TDD using JUnit and Mockito

Introduction to Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development approach in which test cases are written before writing the actual code. This method ensures that the software meets requirements and behaves correctly from the start. The development process follows a strict cycle of writing tests, making the tests pass with minimal code, and then refining the implementation.

Definition and Principles of TDD

Definition

Test-Driven Development (TDD) is a software development practice where developers write automated test cases **before** writing the actual functional code. The goal of TDD is to guide the development process by focusing on writing small, testable units of code that fulfill specific requirements.

Principles of TDD

1. **Write a test before writing code**
    - The test defines what the code should do before the implementation exists.
2. **Write only enough code to pass the test**
    - The initial implementation should be minimal, just enough to make the test pass.
3. **Refactor the code to improve quality**
    - Once the test passes, clean up the code while ensuring that the test still passes.
4. **Repeat the cycle**
    - This process continues iteratively, ensuring that all new features are covered by tests.
5. **Tests act as documentation**
    - Since tests describe expected behavior, they serve as living documentation for the system.

Benefits of TDD in Software Development

TDD brings several advantages that improve the overall quality and maintainability of software.

1. Improves Code Quality

- Writing tests first forces developers to think through requirements, leading to better-designed code.

2. Reduces Bugs and Errors

- Since tests are written before code, they catch potential issues early in the development cycle.

3. Provides Better Code Coverage

- Since every feature has a test before implementation, it ensures all parts of the system are tested.

4. Encourages Simplicity and Modularity

- Since developers only write code to pass tests, unnecessary complexity is avoided.

5. Makes Refactoring Safer

- Developers can confidently refactor code, knowing that tests will catch any unintended changes.

6. Acts as Living Documentation

- Tests clearly describe expected behavior, making it easier for new developers to understand the system.

7. Helps in Continuous Integration and Deployment

- Automated tests make it easier to integrate new changes and deploy software confidently.

TDD Lifecycle: Red-Green-Refactor

TDD follows an iterative development cycle known as **Red-Green-Refactor**.

1. Red (Write a Failing Test)

- Write a test case for a new feature.
- The test fails initially since the feature does not exist.
- Example:

- @Test
- public void testAddition() {
- Calculator calculator = new Calculator();
- assertEquals(5, calculator.add(2, 3));
- }
-
- Since add() method is not yet implemented, this test will fail.

2. Green (Write Minimum Code to Pass the Test)

- Write the simplest implementation that makes the test pass.
- Example:
- public class Calculator {
- public int add(int a, int b) {
- return a + b;
- }
- }
-
- The test now passes.

3. Refactor (Improve Code Quality)

- Optimize the code while keeping tests passing.
- Example:
  - Removing redundant code
  - Improving efficiency
  - Enhancing readability
- Ensure that after refactoring, all tests still pass.

Repeat the Cycle

- Once refactoring is done, the cycle repeats for the next feature.

TDD vs. Traditional Development

Feature	Test-Driven Development (TDD)	Traditional Development
Test Writing	Tests are written **before** writing the code.	Tests are written **after** development.
Code Quality	Results in **clean, modular, and testable** code.	May lead to **tightly coupled** and complex code.
Debugging	Errors are caught **early** in development.	Bugs are found **later**, making debugging harder.
Development Speed	Initial development may be slower, but overall **reduces debugging time**.	Faster in the beginning, but debugging and maintenance take **more time**.
Refactoring	Safe refactoring due to test coverage.	Risky refactoring, as there may be no tests to catch issues.
Documentation	Tests act as **live documentation**.	Requires **separate documentation** for behavior.
Confidence in Changes	High confidence due to automated tests.	Low confidence as changes might break functionality.

Conclusion

Test-Driven Development (TDD) is a powerful approach that helps developers build robust, maintainable, and well-tested software. While it requires discipline and a shift in mindset, the benefits in terms of **code quality, fewer bugs, safer refactoring, and better documentation** make it a highly effective software development practice.

Would you like me to provide real-world examples or guidance on implementing TDD in a specific programming language? 🚀

Getting Started with JUnit

JUnit is a popular **testing framework** for Java applications that allows developers to write and run **unit tests** efficiently. It provides a structured way to verify that individual parts of an application function correctly. JUnit is widely used in **Test-Driven Development (TDD)** and **automated testing**.

