
3. Refactor time-based code to use Clock
4. Write tests for the BookService class

SECTION 2

Sliced Testing

Sliced Testing in Spring Boot

- Test a specific "slice" of the application
- Focus on one layer at a time
- Faster than full application tests
- Clearer test boundaries
- Simplified configuration

Common Test Slices

- `@WebMvcTest` - Controller layer
- `@DataJpaTest` - Repository layer
- `@JsonTest` - JSON serialization/deserialization
- `@RestClientTest` - RestTemplate testing
- `@WebFluxTest` - WebFlux controller testing
- `@JdbcTest` - JDBC testing

When Unit Tests Are Not Enough

- HTTP constraints and semantics
- Security rules and authentication
- Request/response processing
- Data persistence behavior
- Transaction boundaries

@WebMvcTest

```
@WebMvcTest(BookController.class)
class BookControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private BookService bookService;
}
```

- Tests only the web layer
- Auto-configures MockMvc
- Lightweight context with web components
- No full application context

HTTP Semantics with MockMvc

```
@Test
void getBookByIsbnReturnsBookWhenFound() throws Exception {
    Book book = new Book("978-1-2345-6789-0", "Spring Boot Testing", "John Doe");
    when(bookService.findByIsbn("978-1-2345-6789-0")).thenReturn(Optional.of(book));

    mockMvc.perform(get("/api/books/978-1-2345-6789-0"))
           .andExpect(status().isOk())
           .andExpect(content().contentType(MediaType.APPLICATION_JSON))
           .andExpect(jsonPath("$.isbn", is("978-1-2345-6789-0")))
           .andExpect(jsonPath("$.title", is("Spring Boot Testing")));
}
```

Testing HTTP Status Codes

```
@Test
void getBookByIsbnReturnsNotFoundWhenMissing() throws Exception {
    when(bookService.findByIsbn("unknown")).thenReturn(Optional.empty());

    mockMvc.perform(get("/api/books/unknown"))
           .andExpect(status().isNotFound());
}
```

Testing Request Filters

```
@WebMvcTest({BookController.class, RequestLoggingFilter.class})
class FilterTest {
    @Test
    void filterShouldProcessRequest() throws Exception {
        mockMvc.perform(get("/api/books")
                .header("User-Agent", "Test"))
                .andExpect(status().isOk());

        // Verify filter behavior
    }
}
```

Testing Spring Security

```
@Test
@WithMockUser(roles = "ADMIN")
void createBookRequiresAdminRole() throws Exception {
    mockMvc.perform(post("/api/books")
            .contentType(MediaType.APPLICATION_JSON)
            .content("{\"isbn\":\"123\",\"title\":\"Test\"}"))
            .andExpect(status().isCreated());
}

@Test
void createBookWithoutAuthReturns401() throws Exception {
    mockMvc.perform(post("/api/books")
            .contentType(MediaType.APPLICATION_JSON)
            .content("{\"isbn\":\"123\",\"title\":\"Test\"}"))
            .andExpect(status().isUnauthorized());
}
```

Exercise: Testing a Filter and Secured Endpoints

1. Test a request logging filter
2. Test secured endpoints with different authentication scenarios
3. Verify correct HTTP status codes for various security conditions

@DataJpaTest

```java
@DataJpaTest
class BookRepositoryTest {
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private BookRepository bookRepository;
}
```

- Tests JPA repositories
- Auto-configures in-memory database
- Sets up appropriate transaction mgmt
- Provides TestEntityManager

EntityManager vs. Repository

```
@Test
void findByTitleShouldReturnBook() {
    // Using EntityManager to set up data
    Book savedBook = entityManager.persistAndFlush(
        new Book("123", "Spring Data JPA", "John Doe")
    );

    // Using Repository for the test
    List<Book> found = bookRepository.findByTitleContaining("Data");

    assertThat(found).hasSize(1);
    assertThat(found.get(0).getIsbn()).isEqualTo("123");
}
```

Transaction Boundaries and Flushing

```
@Test
void saveAndFlushImmediately() {
    Book book = new Book("123", "Test", "Author");
    bookRepository.save(book);  // No SQL executed yet

    entityManager.flush();  // Forces SQL execution
    entityManager.clear();  // Clears persistence context

    Book found = bookRepository.findById("123").orElse(null);
    assertThat(found).isNotNull();
}
```

