**MOCKITO HANDS-ON EXERCISES**

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.

Steps:

1. Create a mock object for the external API.

2. Stub the methods to return predefined values.

3. Write a test case that uses the mock object.

ExternalApi.java

package com.example;

public interface ExternalApi {

    String getData();

}

MyService.java

package com.example;

public class MyService {

    private final ExternalApi api;

    public MyService(ExternalApi api) {

        this.api = api;

    }

    public String fetchData() {

        return api.getData();

    }

}

MyServiceTest.java

package com.example;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

class MyServiceTest {

    @Test

    void testExternalApi() {

        // Arrange — create a mock of ExternalApi

        ExternalApi mockApi = mock(ExternalApi.class);

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

        // Act — call the service with the mocked dependency

        MyService service = new MyService(mockApi);

        String result = service.fetchData();

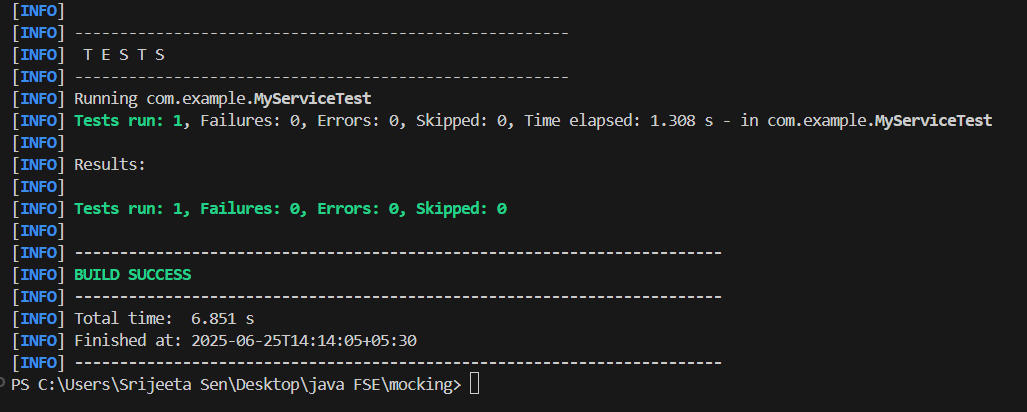
        // Assert — verify behavior & result

        assertEquals("Mock Data", result);

        verify(mockApi).getData();  // optional: verify method was called

    }

}



Exercise 2: Verifying Interactions

Scenario:

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

Steps:

1. Create a mock object.

2. Call the method with specific arguments.

3. Verify the interaction.

ExternalApi.java

package com.example;

public interface ExternalApi {

    String getData();

    void sendData(String data); // <- another method to verify arguments

}

MyService.java

package com.example;

public class MyService {

    private final ExternalApi api;

    public MyService(ExternalApi api) {

        this.api = api;

    }

    public String fetchData() {

        return api.getData();

    }

    public void processAndSendData(String data) {

        api.sendData(data); // this is what we'll verify

    }

}

MyServiceTest.java

package com.example;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

import static org.mockito.Mockito.verify;

class MyServiceTest {

    @Test

    void testVerifyInteraction() {

        // Arrange

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

        MyService service = new MyService(mockApi);

        // Act

        service.fetchData();

        service.processAndSendData("TestValue");