Overview of JUnit Framework

What is JUnit?

JUnit is an open-source testing framework for Java that allows developers to write repeatable tests. It is widely used for **unit testing**, where individual components of a program are tested in isolation.

Key Features of JUnit

- **Simple and Lightweight**: No extra setup is required beyond adding the JUnit dependency.
- **Annotations for Test Methods**: Uses annotations like @Test, @Before, @After, etc., for test execution.
- **Assertions for Validation**: Provides assertion methods to check expected vs. actual results.
- **Automatic Test Execution**: Tests can be executed automatically using test runners.
- **Integration with Build Tools**: Works with **Maven**, **Gradle**, and **IDE plugins** like Eclipse and IntelliJ.

Writing and Running Simple JUnit Tests

Adding JUnit to Your Project

If you are using **Maven**, add the following dependency to pom.xml:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.9.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

For **Gradle**, add this to build.gradle:

```
dependencies {
  testImplementation 'org.junit.jupiter:junit-jupiter:5.9.1'
```

```
}
```

Writing a Simple JUnit Test

Here's a basic example of a **JUnit test for a calculator class**:

Calculator.java (Class to be tested)

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }
}
```

CalculatorTest.java (JUnit Test Class)

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

public class CalculatorTest {

    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result, "2 + 3 should be 5");
    }
```

```
    @Test
    public void testSubtraction() {
        Calculator calculator = new Calculator();
        int result = calculator.subtract(5, 3);
        assertEquals(2, result, "5 - 3 should be 2");
    }
}
```

Running JUnit Tests

- In **IntelliJ IDEA or Eclipse**, right-click the test class and select **Run 'CalculatorTest'**.
- If using **Maven**, run:
- mvn test
-
- If using **Gradle**, run:
- gradle test
-

Annotations in JUnit

JUnit provides various **annotations** to control test execution.

Annotation	Description
@Test	Marks a method as a test case.
@BeforeEach	Runs before each test (e.g., setting up test data).
@AfterEach	Runs after each test (e.g., cleaning up resources).
@BeforeAll	Runs **once** before all tests (for expensive setup).
@AfterAll	Runs **once** after all tests (for cleanup).

Annotation	Description
@Disabled	Skips the test method (useful for incomplete tests).

Example: Using Annotations

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

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

public class CalculatorTest {

  private Calculator calculator;

  @BeforeEach
  public void setup() {
    calculator = new Calculator(); // This runs before each test
  }

  @AfterEach
  public void tearDown() {
    System.out.println("Test case executed!");
  }

  @Test
  public void testAddition() {
    assertEquals(5, calculator.add(2, 3));
  }

  @Test
  public void testSubtraction() {
```

```
    assertEquals(2, calculator.subtract(5, 3));
  }
}
```

In this example:

- @BeforeEach initializes the calculator **before each test**.
- @AfterEach prints a message **after each test**.

Assertions for Verifying Expected Outcomes

JUnit provides assertion methods to check if the **expected output** matches the **actual output**.

Assertion Method	Description
assertEquals(expected, actual)	Checks if two values are equal.
assertNotEquals(expected, actual)	Checks if two values are **not** equal.
assertTrue(condition)	Checks if the condition is **true**.
assertFalse(condition)	Checks if the condition is **false**.
assertNotNull(object)	Checks if an object is **not null**.
assertNull(object)	Checks if an object is **null**.
assertThrows(Exception.class, () -> code)	Checks if an exception is thrown.

Example: Assertions

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

public class AssertionExampleTest {

  @Test
  public void testAssertions() {
```

```
    assertEquals(10, 5 + 5, "Sum should be 10");
    assertNotEquals(10, 3 + 4, "Sum should not be 10");
    assertTrue(3 < 5, "3 should be less than 5");
    assertFalse(10 < 5, "10 should not be less than 5");
    assertNotNull("JUnit", "String should not be null");
    assertThrows(ArithmeticException.class, () -> {
      int result = 10 / 0;
    }, "Should throw ArithmeticException");
  }
}
```

Test Fixture Setup and Teardown

A **test fixture** is the precondition setup required for tests, such as **creating objects, initializing variables, and cleaning up resources**.

