FULL STACK DEV

Writing tests using JUnit5

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

JUNIT

- JUnit is the standard for writing tests in Java. We can use it to write for any layers.
- Let us see how to write test for the controllers.
- We continue with the ProductController and the test will be written in the package test and the name of the class is the controller name with “test” appended
- For testing ProductController, we will use ProductControllerTest.

PREREQUISITES

- Before you start, make sure you have the following installed:
  - Java 17+ (java -version)
  - Maven or Gradle (mvn -v or gradle -v)
  - VS Code IDE
  - Spring Boot Project
- Install These VS Code Extensions
  - In VS Code, go to Extensions (Ctrl+Shift+X or Cmd+Shift+X) and install:
    - Extension Pack for Java (by Microsoft)
    - Spring Boot Extension Pack (optional but recommended)
    - Test Runner for Java (by Microsoft)
    - Debugger for Java
    - Language Support for Java(TM) by Red Hat

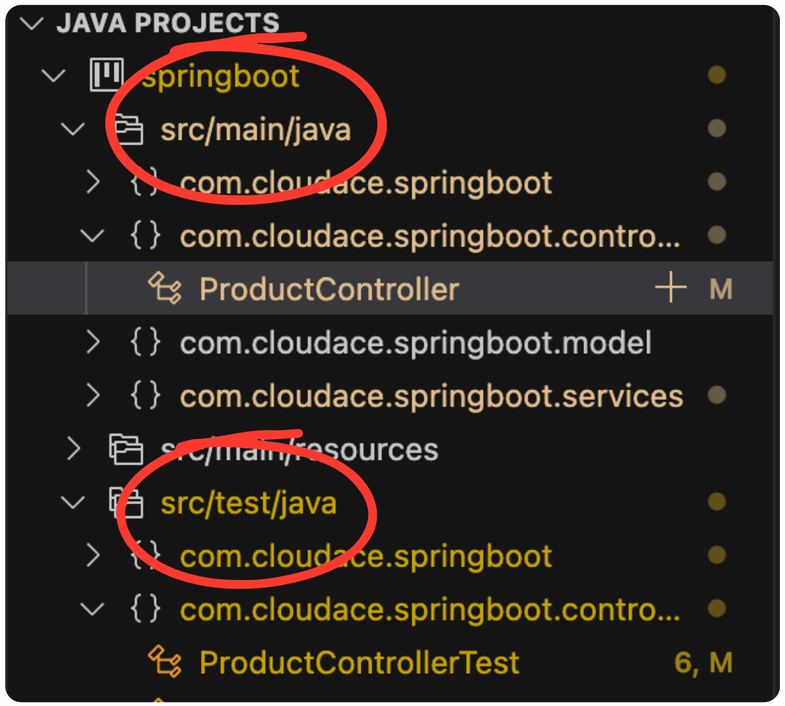
PRODUCTCONTROLLER

```java
@GetMapping("/products/{id}")
public ResponseEntity<Product> getProduct(@PathVariable int id) {
    Product product;
    product = productService.getProductById(id);

    if (product.getName().length() == 0) {
        return ResponseEntity.badRequest().build(); // we are assumin
    }
    return new ResponseEntity<>(product, HttpStatus.OK);
}
```

```java
@GetMapping("/products")
public List<Product> getAllProducts() {
    List<Product> products = productService.getAllProducts();
    return products; // Returning an empty list for now
}
```

SRC/MAIN & SRC/TEST + DEPENDENCY



pom.xml 1, M

pom.xml

```
38
39          <dependency>
40              <groupId>org.springframework.boot</groupId>
41              <artifactId>spring-boot-starter-test</artifactId>
42              <scope>test</scope>
43          </dependency>
44
```

USE RESTTEMPLATE

- It is a Spring Boot-specific test utility used for integration testing of REST endpoints.

- Requires your test class to have:

- @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
- The RANDOM_PORT ensures the server runs on a free port so tests won’t conflict with other apps.

    - You use it only in tests, not in production code.
    - It’s a great way to test the controller layer without mocks.

PRODUCTCONTROLLERTEST

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

    Run Test | Debug Test | Qodo Gen: Test this method | Qodo Gen: Test this method | Qodo Gen: Test this method
    @Test
    public void TestGetProduct() {
        ResponseEntity<Product> response = restTemplate.getForEntity(url: "/products/1", responseType: Prod
        assertEquals(HttpStatus.OK, response.getStatusCode()); // so if evertyhing is ok, we get 200
        assertNotNull(response.getBody());
        assertEquals(expected: "Product 1", response.getBody().getName());
    }

