**JUnit Testing Documentation Report**

**Exercise 1: Setting Up JUnit**

**✅ Scenario:**

Set up JUnit in a Java project using Maven on Replit.

**✅ Steps:**

1. Create a Maven-based Java project on Replit.
2. Add JUnit as a dependency in pom.xml.
3. Create a Java class with basic methods.
4. Write JUnit test cases for the class.
5. Run the tests using Maven.

**✅ Code:**

**pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>JUnitExample</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>17</source>

<target>17</target>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

</project>

**Calculator.java (src/main/java/Calculator.java)**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Cannot divide by zero.");

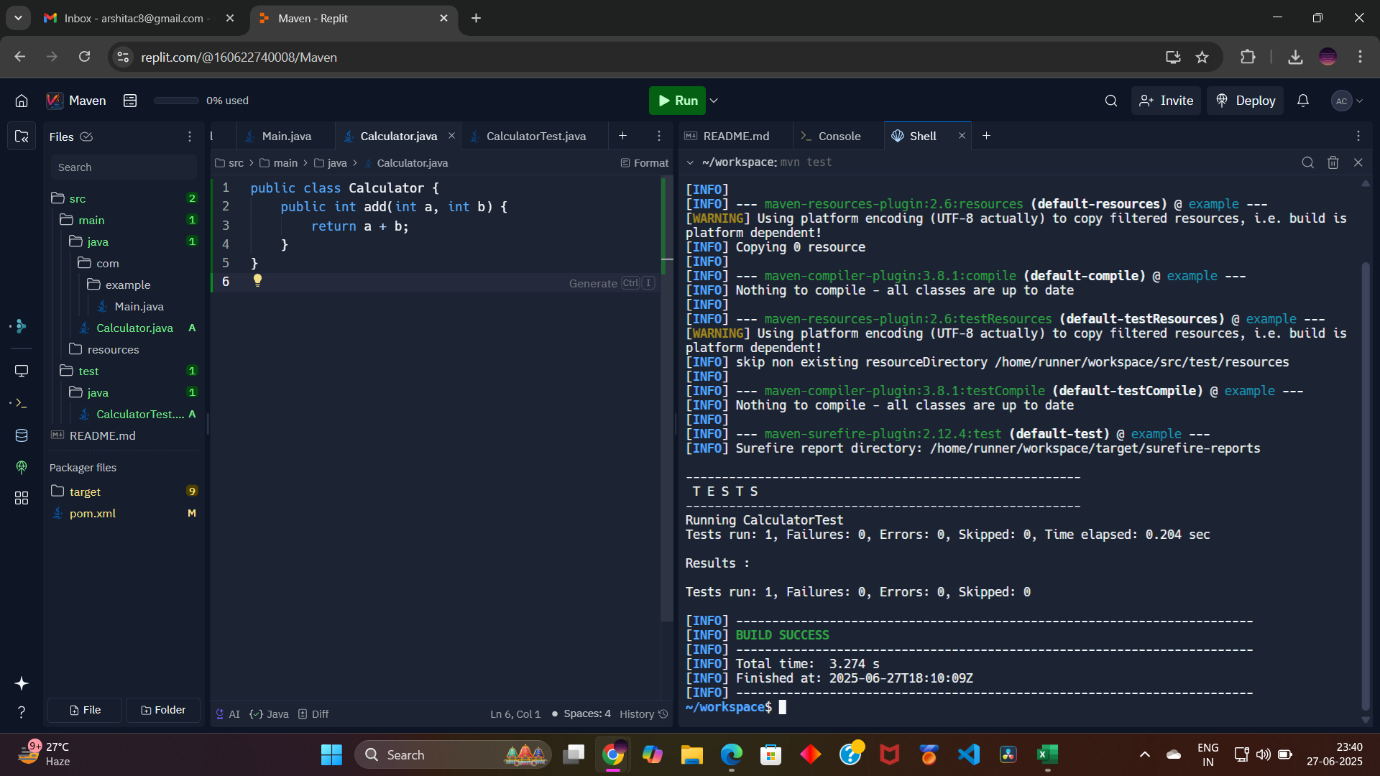
}

return a / b;

}

}

**Output Screenshot for Exercise 1:**

****

**Exercise 2: Writing Basic JUnit Tests**

**✅ Scenario:**

Write basic JUnit tests for the methods in the Calculator class.

