**Week-2**

**Mockito**

**Exercise 1: Mocking and Stubbing**

**Scenario:** You need to test a service that depends on an external API. Use Mockito to mock the external API and stub its methods.

When a class depends on an external component (e.g., API, DB), it's not ideal to test it with real connections.  
Instead, we use Mockito to:

* Create mock objects of the external components
* Stub methods to return predefined (fake) values
* Verify behaviors without calling real systems

This keeps tests fast, independent, and reliable.

**Steps:**

**1.Create a Maven Project in Eclipse**  
Go to File → New → Project → Maven Project  
Choose maven-archetype-quickstart → Fill in GroupId (e.g., com.example) and ArtifactId (e.g., mockito-demo) → Finish.

**2.Add Dependencies in pom.xml**Add these inside <dependencies> tag.

**3.Create Java Interface and Class**

* ExternalApi.java (interface with getData() method)
* MyService.java (uses ExternalApi and has fetchData() method)

**4.Create JUnit Test Class under src/test/java**

* Name it MyServiceTest.java
* Use Mockito.mock() to mock the API and when(...).then Return(...) to stub the return value.

**5.Run the Test**  
Right-click on MyServiceTest.java → Run As → JUnit Test. Test will pass and return "Mock Data" without using the real API.

**ExternalApi.java**

//ExternalApi.java

package com.example.mockito\_demo;

public interface ExternalApi {

String getData();

}

**MyService.java**

//MyService.java

package com.example.mockito\_demo;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData(); // Relies on external API

}

}

**MyServiceTest.java**

//MyServiceTest.java

package com.example.mockito\_demo;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

public class MyServiceTest {

*@Test*

public void testExternalApi() {

// Step 1: Mock the ExternalApi

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

// Step 2: Stub getData() to return fake value

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

// Step 3: Use the mocked API in MyService

MyService service = new MyService(mockApi);

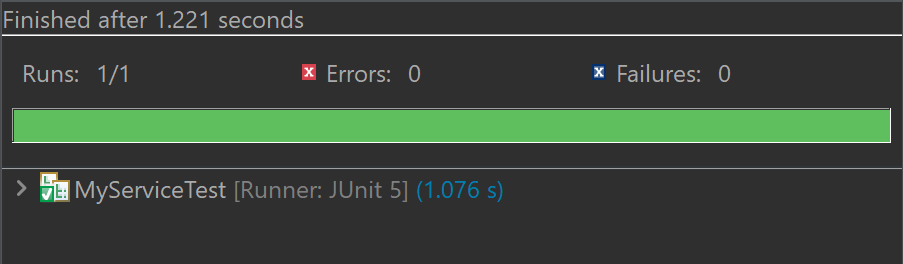
String result = service.fetchData();

// Step 4: Assert the expected result

*assertEquals*("Mock Data", result);

}

}



**Exercise 2: Verifying Interactions**

**Scenario:** You need to ensure that a method is called with specific arguments.

In unit testing, it’s not enough to just check what the output is — sometimes, you also want to ensure how your code interacts with its dependencies.

Mockito allows you to verify if specific methods were called, and even check if they were called with the correct arguments. This is useful when your method doesn’t return a value but performs important actions (like saving to a database or sending a notification).

**Steps:**

**1.Create a Mock Object**  
Use mock() to create a fake version of the dependency (e.g., ExternalApi).

**2.Inject the Mock into Your Class**  
Pass the mock object to your service or class under test (e.g., MyService).

**3.Call the Method to Be Tested**  
Invoke the method that is expected to interact with the mock (e.g., fetchData()).

**4.Verify the Interaction**  
Use verify(mock).methodName() to check that the mock method was actually called.

**ExternalApi.java**

//ExternalApi.java

package com.example.mockito\_demo;

public interface ExternalApi {

String getData();

}

**MyService.java**

//MyService.java

package com.example.mockito\_demo;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData(); // Relies on external API

}

}

**MyServiceTest.java**

//MyServiceTest.java

package com.example.mockito\_demo;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

// Step 1: Create a mock

ExternalApi mockApi = mock(ExternalApi.class);

// Step 2: Inject mock and call the method

MyService service = new MyService(mockApi);

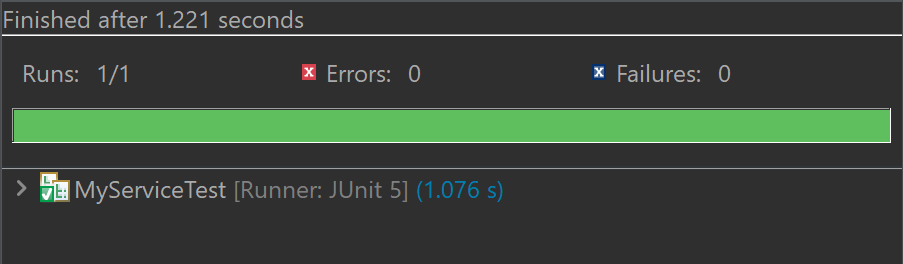
service.fetchData();

// Step 3: Verify that getData() was called

verify(mockApi).getData();

}

}



**Conclusion:**

With Mockito, we also saw how to mock a dependency and then stub method behaviors to create a fine-grained test point.

We also practiced testing interactions: to guarantee that methods are being called correctly during execution.

These methodologies assist in generating robust, responsive, and self-contained unit tests without using real external systems.

**-- THE END --**