Advanced Mockito Hands-On Exercises

# Exercise 1: Mocking Databases and Repositories

***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:

import static org.mockito.Mockito.\*; import org.junit.jupiter.api.Test;

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

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);

}

}

**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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>TDDMockExample</artifactId>

<version>1.0</version>

<name>TDDMockExample</name>

<dependencies>

<!-- JUnit 5 -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<!-- Maven Surefire Plugin for JUnit 5 -->

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<includes>

<include>\*\*/\*Test.java</include>

</includes>

</configuration>

</plugin>

</plugins>

</build>

</project>

**Repository.java**

package com.example.tdd;

public interface Repository {

String getData();

}

**Service.java**

package com.example.tdd;

public class Service {

private Repository repository;

public Service(Repository repository) {

this.repository = repository;

}

public String processData() {

String data = repository.getData();

return "Processed " + data;

}

}

# ServiceTest.java

package com.example.tdd;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class ServiceTest {

*@Test*

public void testServiceWithMockRepository() {

// Step 1: Create mock repository

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

// Step 2: Stub the getData() method

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

// Step 3: Inject mock into service

Service service = new Service(mockRepository);

// Step 4: Test the logic

String result = service.processData();

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

}

}

# Output:

# 

# Exercise 2: Mocking External Services (RESTful APIs)

***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.

## Solution Code:

import static org.mockito.Mockito.\*; import org.junit.jupiter.api.Test;

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

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);

}

}

**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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>MockRestApiExample</artifactId>

<version>1.0</version>

<name>MockRestApiExample</name>

<dependencies>

<!-- JUnit 5 -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<!-- Ensure JUnit 5 works with Maven -->

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<includes>

<include>\*\*/\*Test.java</include>

</includes>

</configuration>

</plugin>

</plugins>

</build>

</project>

**RestClient.java**

package com.example.api;

public interface RestClient {

String getResponse(); // Simulates a REST call (e.g., to a weather API)

}

**ApiService.java**

package com.example.api;

public class ApiService {

private RestClient restClient;

public ApiService(RestClient restClient) {

this.restClient = restClient;

}

public String fetchData() {

String response = restClient.getResponse();

return "Fetched " + response;

}

}

**ApiServiceTest.java**

package com.example.api;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class ApiServiceTest {

*@Test*

public void testServiceWithMockRestClient() {

// Step 1: Create mock REST client

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

// Step 2: Stub getResponse() to return mock data

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

// Step 3: Inject mock into service

ApiService apiService = new ApiService(mockRestClient);

// Step 4: Assert service behavior

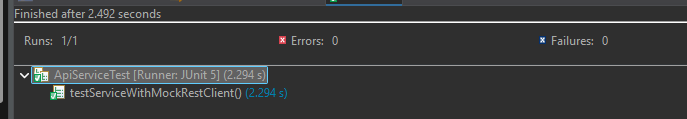
String result = apiService.fetchData();

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

}

}

**Ouput:**

****

# Exercise 3: Mocking File I/O

***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.

## Solution Code:

import static org.mockito.Mockito.\*; import org.junit.jupiter.api.Test;

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

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);

}

}

**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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>MockFileIOExample</artifactId>

<version>1.0</version>

<name>MockFileIOExample</name>

<dependencies>

<!-- JUnit 5 -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<includes>

<include>\*\*/\*Test.java</include>

</includes>

</configuration>

</plugin>

</plugins>

</build>

</project>

**FileReader.java**

package com.example.fileio;

public interface FileReader {

String read();

}

**FileWriter.java**

package com.example.fileio;

public interface FileWriter {

void write(String data);

}

**FileService.java**

package com.example.fileio;

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();

String processed = "Processed " + content;

fileWriter.write(processed);

return processed;

}

}

**FileServiceTest.java**

package com.example.fileio;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class FileServiceTest {

*@Test*

public void testServiceWithMockFileIO() {

// Step 1: Create mocks

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

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

// Step 2: Stub reader

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

// Step 3: Inject mocks into service

FileService fileService = new FileService(mockFileReader, mockFileWriter);

