#### Unit Testing

Testing individual pieces or "units" of code, usually a single function or method, to ensure they work correctly.

Example: If you have a method that calculates the area of a rectangle, a unit test would check if the method returns the correct area for given inputs.

#### System Testing

Testing the entire system as a whole to ensure it meets the specified requirements. This is done after all the individual components have been integrated.

Example: Testing an entire e-commerce website to ensure all features (like browsing products, adding to cart, and checkout) work together properly.

#### Integration Testing

Testing how different components or modules of a system work together. The goal is to find issues in the interactions between them.

Example: After testing a login module and a user profile module separately, integration testing would ensure that logging in correctly leads to the user profile page.

#### Acceptance Testing

Testing done to determine if the system meets the business requirements and is ready for delivery to the user. Often done by the client or end-user.

Example: The client tests the final version of an app to ensure it meets their expectations, like checking if all features are present and work as intended.

#### Performance Testing

Testing how well a system performs under certain conditions, such as load or stress, to ensure it is fast and stable.

Example: Testing a website to see how quickly it loads and how it behaves when 1,000 users are accessing it at the same time.

#### Regression Testing

Testing existing functionality to ensure that recent changes or additions haven't broken anything.

Example: After adding a new feature to an app, regression testing ensures that older features still work correctly.

#### Security Testing

Testing to find vulnerabilities and ensure the system is protected against threats like hacking or data breaches.

Example: Testing an online banking system to ensure that unauthorized users can't access other people's accounts.

#### Load Testing

A type of performance testing where you test how the system behaves under expected user loads to ensure it can handle the traffic.

Example: Testing a new social media platform to see how it performs with 10,000 simultaneous users.

#### End-to-End Testing

Testing the entire flow of an application from start to finish, simulating real-world scenarios to ensure everything works together.

Example: Testing an online purchase process from logging in, selecting products, making a payment, and receiving a confirmation email.

These types of testing help ensure that a system is reliable, secure, and meets the users' needs.

#### Test-driven Development (TDD)

* Write a test before implimentation of code.
* Force the developer to think about requirements before implimentation.
* Tests are not skipped due to time pressure.

This only adds the unit tests, In order to properly test the application, you need to have system and integration tests as well.

#### Unit Testing advantages

* Validates the smallest units of software.
* Find bugs early and easy
* Force Developers to write better and cleaner code

Only performing unit testing will not cache all the bugs

#### What is JUnit

* Unit testing framework for Java
* Enables automated unit testing
* Must-have for TDD

Dependencies for JUnit 5 – org.junit.jupiter and

imports classes are from org.junit.jupiter.api

#### @Test

It marks the method as a test so that JUnit can execute it as part of the test suite.

With @Test, JUnit will automatically execute the test methods when the test suite runs. You don't need to call the methods manually.

#### @DisplayName

Is particularly helpful when you want to convey more detailed information than what the method name alone provides.

Applicable for test methods and test class.

#### Assertions

* Check the outcome of the test
* If the assertion fails, the test fails
* Assertions class available from org.junit.jupiter.api package

##### assertEquals(expected, actual)

checks if two values are **equal**. If the two values are equal, the test passes; if they are not, the test fails.

##### assertNotEquals(unexpected, actual)

checks that two values are **not equal**. If the two values are different, the test passes. If they are equal, the test fails.

##### assertTrue(condition)

checks whether a given condition is true. If the condition is true, the test passes. If it's false, the test fails.

##### assertFalse(condition)

checks whether a given condition is false. If the condition is false, the test passes. If it's true, the test fails.

##### assertNull(object)

The test passes if the object is null, otherwise it fails.

##### assertNotNull(object)

The test passes if the object is **not** null, otherwise it fails.

##### assertThrows(expectedExceptionClass, executable)

* assertThrows ensures that a method throws the correct exception in a specific situation.
* It is used for testing exceptional cases, like invalid inputs or error conditions in your code.

This is useful for ensuring that your code handles error scenarios correctly by throwing the appropriate exceptions.

##### assertAll(executable1, executable2, ...)

It allows you to group multiple assertions together in a single test and ensure that all of them are executed, regardless of whether some fail. This is helpful when you want to check multiple conditions in a test case without stopping at the first failure.