    Run Test | Debug Test | Qodo Gen: Test this method | Qodo Gen: Test this method | Qodo Gen: Test this method
    @Test
    public void TestGetAllProducts() {
        ResponseEntity<Product[]> response = restTemplate.getForEntity(url: "/products", responseType: Prod
        assertEquals(HttpStatus.OK, response.getStatusCode()); // so if evertyhing is ok, we get 200
        assertNotNull(response.getBody());
        assertEquals(expected: 2, response.getBody().length); // we have 2 sample products in the list
    }

```

EXPLANATION

- **@SpringBootTest**(webEnvironment = WebEnvironment.RANDOM_PORT)
  - Start a mini test server
- **@Test**
  - From JUnit 5 (org.junit.jupiter.api.Test), marks a test method.
- **restTemplate**
  - A Spring class for performing HTTP requests during tests.
- **getForEntity()**
  - Makes an HTTP GET request to your endpoint.
- **ResponseEntity<Product>**
  - Captures HTTP response metadata and body.
- **assertEquals, assertNotNull**
  - Standard JUnit assertion methods to validate results.

FULL STACK DEV

Spring Boot®

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

MOCKITO - INTRO

- In Spring Boot, your controller usually depends on a service layer (ProductService) to do the actual business logic.

- With **Mockito**, you can:
  - Replace the real ProductService with a mock.
  - Define predefined behaviors (*what it returns when called*).
  - Focus tests on controller logic only, without hitting a database or real services.

- Remember : controller —> ~~Service —> Hibernate / ORM —> DB~~

MOCKMVC

- MockMvc is a core component of the Spring MVC Test framework.

- It allows you to perform simulated HTTP requests to a Spring controller and verify the response without starting a full web server. It allows you to do :
  - The URL mapping
  - The controller method execution
  - The HTTP status code
  - The content type
  - The JSON or HTML content in the response

- In essence, it acts like a client making an HTTP call to your application and lets you inspect the results.

SETUP: ADD MOCKITO TO YOUR PROJECT

```
45          <dependency>
46              <groupId>org.mockito</groupId>
47              <artifactId>mockito-core</artifactId>
48              <version>5.4.0</version>
49              <scope>test</scope>
50          </dependency>
51
52          <dependency>
53              <groupId>org.mockito</groupId>
54              <artifactId>mockito-junit-jupiter</artifactId>
55              <version>5.4.0</version>
56              <scope>test</scope>
57          </dependency>
```

HOW TO USE IT ?

- We use @WebMvcTest to test only the controller layer of our application

- MockMvc is the tool we use to simulate HTTP requests.

- The when().thenReturn() syntax is where we program the mock object.
- We tell Mockito exactly what to return when a specific method is called on the mock service. This is how we control the behavior of the dependency during the test.

MOCKITBEAN

- The new annotation to use is @MockitoBean.

- By using @MockitoBean, your tests will be more consistent and future-proof (***previously MockBean***).

- What @MockitoBean does
  - Spring creates a Mockito mock of ProductService.
  - The mock is injected into your controller instead of the real bean.
  - When the controller calls productService.getProductById(1), ***it calls the mock, not the real implementation***.

HOW TO WRITE YOUR TESTS ?

- Your test class is preceded by the line :
  - **@WebMvcTest(ProductController.class)**
  - public class ProductControllerWithMTest {
  -
  - }

- Each test throws Exception
  - @Test
  - void testGetProduct() **throws Exception** {
  - }

HOW TO WRITE YOUR TESTS ?

- @Autowired
- private MockMvc mockMvc;
-
- @MockitoBean
- private ProductService productService;

- All your tests contain two sections :
  - **when**(productService.yourMethod()).**thenReturn**(output);

  - **mockMvc.perform**(*endpoints*)
  - .andExpect()
  - .andExpect()
- .....

CUSTOMISE OUTPUT

- Write the beforeEach method which is called on test(s) your run

- @BeforeEach
- void beforeEach(TestInfo testInfo) {
- System.err.println("**********");
- System.out.println(">>> Starting test: " + testInfo.getDisplayName());
- System.err.println("**********");
- }

```
**********
>>> Starting test: testGetProduct()
**********
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.988 s
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

GET A PRODUCT BY ID

```java
@Test
public void testGetProductById() throws Exception {
    Product product = new Product(id: 1L, name: "Laptop", description: "A high-end laptop", price: 1500.

    // Mock the service layer
    when(productService.getProductById(id: 1)).thenReturn(product);

    mockMvc.perform(get(uriTemplate: "/products/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression: "$.name").value(expectedValue: "Laptop"))
            .andExpect(jsonPath(expression: "$.description").value(expectedValue: "A high-end laptop"))
            .andExpect(jsonPath(expression: "$.price").value(expectedValue: 1500.00));

}
```

GET ALL PRODUCTS

```java
@Test
public void testGetAllProducts() throws Exception {

    Product product1 = new Product(id: 1L, name: "Laptop", description: "This is a sample product descr
    Product product2 = new Product(id: 2L, name: "Mouse", description: "This is another product descrip

    // Mock the service layer
    when(productService.getAllProducts()).thenReturn(java.util.Arrays.asList(product1, product2));

    mockMvc.perform(get(uriTemplate: "/products"))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression: "$[0].name").value(expectedValue: "Laptop"))
            .andExpect(jsonPath(expression: "$[0].description").value(expectedValue: "This is a sample
            .andExpect(jsonPath(expression: "$[0].price").value(expectedValue: 19.99));

}
```

TESTING A NON-EXISTING PRODUCT

```
@Test
void testGetProductNotFound() throws Exception {
    // This test will verify that the getProduct method returns a 404 statu
    // You can use mockMvc to perform a GET request with a non-existing pro
    int nonExistingProductId = 999; // Example non-existing product ID
    mockMvc.perform(get("/products/" + nonExistingProductId))
            .andExpect(status().isNotFound()); ⊗ 200 != 404
}
```

FULL STACK DEV

Spring Boot®

cucumber

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

CUCUMBER - INTRO

- Cucumber tests shifts our focus from "how a method works" to "how a user interacts with the system."

