**Exercise 1: Control Structures**

**Scenario 1:** The bank wants to apply a discount to loan interest rates for customers above 60 years old.

**Question:** Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

BEGIN

FOR cust\_rec IN (

SELECT c.CustomerID, l.LoanID

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

WHERE TRUNC(MONTHS\_BETWEEN(SYSDATE, c.DOB)/12) > 60

) LOOP

UPDATE Loans

SET InterestRate = InterestRate - (InterestRate \* 0.01)

WHERE LoanID = cust\_rec.LoanID;

END LOOP;

COMMIT;

END;

**Scenario 2:** A customer can be promoted to VIP status based on their balance.

**Question:** Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.  
BEGIN  
 UPDATE Customers  
 SET IsVIP = 'TRUE'  
 WHERE Balance > 10000;  
END;

**Scenario 3:** The bank wants to send reminders to customers whose loans are due within the next 30 days.

**Question:** Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

BEGIN

FOR loan\_rec IN (

SELECT l.LoanID, l.CustomerID, c.Name, l.EndDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Reminder: Loan ID ' || loan\_rec.LoanID ||

' for Customer ' || loan\_rec.Name ||

' (ID: ' || loan\_rec.CustomerID || ') is due on ' || TO\_CHAR(loan\_rec.EndDate, 'DD-MON-YYYY')

);

END LOOP;

END;

**Exercise 2: Error Handling**

**Scenario 1:** Handle exceptions during fund transfers between accounts.

**Question:** Write a stored procedure **SafeTransferFunds** that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.  
  
CREATE PROCEDURE SafeTransferFunds(p\_from NUMBER, p\_to NUMBER, p\_amount NUMBER) AS  
BEGIN  
 UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from;  
 UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to;  
 COMMIT;  
EXCEPTION  
 WHEN OTHERS THEN  
 ROLLBACK;  
 DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);  
END;

**Scenario 2:** Manage errors when updating employee salaries.

**Question:** Write a stored procedure **UpdateSalary** that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

CREATE PROCEDURE UpdateSalary(p\_empID NUMBER, p\_percent NUMBER) AS  
BEGIN  
 UPDATE Employees SET Salary = Salary + Salary \* (p\_percent / 100) WHERE EmployeeID = p\_empID;  
 IF SQL%ROWCOUNT = 0 THEN  
 RAISE\_APPLICATION\_ERROR(-20001, 'Employee not found');  
 END IF;  
EXCEPTION  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);  
END;

**Scenario 3:** Ensure data integrity when adding a new customer.

**Question:** Write a stored procedure **AddNewCustomer** that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.  
  
CREATE PROCEDURE AddNewCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER) AS  
BEGIN  
 INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)  
 VALUES (p\_id, p\_name, p\_dob, p\_balance, SYSDATE);  
EXCEPTION  
 WHEN DUP\_VAL\_ON\_INDEX THEN  
 DBMS\_OUTPUT.PUT\_LINE('Error: Customer with this ID already exists.');  
END;

**Exercise 3: Stored Procedures**

**Scenario 1:** The bank needs to process monthly interest for all savings accounts.

**Question:** Write a stored procedure **ProcessMonthlyInterest** that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

FOR acc IN (

SELECT AccountID, Balance

FROM Accounts

WHERE AccountType = 'Savings'

) LOOP

UPDATE Accounts

SET Balance = acc.Balance + (acc.Balance \* 0.01)

WHERE AccountID = acc.AccountID;

END LOOP;

COMMIT;

END;

**Scenario 2:** The bank wants to implement a bonus scheme for employees based on their performance.

**Question:** Write a stored procedure **UpdateEmployeeBonus** that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

CREATE PROCEDURE UpdateEmployeeBonus(

p\_department IN VARCHAR2,

p\_bonus\_percent IN NUMBER

) AS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* p\_bonus\_percent / 100)

WHERE Department = p\_department;

COMMIT;

END;

**Scenario 3:** Customers should be able to transfer funds between their accounts.

**Question:** Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance NUMBER;

BEGIN

-- Check if source account has sufficient balance

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_from\_account;

IF v\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance in source account.');

END IF;

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20002, 'One or both account IDs are invalid.');

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error occurred: ' || SQLERRM);

END;

**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

**Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURN NUMBER IS

BEGIN

RETURN FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

END;

