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.

Code :

MyService.java

**package** mockprogram;

**import** java.util.List;

**import** java.util.Collections;

// External API interface that we want to mock

**interface** ExternalApi {

String getData();

String getDataById(Long id);

List<String> getAllData();

**boolean** isServiceAvailable();

}

// Service class that depends on the external API

**class** MyService {

**private** **final** ExternalApi externalApi;

**public** MyService(ExternalApi externalApi) {

**this**.externalApi = externalApi;

}

**public** String fetchData() {

**if** (!externalApi.isServiceAvailable()) {

**return** "Service unavailable";

}

**return** externalApi.getData();

}

**public** String fetchDataById(Long id) {

**if** (id == **null** || id <= 0) {

**throw** **new** IllegalArgumentException("Invalid ID");

}

**return** externalApi.getDataById(id);

}

**public** List<String> fetchAllData() {

List<String> data = externalApi.getAllData();

**return** data != **null** ? data : Collections.*emptyList*();

}

**public** String getFormattedData() {

String data = externalApi.getData();

**return** data != **null** ? "Formatted: " + data : "No data available";

}

}

MyServiceTest.java

**package** mockprogram;

**import** **static** org.mockito.Mockito.\*;

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

**import** org.junit.jupiter.api.Test;

**import** org.junit.jupiter.api.extension.ExtendWith;

**import** org.mockito.Mock;

**import** org.mockito.InjectMocks;

**import** org.mockito.junit.jupiter.MockitoExtension;

**import** java.util.Arrays;

**import** java.util.List;

@ExtendWith(MockitoExtension.**class**)

**public** **class** MyServiceTest {

@Mock

**private** ExternalApi mockApi;

@InjectMocks

**private** MyService myService;

@Test

**public** **void** testFetchData\_Success() {

// Given - Stub the mock methods

*when*(mockApi.isServiceAvailable()).thenReturn(**true**);

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

// When

String result = myService.fetchData();

// Then

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

*verify*(mockApi).isServiceAvailable();

*verify*(mockApi).getData();

}

@Test

**public** **void** testFetchData\_ServiceUnavailable() {

// Given

*when*(mockApi.isServiceAvailable()).thenReturn(**false**);

// When

String result = myService.fetchData();

// Then

*assertEquals*("Service unavailable", result);

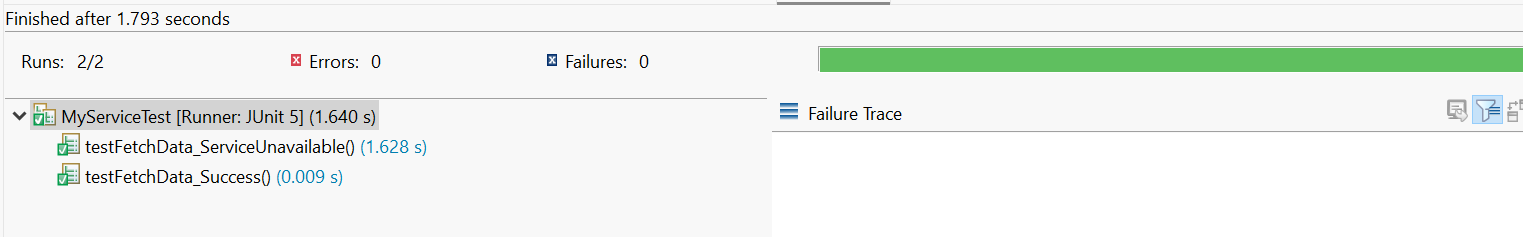
*verify*(mockApi).isServiceAvailable();

*verify*(mockApi, *never*()).getData(); // Verify getData was never called

}

}

Output :



Exercise 2: Verifying Interactions

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

Code :

EmailService.java

//Email service interface that we want to mock

**interface** EmailService {

**void** sendEmail(String recipient, String subject, String body);

}

//Service class that uses the email service

**class** NotificationService {

**private** EmailService emailService;

**public** NotificationService(EmailService emailService) {

**this**.emailService = emailService;

}

**public** **void** sendWelcomeEmail(String userEmail, String userName) {

String subject = "Welcome to our platform!";

String body = "Hello " + userName + ", welcome to our service!";

emailService.sendEmail(userEmail, subject, body);

}

**public** **void** sendPasswordResetEmail(String userEmail) {

String subject = "Password Reset Request";

String body = "Click the link to reset your password.";

emailService.sendEmail(userEmail, subject, body);