**✅ Steps:**

1. Create a test class CalculatorTest.java in src/test/java.
2. Write individual tests for addition, subtraction, multiplication, and division.
3. Include a test case to handle division by zero.

**✅ Code:**

**CalculatorTest.java (src/test/java/CalculatorTest.java)**

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

Calculator calc = new Calculator();

@Test

public void testAddition() {

assertEquals(5, calc.add(2, 3));

}

@Test

public void testSubtraction() {

assertEquals(4, calc.subtract(7, 3));

}

@Test

public void testMultiplication() {

assertEquals(15, calc.multiply(3, 5));

}

@Test

public void testDivision() {

assertEquals(5, calc.divide(10, 2));

}

@Test(expected = IllegalArgumentException.class)

public void testDivideByZero() {

calc.divide(10, 0);

}

}

**Exercise 3: Assertions in JUnit**

**✅ Scenario:**

Use different types of JUnit assertions to validate various conditions.

**✅ Steps:**

1. Create a new test class AssertionsTest.java in src/test/java.
2. Write tests using the following assertions:
   * assertEquals
   * assertTrue
   * assertFalse
   * assertNull
   * assertNotNull

**✅ Code:**

**AssertionsTest.java (src/test/java/AssertionsTest.java)**

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

// Assert equals

assertEquals(5, 2 + 3);

// Assert true

assertTrue(5 > 3);

// Assert false

assertFalse(5 < 3);

// Assert null

Object obj = null;

assertNull(obj);

// Assert not null

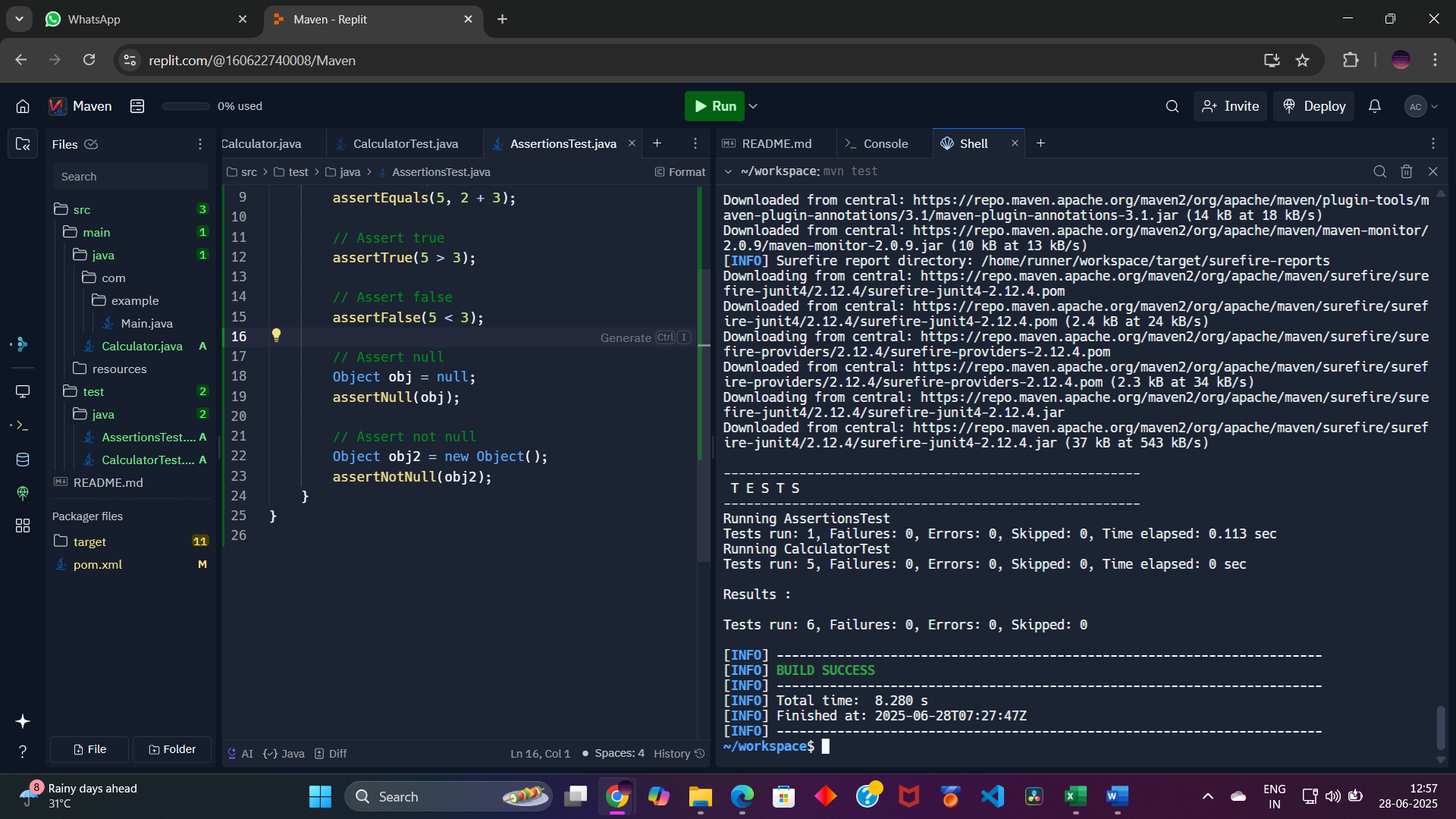
Object obj2 = new Object();