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

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

**public class CalculatorTest {**

**@Test**

**public void testMultipleAssertions() {**

**Calculator calculator = new Calculator();**

**int result = calculator.add(2, 3);**

**int subtractResult = calculator.subtract(5, 2);**

**assertAll(**

**() -> assertEquals(5, result, "Addition result should be 5"),**

**() -> assertEquals(3, subtractResult, "Subtraction result should be 3"),**

**() -> assertTrue(result > 0, "Addition result should be positive")**

**);**

**}**

**}**

##### assertTimeout(duration, executable)

It ensures that a block of code (usually a test) completes within a specified time limit. If the code takes longer than the specified time, the test fails. This is useful for performance testing or verifying that certain operations complete in a reasonable amount of time.

java.time.Duration

Duration.ofSeconds(2)

#### Assumptions

* Setting a condition for executing a test
* If assumption is met, the test will be executed, else the test will not be executed.
* Assumptions are in org.junit.jupiter.api package

##### assumeTrue(condition)

It Runs the test only if the condition is true. Otherwise, the test is skipped.

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assumptions.assumeTrue;

public class EnvironmentTest {

@Test

public void testOnWindows() {

String os = System.getProperty("os.name");

assumeTrue(os.startsWith("Windows"));

// Test code that should only run on Windows

}

}

##### assumeFalse(condition)

It Runs the test only if the condition is false. Otherwise, the test is skipped.

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assumptions.assumeFalse;

public class EnvironmentTest {

@Test

public void testNotOnMac() {

String os = System.getProperty("os.name");

assumeFalse(os.startsWith("Mac"));

// Test code that should not run on macOS

}

}

##### assumingThat(condition, executable)

It Executes a block of code only if the condition is true. Useful for running multiple assertions conditionally.

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assumptions.assumingThat;

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

public class EnvironmentTest {

@Test

public void testWithMultipleAssertions() {

String env = System.getProperty("env");

assumingThat("test".equals(env), () -> {

assertTrue(1 + 1 == 2);

assertTrue("test".length() == 4);

// Other assertions or test code

});

}

}

#### Test Order

* Without specifying, we can’t predict the order
* usually, this is not a problem - tests should be independent

##### @TestMethodOrder

JUnit 5 provides mechanisms for controlling the order of test execution using the @TestMethodOrder annotation and custom ordering strategies. This allows you to specify the order in which tests are executed within a test class.

###### MethodOrderer.OrderAnnotation

* Allows you to specify the order of test methods using the @Order annotation.
* Add the @TestMethodOrder(MethodOrderer.OrderAnnotation.class) to your test class and use the @Order annotation on individual test methods to define their execution order.

import org.junit.jupiter.api.Order;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.TestMethodOrder;

import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;

@TestMethodOrder(OrderAnnotation.class)

public class OrderedTests {

@Test

@Order(1)

public void testFirst() {

System.out.println("First test");

}

@Test

@Order(2)

public void testSecond() {

System.out.println("Second test");

}

@Test

@Order(3)

public void testThird() {

System.out.println("Third test");

}

}

###### MethodOrderer.Alphanumeric

* Executes test methods in alphanumeric order based on their method names.
* Add the @TestMethodOrder(MethodOrderer.Alphanumeric.class) annotation to your test class.

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.TestMethodOrder;

import org.junit.jupiter.api.MethodOrderer.Alphanumeric;

@TestMethodOrder(Alphanumeric.class)

public class AlphanumericOrderedTests {

@Test

public void testA() {

System.out.println("Test A");

}

@Test

public void testB() {

System.out.println("Test B");

}

}

#### @Nested

@Nested in JUnit 5 allows you to group related test classes inside a main test class. It helps organize tests into logical groups, making it easier to test different scenarios or contexts within the same test suite. Each nested class can have its own tests, which are run as part of the main test class, improving the structure and readability of your test code.

#### @RepeatedTest(numberOfRepetitions)

@RepeatedTest in JUnit 5 is used to run the same test multiple times. It helps in scenarios where you want to verify that your test behaves consistently across multiple executions, such as when testing flaky code or performance under repeated conditions.