}

}

NotificationServiceTest.java

**import** **static** org.mockito.Mockito.\*;

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

**import** org.junit.jupiter.api.Test;

**import** org.junit.jupiter.api.BeforeEach;

**public** **class** NotificationServiceTest {

**private** EmailService mockEmailService;

**private** NotificationService notificationService;

@BeforeEach

**public** **void** setUp() {

// Step 1: Create a mock object

mockEmailService = *mock*(EmailService.**class**);

notificationService = **new** NotificationService(mockEmailService);

}

@Test

**public** **void** testSendWelcomeEmail\_VerifySpecificArguments() {

// Step 2: Call the method with specific arguments

[notificationService.sendWelcomeEmail("john@example.com](mailto:notificationService.sendWelcomeEmail("john@example.com)", "John Doe");

// Step 3: Verify the interaction with specific arguments

*verify*(mockEmailService).sendEmail(

"[john@example.com](mailto:john@example.com)",

"Welcome to our platform!",

"Hello John Doe, welcome to our service!"

);

}

@Test

**public** **void** testSendPasswordResetEmail\_VerifyExactArguments() {

// Step 2: Call the method

[notificationService.sendPasswordResetEmail("user@example.com](mailto:notificationService.sendPasswordResetEmail("user@example.com)");

// Step 3: Verify interaction with exact arguments

*verify*(mockEmailService).sendEmail(

"[user@example.com](mailto:user@example.com)",

"Password Reset Request",

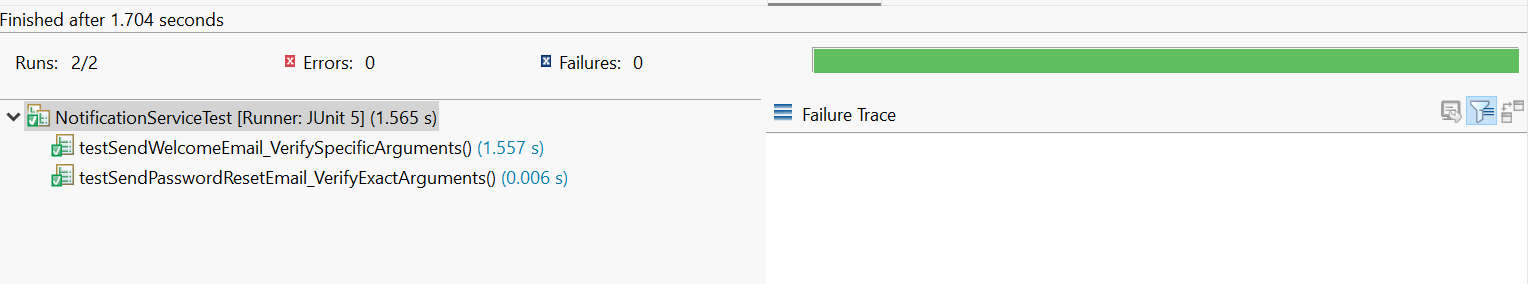
"Click the link to reset your password."

);

}

}

Output :



Exercise 3: Handling Void Methods

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

Code :

DocumentService.java

**package** mockprogram;

//File service interface with void methods

**interface** FileService {

**void** saveFile(String filename, String content);

**void** deleteFile(String filename);

}

//Service class that uses the file service

**class** DocumentService {

**private** FileService fileService;

**public** DocumentService(FileService fileService) {

**this**.fileService = fileService;

}

**public** **void** createDocument(String docName, String content) {

// Save the document using file service

fileService.saveFile(docName + ".txt", content);

// Create a backup

fileService.saveFile(docName + "\_backup.txt", content);

}

}

DocumentServiceTest.java

**package** mockprogram;

**import** **static** org.mockito.Mockito.\*;

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

**import** org.junit.jupiter.api.Test;

**import** org.junit.jupiter.api.BeforeEach;

**public** **class** DocumentServiceTest {

**private** FileService mockFileService;

**private** DocumentService documentService;

@BeforeEach

**public** **void** setUp() {

// Step 1: Create a mock object

mockFileService = *mock*(FileService.**class**);

documentService = **new** DocumentService(mockFileService);

}

@Test

**public** **void** testCreateDocument\_VerifyVoidMethodCalls() {

// Step 2: Stub the void method (optional - void methods don't need stubbing unless you want to throw exceptions)

*doNothing*().when(mockFileService).saveFile(*anyString*(), *anyString*());

// Call the method under test

documentService.createDocument("myDocument", "This is my document content");

// Step 3: Verify the interaction with void methods

*verify*(mockFileService).saveFile("myDocument.txt", "This is my document content");

*verify*(mockFileService).saveFile("myDocument\_backup.txt", "This is my document content");