Using Setup and Teardown Methods

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

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

public class CalculatorTest {

  private Calculator calculator;

  @BeforeAll
  public static void initAll() {
    System.out.println("Setup for all tests (Runs once)");
  }
```

```
    @BeforeEach
    public void init() {
        calculator = new Calculator(); // Runs before each test
        System.out.println("Setting up test case");
    }

    @Test
    public void testAddition() {
        assertEquals(5, calculator.add(2, 3));
    }

    @AfterEach
    public void tearDown() {
        System.out.println("Cleaning up test case");
    }

    @AfterAll
    public static void tearDownAll() {
        System.out.println("Cleanup after all tests (Runs once)");
    }
}
```

Execution Order

1. @BeforeAll (once before all tests)
2. @BeforeEach (before each test)
3. @Test (test execution)
4. @AfterEach (after each test)

5. @AfterAll (once after all tests)

Conclusion

JUnit is a **powerful** framework for writing unit tests in Java. It provides **annotations, assertions, and test fixtures** that help developers write **structured and repeatable** tests. Using JUnit in software development ensures **higher code quality, fewer bugs, and better maintainability**.

Would you like me to cover **mocking with Mockito** or **TDD with JUnit** next? 🚀

Advanced JUnit Features

JUnit offers **advanced testing features** that help improve test efficiency, flexibility, and coverage. These include **parameterized tests, test suites, test execution order, exception testing, and timeout testing**.

---

1. Parameterized Tests

What are Parameterized Tests?

Parameterized tests allow a **single test method** to be executed **multiple times** with different inputs. This eliminates code duplication and improves maintainability.

Setting Up Parameterized Tests (JUnit 5)

JUnit 5 provides the @ParameterizedTest annotation along with **different sources** for test inputs.

Example: Testing a Calculator with Different Inputs

```
import org.junit.jupiter.params.ParameterizedTest;

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

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


public class CalculatorTest {


  private final Calculator calculator = new Calculator();


  @ParameterizedTest
```

```
  @CsvSource({
    "2, 3, 5",
    "5, 5, 10",
    "10, 20, 30"
  })
  public void testAddition(int a, int b, int expected) {
    assertEquals(expected, calculator.add(a, b));
  }
}
```

Other Input Sources for Parameterized Tests

JUnit provides multiple sources for parameterized tests:

Annotation	Description
@ValueSource	Provides a single array of values.
@CsvSource	Uses comma-separated values.
@CsvFileSource	Reads data from a CSV file.
@EnumSource	Uses an Enum for inputs.
@MethodSource	Calls a method to generate test data.

---

2. Test Suites and Categories

What are Test Suites?

A **test suite** groups multiple test classes and runs them together. This is useful when organizing **unit tests, integration tests, and performance tests** separately.

Creating a Test Suite (JUnit 5)

JUnit 5 provides the @Suite annotation to create test suites.

```
import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;
```

```
@Suite
@SelectClasses({CalculatorTest.class, MathOperationsTest.class})
public class TestSuiteExample {
}
```

Running Test Suites

- You can run the test suite in your IDE or with Maven/Gradle.
- This is useful for **running specific groups of tests** (e.g., unit tests, integration tests).

---

3. Test Execution Order

Controlling Test Execution Order

JUnit 5 allows developers to specify the **order of test execution** using @TestMethodOrder.

Example: Running Tests in a Defined Order

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

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class OrderedTestExample {

  @Test
  @Order(2)
  public void secondTest() {
    System.out.println("This runs second");
  }

  @Test
  @Order(1)
```

```
    public void firstTest() {
        System.out.println("This runs first");
    }

    @Test
    @Order(3)
    public void thirdTest() {
        System.out.println("This runs third");
    }
}
```

Other Execution Order Strategies

JUnit provides different ordering strategies:

Strategy	Description
MethodOrderer.OrderAnnotation.class	Orders tests based on @Order annotation.
MethodOrderer.Alphanumeric.class	Runs tests alphabetically by method name.
MethodOrderer.Random.class	Runs tests in a random order.

---

4. Exception Testing

Testing if a Method Throws an Exception

JUnit provides assertThrows() to check if a specific **exception** is thrown.