assertNotNull(obj2);

}

}

**📸 Output Screenshot for Exercise 3:**



**✅ Project Folder Structure:**

JUnitExample/

├── src/

│ ├── main/

│ │ └── java/

│ │ └── Calculator.java

│ └── test/

│ └── java/

│ ├── CalculatorTest.java

│ └── AssertionsTest.java

└── pom.xml

**Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and Teardown Methods in JUnit Scenario: You need to organize your tests using the Arrange-Act-Assert (AAA) pattern and use setup and teardown methods. Steps: 1. Write tests using the AAA pattern. 2. Use @Before and @After annotations for setup and teardown methods.**

**✅ Scenario:**

Organize JUnit tests using the **Arrange-Act-Assert (AAA) pattern** and implement setup and teardown methods using JUnit annotations @Before and @After.

**✅ Steps:**

1. Apply the **Arrange-Act-Assert (AAA)** pattern to write clear and maintainable tests.
2. Use the @Before annotation to set up common objects (**Test Fixtures**) before each test.
3. Use the @After annotation to perform cleanup actions after each test.

**✅ Code:**

**🔹 Calculator.java (src/main/java/Calculator.java)**

java

CopyEdit

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Cannot divide by zero.");

}

return a / b;

}

}

**🔹 CalculatorTest.java (src/test/java/CalculatorTest.java)**

java

CopyEdit

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calc; // Test Fixture

// Setup method runs before each test

@Before

public void setUp() {

calc = new Calculator();

System.out.println("Setup - Calculator instance created.");

}

// Teardown method runs after each test

@After

public void tearDown() {

calc = null;

System.out.println("Teardown - Calculator instance destroyed.");

}

@Test

public void testAddition() {

// Arrange

int a = 10;

int b = 20;

// Act

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

// Assert

assertEquals(30, result);

}

@Test

public void testSubtraction() {

// Arrange

int a = 50;

int b = 30;

// Act

int result = calc.subtract(a, b);

// Assert

assertEquals(20, result);

}

@Test

public void testMultiplication() {

// Arrange

int a = 5;

int b = 4;

// Act

int result = calc.multiply(a, b);

// Assert

assertEquals(20, result);

}

@Test

public void testDivision() {

// Arrange

int a = 100;

int b = 5;

// Act

int result = calc.divide(a, b);

// Assert

assertEquals(20, result);

}

@Test(expected = IllegalArgumentException.class)

public void testDivideByZero() {

// Arrange

int a = 10;

int b = 0;

