Building REST APIs with Spring Boot

3.3 Testing in Spring Boot

Learning objectives

- Explain the difference between unit tests and integration tests in Spring Boot
- Understand how service-layer design improves testability
- Write unit tests for service classes using JUnit 5
- Use @SpringBootTest to load the Spring ApplicationContext
- Autowire controllers/services in integration-style tests
- Prepare for later API testing with Rest Assured

What Is Dependency Injection?

- A way to supply an object with its required collaborators
- Objects do not create their own dependencies
- Dependencies are provided (“injected”) from outside
- Increases modularity and testability
- Reduces coupling between classes
- Central part of the Spring Framework

Why Dependency Injection Matters

- Encourages clean architecture
- Makes code easier to maintain
- Makes components easier to replace
- Enables mocking for testing
- Allows Spring to manage application structure
- Supports scalable application design

The Spring ApplicationContext

- The “container” that manages all Spring beans
- Responsible for creating and configuring objects
- Performs dependency injection automatically
- Scans the application for components
- Lives for the entire lifetime of the application
- Provides beans to controllers, services, and repositories

Spring Beans Explained

- A “bean” is any object managed by Spring
- Created inside the ApplicationContext
- Usually annotated with:
  - @Component or @Service or @Repository or @Controller or @RestController
- Automatically discovered with component scanning
- Available for injection in other beans

Component Scanning Basics

- Spring Boot automatically scans:
  - The main application package and all subpackages
  - Looks for classes annotated with for example:
    - @Component/@Service/@Repository/@Controller/@RestController
- Registers them as beans in the container

Constructor Injection in Spring

- Spring’s recommended injection method
- Dependencies passed via constructor
- Spring automatically resolves parameters
- No need for @Autowired

What Happens at Application Startup

- Spring Boot:
  - 1. Starts the application
  - 2. Creates an ApplicationContext
  - 3. Scans your package structure
  - 4. Detects components and creates beans
  - 5. Builds the dependency graph
  - 6. Injects beans into other beans
  - 7. Starts the web server and waits for requests

Demo, part 1 - Moving Logic Into a Service

- Creating a PersonService
- Annotating it with @Service
- Injecting it into a controller
- Understanding how Spring resolves dependencies
- Seeing how the ApplicationContext manages beans
- Using the service from a REST endpoint

Demo, part 1- Learning objectives

- By the end of the demo you will understand:
  - How services encapsulate business logic
  - How controllers depend on services
  - How Spring performs constructor injection
  - How Spring manages lifecycle of beans
  - How the ApplicationContext wires components
  - How services prepare us for repository usage later

Introduction - Testing in Spring Boot

- Spring Boot includes built-in testing support
- Integrates seamlessly with JUnit 5
- Allows testing components with real Spring context
- Supports unit + integration testing

Why Test Spring Boot Components?

- Ensure business logic behaves as expected
- Catch regressions early
- Improve maintainability
- Increase confidence during refactoring
- Enable continuous delivery
- Testing is easier when architecture uses layers

Why Layers Matter for Testing

- Controllers, Services, and Repositories all play different roles
- Business logic belongs in the Service layer
- Because of this:
  - Services can be unit tested without Spring
  - Controllers become thinner and easier to test
  - Tests become smaller, faster, cleaner

Service Layer = Easy to Test

- If business logic is in services:
  - No web components needed
  - No HTTP requests required
  - Test data can be simple Java objects
- This is why we introduced PersonService before Spring Boot testing.

Controllers Are Harder to Unit Test

- Controllers depend on:
  - HTTP request mapping
  - JSON serialization
  - Spring MVC
  - ApplicationContext
- These require integration tests, not unit tests.
- This highlights the benefit of separating logic into services.

JUnit 5 in Spring Boot

- Spring Boot uses JUnit 5 by default
- No additional setup required
- Located under /src/test/java
- Supports:
  - @Test
  - @BeforeEach, @AfterEach
  - Assertions with the Assertions class

Testing With @SpringBootTest

- Loads full Spring ApplicationContext
- Useful for integration tests
- Validates:
  - Bean creation
  - Auto-configuration
  - Controller + service interactions
- Slower than unit tests

Demo, part 2 - Testing the Person Service

- Write unit tests for PersonService
- Cover:
  - Retrieving all persons
  - Getting one person by ID
  - Creating a new person
  - Increasing age (PATCH behavior)
- Understand that:
  - No Spring Context is required
  - Testing services is extremely simple
- Compare with what would be required to test the controller

Demo Architecture

- We test this class:
  - PersonService
- We do NOT test:
  - Controllers
  - HTTP endpoints
  - JSON serialization
- This demonstrates the benefit of layered design.

Expected Learning Outcomes

- By the end of the demo, students will:
  - Understand how to write unit tests in Spring Boot
  - See how service-centric design improves testability
  - Know when to use plain JUnit and when to involve Spring
  - Understand the difference between unit and integration tests

Lab – Testing the Task Service

- In this lab you will:
  - Write unit tests for the TaskService
  - Practice JUnit 5 basics
  - Test CRUD-style behavior
  - See how service-layer logic is easy to test
  - Get familiar with organizing test code

What You Will Test

- You will write tests for:
  - getAll()
  - getById(int id)
  - create(Task t)
  - toggleComplete(int id)
- Everything will be tested using plain JUnit, no Spring context.
- Extra challenge: Use @SpringBootTest annotation to load application context and inject the TaskService into the test class

Lab Setup

- Open the Task project you created earlier
- Navigate to:
  src/test/java/...
- Create a new test class:
  TaskServiceTest
- Use JUnit 5:
  @Test
  Assertions from org.junit.jupiter.api.Assertions
- Instantiate TaskService manually
  → No Spring annotations needed

Lab - Expected Outcomes

- After completing this lab, you should be able to:
  - Write clean unit tests
  - Test service-layer logic easily
  - Understand why controllers should remain thin
  - Recognize the difference between unit and integration tests

Key Takeaways

- Service-layer logic is easy to unit test
- Controllers depend on Spring, so integration-style tests are often needed
- @SpringBootTest loads the Spring context for realistic tests
- Constructor injection enables Spring to wire beans automatically
- Layered architecture makes testing cleaner, faster, and safer
- Testing early helps catch errors before API endpoints or databases are involved