Unit Testing on

Spring Boot Application

Prepared by

Md Mamun Hossain

Employee ID: 484

Table of Contents

1. Unit Testing

1.1 Unit Testing

Unit testing is a fundamental aspect of software testing where individual components or units of a software application are tested in isolation. A "unit" is the smallest testable part of any software, typically a function, method, or object. This method ensures that each unit of the software performs as expected. The primary goal of unit testing is to validate that each unit of the software performs as expected.

By focusing on small, manageable parts of the application, unit testing helps identify and fix bugs early in the development process, significantly improving code quality and reliability.

2.2 Why Unit Testing?

Unit testing is an essential practice in software development that offers numerous benefits. Here are some key reasons why unit testing is important.

- **Early Bug Detection**
  Unit tests help catch bugs early in the development cycle, allowing developers to identify and fix issues before they escalate. This early detection reduces the cost and effort required to resolve bugs later in the process or after deployment.

- **Improved Code Quality**
  Unit tests ensure that each component of the software performs as expected. This validation helps maintain high code quality by ensuring that the code meets its design and behaves correctly under various conditions.

- **Facilitates Refactoring**
  Unit tests act as a safety net when making changes to the codebase. They allow developers to refactor or update code confidently, knowing that any regressions or unintended side effects will be caught by the tests.

- **Documentation and Understanding**
  Unit tests serve as documentation for the code, providing clear examples of how different components are expected to behave. This helps new developers understand the codebase and the intended functionality of various units.

- **Encourages Modularity**
  Writing unit tests encourages developers to write modular, loosely coupled code. This modularity makes the codebase more maintainable, easier to understand, and simpler to extend or modify.

- **Promotes Best Practices**
  Unit testing promotes best practices in software development, such as writing clean, maintainable code and following design principles like SOLID. It is also a core practice in Test-Driven Development (TDD), which fosters better design and specification adherence.

- **Easier Maintenance**
  A well-tested codebase is easier to maintain and extend. Developers can confidently make changes, knowing that existing functionality is safeguarded by the tests, which helps in long-term project maintenance.

- **Enhances Code Reusability**
  Unit testing helps with code reusability by facilitating the migration of code and test cases. Well-tested code can be confidently reused in different parts of the application or in new projects, ensuring that the reused components behave correctly in their new contexts. This promotes efficient development and reduces redundancy.

2. Environment Setup

To effectively perform unit testing in a Spring Boot project, it's essential to set up a development environment with the required tools and dependencies.

3.1 Development Tools

Before setting up the project, make sure the following development tools are installed

- Java Development Kit (JDK): Ensure JDK 8 or higher is installed on your machine.
- Integrated Development Environment (IDE): An IDE like IntelliJ IDEA, Eclipse, or Visual Studio Code. Here, IntelliJ IDEA is used.
- Build Automation Tool: Maven or Gradle for managing project dependencies and build processes. Here, Maven is used.

3.2 Project Setup

3.2.1 Create a Spring Boot Project

- We can create a Spring Boot project using **Spring Initializr** or directly through your IDE.
- Select the required dependencies, including **Spring Web**, **Spring Data JPA**, and **H2 Database** (for an in-memory database).

3.2.2 Add Dependencies

When creating a Spring Boot project using Spring Initializr or an IDE like IntelliJ IDEA or Eclipse, the **spring-boot-starter-test** dependency is typically included automatically. This is

because Spring Initializr, which is often used by these IDEs to generate new Spring Boot projects, ensures that this essential testing dependency is added by default. The spring-boot-starter-test package includes several crucial libraries like JUnit, Mockito, and AssertJ, which are necessary for unit testing. If any additional testing dependencies are required, they can be added later as needed.

1. **spring-boot-starter-test**

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

2. **h2**

The **H2 Database** dependency is essential when testing the repository layer using the @DataJpaTest annotation.

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>2.3.230</version>
  <scope>test</scope>
</dependency>
```

If any additional dependencies are required for testing, they will be addressed in the subsequent sections.

3.2.3 Configure Application

We need to ensure that application.properties or application.yml file is configured for testing purposes. Here is an example application.properties for a project using Oracle Database, though you can also use MySQL.

```
spring.application.name=employee
server.port=9192

#Database configuration
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=hr
spring.datasource.password=hr
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

#Hibernate Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

3.2.4 Writing Unit Tests

- **Directory Structure**
  Ensure your tests are placed in the **src/test/java** directory of your project, mirroring the package structure of the main source code.