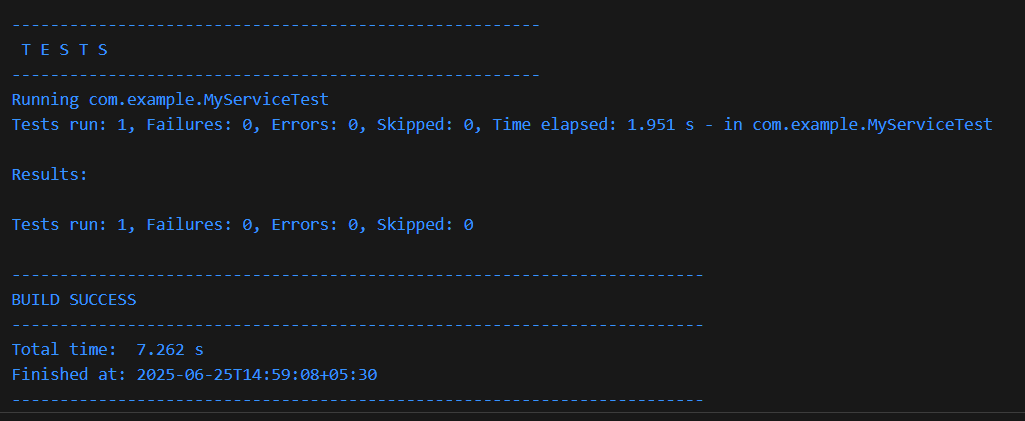
        // Assert (verify interactions)

        verify(mockApi).getData();

        verify(mockApi).sendData("TestValue"); // verify specific argument

    }

}



Exercise 3: Argument Matching

Scenario:

You need to verify that a method is called with specific arguments.

Steps:

1. Create a mock object.

2. Call the method with specific arguments.

3. Use argument matchers to verify the interaction.

ExternalApi.java

package com.example;

public interface ExternalApi {

    void sendData(String data, int priority);

}

MyService.java

package com.example;

public class MyService {

    private final ExternalApi api;

    public MyService(ExternalApi api) {

        this.api = api;

    }

    public void processAndSend(String data) {

        api.sendData(data, 5); // send with fixed priority

    }

}

MyServiceTest.java

package com.example;

import org.junit.jupiter.api.Test;

import static org.mockito.ArgumentMatchers.anyInt;

import static org.mockito.ArgumentMatchers.eq;

import org.mockito.Mockito;

import static org.mockito.Mockito.verify;

class MyServiceTest {

    @Test

    void testVerifyWithArgumentMatchers() {

        // Arrange

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

        MyService service = new MyService(mockApi);

        // Act

        service.processAndSend("Hello World");

        // Assert

        // Verify sendData called with specific string and any int

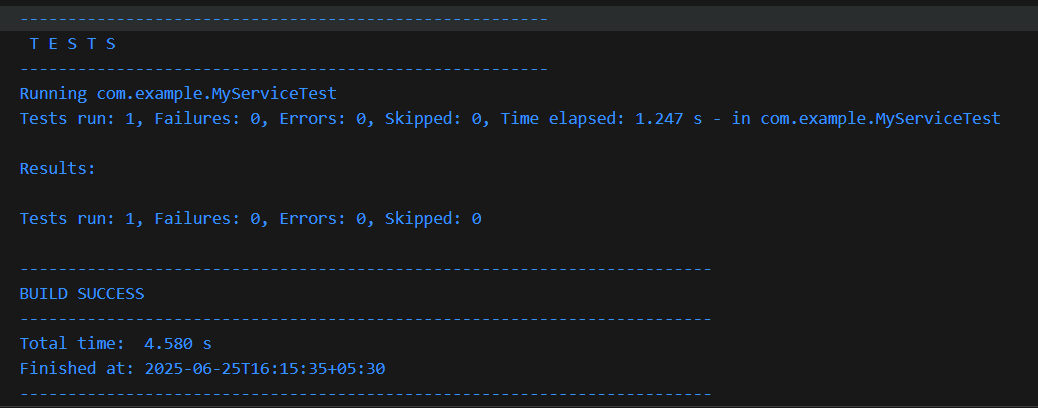
        verify(mockApi).sendData(eq("Hello World"), anyInt());

        // Or you could verify with an exact integer value too:

        verify(mockApi).sendData(eq("Hello World"), eq(5));

    }

}



Exercise 4: Handling Void Methods

Scenario:

You need to test a void method that performs some action.