// Step 4: Call method

String result = fileService.processFile();

// Step 5: Verify output

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

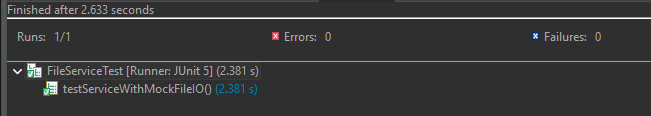
// Step 6: Verify writer called correctly

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

}

}

**Output:**



# Exercise 4: Mocking Network Interactions

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

## Steps:

1. 1. Create a mock network client using Mockito.
2. 2. Stub the network client methods to simulate network interactions.
3. 3. Write a test to verify the service logic using the mocked network client.

## Solution Code:

import static org.mockito.Mockito.\*; import org.junit.jupiter.api.Test;

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

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);

}

}

**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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>MockNetworkInteractionExample</artifactId>

<version>1.0</version>

<name>MockNetworkInteractionExample</name>

<dependencies>

<!-- JUnit 5 -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<includes>

<include>\*\*/\*Test.java</include>

</includes>

</configuration>

</plugin>

</plugins>

</build>

</project>

**NetworkClient.java**

package com.example.network;

public interface NetworkClient {

String connect(); // Simulates connecting to a server

}

**NetworkService.java**

package com.example.network;

public class NetworkService {

private NetworkClient networkClient;

public NetworkService(NetworkClient networkClient) {

this.networkClient = networkClient;

}

public String connectToServer() {

String response = networkClient.connect();

return "Connected to " + response;

}

}

**NetworkServiceTest.java**

package com.example.network;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class NetworkServiceTest {

*@Test*

public void testServiceWithMockNetworkClient() {

// Step 1: Create mock client

NetworkClient mockNetworkClient = *mock*(NetworkClient.class);

// Step 2: Stub method

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

// Step 3: Inject mock into service

NetworkService networkService = new NetworkService(mockNetworkClient);

// Step 4: Assert result

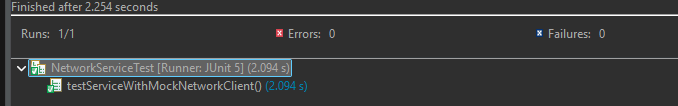
String result = networkService.connectToServer();

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

}

}

**Output:**

****

# Exercise 5: Mocking Multiple Return Values

***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.

## Solution Code:

import static org.mockito.Mockito.\*; import org.junit.jupiter.api.Test;

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

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);

}

}

**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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>MockMultipleReturnExample</artifactId>

<version>1.0</version>

<name>MockMultipleReturnExample</name>

<dependencies>

<!-- JUnit 5 -->

<dependency>

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

<artifactId>junit-jupiter</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.12.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0</version>

<configuration>

<includes>

<include>\*\*/\*Test.java</include>

</includes>

</configuration>

</plugin>

</plugins>

</build>

</project>

**DataProvider.java**

package com.example.multireturn;

public interface DataProvider {

String getData();

}

**MultiCallService.java**

package com.example.multireturn;

public class MultiCallService {

private DataProvider dataProvider;

public MultiCallService(DataProvider dataProvider) {

this.dataProvider = dataProvider;

}

public String combineMultipleDataCalls() {

String first = dataProvider.getData();

String second = dataProvider.getData();

return "Combined: " + first + " + " + second;

}

**MultiCallServiceTest.java**

package com.example.multireturn;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class MultiCallServiceTest {

*@Test*

public void testServiceWithMultipleReturnValues() {

// Step 1: Create mock

DataProvider mockDataProvider = *mock*(DataProvider.class);

// Step 2: Stub to return different values on each call

*when*(mockDataProvider.getData())

.thenReturn("First Call")

.thenReturn("Second Call");

// Step 3: Inject mock and test service logic

MultiCallService service = new MultiCallService(mockDataProvider);

String result = service.combineMultipleDataCalls();

// Step 4: Verify result

*assertEquals*("Combined: First Call + Second Call", result);

}

}

**Output:**

