##### **Testing with JUnit5**

## 1. Introduction to Software Testing

### What is Software Testing?

Software testing is the process of evaluating a software application to ensure that it behaves as expected and is free of bugs or errors. Testing is a critical activity in software development and ensures the product meets the quality standards before being released to the end users.

In simple terms, testing involves running your code under controlled conditions to identify any potential problems in how the software functions.

### Why is Software Testing Important?

Software testing is crucial for a variety of reasons:

1. **Quality Assurance**: Ensures the software functions as intended and meets quality standards.
2. **Bug Detection**: Helps in identifying errors or bugs before the product is released.
3. **User Experience**: Delivers a smooth experience for users by minimizing crashes and malfunctions.
4. **Cost-Efficiency**: Fixing bugs early in development is less expensive than fixing them after deployment.
5. **Performance**: Verifies that the system performs under various conditions and loads.
6. **Security**: Detects vulnerabilities or weaknesses in the system that could be exploited maliciously.

For a **full-stack developer**, software testing is important because it ensures that both the **frontend** (user-facing) and **backend** (server-side) components of a system function seamlessly together. Full-stack developers work with both areas and must ensure that data flows properly between them and that all interactions are thoroughly tested.

### Types of Software Testing

Here are the common types of software testing:

1. **Unit Testing**: Testing individual components or functions in isolation to ensure they work correctly.
2. **Integration Testing**: Testing how various components of the system interact with each other.
3. **System Testing**: Testing the entire system as a whole to verify the software meets its requirements.
4. **Acceptance Testing**: Verifying that the software meets the business requirements and is ready for deployment.
5. **Regression Testing**: Ensuring that new changes or updates have not broken existing functionality.
6. **Performance Testing**: Evaluating how the software performs under stress, load, or high-volume conditions.
7. **Security Testing**: Testing for vulnerabilities and weaknesses in the system to ensure data integrity and protection.
8. **Usability Testing**: Checking the user interface and user experience to ensure it is easy and intuitive.

## 2. Understanding JUnit

### What is JUnit?

JUnit is a widely used testing framework for Java programming that allows developers to write and run tests on their Java code. It helps automate the testing process, making it more efficient and less error-prone. With JUnit, developers can ensure their code is working correctly and can easily track changes over time.

### Why Use JUnit?

JUnit provides several benefits:

1. **Automation**: Automates the process of running tests, which improves productivity.
2. **Consistency**: Helps in maintaining consistency in testing.
3. **Early Detection**: Detects errors early, improving the overall quality of the software.
4. **Test-Driven Development (TDD)**: Encourages the practice of writing tests before the actual code, leading to better design and fewer bugs.
5. **Integration with Build Tools**: JUnit integrates easily with tools like Maven and Gradle, which helps in continuous integration.

### Key Features of JUnit5

JUnit5 is the latest version of the JUnit framework, which consists of the following key features:

* **Modular Architecture**: JUnit5 is broken down into three main modules: JUnit Platform, JUnit Jupiter, and JUnit Vintage.
  + **JUnit Platform**: Provides a foundation for running tests.
  + **JUnit Jupiter**: Contains the new programming model and extension model for writing tests.
  + **JUnit Vintage**: Allows you to run older versions of JUnit (JUnit 3 and JUnit 4) tests on the JUnit 5 platform.
* **Annotations**: Introduced new annotations to simplify writing tests and improve readability.
* **Extensions**: JUnit5 allows custom extensions to extend the framework's functionality.

## 3. JUnit5 Methods

JUnit provides a variety of methods for performing tests, asserting outcomes, and managing the test lifecycle.

### List of Methods in JUnit5

#### Assertions

Assertions are used to validate the expected results in tests. Here are some of the common assertion methods:

* **assertEquals(expected, actual)**: Asserts that two values are equal.
* **assertNotEquals(expected, actual)**: Asserts that two values are not equal.
* **assertTrue(condition)**: Asserts that the condition is true.
* **assertFalse(condition)**: Asserts that the condition is false.
* **assertNull(value)**: Asserts that the value is null.
* **assertNotNull(value)**: Asserts that the value is not null.
* **assertNotSame(expected, actual)**: Asserts that two objects are not the same (i.e., they don’t reference the same object in memory).
* **assertSame(expected, actual)**: Asserts that two objects are the same (i.e., they reference the same object in memory).
* **assertThrows(expectedType, executable)**: Asserts that a specific exception is thrown.
* **assertDoesNotThrow(executable)**: Asserts that no exception is thrown.