Steps:

1. Create a mock object.

2. Stub the void method.

3. Verify the interaction.

ExternalApi.java

package com.example;

public interface ExternalApi {

void sendNotification(String message); // <- void method

}

MyService.java

package com.example;

public class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public void notifyUser() {

api.sendNotification("Welcome!");

}

}

MyServiceTest.java

package com.example;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

import static org.mockito.ArgumentMatchers.\*;

import static org.mockito.Mockito.\*;

class MyServiceTest {

@Test

void testVoidMethod() {

// Arrange

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

MyService service = new MyService(mockApi);

// Stub the void method if needed

doNothing().when(mockApi).sendNotification(anyString());

// Act

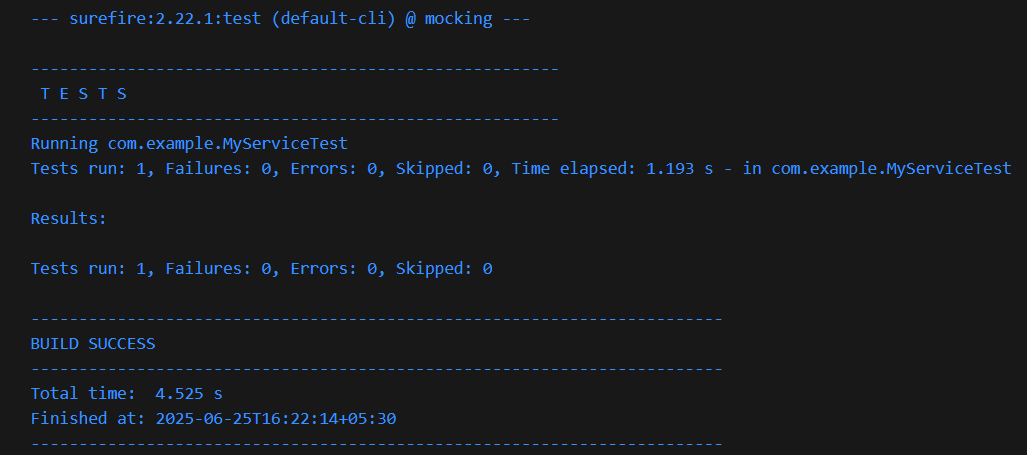
service.notifyUser();

// Assert

verify(mockApi).sendNotification("Welcome!"); // verify the call

}

}



Exercise 5: Mocking and Stubbing with Multiple Returns

Scenario:

You need to test a service that depends on an external API with multiple return values.

Steps:

1. Create a mock object for the external API.

2. Stub the methods to return different values on consecutive calls.

3. Write a test case that uses the mock object.

ExternalApi.java

package com.example;

public interface ExternalApi {

String getStatus();

}

MyService.java

package com.example;

public class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String checkStatus() {

return api.getStatus();

}

}

MyServiceTest.java

package com.example;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

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

import static org.mockito.Mockito.\*;

class MyServiceTest {

@Test

void testMultipleReturns() {

// Arrange

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

MyService service = new MyService(mockApi);

// Stubbing with multiple return values:

when(mockApi.getStatus())

.thenReturn("OK") // 1st call

.thenReturn("WARN") // 2nd call

.thenReturn("ERROR"); // 3rd call

// Act & Assert

assertEquals("OK", service.checkStatus());

assertEquals("WARN", service.checkStatus());

assertEquals("ERROR", service.checkStatus());

// Verify the method was called 3 times

verify(mockApi, times(3)).getStatus();

}

}



Exercise 6: Verifying Interaction Order

Scenario:

You need to ensure that methods are called in a specific order.

