

# Mastering Spring Boot Testing with JUnit and Mockito — Quick Start Guide



Alexander Obregon · [Follow](#)

3 min read · Mar 25, 2023



Listen



Share



More



[Image Source](#)

## Introduction

Writing tests for your Spring Boot applications is important to ensure their reliability, maintainability, and quality. JUnit and Mockito are popular libraries for writing unit and integration tests in Java applications. In this article, we'll explore how to effectively test Spring Boot applications using JUnit and Mockito.

## Setting Up Testing Dependencies

To get started, add the following testing dependencies to your project:

```
<!-- Maven -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <scope>test</scope>
</dependency>

// Gradle
testImplementation 'org.springframework.boot:spring-boot-starter-test'
testImplementation 'org.mockito:mockito-core'
```

The `spring-boot-starter-test` dependency includes JUnit, Mockito, and other useful testing libraries.

## Writing Unit Tests with JUnit

JUnit is a widely-used testing framework for Java applications. It provides annotations like `@Test`, `@BeforeEach`, and `@AfterEach` for structuring your tests. Here's an example of a simple JUnit test for a calculator service:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorServiceTest {

    private final CalculatorService calculatorService = new CalculatorService();

    @Test
    public void testAddition() {
        int result = calculatorService.add(3, 5);
        assertEquals(8, result, "3 + 5 should equal 8");
    }
}
```

## Mocking Dependencies with Mockito

Mockito is a powerful mocking framework for Java applications. It allows you to create mock objects, define their behavior, and verify interactions. To mock dependencies in your unit

tests, use Mockito's `@Mock` annotation and `@InjectMocks` annotation to inject the mocks into the class under test:

```
import com.example.service.UserService;
import com.example.repository.UserRepository;
```

[Open in app](#) ↗

```
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    // Write your tests here
}
```

## Testing Spring Boot Components

Spring Boot provides built-in support for testing various components such as controllers, services, and repositories. Use the `@SpringBootTest` annotation to load the entire application context, or use `@WebMvcTest`, `@DataJpaTest`, and `@RestClientTest` for testing specific components:

```
import com.example.controller.UserController;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@ExtendWith(SpringExtension.class)
@WebMvcTest(UserController.class)
public class UserControllerTest {
```

```

@Autowired
private MockMvc mockMvc;

@MockBean
private UserService userService;

@Test
public void testGetUsers() throws Exception {
    mockMvc.perform(get("/users"))
        .andExpect(status().isOk());
}
}

```

## Testing with Test Slices

Spring Boot offers test slices to load only specific parts of the application context for faster test execution. Use test slices like `@WebMvcTest`, `@DataJpaTest`, and `@RestClientTest` to focus testing a single layer of your application:

```

import com.example.repository.UserRepository;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@DataJpaTest
public class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    // Write your tests here
}

```

## Integration Testing with @SpringBootTest

Use the `@SpringBootTest` annotation to create integration tests that load the entire application context. This approach is useful for testing the interaction between multiple components, but it can be slower than using test slices:

```

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@SpringBootTest
public class ApplicationIntegrationTest {

    @Autowired
    private UserService userService;

    // Write your tests here
}
```

## Test Configuration and Profiles

Use `@TestConfiguration` to define beans specifically for testing, and use `@ActiveProfiles` to activate specific Spring profiles during testing:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("test")
public class TestConfiguration {

    @Bean
    public CustomTestService customTestService() {
        return new CustomTestService();
    }
}
```

**Then, activate the test profile in your test class:**

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@SpringBootTest
@ActiveProfiles("test")
public class CustomTestServiceTest {
```

```
@Autowired
private CustomTestService customTestService;