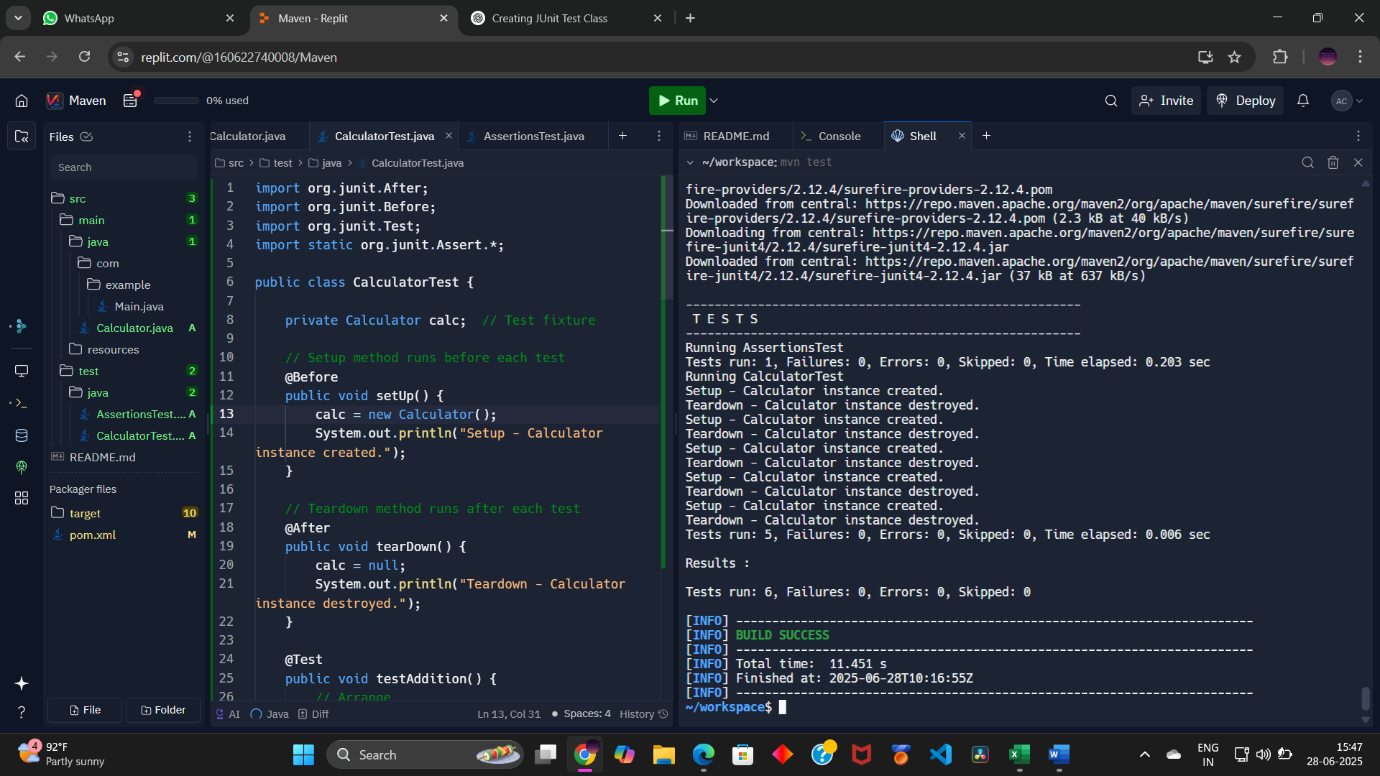
// Act

calc.divide(a, b);

// Assert is handled by expected exception

}

}



**📝 JUnit with Mockito**

**Exercise 1: Mocking and Stubbing**

**✅ Question / Scenario:**

A service depends on an external API. Test the service without relying on the actual API by using **Mockito** to create a **mock object** and **stub its methods** to return predefined values.

**✅ Steps:**

1. Create a **mock object** for the external API using Mockito.
2. **Stub the method** getData() to return "Mock Data".
3. Write a test case that uses the mock object and checks whether the service returns the stubbed data.