Steps:

1. Create a mock object.

2. Call the methods in a specific order.

3. Verify the interaction order.

UserRepository.java

package com.example;

public interface UserRepository {

    void connect();

    void saveUser(String name);

    void disconnect();

}

UserService.java

package com.example;

public class UserService {

    private final UserRepository repository;

    public UserService(UserRepository repository) {

        this.repository = repository;

    }

    public void registerUser(String name) {

        repository.connect();

        repository.saveUser(name);

        repository.disconnect();

    }

}

UserServiceTest.java

package com.example;

import org.junit.jupiter.api.Test;

import org.mockito.InOrder;

import static org.mockito.Mockito.inOrder;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.verifyNoMoreInteractions;

class UserServiceTest {

    @Test

    void testVerifyInteractionOrder() {

        // Arrange

        UserRepository mockRepo = mock(UserRepository.class);

        UserService service = new UserService(mockRepo);

        // Act

        service.registerUser("Alice");

        // Assert — verify order of interactions

        InOrder inOrder = inOrder(mockRepo);

        inOrder.verify(mockRepo).connect();

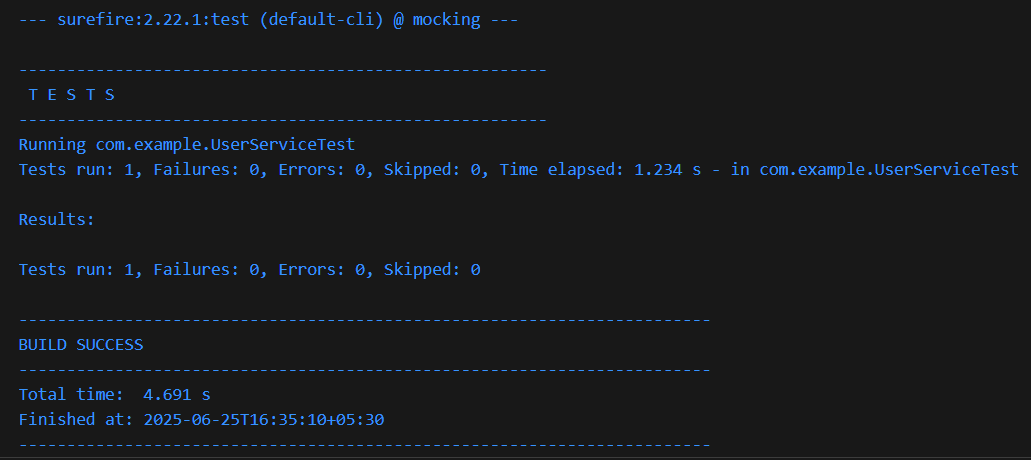
        inOrder.verify(mockRepo).saveUser("Alice");

        inOrder.verify(mockRepo).disconnect();

        verifyNoMoreInteractions(mockRepo);

    }

}



Exercise 7: Handling Void Methods with Exceptions

Scenario:

You need to test a void method that throws an exception.

Steps:

1. Create a mock object.

2. Stub the void method to throw an exception.

3. Verify the interaction.

NotificationService.java

package com.example;

public interface NotificationService {

    void sendNotification(String message);

}

UserProcessor.java

package com.example;

public class UserProcessor {

    private final NotificationService notificationService;

    public UserProcessor(NotificationService notificationService) {

        this.notificationService = notificationService;

    }

    public void processUser(String username) {

        // Some processing

        notificationService.sendNotification("Welcome " + username);

    }

}

UserProcessorTest.java

package com.example;

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

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

import org.junit.jupiter.api.Test;

import static org.mockito.ArgumentMatchers.anyString;

import static org.mockito.Mockito.doThrow;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.times;

import static org.mockito.Mockito.verify;

class UserProcessorTest {

    @Test

    void testVoidMethodThrowsException() {

        // Arrange

        NotificationService mockService = mock(NotificationService.class);

        UserProcessor processor = new UserProcessor(mockService);

        // Stub the void method to throw exception

        doThrow(new RuntimeException("Service failed"))

                .when(mockService).sendNotification(anyString());

        // Act & Assert — verify the exception is thrown

        RuntimeException thrown = assertThrows(

                RuntimeException.class,

                () -> processor.processUser("Alice"),

                "Expected exception not thrown"

        );

        assertEquals("Service failed", thrown.getMessage());

        verify(mockService, times(1)).sendNotification("Welcome Alice");

    }

}