**Scenario 2:** The bank needs to compute the monthly installment for a loan.

**Question:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p\_loan NUMBER, p\_rate NUMBER, p\_years NUMBER) RETURN NUMBER IS

v\_months NUMBER := p\_years \* 12;

v\_r NUMBER := p\_rate / (12 \* 100);

BEGIN

RETURN (p\_loan \* v\_r \* POWER(1 + v\_r, v\_months)) / (POWER(1 + v\_r, v\_months) - 1);

END;

**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

**Question:** Write a function **HasSufficientBalance** that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

CREATE OR REPLACE FUNCTION HasSufficientBalance(p\_accountid NUMBER, p\_amount NUMBER) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_accountid;

RETURN v\_balance >= p\_amount;

END;

**Exercise 5: Triggers**

**Scenario 1:** Automatically update the last modified date when a customer's record is updated.

**Question:** Write a trigger **UpdateCustomerLastModified** that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

**Scenario 2:** Maintain an audit log for all transactions.

**Question:** Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (Action, ActionDate, Details)

VALUES ('Transaction Inserted', SYSDATE, 'AccountID: ' || :NEW.AccountID || ', Amount: ' || :NEW.Amount);

END;

**Scenario 3:** Enforce business rules on deposits and withdrawals.

**Question:** Write a trigger **CheckTransactionRules** that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;

IF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Withdrawal exceeds balance');

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20005, 'Deposit must be positive');

END IF;

END IF;

END;

**Exercise 6: Cursors**

**Scenario 1:** Generate monthly statements for all customers.

**Question:** Write a PL/SQL block using an explicit cursor **GenerateMonthlyStatements** that retrieves all transactions for the current month and prints a statement for each customer.

DECLARE

CURSOR cur IS SELECT \* FROM Transactions WHERE EXTRACT(MONTH FROM TransactionDate) = EXTRACT(MONTH FROM SYSDATE);

BEGIN

FOR rec IN cur LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer statement: AccountID ' || rec.AccountID || ', Amount: ' || rec.Amount);

END LOOP;

END;

**Scenario 2:** Apply annual fee to all accounts.

**Question:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

DECLARE

CURSOR cur IS SELECT \* FROM Accounts;

BEGIN

FOR rec IN cur LOOP

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = rec.AccountID;

END LOOP;

COMMIT;

END;

**Scenario 3:** Update the interest rate for all loans based on a new policy.

**Question:** Write a PL/SQL block using an explicit cursor **UpdateLoanInterestRates** that fetches all loans and updates their interest rates based on the new policy.

DECLARE

CURSOR cur IS SELECT \* FROM Loans;

BEGIN

FOR rec IN cur LOOP

UPDATE Loans SET InterestRate = rec.InterestRate + 0.5 WHERE LoanID = rec.LoanID;

END LOOP;

COMMIT;

END;

**Exercise 7: Packages**

**Scenario 1:** Group all customer-related procedures and functions into a package.

**Question:** Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(...);

PROCEDURE UpdateCustomer(...);

FUNCTION GetBalance(p\_id NUMBER) RETURN NUMBER;

END;

**Scenario 2:** Create a package to manage employee data.

**Question:** Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(...);

PROCEDURE UpdateEmployee(...);

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END;

**Scenario 3:** Group all account-related operations into a package.

**Question:** Create a package **AccountOperations** with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(...);

PROCEDURE CloseAccount(...);

FUNCTION TotalBalance(p\_custid NUMBER) RETURN NUMBER;

END;

**JUnit Testing Exercises**

**Exercise 1: Setting Up JUnit**

**import static org.junit.Assert.\*;**

**import org.junit.Test;**

**public class BasicTest {**

**@Test**

**public void sampleTest() {**

**assertTrue(true);**

**}**

**}**

**Exercise 2: Writing Basic JUnit Tests**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

}

import static org.junit.Assert.\*;

import org.junit.Test;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

assertEquals(5, calc.add(2, 3));

}

@Test

public void testMultiply() {

Calculator calc = new Calculator();

assertEquals(6, calc.multiply(2, 3));

}

}

**Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and**

**Teardown Methods in JUnit**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

public void setUp() {

calculator = new Calculator();

}

public void tearDown() {

calculator = null;

}

public void testAddition() {

int result = calculator.add(2, 3);

assertEquals(5, result);

}

public void testSubtraction() {

int result = calculator.subtract(10, 4);

assertEquals(6, result);

}

}

class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

**Exercise 1: Parameterized Tests**

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

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

public class EvenCheckerTest {

@ParameterizedTest

@ValueSource(ints = {2, 4, 6, 8})

void testIsEven(int number) {

assertTrue(EvenChecker.isEven(number));