**✅ Code:**

**🔹 ExternalApi.java (src/main/java/ExternalApi.java)**

public interface ExternalApi {

String getData();

}

**🔹 MyService.java (src/main/java/MyService.java)**

public class MyService {

private ExternalApi externalApi;

public MyService(ExternalApi externalApi) {

this.externalApi = externalApi;

}

public String fetchData() {

return externalApi.getData();

}

}

**🔹 MyServiceTest.java (src/test/java/MyServiceTest.java)**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testExternalApi() {

// Create a mock object for ExternalApi

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

// Stub the getData() method to return "Mock Data"

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

// Inject mock into MyService

MyService service = new MyService(mockApi);

// Call the method under test

String result = service.fetchData();

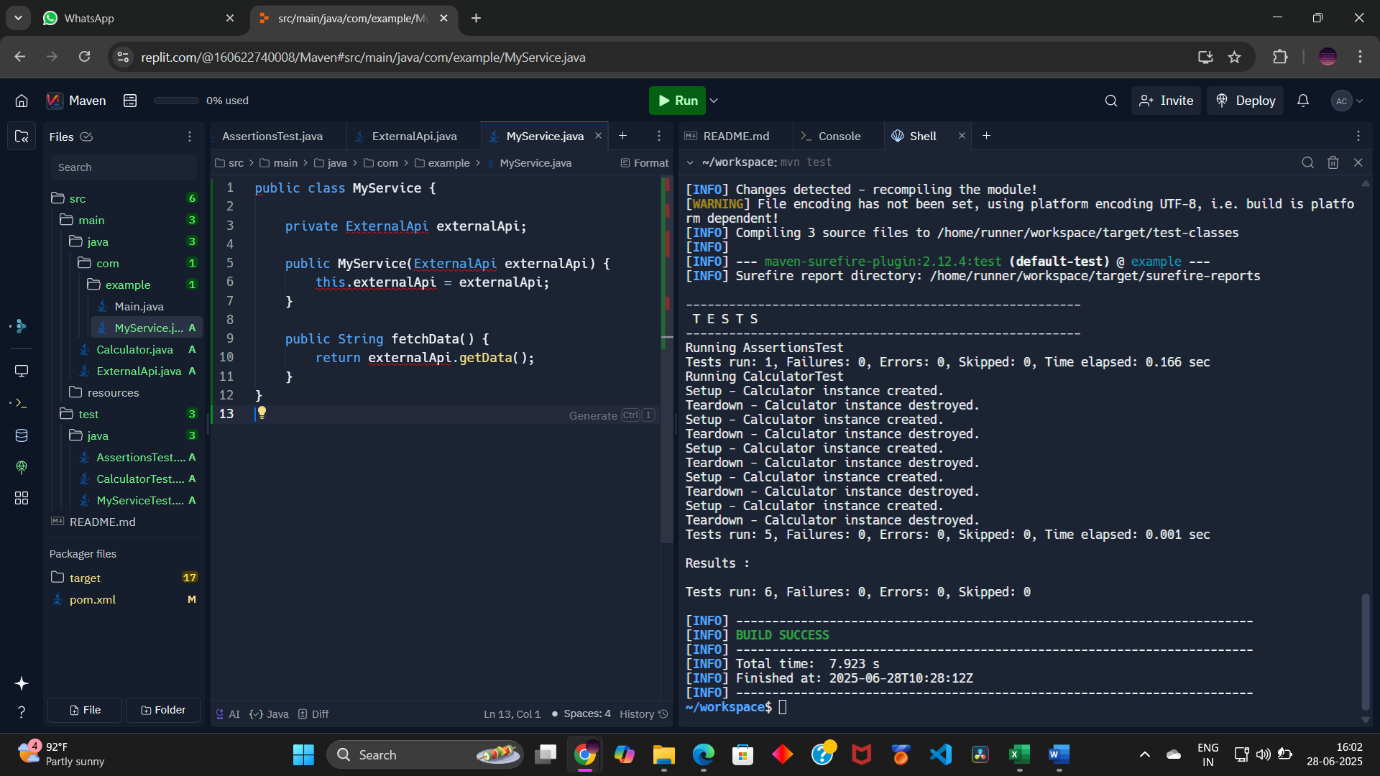
// Assert the result

assertEquals("Mock Data", result);

}

}

**📸 Output Screenshot for Exercise 1:**

****

**✅ Explanation:**

* The ExternalApi interface is mocked.
* Its method getData() is stubbed to return “Mock Data”.
* The test verifies that fetchData() from MyService returns this mocked value correctly.