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

public class AssertionTest {

@Test

public void testAssertions() {

// assertEquals(expected, actual)

int expected = 5;

int actual = 5;

assertEquals(expected, actual, "The values should be equal");

// assertNotEquals(expected, actual)

int unexpected = 10;

assertNotEquals(unexpected, actual, "The values should not be equal");

// assertTrue(condition)

boolean isTrue = 10 > 5;

assertTrue(isTrue, "The condition should be true");

// assertFalse(condition)

boolean isFalse = 5 > 10;

assertFalse(isFalse, "The condition should be false");

// assertNull(value)

String str = null;

assertNull(str, "The value should be null");

// assertNotNull(value)

str = "Hello";

assertNotNull(str, "The value should not be null");

// assertNotSame(expected, actual)

Object obj1 = new Object();

Object obj2 = new Object();

assertNotSame(obj1, obj2, "The objects should not be the same instance");

// assertSame(expected, actual)

Object obj3 = obj1;

assertSame(obj1, obj3, "The objects should be the same instance");

// assertThrows(expectedType, executable)

assertThrows(ArithmeticException.class, () -> {

int result = 10 / 0; // This will throw ArithmeticException

}, "Division by zero should throw ArithmeticException");

// assertDoesNotThrow(executable)

assertDoesNotThrow(() -> {

int result = 10 / 2; // This should not throw any exception

}, "The division should not throw an exception");

}

}

## 4. JUnit5 Annotations

### Overview of JUnit5 Annotations

Annotations in JUnit provide metadata that define the test lifecycle and configure different aspects of the test behavior. These annotations are essential for marking the start and end of tests, setting up the environment, and asserting test results.

### List of Common Annotations in JUnit5

* **@Test**: Denotes a method as a test method.
* **@BeforeAll**: Runs once before all tests in the test class (static method).
* **@AfterAll**: Runs once after all tests in the test class (static method).
* **@BeforeEach**: Runs before each test method.
* **@AfterEach**: Runs after each test method.
* **@DisplayName**: Provides a custom name for the test.
* **@Disabled**: Disables a test or test class
* **@RepeatedTest**: Marks a test to be run multiple times.
* **@ParameterizedTest**: Runs the same test with multiple sets of data.
* **@ValueSource**: Used in conjunction with @ParameterizedTest to provide test data (e.g., for integers, strings).
* **@CsvSource**: Provides a set of CSV data for parameterized tests.
* **@Nested**: Marks a nested class that can contain tests (used for organizing tests).
* **@ExtendWith**: Registers an extension to be applied to the test.

**Class with Methods to Test (Calculator)**

import java.util.List;

public class Calculator {

// Method for addition

public int add(int a, int b) {

return a + b;

}

// Method for subtraction

public int subtract(int a, int b) {

return a - b;

}

// Method for multiplication

public int multiply(int a, int b) {

return a \* b;

}

// Method for division

public double divide(int a, int b) {

if (b == 0) {

throw new ArithmeticException("Cannot divide by zero");

}

return (double) a / b;

}

// Method to check if a number is even

public boolean isEven(int number) {

return number % 2 == 0;

}

// Method to find the largest number in a list

public int findLargest(List<Integer> numbers) {

if (numbers == null || numbers.isEmpty()) {

throw new IllegalArgumentException("List cannot be null or empty");

}

return numbers.stream().max(Integer::compare).orElseThrow();

}

}

**Test Cases Using JUnit Annotations**

Now, let's define more realistic use cases for each method and organize them using JUnit annotations.

**1. Simple Test Cases Using @Test**

In this example, we will test basic operations like addition, subtraction, multiplication, and division with different inputs.