- This approach is known as **Behavior-Driven Development** (BDD)

- An approach where you define the behavior of the application from the user's perspective before you write any code.

- This directly supports the core Agile principle of collaboration and shared understanding.

HOW TO USE CUCUMBER ?

- **Three steps**

- **Feature Files (.feature)**: You write plain-text files that describe the application's behavior using a language called Gherkin.

- **Step Definitions**: The developer writes the Java code that "glues" each line of the Gherkin feature file to the actual logic (MockMvc)

- **Test Runner**: A JUnit test class acts as the entry point, telling Cucumber to find and run all the feature files.

OVERALL

FEATURE IN GHERKINS -> JAVA CODE -> MOCKITO (UNIT TESTING) -> JUNIT (INTEGRATION TESTING)

EXAMPLE #1 - REQUIREMENTS

- **Feature**: Add new products to the inventory
  - *As an authenticated user*
  - *I want to add products to the inventory*
  - *So that I can keep track of all available products*

- Written by Product Owner / Business Analyst

- Next step → Developers

EXAMPLE 1 - FUNCTIONALITY

- **Scenario 1**: Successfully add a new product with all required details

- Given I am authenticated as an administrator

- When I submit a request to add a product with the following details:
- | name | description | price
- | "Laptop" | "High-performance laptop" | 1200

- Then the product "Laptop" should be created successfully
- And the system should respond with a 201 status code

HOW TO WRITE YOUR TESTS ?

- **Scenario 2**: Fail to add a product due to missing required fields

- Given I am authenticated as an administrator

- When I submit a request to add a product with the following details:
- | name | description | price |
- | "Monitor" | "4K display" | 350 |

- Then the product "Monitor" should not be created
- And the system should respond with a 400 status code

AUTOMATED ACCEPTANCE TESTS

- Cucumber scenarios are, at their core, automated acceptance tests. They verify that the functionality meets the business requirements.
    - **Developers** can use these tests to ensure their code works as expected.
    - **QA engineers** can use the same tests to validate the feature.
- In short, Cucumber acts as a bridge between the business requirements and the technical implementation

THE ORDER OF A BDD TEST RUN - 1

- **1. Cucumber reads the .feature file.**

- The process begins with the plain-language .feature file you wrote - it is the single source of truth for the test's behavior.

- **2. JUnit launches the Cucumber test runner.**

- You typically have a test runner class annotated with @RunWith(Cucumber.class) or a similar JUnit 5 annotation.

- JUnit is the framework that starts the process. It hands control over to the Cucumber engine.

THE ORDER OF A BDD TEST RUN - 2

- **3. Cucumber finds and executes the matching Step Definitions.**
- This is done for each Given, When, and Then step in the .feature file

- **4. The Java method uses MockMvc or Mockito to test your application code.**
- This is where the actual work happens. Inside the Java method that Cucumber called, you write the code that performs the verification.
  - *MockMvc is used for integration testing*
  - *Mockito is used for unit testing*

- **5. Cucumber reports the result.**
- After all the Java code for a scenario has run, Cucumber collects the results of the assertions

THE BDD WORKFLOW IN A TEAM

- 1. Defining the requirements - understanding what should de done
  - Meeting with Product owner - QA - Developer
- 2: The QA/Tester Writes the Feature File
- 3: The Developer Creates the Step Definitions
- 4: The Developer Writes the Application Code
- 5: The Tests Become Part of the CI/CD Pipeline

*This is carried out automatically by **Github actions** or **Jenkins** while deploying to DockerHub !!* ***If any test fails, then Deployment fails***

SCOPE OF JUNIT VS CUCUMBER

Type	Purpose	Example
JUnit Test (ProductControllerTest.java)	Test internal logic, small isolated behaviors	"Does /products/1 return HTTP 200?"
Cucumber Test (Product.feature + ProductStepDefinitions.java)	Test business workflows in human-readable form	"When I add a product, it should appear in the product list"

LET'S TEST CUCUMBER

- **1. The Maven Dependency (pom.xml)**
- <dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>7.16.1</version>
  <scope>test</scope>
- </dependency>

- <dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit-platform-engine</artifactId>
  <version>7.16.1</version>
  <scope>test</scope>
- </dependency>

LET'S TEST CUCUMBER

- **2. Create the following folders in test**
  - resources
    - /features : Contains your feature file (ex hello.feature)
    - / : Contains cucumber.properties
  - your_package/runner
    - Contains your cucumber test file (ex CucumberTest.java)
  - your_package/steps
    - Contains your steps files (ex HelloSteps.java)