}

}

class EvenChecker {

public static boolean isEven(int number) {

return number % 2 == 0;

}

}

**Exercise 2: Test Suites and Categories**

import org.junit.platform.suite.api.SelectClasses;

import org.junit.platform.suite.api.Suite;

@Suite

@SelectClasses({CalculatorTest.class, EvenCheckerTest.class})

public class AllTests {

}

**Exercise 3: Test Execution Order**

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

import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;

@TestMethodOrder(OrderAnnotation.class)

public class OrderedTests {

@Test

@Order(1)

void testFirst() {

System.out.println("First Test");

}

@Test

@Order(2)

void testSecond() {

System.out.println("Second Test");

}

}

****Exercise 4: Exception Testing****

import org.junit.jupiter.api.Test;

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

public class ExceptionThrowerTest {

@Test

void testThrowException() {

assertThrows(IllegalArgumentException.class, () -> {

ExceptionThrower.throwException();

});

}

}

class ExceptionThrower {

public static void throwException() {

throw new IllegalArgumentException("This is an error");

}

}

****Exercise 5: Timeout and Performance Testing****

import org.junit.jupiter.api.Test;

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

import java.time.Duration;

public class PerformanceTesterTest {

@Test

void testPerformTask() {

assertTimeout(Duration.ofSeconds(1), () -> {

PerformanceTester.performTask();

});

}

}

class PerformanceTester {

public static void performTask() {

// Simulate a quick task

for (int i = 0; i < 1000000; i++) {

int x = i \* i;

}

}

}

**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.

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

class ExternalApi {

String getData() {

return "Real Data";

}

}

class MyService {

private ExternalApi api;

MyService(ExternalApi api) {

this.api = api;

}

String fetchData() {

return api.getData();

}

}

public class MyServiceTest {

@Test

public void testExternalApi() {

ExternalApi mockApi = mock(ExternalApi.class);

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

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

### ****Exercise 2: Verifying Interactions****

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

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

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.fetchData();

verify(mockApi).getData();

}

}

### ****Exercise 3: Argument Matching****

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

import static org.mockito.Mockito.\*;

import static org.mockito.ArgumentMatchers.\*;

import org.junit.jupiter.api.Test;

class Logger {

void log(String message) {}

}

public class LoggerTest {

@Test

public void testArgumentMatching() {

Logger mockLogger = mock(Logger.class);

mockLogger.log("Hello");

verify(mockLogger).log(anyString());

}

}

### ****Exercise 4: Handling Void Methods****

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

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

class Notifier {

void sendNotification(String msg) {}

}

public class NotifierTest {

@Test

public void testVoidMethod() {

Notifier mockNotifier = mock(Notifier.class);

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

mockNotifier.sendNotification("Hello");

verify(mockNotifier).sendNotification("Hello");

}

}

### ****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.

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

public class MultipleReturnsTest {

@Test

public void testMultipleReturns() {

List mockList = mock(List.class);

when(mockList.get(0)).thenReturn("First").thenReturn("Second");

assertEquals("First", mockList.get(0));

assertEquals("Second", mockList.get(0));

}

}

### ****Exercise 6: Verifying Interaction Order****

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

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

class Service {

void stepOne() {}

void stepTwo() {}

}

public class ServiceTest {

@Test

public void testOrder() {

Service mockService = mock(Service.class);

mockService.stepOne();

mockService.stepTwo();

InOrder inOrder = inOrder(mockService);

inOrder.verify(mockService).stepOne();

inOrder.verify(mockService).stepTwo();

}

}

### ****Exercise 7: Handling Void Methods with Exceptions****

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

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

class Failer {

void failIfNeeded() {

throw new RuntimeException("Failure");

}

}

public class FailerTest {

@Test

public void testVoidException() {

Failer mockFailer = mock(Failer.class);

doThrow(new RuntimeException("Failure")).when(mockFailer).failIfNeeded();

assertThrows(RuntimeException.class, () -> mockFailer.failIfNeeded());

}

}

**Advanced Mockito Hands-On Exercises**

### ****Exercise 1: Mocking Databases and Repositories****

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);

}

}

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

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);

}

}

### ****Exercise 3: Mocking File I/O****

**Scenario:**  
Test a service that reads/writes to files.

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);

}

}

### ****Exercise 4: Mocking Network Interactions****

**Scenario:**  
Test a service that connects to a network.

**Solution:**

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);

