**Advanced Mockito Hands-On Exercises (Everything under this section is Additional)**

# **Exercise 1: Mocking Databases and Repositories (Additional Exercise)**

**You need to test a service that interacts with a database repository.**

**Steps:**

1. **Create a mock repository using Mockito.**
2. **Stub the repository methods to return predefined data.**
3. **Write a test to verify the service logic using the mocked repository.**

**Solution Code:**

**ServiceTest.java (Given Solution Code)**

package E1\_MockingDatabasesAndRepositories;

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

public class ServiceTest {

*@Test*

public void testServiceWithMockRepository() {

Repository mockRepository = *mock*(Repository.class);

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

Service service = new Service(mockRepository);

String result = service.processData();

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

}

}

**Repository.java**

package E1\_MockingDatabasesAndRepositories;

public interface Repository {

String getData();

}

**Service.java**

package E1\_MockingDatabasesAndRepositories;

public class Service {

private Repository repository;

public Service(Repository repository) {

this.repository = repository;

}

public String processData() {

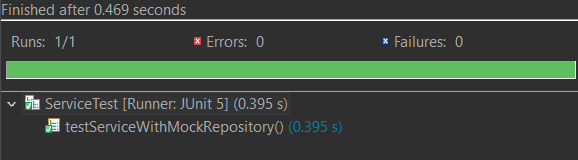
String data = repository.getData();

return "Processed " + data;

}

}

**Output:**



# **Exercise 2: Mocking External Services (RESTful APIs) (Additional Exercise)**

**You need to test a service that calls an external RESTful API.**

**Steps:**

1. **Create a mock REST client using Mockito.**
2. **Stub the REST client methods to return predefined responses.**
3. **Write a test to verify the service logic using the mocked REST client.**

**ApiServiceTest.java (Given Solution Code)**

package E2\_MockingExternalServices;

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

public class ApiServiceTest {

*@Test*

public void testServiceWithMockRestClient() {

RestClient mockRestClient = *mock*(RestClient.class);

*when*(mockRestClient.getResponse()).thenReturn("Mock Response");

ApiService apiService = new ApiService(mockRestClient);

String result = apiService.fetchData();

*assertEquals*("Fetched Mock Response", result);

}

}

**ApiService.java**

public class ApiService {

private RestClient restClient;

public ApiService(RestClient restClient) {

this.restClient = restClient;

}

public String fetchData() {

return "Fetched " + restClient.getResponse();

}

}

**RestClient.java**

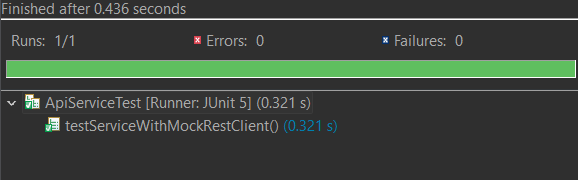
package E2\_MockingExternalServices;

public interface RestClient {

String getResponse();

}

**Output:**

****

# **Exercise 3: Mocking File I/O (Additional Exercise)**

**You need to test a service that reads from and writes to files.**

**Steps:**

1. **Create a mock file reader and writer using Mockito.**
2. **Stub the file reader and writer methods to simulate file operations.**
3. **Write a test to verify the service logic using the mocked file reader and writer.**

**FileServiceTest.java**

package E3\_MockingFileIO;

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

public class FileServiceTest {

*@Test*

public void testServiceWithMockFileIO() {

FileReader mockFileReader = *mock*(FileReader.class);

FileWriter mockFileWriter = *mock*(FileWriter.class);

*when*(mockFileReader.read()).thenReturn("Mock File Content");

FileService fileService = new FileService(mockFileReader, mockFileWriter);

String result = fileService.processFile();

*assertEquals*("Processed Mock File Content", result);

*verify*(mockFileWriter).write("Processed Mock File Content");