Example: Testing Division by Zero

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

public class ExceptionTestExample {
```

```
  @Test
  public void testDivisionByZero() {
    ArithmeticException exception = assertThrows(ArithmeticException.class, () -> {
      int result = 10 / 0;
    });

    assertEquals("/ by zero", exception.getMessage());
  }
}
```

- assertThrows() ensures the correct exception is thrown.
- getMessage() verifies the exception message.

---

5. Timeout and Performance Testing

What is Timeout Testing?

Timeout tests ensure that a method executes within a specified time **to detect performance issues**.

Setting a Timeout for a Test

JUnit provides assertTimeout() to **set a time limit** for test execution.

Example: Ensuring a Method Runs Within 500ms

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import java.time.Duration;

public class TimeoutTestExample {

  @Test
  public void testPerformance() {
```

```
    assertTimeout(Duration.ofMillis(500), () -> {
      Thread.sleep(400); // Simulating a delay
    });
  }
}
```

- If the method **exceeds 500ms**, the test **fails**.

Forcing a Timeout Failure with assertTimeoutPreemptively()

```
@Test
public void testPreemptiveTimeout() {
  assertTimeoutPreemptively(Duration.ofMillis(500), () -> {
    Thread.sleep(600); // This will fail
  });
}
```

- assertTimeoutPreemptively() **stops execution** immediately when the time limit is exceeded.

Conclusion

JUnit provides powerful **advanced testing features** to improve code quality and efficiency:

Feature	Benefit
Parameterized Tests	Reduce duplication by testing multiple inputs.
Test Suites	Group tests for better organization.
Test Execution Order	Control the sequence of test execution.
Exception Testing	Verify if expected exceptions occur.
Timeout Testing	Ensure methods run within a specific time.

Would you like me to demonstrate **mocking with Mockito** or **integrating JUnit with Spring Boot** next? 🚀

Mockito Basics

Mockito is a **popular Java framework** for **unit testing** that allows you to create **mock objects** to simulate dependencies in your code. It is mainly used to **mock external dependencies** (e.g., database calls, web services) so that tests can focus only on the logic of the class being tested.

---

1. Introduction to Mockito

What is Mockito?

Mockito is an open-source **mocking framework** for Java that allows developers to create **test doubles (mocks)** instead of using real objects.

Why Use Mockito?

- **Isolates unit tests** by replacing dependencies with mocks.
- **Removes the need for real databases, APIs, or external systems**.
- **Increases test reliability** by avoiding flaky external dependencies.
- **Improves test speed** since mocks are lightweight and faster.

Adding Mockito to Your Project

For Maven (pom.xml)

```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.2.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

For Gradle (build.gradle)

```
dependencies {
    testImplementation 'org.mockito:mockito-core:5.2.0'
}
```

---

2. Mocking and Stubbing

Mocking vs. Stubbing

Concept	Description
Mocking	Creating a fake version of an object to track interactions.
Stubbing	Defining fake return values when specific methods are called.

Creating a Mock Object

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import static org.mockito.Mockito.*;

public class MockExampleTest {

    @Test
    public void testMock() {
        // Create a mock object
        Calculator calculator = mock(Calculator.class);

        // Stubbing: Define behavior when the method is called
        when(calculator.add(2, 3)).thenReturn(5);

        // Verify that the mocked method returns the expected value
        System.out.println(calculator.add(2, 3)); // Output: 5
```

```
  }
}
```

- mock(ClassName.class): Creates a mock instance.
- when(...).thenReturn(...): Defines a return value for a method call.

---

3. Verifying Interactions

Checking Method Calls

Mockito allows verifying **how many times a method is called** and **with what parameters**.

Example: Verifying Method Calls

```
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

public class VerificationTest {

  @Test
  public void testVerification() {
    Calculator calculator = mock(Calculator.class);

    // Calling mocked methods
    calculator.add(2, 3);
    calculator.add(2, 3);
    calculator.subtract(5, 3);

    // Verify if add(2,3) was called exactly 2 times
    verify(calculator, times(2)).add(2, 3);
```

```
        // Verify if subtract(5,3) was called once
        verify(calculator, times(1)).subtract(5, 3);

        // Verify that multiply() was never called
        verify(calculator, never()).multiply(anyInt(), anyInt());
    }
}
```

Verification Methods

Method	Description
verify(mock).method(args)	Checks if a method was called with specific arguments.
verify(mock, times(n)).method(args)	Checks if a method was called **n** times.
verify(mock, never()).method(args)	Ensures a method **was never called**.
verify(mock, atLeast(n)).method(args)	Checks if a method was called **at least n times**.
verify(mock, atMost(n)).method(args)	Checks if a method was called **at most n times**.