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class BasicOperationsTest {

Calculator calculator = new Calculator();

@Test

void testAddition() {

// Use case: Adding two positive numbers

assertEquals(7, calculator.add(3, 4), "Addition of 3 and 4 should be 7");

// Use case: Adding a positive and a negative number

assertEquals(1, calculator.add(5, -4), "Addition of 5 and -4 should be 1");

// Use case: Adding two negative numbers

assertEquals(-7, calculator.add(-3, -4), "Addition of -3 and -4 should be -7");

}

@Test

void testSubtraction() {

// Use case: Subtracting two numbers

assertEquals(4, calculator.subtract(9, 5), "9 - 5 should be 4");

// Use case: Subtracting a larger number from a smaller one

assertEquals(-2, calculator.subtract(3, 5), "3 - 5 should be -2");

// Use case: Subtracting negative numbers

assertEquals(7, calculator.subtract(5, -2), "5 - (-2) should be 7");

}

@Test

void testMultiplication() {

// Use case: Multiplying two positive numbers

assertEquals(20, calculator.multiply(4, 5), "4 \* 5 should be 20");

// Use case: Multiplying by zero

assertEquals(0, calculator.multiply(0, 5), "0 \* 5 should be 0");

// Use case: Multiplying two negative numbers

assertEquals(12, calculator.multiply(-3, -4), "-3 \* -4 should be 12");

}

@Test

void testDivision() {

// Use case: Dividing two numbers

assertEquals(5.0, calculator.divide(10, 2), "10 / 2 should be 5.0");

// Use case: Dividing by zero

assertThrows(ArithmeticException.class, () -> calculator.divide(10, 0), "Division by zero should throw ArithmeticException");

// Use case: Dividing a negative number

assertEquals(-2.5, calculator.divide(-5, 2), "-5 / 2 should be -2.5");

}

}

**2. Setup and Teardown Using @BeforeAll, @AfterAll, @BeforeEach, @AfterEach, and @DisplayName**

In this example, we'll simulate a scenario where we want to initialize and clean up before and after each test.

import org.junit.jupiter.api.\*;

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

public class LifecycleCalculatorTest {

Calculator calculator;

@BeforeAll

static void beforeAllTests() {

// Simulate setup that happens once for all tests (e.g., establishing DB connections)

System.out.println("Before all tests");

}

@AfterAll

static void afterAllTests() {

// Simulate cleanup that happens once after all tests (e.g., closing DB connections)

System.out.println("After all tests");

}

@BeforeEach

void setUp() {

// Setup method runs before each test, instantiate the calculator for each test

calculator = new Calculator();

System.out.println("Before each test");

}

@AfterEach

void tearDown() {

// Cleanup method runs after each test

System.out.println("After each test");

}

@Test

@DisplayName("Test Addition")

void testAddition() {

// Use case: Add two numbers

int result = calculator.add(10, 20);

assertEquals(30, result, "10 + 20 should be 30");

}

@Test

@DisplayName("Test Subtraction")

void testSubtraction() {

// Use case: Subtract two numbers

int result = calculator.subtract(10, 4);

assertEquals(6, result, "10 - 4 should be 6");

}

}

**3. Repeated and Parameterized Tests Using @RepeatedTest, @ParameterizedTest, @ValueSource, and @CsvSource**

In this example, we demonstrate how to repeat tests and use multiple sets of input data.

import org.junit.jupiter.api.\*;

import org.junit.jupiter.params.\*;

import org.junit.jupiter.params.provider.\*;

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

public class ParameterizedCalculatorTest {

Calculator calculator = new Calculator();

@RepeatedTest(5)

void repeatedAdditionTest() {

// Use case: Repeatedly test addition of same values

int result = calculator.add(5, 5);

assertEquals(10, result, "5 + 5 should be 10");

}

@ParameterizedTest

@ValueSource(ints = {2, 4, 6, 8})

void testIsEven(int number) {

// Use case: Check if the number is even

boolean result = calculator.isEven(number);

assertEquals(true, result, number + " should be even");

}

@ParameterizedTest

@CsvSource({"2, 3, 5", "10, 20, 30", "7, 8, 15"})

void testAdditionWithCsvSource(int a, int b, int expected) {

// Use case: Test addition with CSV source data

int result = calculator.add(a, b);