**Exercise 2: Verifying Interactions**

**✅ Question / Scenario:**

Ensure that a method (getData()) from the mocked external API is actually called during the execution of the service method. This validates the interaction between the service and the API.

**✅ Steps:**

1. Create a **mock object** for the external API.
2. Call the service method fetchData().
3. Use Mockito’s verify() function to check whether the getData() method was called.

**✅ Code:**

**🔹 MyServiceTest.java (Continued in the same file)**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

// Create a mock object for ExternalApi

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

// Inject mock into MyService

MyService service = new MyService(mockApi);

// Call the method under test

service.fetchData();

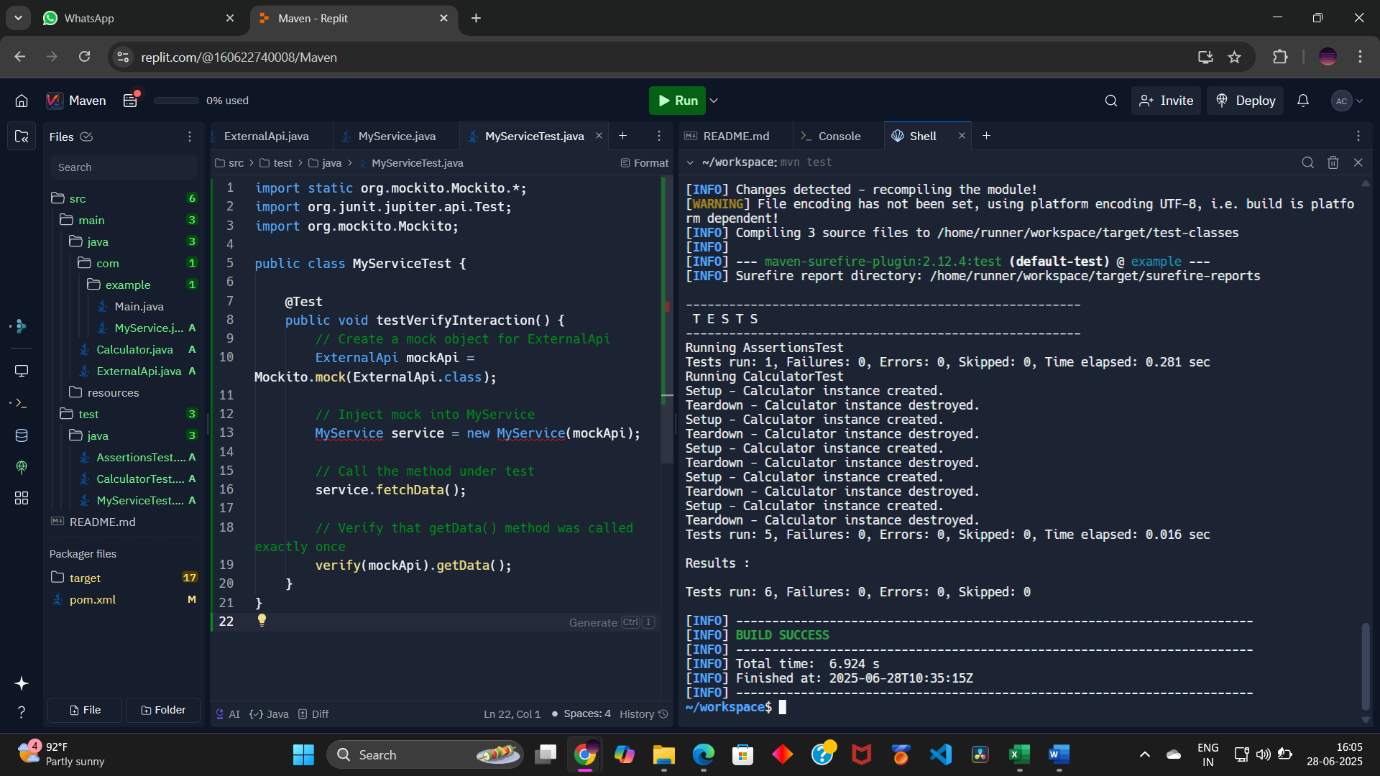
// Verify that getData() method was called exactly once

verify(mockApi).getData();

}

}

**📸 Output Screenshot for Exercise 2:**

****

**✅ Explanation:**

* A mock of ExternalApi is created.
* The service calls fetchData().
* Mockito’s verify() checks whether getData() was invoked.
* The test passes if the method was called; otherwise, it fails.