---

4. Argument Matching

Mockito provides **flexible argument matchers** to verify interactions **even when exact values are unknown**.

Example: Using Argument Matchers

```
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

public class ArgumentMatcherTest {
```

```
    @Test
    public void testArgumentMatchers() {
        Calculator calculator = mock(Calculator.class);

        // Stubbing with any integer arguments
        when(calculator.add(anyInt(), anyInt())).thenReturn(100);

        // Calling method with different arguments
        System.out.println(calculator.add(10, 20)); // Output: 100
        System.out.println(calculator.add(5, 15));  // Output: 100

        // Verify that add() was called with any integer arguments
        verify(calculator, times(2)).add(anyInt(), anyInt());
    }
}
```

Common Argument Matchers

Matcher	Description
anyInt()	Matches any int value.
anyString()	Matches any String value.
anyList()	Matches any List<?> value.
eq(value)	Matches exactly the given value.

---

5. Handling Void Methods

Mocking Void Methods

Mockito provides doNothing(), doThrow(), and doAnswer() to handle **void methods**.

Example: Using doNothing()

```
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

public class VoidMethodTest {

  @Test
  public void testVoidMethod() {
    Logger logger = mock(Logger.class);

    // Stubbing a void method
    doNothing().when(logger).log("Hello");

    // Calling the void method
    logger.log("Hello");

    // Verify the method was called
    verify(logger, times(1)).log("Hello");
  }
}
```

- doNothing().when(mock).method(args): Prevents a void method from doing anything.

Example: Throwing an Exception for a Void Method

```
@Test
public void testVoidMethodThrowsException() {
  Logger logger = mock(Logger.class);

  doThrow(new RuntimeException("Logging failed")).when(logger).log("Error");
```

```
    // Calling the void method should throw an exception
    assertThrows(RuntimeException.class, () -> logger.log("Error"));
}
```

- doThrow().when(mock).method(args): Forces a void method to throw an exception.

Conclusion

Mockito simplifies **unit testing** by allowing developers to mock dependencies, control method behavior, and verify interactions.

Feature	Benefit
Mocking & Stubbing	Replace real dependencies with test doubles.
Verifying Interactions	Ensure methods are called with expected arguments.
Argument Matching	Use flexible matchers for verifying method calls.
Handling Void Methods	Control behavior of methods that return void.

Would you like to explore **Mockito with Spring Boot** or **Mocking static methods using Mockito** next? 🚀

Testing Spring Applications with JUnit and Mockito

Spring Boot applications rely on different layers, including **controllers, services, and repositories**. Using **JUnit and Mockito**, we can efficiently test each layer in isolation and conduct **integration testing** to ensure the entire application works as expected.

---

1. Overview of Spring Testing

Spring provides **built-in support** for testing via the spring-boot-starter-test dependency, which includes:

- **JUnit 5** (for unit testing)
- **Mockito** (for mocking dependencies)
- **Spring Test** (for integration testing)

- **AssertJ** (for fluent assertions)

Adding Dependencies

For **Maven (pom.xml)**:

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

For **Gradle (build.gradle)**:

```
dependencies {
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

---

2. Testing Service Layer with Mockito

Scenario: Testing a Service that Fetches Users

We will test a **UserService** that depends on a **UserRepository**.

Service Code (UserService.java)

```
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class UserService {

  private final UserRepository userRepository;

  public UserService(UserRepository userRepository) {
```

```
        this.userRepository = userRepository;
    }

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }
}
```

Test for Service Layer (UserServiceTest.java)

```
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class UserServiceTest {
```

```
@Mock
private UserRepository userRepository;

@InjectMocks
private UserService userService;

@BeforeEach
void setUp() {
    MockitoAnnotations.openMocks(this);
}

@Test
public void testGetAllUsers() {
    // Arrange
    User user1 = new User(1L, "John Doe");
    User user2 = new User(2L, "Jane Doe");
    when(userRepository.findAll()).thenReturn(Arrays.asList(user1, user2));

