**Module: Mockito Testing with Java**

**Objective**

This document demonstrates the use of Mockito for unit testing in Java, focusing on mocking, stubbing, and verifying interactions. The aim is to simulate dependencies such as external APIs and ensure test reliability by controlling the test environment using mock objects. By decoupling external systems during testing, we improve test precision and isolation, enabling robust development practices.

**Exercise 1: Mocking and Stubbing**

In real-world applications, services often rely on external APIs for data. However, during unit testing, calling live APIs is not practical due to latency, unpredictability, or unavailability. Mockito offers an elegant solution to this by allowing developers to create mock objects and stub method behavior with predictable outputs. This exercise focuses on simulating an external API using Mockito.

In this implementation, the ExternalApi class is assumed to have a method getData() that retrieves data from an external source. The goal was to test a MyService class which uses ExternalApi internally, without making actual API calls. Mockito was used to mock the API, and the getData() method was stubbed to return a predefined value.

**Code Implementation**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

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

public class MyServiceTest {

@Test

public void testExternalApi() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

when(mockApi.getData()).thenReturn("Mock Data");

MyService service = new MyService(mockApi);

String result = service.fetchData();

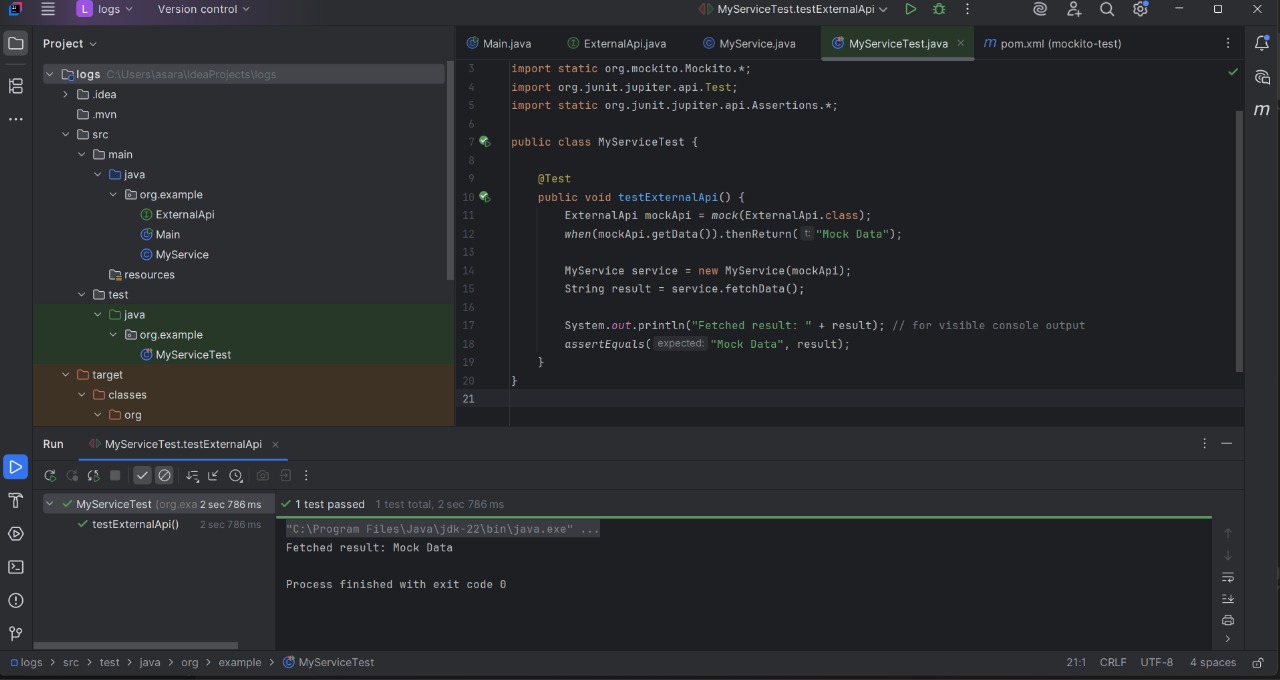
assertEquals("Mock Data", result);

}

}

This test verifies that the service correctly fetches and returns the mocked data. The external dependency was effectively replaced by a controlled simulation. As a result, the test becomes faster, independent, and stable.

**Output:**



**Exercise 2: Verifying Interactions**

Apart from controlling the behavior of dependencies, it is equally important to verify how those dependencies are used. This exercise focuses on verifying whether specific methods were invoked, and whether they were called with the correct parameters.

In this scenario, the same ExternalApi was mocked, and the interaction between MyService and the mock object was tracked using Mockito’s verify() feature. The test ensures that the getData() method was indeed called when the service logic was executed.

**Code Implementation**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.fetchData();

verify(mockApi).getData();

}

}

This test does not focus on return values, but on method invocation. If the method getData() was not called, the test would fail, clearly indicating a flaw in the logic or behavior of the MyService class.

**Output:**