**✅ Project Structure:**

MockitoExample/

├── src/

│ ├── main/

│ │ └── java/

│ │ ├── ExternalApi.java

│ │ └── MyService.java

│ └── test/

│ └── java/

│ └── MyServiceTest.java

└── pom.xml

**✅ pom.xml Dependencies (Summary):**

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.9.3</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.7.0</version>

<scope>test</scope>

</dependency>

</dependencies>

**Logging in Java using SLF4J – Documentation**

**✅ Exercise 1: Logging Error Messages and Warning Levels**

**Question / Scenario:**

Create a Java application that demonstrates **logging error messages and warning levels** using **SLF4J with Logback**.

**Steps:**

1. Add **SLF4J** and **Logback** dependencies to pom.xml.
2. Create a Java class that uses SLF4J for logging error and warning messages.
3. Run the program and observe logs printed in the **Console**.

**Code:**

🔸 **pom.xml Dependencies:**

<dependencies>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

</dependencies>

🔸 **LoggingExample.java** (src/main/java/LoggingExample.java)

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingExample {

private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);

public static void main(String[] args) {

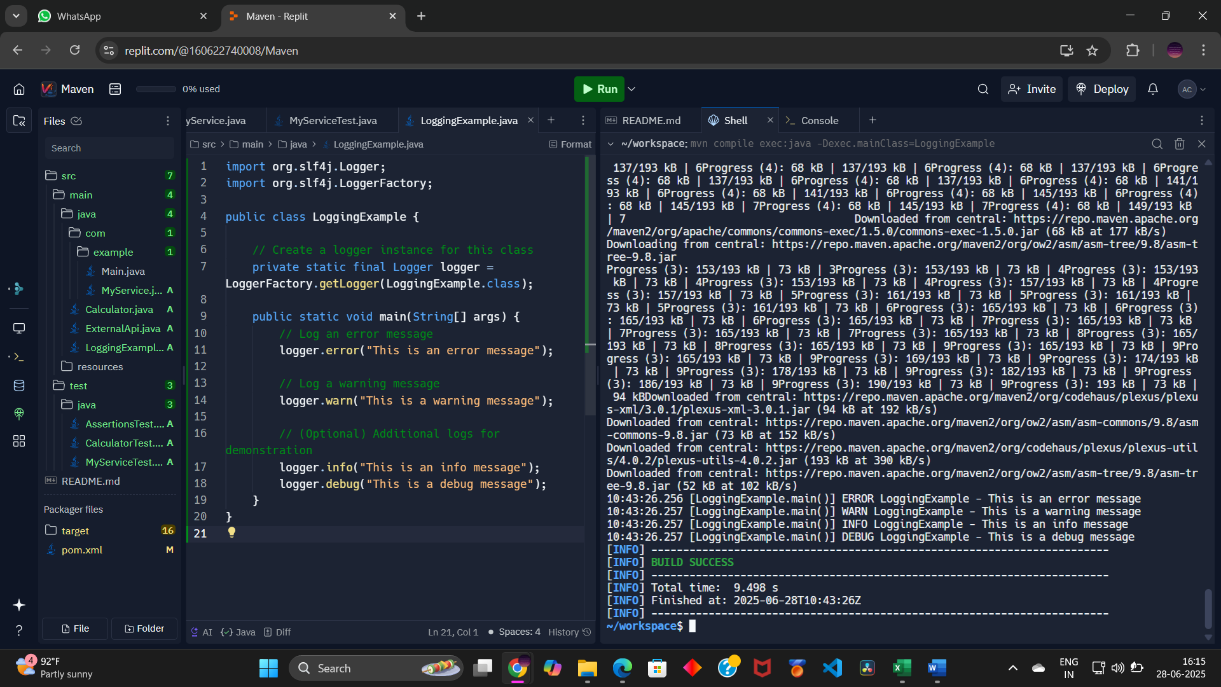
logger.error("This is an error message");

logger.warn("This is a warning message");

}

}

**📸 Output Screenshot Placeholder:**

**