// Write your tests here
}
```

## Conclusion

Testing your Spring Boot applications using JUnit and Mockito is essential for ensuring the reliability and quality. By leveraging Spring Boot's testing support, test slices, and built-in tools, you can create comprehensive unit and integration tests for your applications, resulting in a stronger and maintainable codebase.

1. [Spring Boot Testing Overview](#)
2. [Mockito Documentation](#)
3. [Spring Boot Test Slices](#)



[Spring Boot](#) icon by [Icons8](#)

Spring Boot

JUnit

Mockito

Java

Testing



Follow



## Written by Alexander Obregon

24K Followers

Software Engineer, fervent coder & writer. Devoted to learning & assisting others. Connect on LinkedIn:  
<https://www.linkedin.com/in/alexander-obregon-97849b229/>

### More from Alexander Obregon



Alexander Obregon

## Enhancing Logging with @Log and @Slf4j in Spring Boot Applications

Introduction

8 min read · Sep 22, 2023



278



4





 Alexander Obregon

## Java Memory Leaks: Detection and Prevention

Introduction

8 min read · Nov 13, 2023



641



3



 Alexander Obregon


## Optimizing Queries with @Query Annotation in Spring Data JPA

Introduction



16 min read · Aug 29, 2023

 492  6



 Alexander Obregon

## Using Spring's @Retryable Annotation for Automatic Retries

Introduction

11 min read · Sep 17, 2023

 305  5

See all from Alexander Obregon

Recommended from Medium

# SPRING BOOT TESTING TUTO

 Wensen Ma

## Comprehensive Guide to Spring Boot Testing

Testing is an integral part of the software development process, especially in complex frameworks like Spring Boot. This guide aims to...

3 min read · Jan 21, 2024

 4 



 Ashok Sasidharan Nair

## Unit Testing a Spring Boot service with Mockito and JUnit 5

Using Mockito with JUnit 5 to design parameterized and non-parameterized unit tests for a Spring Boot service.

8 min read · Jan 24, 2024

 20  1

## Lists



### General Coding Knowledge

20 stories · 1272 saves



### Business

41 stories · 115 saves



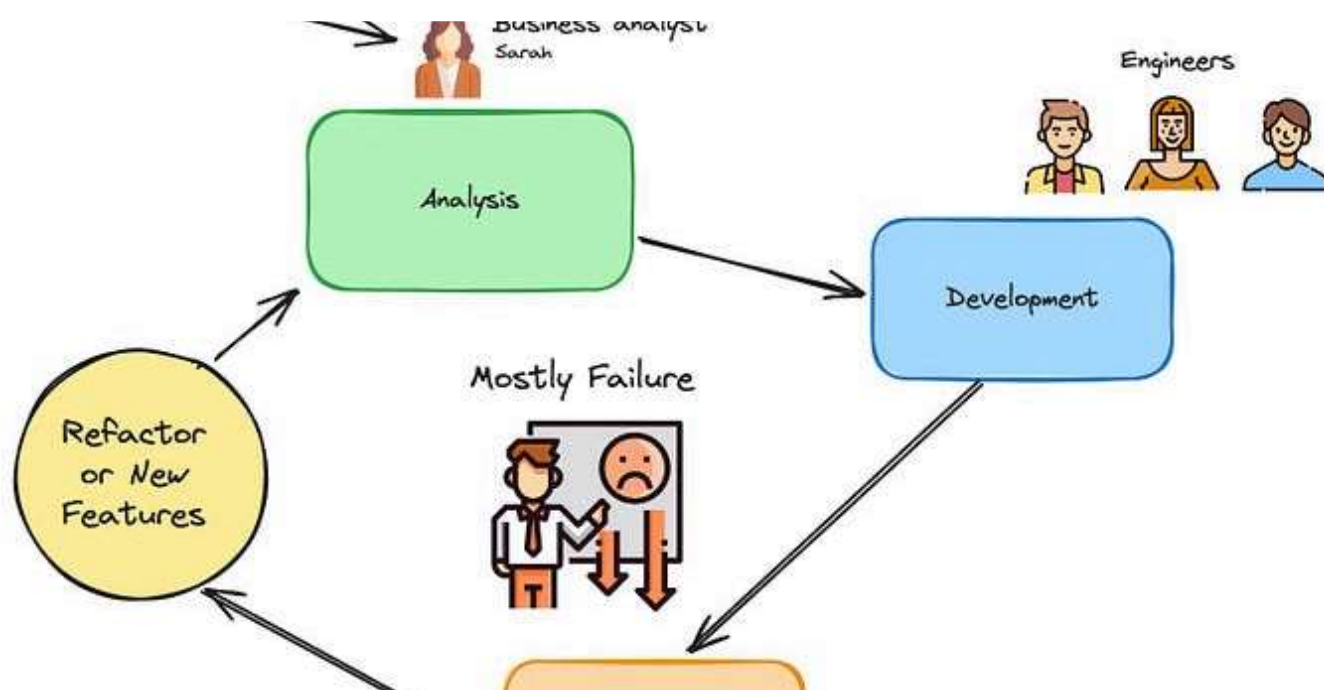
### data science and AI


40 stories · 175 saves



### Natural Language Processing

1492 stories · 1008 saves



 Berat Yesbek

## Unit Test on Spring Boot, Mock, Integration Test with Test Container, and Argument Capture

Why should unit tests be written?

9 min read · Jan 15, 2024



# Unit Testing

👤 Hüsna POYRAZ

## Spring Data JPA Unit Test: Repository Layer

Hello, I will explain how to write unit tests for the repository layer in a simple Spring Boot project based on the following topic...

7 min read · Mar 19, 2024



## Mastering Unit Testing with JUnit and Mockito: A Comprehensive Guide for Developers

Apidog is an integrated collaboration platform for API documentation, API debugging, API mocking, and API automated testing. It's the best...

5 min read · Mar 21, 2024



## End-To-End Testing Spring Boot REST APIs with Rest Assured

Testing has proved to be a very crucial process that ensures the delivery of high-quality software. A comprehensive test strategy would...

6 min read · Mar 26, 2024



See more recommendations