}

}

**FileService.java**

package E3\_MockingFileIO;

public class FileService {

private FileReader fileReader;

private FileWriter fileWriter;

public FileService(FileReader fileReader, FileWriter fileWriter) {

this.fileReader = fileReader;

this.fileWriter = fileWriter;

}

public String processFile() {

String content = fileReader.read();

fileWriter.write("Processed " + content);

return "Processed " + content;

}

}

**FileReader.java**

package E3\_MockingFileIO;

public interface FileReader {

String read();

}

**FileWriter.java**

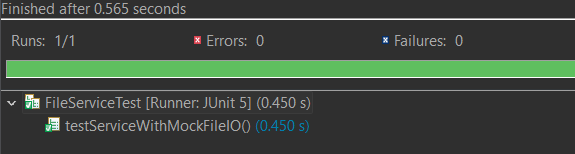
package E3\_MockingFileIO;

public interface FileWriter {

void write(String content);

}

**Output:**



# **Exercise 4: Mocking Network Interactions (Additional Exercise)**

**You need to test a service that interacts with network resources.**

**Steps:**

**1. Create a mock network client using Mockito.**

**2. Stub the network client methods to simulate network interactions.**

**3. Write a test to verify the service logic using the mocked network client.**

**NetworkServiceTest.java**

package E4\_MockingNetworkInteractions;

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

public class NetworkServiceTest {

@Test

public void testServiceWithMockNetworkClient() {

NetworkClient mockNetworkClient = mock(NetworkClient.class);

when(mockNetworkClient.connect()).thenReturn("Mock Connection");

NetworkService networkService = new NetworkService(mockNetworkClient);

String result = networkService.connectToServer();

assertEquals("Connected to Mock Connection", result);

}

}

**NetworkService.java**

package E4\_MockingNetworkInteractions;

public class NetworkService {

private NetworkClient networkClient;

public NetworkService(NetworkClient networkClient) {

this.networkClient = networkClient;

}

public String connectToServer() {

return "Connected to " + networkClient.connect();

}

}

**NetworkClient.java**

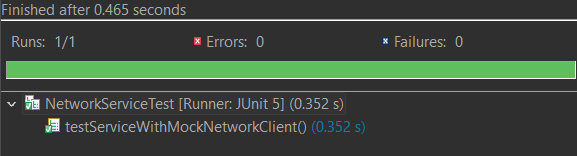
package E4\_MockingNetworkInteractions;

public interface NetworkClient {

String connect();

}

**Output:**



# **Exercise 5: Mocking Multiple Return Values (Additional Exercise)**

**You need to test a service that calls a method multiple times with different return values.**

**Steps:**

1. **Create a mock object using Mockito.**
2. **Stub the method to return different values on consecutive calls.**
3. **Write a test to verify the service logic using the mocked object.**

**MultiReturnServiceTest.java**

package E5\_MockingMultipleRunValues;

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

public class MultiReturnServiceTest {

@Test

public void testServiceWithMultipleReturnValues() {

Repository mockRepository = mock(Repository.class);

when(mockRepository.getData())

.thenReturn("First Mock Data")

.thenReturn("Second Mock Data");

Service service = new Service(mockRepository);

String firstResult = service.processData();

String secondResult = service.processData();

assertEquals("Processed First Mock Data", firstResult);

assertEquals("Processed Second Mock Data", secondResult);

}

}

**ServiceTest.java**

package E5\_MockingMultipleRunValues;

public class Service {

private Repository repository;

public Service(Repository repository) {

this.repository = repository;

}

public String processData() {

return "Processed " + repository.getData();

}

}

**Repository.java**

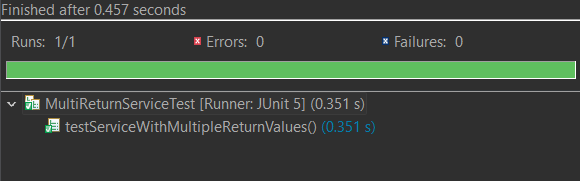
package E5\_MockingMultipleRunValues;

public interface Repository {

String getData();

}

**Output:**

****