When SQL Gets Executed

- `save()` - No immediate SQL (only on flush)
- `saveAndFlush()` - Executes SQL immediately
- `findById()` - Executes SQL if entity not in context
- Transaction completion - Triggers flush
- `EntityManager.flush()` - Manual flush

Testing Native Queries

```
@Query(value = "SELECT * FROM books WHERE " +
               "LOWER(title) LIKE LOWER(CONCAT('%', :keyword, '%')) OR " +
               "LOWER(author) LIKE LOWER(CONCAT('%', :keyword, '%'))",
       nativeQuery = true)
List<Book> search(String keyword);
```

How to test?

- Ensure correct data setup
- Verify query results with different parameters
- Test edge cases and special characters

Lazy Loading

```java
@Entity
public class Book {
    // ...

    @OneToMany(mappedBy = "book", fetch = FetchType.LAZY)
    private Set<Review> reviews;
}

@Test
void demonstrateLazyLoading() {
    Book book = entityManager.find(Book.class, "123");

    // reviews not loaded yet
    assertFalse(Hibernate.isInitialized(book.getReviews()));

    // accessing the collection triggers loading
    int reviewCount = book.getReviews().size();

    // now it's loaded
    assertTrue(Hibernate.isInitialized(book.getReviews()));
}
```

Entity Lifecycle Requirements

- Entities need no-arg constructor (can be protected)
- Prefer field access or property access consistently
- Understand cascade types
- Use proper fetch types (LAZY vs EAGER)

Exercise: Working with JPA Tests

1. Write DataJpaTest for custom repository methods
2. Test lazy loading behavior
3. Test entity lifecycle and transaction boundaries
4. Write test for a native query

Lab 2: Putting It All Together

- Test controllers with security constraints
- Test repository methods with proper transaction handling
- Understand when SQL statements are executed
- Verify proper HTTP semantics and JPA behavior

SECTION 3

Integration Testing

Integration Testing with Spring Boot

- Test multiple application layers together
- Verify component interactions
- Test with real external dependencies
- Ensure end-to-end functionality

@SpringBootTest

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class BookApplicationIntegrationTest {
    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private BookRepository bookRepository;
}
```

- Loads full application context
- Starts embedded server with random port
- Configures TestRestTemplate
- Much heavier than sliced tests

Testing from the Outside

```
@Test
void createBookShouldStoreInDatabase() {
    // Prepare test data
    Book newBook = new Book("123", "Test Book", "Test Author");

    // Make HTTP request
    ResponseEntity<Book> response = restTemplate.postForEntity(
        "/api/books", newBook, Book.class
    );

    // Verify HTTP response
    assertEquals(HttpStatus.CREATED, response.getStatusCode());
    assertNotNull(response.getBody());

    // Verify database state
    Optional<Book> stored = bookRepository.findById("123");
    assertTrue(stored.isPresent());
    assertEquals("Test Book", stored.get().getTitle());
}
```

External Dependencies in Tests

- Need real databases, message brokers, services
- Cannot use in-memory alternatives for everything
- Must be repeatable and isolated
- Should be fast and reliable

Enter: **Testcontainers**

Testcontainers

- Docker containers for tests
- Real database instances
- Consistent test environment
- Automatic cleanup
- Wide range of supported systems

PostgreSQL with Testcontainers

```
@SpringBootTest
@Testcontainers
class DatabaseIntegrationTest {
    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:14.5")
        .withDatabaseName("testdb")
        .withUsername("test")
        .withPassword("test");

    @DynamicPropertySource
    static void properties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }
}
```

Test Data Management

- Each test should start with a known state
- Tests should not interfere with each other
- Options:
  - Truncate tables between tests
  - Transaction rollback
  - Separate schemas per test
  - Database resets

Test Data Cleanup Example

```
@BeforeEach
void setUp() {
    // Clean database before test
    jdbcTemplate.execute("DELETE FROM book_loans");
    jdbcTemplate.execute("DELETE FROM book_reviews");
    jdbcTemplate.execute("DELETE FROM books");