#### @ParameterizedTest

@ParameterizedTest in JUnit 5 allows you to run a test multiple times with different inputs. It helps in testing the same logic with various sets of data, ensuring that the code behaves correctly across different scenarios.

import org.junit.jupiter.params.ParameterizedTest;

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

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

public class ParameterizedTestExample {

@ParameterizedTest

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

public void testEvenNumbers(int number) {

assertTrue(number % 2 == 0); // Test will run with each number

}

}

##### @CsvSource

Is used with @ParameterizedTest to pass multiple sets of comma-separated values (CSV) as inputs to a test. It allows you to supply multiple arguments for each test run, making it useful for testing with various combinations of data.

@ParameterizedTest

@CsvSource({ "apple, 1", "banana, 2", "cherry, 3" })

public void testWithCsvSource(String fruit, int rank) {

assertEquals(fruit.length(), rank); // Example test logic

}

**@CsvFileSource(resources = "/data.csv", delimiter= ‘;’)**

#### @Timeout

Is used to specify a time limit for the execution of a test. If the test exceeds this time limit, it fails. This is useful for ensuring that tests do not run indefinitely and helps identify performance issues or infinite loops in the code.

@Timeout(value = 5, unit = TimeUnit.SECONDS) // can use test case alone or at class level

#### @Execution(ExecutionMode.CONCURRENT)

The tests annotated with this mode will run concurrently, meaning they can run in parallel, which can speed up test execution.

Annotation can be used at class level or method level

#### @BeforeAll

Executes a method once before all tests in the test class.

#### @AfterAll

Executes a method once after all tests in the test class have completed.

These annotations are typically used for initializing shared resources (like opening a database connection or starting a server) and cleaning up those resources after all tests have run.

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

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

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

**public class ExampleTest {**

**@BeforeAll**

**static void setup() {**

**System.out.println("Before all tests: Initialize shared resources");**

**}**

**@AfterAll**

**static void cleanup() {**

**System.out.println("After all tests: Cleanup shared resources");**

**}**

**@Test**

**void test1() {**

**System.out.println("Test 1 is running");**

**}**

**@Test**

**void test2() {**

**System.out.println("Test 2 is running");**

**}**

**}**

The methods annotated with @BeforeAll and @AfterAll must be static because they are executed at the class level, not at the instance level (i.e., before/after the entire test class rather than each individual test method).

#### @BeforeEach and @AfterEach

Are annotations in JUnit 5 used to run setup and teardown code before and after each individual test method within a test class.

**These methods are used to maintain isolation between tests, ensuring that the outcome of one test does not affect the others.**

#### Custom messages and reports

1. @DisplayName,
2. 3rd parameter in Assertions
3. By adding the below plugin in pom.xml and running below commands in the terminal , unit texts will be executed and reports will be available under the target folder.

**<plugins>**

**<plugin>**

**<artifactId>maven-surefire-plugin</artifactId> <version>2.22.2</version>**

**</plugin>**

**<plugin>**

**<artifactId>maven-failsafe-plugin</artifactId> <version>2.22.2</version>**

**</plugin>**

**</plugins>**

**mvn surefire-report:report**

**Targets → site → surefire-report.html will have a detailed test report.**

#### Conditional Test executions:

@EnabledOnOs({OS.MAC})

@EnabledOnOs({OS.WINDOWS})

@EnabledOnJre({JRE.JAVA\_16})

@DisabledOnJre({JRE.JAVA\_16})

@EnabledIfSystemProperty(named=”os.version”, matches=”x”)

@DisabledIfSystemProperty(named=”os.version”, matches=”x”)

@EnabledIfEnvironmentVariable(named=”USERNAME”, matches=”xyz”)

If condition true → test runs , else → test Ignored

#### Disabling unit tests

**@Disabled / @Disabled(“message”)**

## **TDD - Spring Code Test**

#### @RunWith(SpringRunner.class)

// Instructs JUnit to use SpringRunner to run the test

#### @SpringBootTest

// Loads the full application context for testing

@SpringBootTest(

classes = MyApp.class,

webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT,

properties = {"spring.main.banner-mode=off"},

args = "--custom.arg=foo",

activeProfiles = "test"

)

public class MyCombinedTest {

@Test

public void testSomething() {

// Your test code here

}

}

**WebEnvironment.MOCK**: Loads a mock servlet environment.

**WebEnvironment.RANDOM\_PORT**: Loads an actual servlet environment and starts the server on a random port.

**WebEnvironment.DEFINED\_PORT**: Uses the port defined in the application properties.

**WebEnvironment.NONE**: No web environment is loaded.

#### @RunWith(MockitoJUnitRunner.class)

This tells JUnit to use the MockitoJUnitRunner to run the tests. This runner automatically initializes any fields annotated with @Mock, @Spy, etc.

import org.junit.runner.RunWith;

import org.mockito.junit.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)

#### @Mock

Annotates a field that should be mocked, meaning Mockito will create a mock instance of a Service.