}

}

### ****Exercise 5: Mocking Multiple Return Values****

**Scenario:**  
Test a service method that’s called multiple times with different results.

**Solution:**

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);

}

}

**Spring Testing Exercises**

### ****Exercise 1: Basic Unit Test for a Service Method****

**Service Code:**

@Servicepublic class CalculatorService {

public int add(int a, int b) {

return a + b;

}

}

**Test Code:**

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

public class CalculatorServiceTest {

private final CalculatorService calculatorService = new CalculatorService();

@Test

public void testAdd() {

assertEquals(5, calculatorService.add(2, 3));

}

}

### ****Exercise 2: Mocking a Repository in a Service Test****

**Entity:**

@Entitypublic class User {

@Id

private Long id;

private String name;

}

**Repository:**

public interface UserRepository extends JpaRepository<User, Long> {

}

**Service:**

@Servicepublic class UserService {

@Autowired

private UserRepository userRepository;

public User getUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

}

**Test Code:**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import java.util.Optional;

public class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

public UserServiceTest() {

MockitoAnnotations.openMocks(this);

}

@Test

public void testGetUserById() {

User user = new User();

user.setId(1L);

user.setName("Alice");

when(userRepository.findById(1L)).thenReturn(Optional.of(user));

User result = userService.getUserById(1L);

assertEquals("Alice", result.getName());

}

}

### ****Exercise 3: Testing a REST Controller with MockMvc****

**Controller:**

@RestController@RequestMapping("/users")public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

public ResponseEntity<User> getUser(@PathVariable Long id) {

return ResponseEntity.ok(userService.getUserById(id));

}

}

**Test Code:**

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

import org.junit.jupiter.api.Test;import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.boot.test.mock.mockito.MockBean;import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest(UserController.class)public class UserControllerTest {

@Autowired

private MockMvc mockMvc;

@MockBean

private UserService userService;

@Test

public void testGetUser() throws Exception {

User user = new User();

user.setId(1L);

user.setName("Alice");

when(userService.getUserById(1L)).thenReturn(user);

mockMvc.perform(get("/users/1"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("Alice"));

}

}

### ****Exercise 4: Integration Test with Spring Boot****

@SpringBootTest@AutoConfigureMockMvcpublic class UserIntegrationTest {

@Autowired

private MockMvc mockMvc;

@Test

public void testFullFlow() throws Exception {

mockMvc.perform(post("/users")

.contentType("application/json")

.content("{\"id\":1,\"name\":\"Alice\"}"))

.andExpect(status().isOk());

mockMvc.perform(get("/users/1"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("Alice"));

}

}

### ****Exercise 5: Test Controller POST Endpoint****

**POST Endpoint:**

@PostMappingpublic ResponseEntity<User> createUser(@RequestBody User user) {

return ResponseEntity.ok(userService.saveUser(user));

}

**Test:**

@Testpublic void testCreateUser() throws Exception {

User user = new User();

user.setId(1L);

user.setName("Bob");

when(userService.saveUser(any(User.class))).thenReturn(user);

mockMvc.perform(post("/users")

.contentType("application/json")

.content("{\"id\":1,\"name\":\"Bob\"}"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("Bob"));

}

### ****Exercise 6: Test Service Exception Handling****

@Testpublic void testGetUserById\_NotFound() {

when(userRepository.findById(99L)).thenReturn(Optional.empty());

assertNull(userService.getUserById(99L));

}

### ****Exercise 7: Test Custom Repository Query****

**Repository:**

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByName(String name);

}

**Test:**

@Testpublic void testFindByName() {

User user = new User();

user.setId(1L);

user.setName("Alice");

when(userRepository.findByName("Alice")).thenReturn(List.of(user));

List<User> result = userRepository.findByName("Alice");

assertEquals(1, result.size());

}

### ****Exercise 8: Test Controller Exception Handling****

**Exception Handler:**

@ControllerAdvicepublic class GlobalExceptionHandler {

@ExceptionHandler(NoSuchElementException.class)

public ResponseEntity<String> handleNotFound(NoSuchElementException ex) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body("User not found");

}

}

**Test:**

@Testpublic void testControllerThrowsException() throws Exception {

when(userService.getUserById(99L)).thenThrow(new NoSuchElementException("User not found"));

mockMvc.perform(get("/users/99"))

.andExpect(status().isNotFound())

.andExpect(content().string("User not found"));