    // Insert test data
    jdbcTemplate.update(
        "INSERT INTO books VALUES (?, ?, ?, ?, ?)",
        "123", "Test Book", "Test Author", LocalDate.now(), "AVAILABLE"
    );
}
```

Spring Test Context Caching

- Creating application contexts is expensive
- Spring caches contexts between tests
- Same configuration = reused context
- Different configuration = new context

Context Caching Issues

Common problems that break caching:

1. Different context configurations
2. @DirtiesContext usage
3. Modifying beans in tests
4. Different property settings
5. Different active profiles

Optimizing Context Caching

```
// Bad - different properties = different contexts
@SpringBootTest(properties = "spring.profiles.active=test1")
class Test1 { }

@SpringBootTest(properties = "spring.profiles.active=test2")
class Test2 { }

// Good - shared configuration = single context
@SpringBootTest
@ActiveProfiles("test")
class Test1 { }

@SpringBootTest
@ActiveProfiles("test")
class Test2 { }
```

Avoiding @DirtiesContext

```
// Bad - marks context as dirty after each test
@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
void testThatModifiesState() {
    // ...
}

// Good - manually cleans up state
@BeforeEach
void setUp() {
    // Reset database or other state
    jdbcTemplate.execute("DELETE FROM books");
}
```

Exercise: Fix Context Caching Issues

Identify and fix three integration tests that break context caching:

1. Tests with different property configurations
2. Tests using @DirtiesContext unnecessarily
3. Tests that modify shared beans

Testing HTTP APIs

- TestRestTemplate - Synchronous REST client
- WebTestClient - Reactive REST client
- RestAssured - BDD-style REST testing

TestRestTemplate Example

```
@Test
void getBookByIsbnReturnsCorrectData() {
    // Prepare database with test data
    insertTestBook("123", "Test Title", "Test Author");

    // Make HTTP request
    ResponseEntity<Book> response = restTemplate.getForEntity(
        "/api/books/123", Book.class
    );

    // Verify response
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals("Test Title", response.getBody().getTitle());
}
```

Working with Collections

```
@Test
void getAllBooksReturnsAllBooks() {
    // Setup test data
    insertTestBooks();

    // Request collection
    ResponseEntity<List<Book>> response = restTemplate.exchange(
        "/api/books",
        HttpMethod.GET,
        null,
        new ParameterizedTypeReference<List<Book>>() {}
    );

    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals(2, response.getBody().size());
}
```

Exercise: Full HTTP API Test

Write a comprehensive test for the books API:

1. Set up test data with Testcontainers and PostgreSQL
2. Test CRUD operations via HTTP endpoints
3. Verify database state after operations
4. Handle collections and complex responses

Lab 3: Summary

- Integration testing with full application context
- Using Testcontainers for real database testing
- Managing test data effectively
- Optimizing Spring Test Context caching
- Testing HTTP APIs from the outside

SECTION 4

Best Practices & Pitfalls

Common Spring Boot Testing Pitfalls

1. Using @SpringBootTest everywhere
2. Mixing JUnit 4 and JUnit 5
3. Not leveraging context caching
4. Missing test isolation
5. Brittle tests with implementation details

Pitfall #1: @SpringBootTest Everywhere

L Problem:

- Using @SpringBootTest for every test
- Heavy test startup time
- Slow test execution

Solution:

- Use the right test slice for the job
- Prefer unit tests for isolated components
- Reserve full context tests for integration needs

Choosing the Right Test Type

Test Type	When to Use	Speed	Isolation
Unit Test	Testing single components	Very Fast	High
Sliced Test	Testing specific layers	Fast	Medium
Integration Test	Testing multiple components together	Slow	Low
End-to-End Test	Testing complete functionality	Very Slow	Very Low

Pitfall #2: JUnit 4 vs 5 Issues

L Problem:

- Mixing JUnit 4 and JUnit 5 annotations
- Using wrong extension mechanisms
- Incompatible lifecycle methods

Solution:

- Use JUnit 5 consistently
- Use appropriate extension mechanisms
- Understand lifecycle differences

JUnit 4 vs 5 Comparison

JUnit 4	JUnit 5
@Before, @After	@BeforeEach, @AfterEach
@BeforeClass, @AfterClass	@BeforeAll, @AfterAll
@RunWith	@ExtendWith
@Rule, @ClassRule	@ExtendWith + custom Extension
@Ignore	@Disabled
@Category	@Tag

Pitfall #3: Not Using Context Caching

L Problem:

- Creating new contexts for every test class
- Slow test execution
- High resource usage

Solution:

- Design tests to share contexts
- Consistent configuration across test classes
- Avoid unnecessary @DirtiesContext

Visualizing Context Caching

- Same configuration = Same context
- Different configuration = Different context
- @DirtiesContext = New context

Pitfall #4: Missing Test Isolation

L Problem:

- Tests affecting each other
- Order-dependent tests
- Shared mutable state

Solution:

- Reset state between tests
- Use test-specific data
- Avoid static mutable state
- Clean up resources properly

Test Isolation Techniques

- Database: Truncate tables or use transactions
- Files: Use temporary directories
- In-memory state: Reset before each test
- Static state: Avoid or reset explicitly
- Mocks: Create fresh instances for each test

Pitfall #5: Brittle Tests

L Problem:

- Tests coupled to implementation details
- Breaking with minor refactoring
- Overuse of mocks and verification

Solution:

- Test behavior, not implementation
- Use meaningful assertions
- Mock at boundaries, not internals
- Focus on outputs for given inputs

Behavior vs Implementation Testing

Implementation Testing:

```
void testImplementationDetails() {
    service.processBook(book);

    verify(repository).findById(book.getId());
    verify(validator).validateBookData(book);
    verify(repository).save(book);
}
```

Behavior Testing:

```
void testProcessBookBehavior() {
    when(repository.findById(book.getId())).thenReturn(Optional.empty());

    ProcessingResult result = service.processBook(book);

    assertEquals(ProcessingStatus.CREATED, result.getStatus());
    assertTrue(repository.existsById(book.getId()));
}
```

Best Practices: Test Organization

1. Use descriptive test names
2. Follow a consistent pattern (Given-When-Then, Arrange-Act-Assert)
3. Group related tests in nested classes
4. Separate test utilities and fixtures
5. Keep test code clean and maintained

Best Practice: Descriptive Test Names

```
// Not descriptive
@Test
void testGetBook() { /* ... */ }

// Descriptive - explains behavior and expectations
@Test
void getBookByIsbn_whenBookExists_returnsBookDetails() { /* ... */ }

// Alternative with @DisplayName
@Test
@DisplayName("Get book by ISBN returns book details when book exists")
void getBookByIsbn() { /* ... */ }
```

Best Practice: Test Structure

```
@Test
void createBook_withValidData_savesBookAndReturnsCreated() {
    // Given
    Book newBook = new Book("123", "Title", "Author");
    when(repository.existsById("123")).thenReturn(false);

    // When
    ResponseEntity<Book> response = controller.createBook(newBook);

    // Then
    assertEquals(HttpStatus.CREATED, response.getStatusCode());
    verify(repository).save(newBook);
}
```

Best Practice: Nested Tests

```
@Nested
class GetBookByIsbnTests {
    @Test
    void returnsBookWhenFound() { /* ... */ }

    @Test
    void returns404WhenNotFound() { /* ... */ }
}

@Nested
class CreateBookTests {
    @Test
    void savesValidBook() { /* ... */ }

    @Test
    void rejects400ForDuplicateIsbn() { /* ... */ }
}
```

AI-Assisted Testing

- Code generation for tests
- Test case identification
- Edge case suggestion
- Refactoring existing tests
- Documentation generation

Workshop Recap

- Unit testing with JUnit 5 and Mockito
- Sliced testing with @WebMvcTest and @DataJpaTest
- Integration testing with @SpringBootTest and Testcontainers
- Test context caching optimization
- Best practices and pitfalls

Next Steps

1. Review workshop materials and lab exercises
2. Apply techniques to your own projects
3. Explore advanced topics:
    - Contract testing
    - Performance testing
    - Chaos engineering
    - Test-driven development

Resources

- Official Spring Testing Documentation
- Testing Spring Boot Applications Masterclass
- Spring Boot Testing Starter Guide
- Testcontainers Documentation

Testing Spring Boot Applications Masterclass

- Comprehensive online course
- 40+ video lessons
- Hands-on exercises
- Advanced testing techniques
- [Link to course]

Thank You!

Questions?

[Your contact information]