#### **@WebMvcTest**

Is a specialized annotation in Spring Boot used to test the **web layer** of your application, specifically controllers. It focuses on testing the web layer in isolation, without loading the entire Spring application context, which makes the tests faster and more focused.

@WebMvcTest is used to test Spring MVC controllers and their behavior.

It automatically configures Spring MVC infrastructure and components like MockMvc, Jackson, ViewResolvers, etc.

It’s ideal for testing the endpoints, request handling, and response content of a controller without involving service or repository layers.

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.boot.test.mock.mockito.MockBean;

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;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

@WebMvcTest(MyController.class) // Specifies that only the web layer should be tested

public class MyControllerTest {

@Autowired

private MockMvc mockMvc; // Injected MockMvc for testing HTTP requests

@MockBean

private MyService myService; // Mocked service that the controller depends on

@Test

public void testGetEndpoint() throws Exception {

// Mock service method if necessary using Mockito

// Mockito.when(myService.someMethod()).thenReturn(someValue);

mockMvc.perform(get("/my-endpoint")) // Perform a GET request to the controller's endpoint

.andExpect(status().isOk()) // Expect the HTTP status to be 200 OK

.andExpect(content().string("Expected response content")); // Expect specific response content

}

}

#### **@DataJpaTest**

Is a specialized annotation in Spring Boot designed for testing the JPA (Java Persistence API) layer of your application. This annotation focuses on testing the repository layer, ensuring that the interactions with the database through JPA are working as expected.

* @DataJpaTest is used to test JPA repositories and other database-related components.
* It sets up an in-memory database (like H2) by default, which allows for fast and isolated tests without affecting the actual database.
* It configures only the necessary components related to JPA, like EntityManager, JpaRepositories, and DataSource.

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;

import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace;

import org.springframework.test.annotation.Rollback;

import static org.assertj.core.api.Assertions.assertThat;

@DataJpaTest

@AutoConfigureTestDatabase(replace = Replace.NONE) // Optionally configure the database behavior

public class MyRepositoryTest {

@Autowired

private MyRepository myRepository;

@Test

@Rollback(false) // Optionally control transaction rollback behavior

public void testSaveAndFindEntity() {

MyEntity entity = new MyEntity();

entity.setName("Test Name");

myRepository.save(entity);

MyEntity foundEntity = myRepository.findById(entity.getId()).orElse(null);

assertThat(foundEntity).isNotNull();

assertThat(foundEntity.getName()).isEqualTo("Test Name");

}

}

## Mocking Static Methods

Mockito cannot mock static methods, but PowerMockito can. Static methods are often considered bad design for testability because they are tightly coupled to the class they are defined in, making them hard to replace in a test. However, if you’re working with legacy code or a codebase you cannot refactor, you might need to mock static methods.

**public class Utility {**

**public static String staticMethod() {**

**return "Hello, World!";**

**}**

**}**

**// Test**

**@RunWith(PowerMockRunner.class)**

**@PrepareForTest(Utility.class)**

**public class MyTest {**

**@Test**

**public void testStaticMethod() {**

**PowerMockito.mockStatic(Utility.class);**

**Mockito.when(Utility.staticMethod()).thenReturn("Mocked Hello, World!");**

**String result = Utility.staticMethod();**

**assertEquals("Mocked Hello, World!", result);**

**}**

**}**

## **Mocking Private Methods**

Private methods cannot be directly mocked with Mockito, but they can be mocked using PowerMockito. This can be helpful in testing legacy code or highly encapsulated logic.