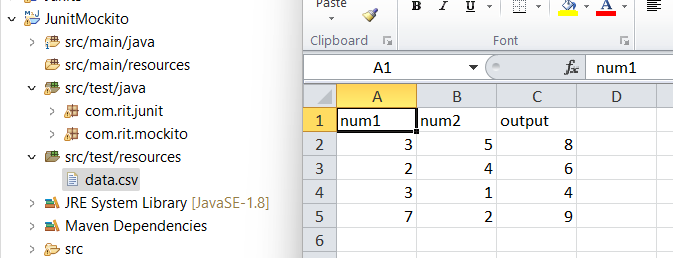
assertEquals(expected, result, "The sum should be correct");

}

}

**4. Parameterized Tests Using @CsvFileSource**

In this example, we demonstrate how to read input data from a csv file

****

package com.rit.junit;

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

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.CsvFileSource;

public class T6CSVFileTest {

Calculator calculator = new Calculator();

@ParameterizedTest

@CsvFileSource(resources = "/data.csv", numLinesToSkip=1)

void testAdditionWithCsvSource(int a, int b, int expected) {

// Use case: Test addition with CSV source data

int result = calculator.add(a, b);

assertEquals(expected, result, "The sum should be correct");

}

}

**5. Test Disabling Using @Disabled**

This example shows how to disable and conditionally enable tests based on system properties or time.

import org.junit.jupiter.api.\*;

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

public class ConditionalCalculatorTest {

Calculator calculator = new Calculator();

@Test

@Disabled("This test is temporarily disabled due to a known issue")

void disabledTest() {

// This test will not run

assertEquals(10, calculator.add(5, 5), "This test should not run");

}

}

**6. Nested Tests Using @Nested**

In this example, we demonstrate how to use @Nested to organize related test cases inside a nested class.

import org.junit.jupiter.api.\*;

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

public class CalculatorNestedTest {

@Nested

@DisplayName("Addition Tests")

class AdditionTests {

Calculator calculator = new Calculator();

@Test

@DisplayName("Test Add Positive Numbers")

void testAddPositiveNumbers() {

int result = calculator.add(5, 7);

assertEquals(12, result, "5 + 7 should be 12");

}

@Test

@DisplayName("Test Add Negative Numbers")

void testAddNegativeNumbers() {

int result = calculator.add(-3, -4);

assertEquals(-7, result, "-3 + (-4) should be -7");

}

}

@Nested

@DisplayName("Subtraction Tests")

class SubtractionTests {

Calculator calculator = new Calculator();

@Test

@DisplayName("Test Subtract Positive Numbers")

void testSubtractPositiveNumbers() {

int result = calculator.subtract(10, 4);

assertEquals(6, result, "10 - 4 should be 6");

}

@Test

@DisplayName("Test Subtract Negative Numbers")

void testSubtractNegativeNumbers() {

int result = calculator.subtract(-5, -3);

assertEquals(-2, result, "-5 - (-3) should be -2");

}

}

@Nested

@DisplayName("Multiplication Tests")

class MultiplicationTests {

Calculator calculator = new Calculator();

@Test

@DisplayName("Test Multiply Positive Numbers")

void testMultiplyPositiveNumbers() {

int result = calculator.multiply(4, 5);

assertEquals(20, result, "4 \* 5 should be 20");

}

@Test

@DisplayName("Test Multiply By Zero")

void testMultiplyByZero() {

int result = calculator.multiply(0, 7);

assertEquals(0, result, "0 \* 7 should be 0");

}

}

}

**7. Extended Tests Using @ExtendWith**

In this example, we'll demonstrate how to use @ExtendWith to register a custom extension in a JUnit test.

Custom Extension Class (Using BeforeEach and AfterEach)

import org.junit.jupiter.api.extension.\*;

public class CustomExtension implements BeforeEachCallback, AfterEachCallback {

@Override

public void beforeEach(ExtensionContext context) {

System.out.println("Before each test with Custom Extension");

}

@Override

public void afterEach(ExtensionContext context) {

System.out.println("After each test with Custom Extension");

}

}

Test Class Using @ExtendWith (to Use the Custom Extension)

import org.junit.jupiter.api.\*;

import org.junit.jupiter.api.extension.\*;

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

@ExtendWith(CustomExtension.class) // Register the custom extension

public class CalculatorWithExtensionTest {

Calculator calculator;

@BeforeEach

void setUp() {

calculator = new Calculator();