- **Test Annotations.**
  Familiarize with the key annotations used in Spring Boot testing: We will address them when we use.

- **Creating Test Cases**
  Write test cases for each layer method, focusing on testing the behavior in isolation.

3. Assertion

3.1 JUnit Assertions

Assertions is a JUnit API or library of functions that provides a library of functions to verify test outcomes. The Assertions class in JUnit 5 offers a set of methods to validate conditions and ensure that code behaves as expected. If an assertion fails, it throws an AssertionError, indicating that the test did not pass.

These methods allow you to compare actual results with expected outcomes, covering conditions such as equality, nullity, and truthiness. Assertions are essential for effective unit testing, as they help ensure that the actual output matches the expected result for each test case.

Advantages of Using Assertions

1. Assertions provide a quick and effective method for detecting and fixing bugs.
2. They help ensure that your assumptions about the output of specific conditions are correct.
3. Assertions can serve as preconditions or postconditions for function calls, enhancing the reliability of your code.

Implementation of JUnit Assertions

When examining the implementation of assertions, it's important to note that JUnit 4 and JUnit 5 use different classes for their assert methods. In JUnit 4, assert methods are contained within the Assert class. In contrast, JUnit 5 uses the Assertions class for assert methods.

For JUnit 5, import the Assertions class from the package **org.junit.jupiter.api** , for JUnit 4, import the Assert class from **org. junit** .

3.2 Common Assertions in JUnit 5

Here are common assertions in **JUnit 5**

- **assertEquals(expected, actual)**: Asserts that the expected and actual values are equal.
- **assertNotEquals(expected, actual)**: Asserts that the expected and actual values are not equal.
- **assertTrue(condition)**: Asserts that the given condition is true. The test passes if true and fails if false.
- **assertFalse(condition)**: Asserts that the given condition is false. The test passes if false and fails if true.
- **assertNull(value)**: Asserts that the given value is null. The test passes if null and fails if not.
- **assertNotNull(value)**: Asserts that the given value is not null. The test passes if not null and fails if null.
- **assertArrayEquals(expectedArray, actualArray)**: Asserts that the expected and actual arrays are equal. The test passes if they are equal and fails if not.
- **assertSame(expected, actual)**: Asserts that the expected and actual references point to the same object. The test passes if they are the same and fails if not.
- **assertNotSame(expected, actual)**: Asserts that the expected and actual references do not point to the same object. The test passes if they are different and fails if not.
- **assertThrows(exceptionType, executable)**: Asserts that the executable throws an exception of the specified type. The test passes if the exception is thrown and fails if not.

These assertions help validate different aspects of your code, ensuring it behaves as expected under various conditions. Using these assertions effectively can significantly enhance the reliability and correctness of your unit tests.

4. Project Overview

Project Name: mPack (Management Pack)

The mPack project is a Spring Boot application designed for managing employee information. It provides functionalities for creating, updating, retrieving, and deleting employee records. The application utilizes an Oracle relational database for data persistence and Spring Data JPA for data access. Additionally, the project incorporates Data Transfer Objects (DTOs) for effective data encapsulation and mapping.

Project Structure:

1. **Entity Layer:**

    - **Employee Entity:** Represents the employee data model with fields such as id, name, email, designation, department, division, mobile, salary, address, and bloodGroup.

This entity is mapped to the employeeManagement table in the database, with unique constraints on email and mobile.

2. **Repository Layer:**

   - **EmployeeRepository:** Provides data access methods for the Employee entity. It extends JpaRepository and includes custom query methods to find employees by attributes such as email, mobile, designation, department, division, and bloodGroup, as well as by a combination of department and division.

3. **Service Layer:**

   - **EmployeeService:** Defines service methods for CRUD operations and querying employee data.
   - **EmployeeServiceImpl:** Implements the EmployeeService interface, including methods for creating, updating, deleting, and retrieving employee records. It uses ModelMapper to map between entity and DTO objects.

4. **DTO Layer:**

   - **EmployeeDto:** Represents employee details for data transfer.
   - **CustomizeEmpDto:** Provides customized responses related to employee data.
   - **ResponseEmpDto:** Used for responses related to operations such as updates and deletions.

