### **A Guide to JUnit & Mockito Fundamentals in Spring Boot**

This document serves as a comprehensive learning guide for understanding and applying core unit testing principles using JUnit 5 and Mockito within a Spring Boot environment.

### **JUnit & Mockito Fundamentals**

#### **Core Content: JUnit 5 & Mockito Essentials**

##### **JUnit 5 Essentials**

JUnit 5 is the de facto standard for unit testing in Java. Its core features help you write readable and maintainable tests.

* **Annotations**:
  + @Test: Marks a method as a test method. This is the most fundamental annotation.
  + @BeforeEach: A method annotated with this will run before each test method. It's perfect for setting up a clean state for every test, such as initializing objects.
  + @AfterEach: A method with this annotation will run after each test method. It's used for cleanup, like closing resources.
  + @ParameterizedTest: Allows you to run the same test method with different sets of arguments, which is great for testing methods with various inputs.
* **Assertions**: Assertions are methods that verify the expected outcome of a test. For example, Assertions.assertEquals(expected, actual) checks if two values are equal. When an assertion fails, the test stops and is marked as failed.

##### **Mockito Framework**

Mockito is a powerful mocking framework for Java. The main purpose of mocking is to **isolate the code you are testing** by replacing its dependencies with fake objects called "mocks." This prevents your unit tests from failing due to issues in other classes or external services.

* **Creating Mocks**: You can create a mock of a dependency using the @Mock annotation.
* **Stubbing**: Stubbing defines the behavior of a mock object. You use the when().thenReturn() syntax to tell the mock what to return when a specific method is called.
* **Verification**: After calling the method under test, you use verify() to ensure that the method on the mock was called as expected.
* **@InjectMocks**: This annotation automatically injects the mock objects (annotated with @Mock) into the fields of the class you are testing.

#### **Hands-on Project Activity #1: "User Registration Service Testing"**

**Project Setup**: We will test a UserRegistrationService that has three dependencies: a UserRepository, an EmailService, and a PasswordValidator.

##### **Step 1: Set Up the Classes to Be Tested**

Here are the classes that make up our service. You do not need to implement the actual logic for the dependencies, as we will mock them.

Java

// --- User.java (Data Model) ---  
public class User {  
 private String email;  
 private String password;  
 private boolean isRegistered;  
 // Getters and setters  
}  
  
// --- UserRepository.java (Dependency) ---  
// We will mock this interface  
public interface UserRepository {  
 User findByEmail(String email);  
 User save(User user);  
}  
  
// --- EmailService.java (Dependency) ---  
// We will mock this interface  
public interface EmailService {  
 void sendRegistrationEmail(String email);  
}  
  
// --- PasswordValidator.java (Dependency) ---  
// We will mock this interface  
public interface PasswordValidator {  
 boolean isValid(String password);  
}  
  
// --- UserRegistrationService.java (Class Under Test) ---  
public class UserRegistrationService {  
 private final UserRepository userRepository;  
 private final EmailService emailService;  
 private final PasswordValidator passwordValidator;  
  
 public UserRegistrationService(UserRepository userRepository, EmailService emailService, PasswordValidator passwordValidator) {  
 this.userRepository = userRepository;  
 this.emailService = emailService;  
 this.passwordValidator = passwordValidator;  
 }  
  
 public boolean registerUser(String email, String password) {  
 if (!passwordValidator.isValid(password)) {  
 return false;  
 }  
 if (userRepository.findByEmail(email) != null) {  
 return false;  
 }  
 User newUser = new User();  
 newUser.setEmail(email);  
 newUser.setPassword(password);  
 newUser.setRegistered(true);  
 userRepository.save(newUser);  
 emailService.sendRegistrationEmail(email);  
 return true;  
 }  
}

##### **Step 2: Create the Test Class**

We will create a UserRegistrationServiceTest class and use Mockito to inject the mock dependencies.

Java

// --- UserRegistrationServiceTest.java ---  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.extension.ExtendWith;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.junit.jupiter.MockitoExtension;  
import static org.junit.jupiter.api.Assertions.\*;  
import static org.mockito.Mockito.\*;  
  
@ExtendWith(MockitoExtension.class)  
public class UserRegistrationServiceTest {  
  
 @Mock  
 private UserRepository userRepository;  
  
 @Mock  
 private EmailService emailService;  
  
 @Mock  
 private PasswordValidator passwordValidator;  
  
 @InjectMocks  
 private UserRegistrationService userRegistrationService;  
  
 private User sampleUser;  
  
 @BeforeEach  
 void setUp() {  
 sampleUser = new User();  
 [sampleUser.setEmail("test@example.com](mailto:sampleUser.setEmail("test@example.com)");  
 }  
   
 // --- Test Scenarios ---  
   
 // 1. Positive Test: Successful Registration  
 @Test  
 void testSuccessfulRegistration() {  
 // AAA Pattern  
 // Arrange  
 when(passwordValidator.isValid(anyString())).thenReturn(true);  
 when(userRepository.findByEmail(anyString())).thenReturn(null);  
  
 // Act  
 boolean result = [userRegistrationService.registerUser("newuser@example.com](mailto:userRegistrationService.registerUser("newuser@example.com)", "SecurePassword123!");  
  
 // Assert  
 assertTrue(result);  
 verify(userRepository, times(1)).save(any(User.class));  
 verify(emailService, [times(1)).sendRegistrationEmail("newuser@example.com](mailto:times(1)).sendRegistrationEmail("newuser@example.com)");  
 }  
  