**public class MyClass {**

**private String privateMethod() {**

**return "Hello, World!";**

**}**

**public String publicMethod() {**

**return privateMethod();**

**}**

**}**

**// Test**

**@RunWith(PowerMockRunner.class)**

**@PrepareForTest(MyClass.class)**

**public class MyTest {**

**@Test**

**public void testPrivateMethod() throws Exception {**

**MyClass myClass = PowerMockito.spy(new MyClass());**

**PowerMockito.doReturn("Mocked Hello, World!").when(myClass, "privateMethod");**

**String result = myClass.publicMethod();**

**assertEquals("Mocked Hello, World!", result);**

**}**

**}**

## **Mocking Final Classes and Methods**

Mockito cannot mock final classes and methods because they are not designed to be overridden. PowerMockito, however, can mock final methods and classes, making it useful when dealing with unmodifiable code.

**public final class FinalClass {**

**public final String finalMethod() {**

**return "Hello, World!";**

**}**

**}**

**// Test**

**@RunWith(PowerMockRunner.class)**

**@PrepareForTest(FinalClass.class)**

**public class MyTest {**

**@Test**

**public void testFinalMethod() {**

**FinalClass finalClassMock = PowerMockito.mock(FinalClass.class);**

**Mockito.when(finalClassMock.finalMethod()).thenReturn("Mocked Hello, World!");**

**String result = finalClassMock.finalMethod();**

**assertEquals("Mocked Hello, World!", result);**

**}**

**}**

# **Mockito**

Is a popular Java testing framework used to create mock objects and write unit tests. Here's a comprehensive guide to the most commonly used methods in Mockito, along with examples.

## **1. Creating Mocks**

Mockito.mock(): Creates a mock instance of a class or interface.

**MyService mockService = Mockito.mock(MyService.class);**

## **2. Stubbing Method Calls**

Mockito.when(): Defines behavior for a method call on a mock object.

Mockito.thenReturn(): Specifies the value to return when a mocked method is called.

**Mockito.when(mockService.getMessage()).thenReturn("Mocked Message");**

Mockito.thenThrow(): Specifies an exception to throw when a mocked method is called.

**Mockito.when(mockService.getMessage()).thenThrow(new RuntimeException("Error"));**

Mockito.thenAnswer(): Allows custom behavior using an Answer object.

**Mockito.when(mockService.getMessage()).thenAnswer(invocation -> "Dynamic Message");**

Mockito.thenCallRealMethod(): Calls the real method instead of mocking it.

**Mockito.when(mockService.getMessage()).thenCallRealMethod();**

## **3. Verification**

Mockito.verify(): Verifies that a method was called on the mock object.

**Mockito.verify(mockService).getMessage();**

Mockito.verify() with times: Verifies the number of times a method was called.

**Mockito.verify(mockService, Mockito.times(1)).getMessage();**

Mockito.verifyNoMoreInteractions(): Ensures that no other interactions happen on the mock.

**Mockito.verifyNoMoreInteractions(mockService);**

Mockito.verifyZeroInteractions(): Verifies that no interactions occurred on the mock.

**Mockito.verifyZeroInteractions(mockService);**

## **4. Argument Matching**

Mockito.any(): Matches any object of the given type.

**Mockito.when(mockService.process(Mockito.any(String.class))).thenReturn("Processed");**

Mockito.eq(): Matches an exact value.

**Mockito.when(mockService.process(Mockito.eq("Input"))).thenReturn("Processed");**

## **5. Spying**

Mockito.spy(): Creates a spy of a real object, allowing to stub and verify calls.

**MyService spyService = Mockito.spy(new MyService());**

**Mockito.when(spyService.getMessage()).thenReturn("Spy Message");**

## **6. Exception Handling**

Mockito.doThrow(): Specifies an exception to throw when a method is called on a mock or spy.

**Mockito.doThrow(new RuntimeException()).when(mockService).process("ErrorInput");**

## **7. Mocking Void Methods**

Mockito.doNothing(): Does nothing when a void method is called (default behavior).

**Mockito.doNothing().when(mockService).voidMethod();**

Mockito.doThrow(): Throws an exception when a void method is called.

**Mockito.doThrow(new RuntimeException()).when(mockService).voidMethod();**

## **8. Timeout Verification**

Mockito.timeout(): Verifies that a method was called within a specific time.

**Mockito.verify(mockService, Mockito.timeout(100)).getMessage();**