    // Act
    List<User> users = userService.getAllUsers();

    // Assert
    assertEquals(2, users.size());
    verify(userRepository, times(1)).findAll();
}

@Test
public void testGetUserById_Found() {
```

```
        User user = new User(1L, "John Doe");
        when(userRepository.findById(1L)).thenReturn(Optional.of(user));

        User foundUser = userService.getUserById(1L);

        assertNotNull(foundUser);
        assertEquals("John Doe", foundUser.getName());
    }

    @Test
    public void testGetUserById_NotFound() {
        when(userRepository.findById(1L)).thenReturn(Optional.empty());

        User user = userService.getUserById(1L);

        assertNull(user);
    }
}
```

Key Concepts Used

- **Mocking** the repository using @Mock.
- **Injecting mocks** into UserService using @InjectMocks.
- **Setting up mocks** using when(...).thenReturn(...).
- **Verifying interactions** with verify().

---

3. Testing Controller Layer with MockMvc

Scenario: Testing a REST Controller

Spring provides MockMvc to test controllers **without starting the server**.

Controller Code (UserController.java)

```
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return userService.getUserById(id);
    }
}
```

Test for Controller Layer (UserControllerTest.java)

```
import static org.mockito.Mockito.*;
```

```
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import java.util.Arrays;

public class UserControllerTest {

    private MockMvc mockMvc;

    @Mock
    private UserService userService;

    @InjectMocks
    private UserController userController;

    @BeforeEach
    void setUp() {
        mockMvc = MockMvcBuilders.standaloneSetup(userController).build();
    }
```

```
    @Test
    public void testGetAllUsers() throws Exception {
        when(userService.getAllUsers()).thenReturn(Arrays.asList(new User(1L, "John Doe")));

        mockMvc.perform(get("/users"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.size()").value(1))
                .andExpect(jsonPath("$[0].name").value("John Doe"));
    }

    @Test
    public void testGetUserById() throws Exception {
        when(userService.getUserById(1L)).thenReturn(new User(1L, "John Doe"));

        mockMvc.perform(get("/users/1"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name").value("John Doe"));
    }
}
```

Key Concepts Used

- **MockMvc**: Used to test REST controllers.
- **jsonPath()**: Verifies JSON response fields.
- **Mocking service layer** to isolate controller tests.

---

4. Integration Testing with Spring Boot

What is Integration Testing?

- Ensures that **multiple components work together** (e.g., controller, service, repository).
- Uses **@SpringBootTest** to load the full application context.
- Uses **an in-memory database (H2)** for real database interaction.

Example: Integration Test (UserIntegrationTest.java)

```
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;

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


import org.junit.jupiter.api.Test;

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

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

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.web.servlet.MockMvc;


@SpringBootTest

@AutoConfigureMockMvc

public class UserIntegrationTest {


  @Autowired

  private MockMvc mockMvc;


  @Test

  public void testGetAllUsers() throws Exception {

    mockMvc.perform(get("/users"))

        .andExpect(status().isOk());

  }

}
```

Key Concepts Used

- **@SpringBootTest**: Loads the entire Spring context.
- **@AutoConfigureMockMvc**: Configures MockMvc for integration tests.
- **Uses a real database** for actual persistence testing.

Conclusion

Test Type	What It Tests	Tools Used
Unit Test (Service Layer)	Tests business logic	Mockito (@Mock, @InjectMocks)
Controller Test	Tests REST API endpoints	MockMvc (jsonPath(), perform())
Integration Test	Tests the whole Spring Boot app	@SpringBootTest, @AutoConfigureMockMvc

Would you like to explore **mocking static methods** or **testing databases with Testcontainers** next? 🚀

Mocking External Dependencies in Java with Mockito

When testing applications, we often encounter **external dependencies** such as:

- **Databases (JPA, Hibernate, JDBC)**
- **External REST APIs (WebClient, RestTemplate)**
- **File I/O and network calls**

Mocking these dependencies ensures that our **unit tests** remain **isolated**, **fast**, and **reliable** without relying on actual external resources.

---

1. Mocking Databases and Repositories

Scenario: Mocking a JPA Repository

Spring applications commonly use **Spring Data JPA** repositories. Instead of using a real database, we mock the repository using **Mockito**.

Example: Service Layer with a JPA Repository

User Entity (User.java)