}

@Test

void testAddition() {

// Use case: Addition of two positive numbers

int result = calculator.add(3, 4);

assertEquals(7, result, "3 + 4 should be 7");

}

@Test

void testMultiplication() {

// Use case: Multiplying two positive numbers

int result = calculator.multiply(5, 6);

assertEquals(30, result, "5 \* 6 should be 30");

}

}

## 5. Best Practices for JUnit Testing

1. **Write Readable Test Names**: Test method names should be descriptive of what they test.
2. **Keep Tests Independent**: Ensure that tests do not depend on each other.
3. **Test One Thing at a Time**: Each test should focus on testing a single unit of functionality.
4. **Use Assertions Liberally**: Always assert the expected results.
5. **Follow Naming Conventions**: Name your test methods in a standardized way, such as methodName\_condition\_expectedResult.
6. **Mock Dependencies**: Use mock objects or frameworks like Mockito for testing code that interacts with external systems or services.

**Scenario based Task**

To implement testing for a method that validates a username in an application, you'll need to identify various validation rules for the username, such as:

* It must not be null or empty.
* It must have a minimum and maximum length (e.g., 3-15 characters).
* It should only contain allowed characters (e.g., alphanumeric and underscores).
* It should not contain special characters or spaces.
* It may not contain forbidden words (e.g., "admin").

**Steps to Implement**

**Step 1: Write the Username Validation Method**

First, create the UsernameValidator class with a validate method to perform all these checks.

**UsernameValidator.java**

public class UsernameValidator {

// Validates the username according to the rules

public boolean validate(String username) {

if (username == null || username.isEmpty()) {

return false; // Username cannot be null or empty

}

if (username.length() < 3 || username.length() > 15) {

return false; // Username must be between 3 and 15 characters

}

// Only allow alphanumeric characters and underscores

if (!username.matches("[a-zA-Z0-9\_]+")) {

return false;

}

// Check for forbidden words (e.g., "admin")

if (username.toLowerCase().contains("admin")) {

return false;

}

return true; // Passed all validation checks

}

}

**Step 2: Create the Unit Tests**

Now, write a test class to cover different possible cases for the username validation. This includes valid, invalid, and edge cases.

**UsernameValidatorTest.java**

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

import org.junit.jupiter.params.provider.CsvSource;

public class UsernameValidatorTest {

private final UsernameValidator validator = new UsernameValidator();

// Test with various valid usernames

@Test

public void testValidUsername() {

assertTrue(validator.validate("validUsername123"));

assertTrue(validator.validate("user\_name"));

assertTrue(validator.validate("JohnDoe"));

}

// Test with invalid usernames (empty, null, too short, or too long)

@Test

public void testInvalidUsername() {

assertFalse(validator.validate("")); // Empty username

assertFalse(validator.validate("ab")); // Too short

assertFalse(validator.validate("a".repeat(16))); // Too long (16 characters)

assertFalse(validator.validate(null)); // Null username

}

// Test with usernames that contain special characters or spaces

@Test

public void testUsernameWithSpecialCharacters() {

assertFalse(validator.validate("user!name")); // Special character

assertFalse(validator.validate("user name")); // Space in username

assertFalse(validator.validate("user@name")); // Special character '@'

}

// Test with usernames containing the forbidden word "admin"

@Test

public void testUsernameWithForbiddenWord() {

assertFalse(validator.validate("admin123")); // Contains forbidden word "admin"

assertFalse(validator.validate("superadmin")); // Contains forbidden word "admin"

}

// Test valid usernames using parameterized test

@ParameterizedTest

@ValueSource(strings = {"validUser", "John\_Doe123", "user\_1"})

public void testValidUsernames(String username) {

assertTrue(validator.validate(username));

}

// Test invalid usernames using parameterized test

@ParameterizedTest

@CsvSource({

"'', false", // Empty username

"'ab', false", // Too short

"'a'.repeat(16), false", // Too long

"'user@name', false", // Special character

"'admin123', false" // Forbidden word "admin"

})

public void testInvalidUsernames(String username, boolean expectedResult) {

assertEquals(expectedResult, validator.validate(username));

}

}

**Real-time scenario**

Let's design a comprehensive real-time scenario. We’ll create a basic application involving a UserService that manages users and their details. We'll build out the service and then write JUnit 5 test cases to validate its functionality.