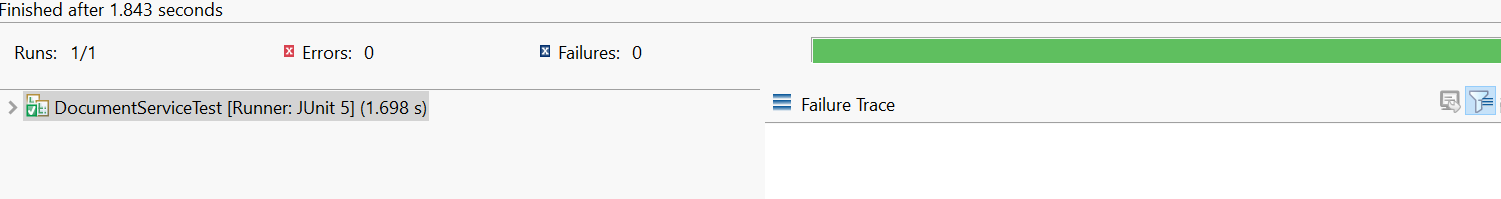
// Verify the saveFile method was called exactly 2 times

*verify*(mockFileService, *times*(2)).saveFile(*anyString*(), *anyString*());

}

}

Output :



Exercise 4: Mocking and Stubbing with Multiple Returns

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

Code :

WeatherService.java

**package** mockprogram;

//External API interface

**interface** WeatherApi {

String getTemperature();

}

//Service class that depends on the weather API

**class** WeatherService {

**private** WeatherApi weatherApi;

**public** WeatherService(WeatherApi weatherApi) {

**this**.weatherApi = weatherApi;

}

**public** String getWeatherReport() {

// Call the API three times to get temperature readings

String temp1 = weatherApi.getTemperature();

String temp2 = weatherApi.getTemperature();

String temp3 = weatherApi.getTemperature();

**return** "Temperature readings: " + temp1 + ", " + temp2 + ", " + temp3;

}

}

WeatherServiceTest.java

p **ackage** mockprogram;

**import** **static** org.mockito.Mockito.\*;

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

**import** org.junit.jupiter.api.Test;

**import** org.junit.jupiter.api.BeforeEach;

//Test class demonstrating multiple returns

**public** **class** WeatherServiceTest {

**private** WeatherApi mockWeatherApi;

**private** WeatherService weatherService;

@BeforeEach

**public** **void** setUp() {

// Step 1: Create a mock object for the external API

mockWeatherApi = *mock*(WeatherApi.**class**);

weatherService = **new** WeatherService(mockWeatherApi);

}

@Test

**public** **void** testGetWeatherReport\_MultipleReturns() {

// Step 2: Stub the methods to return different values on consecutive calls

*when*(mockWeatherApi.getTemperature())

.thenReturn("25°C")

.thenReturn("27°C")

.thenReturn("24°C");

// Step 3: Write a test case that uses the mock object

String result = weatherService.getWeatherReport();

// Verify the result contains all three different temperature readings

*assertEquals*("Temperature readings: 25°C, 27°C, 24°C", result);

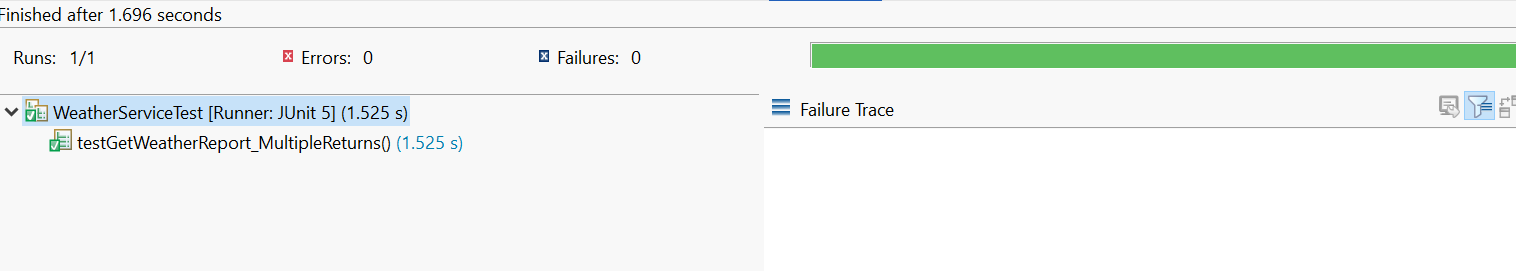
// Verify that the API was called exactly 3 times

*verify*(mockWeatherApi, *times*(3)).getTemperature();

}

}

Output :



Exercise 5: Verifying Interaction Order

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

Code :