}

### ****Exercise 9: Parameterized Test with JUnit****

import org.junit.jupiter.params.ParameterizedTest;import org.junit.jupiter.params.provider.ValueSource;

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

public class CalculatorTest {

@ParameterizedTest

@ValueSource(ints = {2, 4, 6, 8})

public void testIsEven(int number) {

assertTrue(number % 2 == 0);

}

}

**Mocking Dependencies in Spring Tests using Mockito**

### ****Exercise 1: Mocking a Service Dependency in a Controller Test****

**Goal:** Write a unit test for a Spring controller that uses a service to fetch data. Mock the service using Mockito.

**Controller:**

@RestController@RequestMapping("/users")public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

public ResponseEntity<User> getUser(@PathVariable Long id) {

return ResponseEntity.ok(userService.getUserById(id));

}

}

**Test Code:**

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.mock.mockito.MockBean;

import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest(UserController.class)public class UserControllerTest {

@Autowired

private MockMvc mockMvc;

@MockBean

private UserService userService;

@Test

public void testGetUser() throws Exception {

User user = new User();

user.setId(1L);

user.setName("Alice");

when(userService.getUserById(1L)).thenReturn(user);

mockMvc.perform(get("/users/1"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("Alice"));

}

}

### ****Exercise 2: Mocking a Repository in a Service Test****

**Goal:** Write a unit test for a Spring service using a mocked repository.

**Test Code:**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import java.util.Optional;

public class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

public UserServiceTest() {

MockitoAnnotations.openMocks(this);

}

@Test

public void testGetUserById() {

User user = new User();

user.setId(1L);

user.setName("Alice");

when(userRepository.findById(1L)).thenReturn(Optional.of(user));

User result = userService.getUserById(1L);

assertEquals("Alice", result.getName());

}

}

**Exercise 3: Mocking a Service Dependency in an Integration Test**

**Goal:** Write a Spring Boot integration test that mocks a service.

**Test Code:**

import static org.mockito.Mockito.\*;import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

import org.junit.jupiter.api.Test;import org.springframework.boot.test.context.SpringBootTest;import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;import org.springframework.boot.test.mock.mockito.MockBean;import org.springframework.beans.factory.annotation.Autowired;import org.springframework.test.web.servlet.MockMvc;

@SpringBootTest@AutoConfigureMockMvcpublic class UserIntegrationTest {

@Autowired

private MockMvc mockMvc;

@MockBean

private UserService userService;

@Test

public void testGetUser() throws Exception {

User user = new User();

user.setId(1L);

user.setName("Alice");

when(userService.getUserById(1L)).thenReturn(user);

mockMvc.perform(get("/users/1"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("Alice"));

}

}

**Logging using SLF4J**

### ****Exercise 1: Logging Error Messages and Warning Levels****

**Goal:** Demonstrate logging error and warning levels using SLF4J.

**Step 1: Add SLF4J + Logback dependencies (**pom.xml**):**

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version></dependency><dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version></dependency>

**Step 2: Java Logging Example**

import org.slf4j.Logger;import org.slf4j.LoggerFactory;

public class LoggingExample {

private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);

public static void main(String[] args) {

logger.error("This is an error message");

logger.warn("This is a warning message");

}

}

### ****Exercise 2: Parameterized Logging****

**Goal:** Demonstrate how to use placeholders ({}) in SLF4J logs.

import org.slf4j.Logger;import org.slf4j.LoggerFactory;

public class ParameterizedLogging {

private static final Logger logger = LoggerFactory.getLogger(ParameterizedLogging.class);

public static void main(String[] args) {

String username = "Alice";

int age = 30;

logger.info("User {} has logged in", username);

logger.debug("User {} is {} years old", username, age);

}

}

### ****Exercise 3: Using Different Appenders****

**Goal:** Configure and use console and file appenders in SLF4J via logback.xml.

logback.xml

**Configuration**

<configuration>

<appender name="console" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<appender name="file" class="ch.qos.logback.core.FileAppender">

<file>app.log</file>

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="debug">

<appender-ref ref="console" />

<appender-ref ref="file" />

</root></configuration>

**Step 2: Java Code Example**

import org.slf4j.Logger;import org.slf4j.LoggerFactory;

public class MultiAppenderLogger {

private static final Logger logger = LoggerFactory.getLogger(MultiAppenderLogger.class);

public static void main(String[] args) {

logger.info("This message goes to both console and file.");

logger.error("This is a logged error with both appenders.");

}

}