**Project Structure:**

* src/main/java/
  + com/example/userservice/
    - User.java
    - UserService.java
* src/test/java/
  + com/example/userservice/
    - UserServiceTest.java

Here’s a step-by-step implementation:

**1. Define the User Model**

**File: User.java**

**package** com.rit.userservice;

**public** **class** User {

**private** String username;

**private** String email;

**public** User(String username, String email) {

**this**.username = username;

**this**.email = email;

}

*// Getters*

**public** String getUsername() {

**return** username;

}

**public** String getEmail() {

**return** email;

}

*// Setters*

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

*// Override toString for easy debugging*

@Override

**public** String toString() {

**return** "User{" +

"username='" + username + '\'' +

", email='" + email + '\'' +

'}';

}

}

**2. Create the UserService**

**File: UserService.java**

**package** com.rit.userservice;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** UserService {

**private** Map<String, User> userMap = **new** HashMap<>();

**public** UserService() {

*// Initial data*

userMap.put("john\_doe", **new** User("john\_doe", "john@example.com"));

}

**public** User getUser(String username) {

**return** userMap.get(username);

}

**public** **void** addUser(User user) {

userMap.put(user.getUsername(), user);

}

**public** **boolean** removeUser(String username) {

**return** userMap.remove(username) != **null**;

}

}

**3. Write JUnit 5 Test Cases**

**File: UserServiceTest.java**

**package** com.rit.userservice;

**import** org.junit.jupiter.api.AfterEach;

**import** org.junit.jupiter.api.BeforeEach;

**import** org.junit.jupiter.api.DisplayName;

**import** org.junit.jupiter.api.Nested;

**import** org.junit.jupiter.api.Test;

**import** org.junit.jupiter.api.Timeout;

**import** java.time.Duration;

**import** java.util.NoSuchElementException;

**import** static org.junit.jupiter.api.Assertions.\*;

**class** UserServiceTest {

**private** UserService userService;

@BeforeEach

**void** setUp() {

userService = **new** UserService();

}

@AfterEach

**void** tearDown() {

userService = **null**;

}

@Nested

@DisplayName("Tests for getUser method")

**class** GetUserTests {

@Test

@DisplayName("Should return user for valid username")

**void** shouldReturnUserForValidUsername() {

User user = userService.getUser("john\_doe");

assertNotNull(user, "User should not be null");

assertEquals("john\_doe", user.getUsername(), "Username should match");

assertEquals("john@example.com", user.getEmail(), "Email should match");

}

@Test

@DisplayName("Should return null for invalid username")

**void** shouldReturnNullForInvalidUsername() {

User user = userService.getUser("non\_existent");

assertNull(user, "User should be null for non-existent username");

}

}

@Nested

@DisplayName("Tests for addUser method")

**class** AddUserTests {

@Test

@DisplayName("Should add a new user")

**void** shouldAddNewUser() {

User newUser = **new** User("jane\_doe", "jane@example.com");

userService.addUser(newUser);

User retrievedUser = userService.getUser("jane\_doe");

assertNotNull(retrievedUser, "User should not be null after addition");

assertEquals("jane\_doe", retrievedUser.getUsername(), "Username should match");

assertEquals("jane@example.com", retrievedUser.getEmail(), "Email should match");

}

}

@Nested

@DisplayName("Tests for removeUser method")

**class** RemoveUserTests {

@Test

@DisplayName("Should remove an existing user")

**void** shouldRemoveExistingUser() {

**boolean** result = userService.removeUser("john\_doe");

assertTrue(result, "User should be removed successfully");

User user = userService.getUser("john\_doe");

assertNull(user, "User should be null after removal");

}

@Test

@DisplayName("Should not remove a non-existing user")

**void** shouldNotRemoveNonExistingUser() {

**boolean** result = userService.removeUser("non\_existent");

assertFalse(result, "Removing a non-existing user should return false");

}

}

@Test

@DisplayName("Should timeout if operation takes too long")

@Timeout(value = 1, unit = TimeUnit.SECONDS)

**void** shouldTimeoutIfOperationTakesTooLong() **throws** InterruptedException {

Thread.sleep(500); *// Simulate long operation*

assertTrue(**true**);

}

}