```
import jakarta.persistence.*;

@Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;

  // Constructors, Getters, and Setters
}
```

Repository (UserRepository.java)

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

Service Class (UserService.java)

```
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class UserService {
  private final UserRepository userRepository;

  public UserService(UserRepository userRepository) {
    this.userRepository = userRepository;
```

```
    }

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }
}
```

Testing Service with Mocked Repository

UserServiceTest.java

```
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import java.util.Arrays;
import java.util.List;

public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;
```

```
    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testGetAllUsers() {
        // Mock database response
        when(userRepository.findAll()).thenReturn(Arrays.asList(new User(1L, "Alice"), new User(2L, "Bob")));

        // Call method
        List<User> users = userService.getAllUsers();

        // Verify results
        assertEquals(2, users.size());
        verify(userRepository, times(1)).findAll();
    }
}
```

Key Takeaways

✓ **Mocking repositories** prevents actual database access.

✓ **Mockito's when().thenReturn()** simulates database queries.

✓ **verify() ensures interactions with the repository**.

---

2. Mocking External REST APIs

Applications often **consume REST APIs** using Spring's RestTemplate or WebClient. Instead of calling **real APIs**, we mock them.

Scenario: Calling a Third-Party API for Weather Data

WeatherService.java

```
import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;


@Service

public class WeatherService {

  private final RestTemplate restTemplate;


  public WeatherService(RestTemplate  restTemplate) {

    this.restTemplate = restTemplate;

  }


  public String getWeather(String city) {

    String url = "<https://api.weather.com/v3/weather/>"  + city;

    return restTemplate.getForObject(url,  String.class);

  }

}
```

Mocking RestTemplate in Unit Test

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

import static org.mockito.Mockito.*;


import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;
```

```
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.springframework.web.client.RestTemplate;

public class WeatherServiceTest {

  @Mock
  private RestTemplate restTemplate;

  @InjectMocks
  private WeatherService weatherService;

  @BeforeEach
  void setUp() {
    MockitoAnnotations.openMocks(this);
  }

  @Test
  public void testGetWeather() {
    // Mock API response
    when(restTemplate.getForObject("<https://api.weather.com/v3/weather/London>",
String.class))
        .thenReturn("Sunny");

    // Call method
    String weather = weatherService.getWeather("London");

    // Verify response
```

```
        assertEquals("Sunny", weather);
        verify(restTemplate, times(1)).getForObject(anyString(), eq(String.class));
    }
}
```

Key Takeaways

✓ **Mocking RestTemplate prevents real API calls**.

✓ **when().thenReturn() stubs API responses**.

✓ **verify() ensures API methods are invoked correctly**.

---

3. Mocking File I/O and Network Interactions

Scenario: Reading from a File

FileService.java

```
import org.springframework.stereotype.Service;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

@Service
public class FileService {
    public List<String> readFile(String path) throws IOException {
        return Files.readAllLines(Paths.get(path));
    }
}
```

Mocking File Operations

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

```
import static org.mockito.Mockito.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.List;

public class FileServiceTest {

    @InjectMocks
    private FileService fileService;

    @Mock
    private Files files;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testReadFile() throws IOException {
```

```
        // Mock file reading
        when(Files.readAllLines(Paths.get("test.txt"))).thenReturn(Arrays.asList("Line 1", "Line 2"));

        // Call method
        List<String> lines = fileService.readFile("test.txt");

        // Verify output
        assertEquals(2, lines.size());
    }
}
```

Key Takeaways

✓ Mocking Files.readAllLines() avoids actual file system access.

✓ Prevents file-not-found errors in tests.

---

4. Strategies for Testing Code with External Dependencies

Strategy	Description	Example
Mockito	Replaces real dependencies with mocks.	Mocking RestTemplate, JPA Repositories.
Testcontainers	Runs real databases in Docker for integration tests.	Testing PostgreSQL, MySQL.
WireMock	Mocks REST APIs using HTTP stubs.	Simulating third-party APIs.
In-Memory Databases	Uses H2, HSQLDB, or embedded databases instead of real ones.	Testing repository queries.

Conclusion