 // 2. Negative Test: Invalid Password  
 @Test  
 void testRegistrationFailsWithInvalidPassword() {  
 // Arrange  
 when(passwordValidator.isValid(anyString())).thenReturn(false);  
  
 // Act  
 boolean result = [userRegistrationService.registerUser("test@example.com](mailto:userRegistrationService.registerUser("test@example.com)", "weak");  
  
 // Assert  
 assertFalse(result);  
 verify(userRepository, never()).findByEmail(anyString());  
 verify(emailService, never()).sendRegistrationEmail(anyString());  
 }  
  
 // 3. Negative Test: User Already Exists  
 @Test  
 void testRegistrationFailsIfUserExists() {  
 // Arrange  
 when(passwordValidator.isValid(anyString())).thenReturn(true);  
 when(userRepository.findByEmail(anyString())).thenReturn(sampleUser);  
  
 // Act  
 boolean result = [userRegistrationService.registerUser("test@example.com](mailto:userRegistrationService.registerUser("test@example.com)", "ValidPassword123");  
  
 // Assert  
 assertFalse(result);  
 verify(userRepository, [times(1)).findByEmail("test@example.com](mailto:times(1)).findByEmail("test@example.com)");  
 verify(userRepository, never()).save(any(User.class));  
 verify(emailService, never()).sendRegistrationEmail(anyString());  
 }  
}

#### **Situational Exercise #2: "The Flaky Email Test"**

**Problem**: A test for the UserRegistrationService sometimes fails in the CI/CD pipeline. The team suspects it's related to the EmailService dependency.

**Root Cause**: The flaky test likely makes a real network call to an external email service. In a CI/CD environment, this can fail for various reasons: a network timeout, an API rate limit, or the email service being temporarily unavailable. A true unit test should **never** rely on external services.

**Solution**: The correct approach is to mock the EmailService. By using Mockito, we can verify that the sendRegistrationEmail method was **called** with the correct arguments, without actually making a network call.

##### **Flaky Test (Incorrect)**

Java

// --- FlakyTest.java ---  
// This test is bad because it relies on a real, external EmailService.  
// It will be slow and may fail unpredictably.  
public class FlakyEmailTest {  
 // This test relies on a real instance of EmailService which is a dependency  
 // that depends on the network and other infrastructure  
 // which makes it slow and unpredictable.  
}

##### **Corrected Test (Fixed with Mockito)**

The corrected test is what we already have in our UserRegistrationServiceTest.java above. It correctly mocks the EmailService and uses verify() to ensure the sendRegistrationEmail method was called exactly once, without making a real API call. This test is fast, predictable, and isolated.

### **How to Execute Tests in VS Code**

VS Code, with the **Java Test Runner** extension, makes executing JUnit tests a breeze.

1. **Install the Extension**: Open the Extensions view (Ctrl+Shift+X), search for "Java Test Runner," and click **Install**.
2. **Test Explorer**: A new beaker icon will appear in the VS Code sidebar. Click it to open the **Test Explorer**, which automatically discovers all your test classes.
3. **Run Tests**: In your UserRegistrationServiceTest.java file, VS Code will display a green play button next to the class name and each individual @Test method.
   1. Click the play button next to the class name to run all tests in that file.
   2. Click the play button next to a specific @Test to run only that test.
   3. You can also click the debug icon next to the play button to run the test in a debugger.
4. **View Results**: The Test Explorer panel will display a summary of your test run. Green checkmarks indicate passed tests, while a red "X" signifies a failure. Clicking on a failed test will show you the exact reason for the failure in the output console.

### **Advanced Mocking & Spring Boot Testing**

This section serves as a learning guide for advanced testing techniques in Spring Boot, covering key concepts, code examples, and execution within VS Code.

#### **Core Concepts & Hands-on Activity**

##### **Advanced Mockito Patterns**

* **@Mock vs. @MockBean**: Use @Mock for simple unit tests without a Spring context. Use @MockBean for slice tests (@WebMvcTest, @DataJpaTest) to replace a real bean with a mock.
* **Partial Mocking with @Spy**: Use @Spy to test a real object while stubbing out a few specific methods.
* **Mocking Static Methods**: Use Mockito.mockStatic() (with the mockito-inline dependency) to mock static methods.

##### **Spring Boot Testing Annotations**

* @WebMvcTest: Use for testing the web layer (controllers).
* @DataJpaTest: Use for testing the persistence layer (repositories).
* @SpringBootTest: Use for full integration tests, as it loads the entire application context.

##### **Example: Testing with @WebMvcTest and @DataJpaTest**

#### **Integration Testing & VS Code Execution**

##### **Integration Testing Tools & Techniques**

* **TestContainers**: A library that spins up real databases (like PostgreSQL) in Docker containers for your tests, ensuring a realistic environment.
* **WireMock**: A mock server that simulates external API calls, allowing you to test how your application handles different responses without relying on a real third-party service.

##### **Example: Integration Test with WireMock**

##### **How to Execute Tests in VS Code**

VS Code provides a straightforward way to run tests using the **Java Test Runner** extension.

1. **Install the Extension**: Install "Java Test Runner" from the VS Code Extensions view (Ctrl+Shift+X).
2. **Open the Test Explorer**: Click the beaker icon on the sidebar to open the **Test Explorer**. The extension will automatically discover and list your test classes and methods.
3. **Run Tests**: You can run all tests in a class by clicking the green play button next to the class name, or run a single test by clicking the button next to the method name.
4. **View Results**: The Test Explorer will display the results (pass/fail), and you can click on a failed test to see the error message and stack trace. This provides instant feedback without needing to use the command line.