UserService.java

**package** mockprogram;

//Database service interface

**interface** DatabaseService {

**void** openConnection();

**void** executeQuery(String query);

**void** closeConnection();

}

//Service class that uses the database service

**class** UserService {

**private** DatabaseService databaseService;

**public** UserService(DatabaseService databaseService) {

**this**.databaseService = databaseService;

}

**public** **void** createUser(String username) {

// These methods must be called in specific order

databaseService.openConnection();

databaseService.executeQuery("INSERT INTO users (name) VALUES ('" + username + "')");

databaseService.closeConnection();

}

}

UserServiceTest.java

**package** mockprogram;

**import** **static** org.mockito.Mockito.\*;

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

**import** org.junit.jupiter.api.Test;

**import** org.mockito.InOrder;

**import** org.junit.jupiter.api.BeforeEach;

//Test class demonstrating interaction order verification

**public** **class** UserServiceTest {

**private** DatabaseService mockDatabaseService;

**private** UserService userService;

@BeforeEach

**public** **void** setUp() {

// Step 1: Create a mock object

mockDatabaseService = *mock*(DatabaseService.**class**);

userService = **new** UserService(mockDatabaseService);

}

@Test

**public** **void** testCreateUser\_VerifyInteractionOrder() {

// Step 2: Call the methods in a specific order

userService.createUser("john\_doe");

// Step 3: Verify the interaction order

InOrder inOrder = *inOrder*(mockDatabaseService);

inOrder.verify(mockDatabaseService).openConnection();

inOrder.verify(mockDatabaseService).executeQuery("INSERT INTO users (name) VALUES ('john\_doe')");

inOrder.verify(mockDatabaseService).closeConnection();

// Also verify that all methods were called exactly once

*verify*(mockDatabaseService, *times*(1)).openConnection();

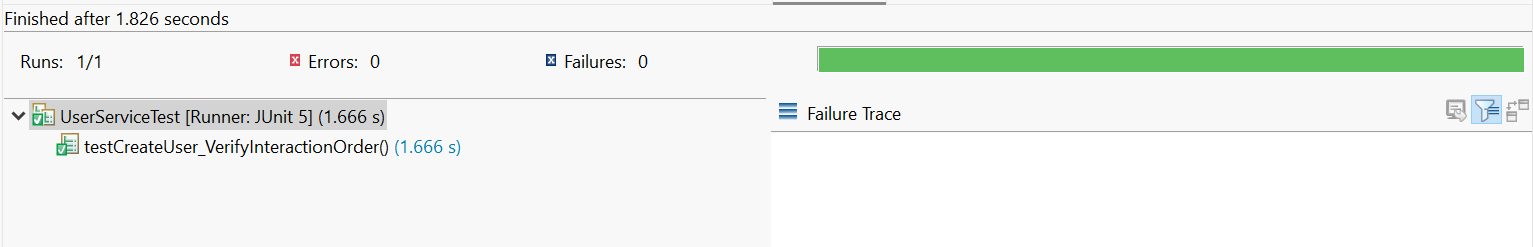
*verify*(mockDatabaseService, *times*(1)).executeQuery(*anyString*());

*verify*(mockDatabaseService, *times*(1)).closeConnection();

}

}

Output :



Exercise 6: Handling Void Methods with Exceptions

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

Code :

DocumentService.java

**package** mockprogram;

**interface** FileService {

**void** saveFile(String filename, String content) **throws** Exception;

}

// Service class that uses the file service

**class** DocumentService {

**private** FileService fileService;

**public** DocumentService(FileService fileService) {

**this**.fileService = fileService;

}

**public** String createDocument(String docName, String content) {

**try** {

fileService.saveFile(docName + ".txt", content);

**return** "Document saved successfully";

} **catch** (Exception e) {

**return** "Failed to save document: " + e.getMessage();

}

}

}

DocumentServiceTest.java

**package** mockprogram;

**import** **static** org.mockito.Mockito.\*;

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

**import** org.junit.jupiter.api.Test;

**import** org.junit.jupiter.api.BeforeEach;

//Test class demonstrating void method exception handling

**public** **class** DocumentServiceTest {

**private** FileService mockFileService;

**private** DocumentService documentService;

@BeforeEach

**public** **void** setUp() {

// Step 1: Create a mock object

mockFileService = *mock*(FileService.**class**);

documentService = **new** DocumentService(mockFileService);

}

@Test

**public** **void** testCreateDocument\_VoidMethodThrowsException() **throws** Exception {

// Step 2: Stub the void method to throw an exception

*doThrow*(**new** RuntimeException("Disk full")).when(mockFileService).saveFile("myDocument.txt", "content");

// Call the method under test

String result = documentService.createDocument("myDocument", "content");

// Step 3: Verify the interaction and exception handling

*verify*(mockFileService).saveFile("myDocument.txt", "content");

// Verify that the exception was handled correctly

*assertEquals*("Failed to save document: Disk full", result);

}

}

Output :