5. **Controller Layer:**

   - **EmployeeController:** A REST controller that provides endpoints for managing employees. It includes methods for creating, retrieving, updating, and deleting employee records, as well as querying by various attributes.

6. **Exception Layer:**

   - **Custom Exceptions:** Handles application-specific exceptions. For example: **ResourceNotFoundException:** Thrown when a requested resource (e.g., an employee record) is not found in the database.
   - **Exception Handling:** Utilizes @ControllerAdvice to handle exceptions globally and provide consistent error responses across the application.

7. **Configuration:**

   - **application.properties:** Contains configuration settings for database connections, JPA settings, and application port.

Testing:

The application includes unit testing for various layers:

- **Repository Layer:** Tests for CRUD operations and custom queries.
- **Service Layer:** Tests for service methods and business logic.
- **Controller Layer:** Tests for REST endpoints and request handling.

5. Testing on Repository Layer

6.1 Repository Layer

The repository layer in your Spring Boot project acts as a bridge between your application and the database. It is responsible for managing data access, storage, and retrieval operations. In this project, the repository layer uses Spring Data JPA to interact with an Oracle database, which simplifies data access by providing a set of high-level abstractions.

Repository of mPack:

```
package com.example.mPack.employee.repository;
import com.example.mPack.employee.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import java.util.List;
10 usages
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    Employee findByEmail (String email);

    @Query(value = "select e from Employee e where e.mobile=:mobile")
    Employee findByMobile (@Param("mobile")String mobile);
    6 usages
    List<Employee> findAllByDesignation(String designation);

    List<Employee> findAllByDepartment(String department);
    6 usages
    List<Employee> findAllByDivision(String division);
    5 usages
    List<Employee> findAllByBloodGroup (String bloodGroup);
    6 usages
    @Query (value = "select e from Employee e where e.division=:division " +
            "and e.department=:department")
    List<Employee> findByDivisionDepartment (@Param("department") String department,
                                             @Param("division") String division);

}
```

6.2 Test using @SpringBootTest

The **@SpringBootTest** annotation is a key component of the Spring Boot testing framework, and is used to set up an application context for testing purposes. It is designed to load the complete Spring application context, making it suitable for both unit and integration testing.
When we use **@SpringBootTest**

- **Full Application Context:** It loads the entire Spring application context, making it suitable for testing.
- **Real Database Interaction:** Tests interact with the actual database, ensuring that the data access layer works correctly with the database. Suppose, when an object is saved, it is permanently stored in the database, validating the persistence operations.

Necessary annotations

@Autowired: Injects dependencies into a class. In testing, it's commonly used to provide instances of beans or other components into the test class.

@BeforeEach: Runs a method before each test case execution. It's useful for setting up common test data or state.

@Test: Marks a method as a test case. The method will be executed by the testing framework as part of the test suite.

@SpringBootTest: Loads the full application context for integration tests. It provides a way to test the application with a real Spring context, ensuring that configurations and beans are loaded as they would be in production.

Let's start with test class

```
package com.example.mPack.employee.repository;
import com.example.mPack.employee.entity.Employee;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import java.util.List;
import java.util.Optional;
import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest
class EmployeeRepositoryTest {

    @Autowired
    EmployeeRepository employeeRepository;
    private Employee employee1;
    private Employee employee2;

    @BeforeEach
```

```
    public void setup() {
        employee1 = Employee.builder()
                .name("Bijoy Ahmed")
                .email("bijoy@gmail.com")
                .designation("Associate Engineer")
                .department("JAVA DEV. PLATFORMS")
                .division("SOFTWARE ENGINEERING")
                .mobile("01711289334")
                .salary(60000L)
                .address("Rajshahi")
                .bloodGroup("B+")
                .build();

        employee2 = Employee.builder()
                .name("Wakil Ahmed")
                .email("wakil@gmail.com")
                .designation("Data Analyst")
                .department("Analytics")
                .division("Research")
                .mobile("01722334445")
                .salary(50000L)
                .address("Dhaka")
                .bloodGroup("O+")
                .build();
    }
```

Setup Method:
The setup method initializes the test data before each test case. This ensures that we have consistent data for our tests.

6.2.1 Test Cases

TC 1: saveEmployeeTest

Purpose: Verify that the Employee object is correctly saved to the database using the employeeRepository.

```
@Test
void saveEmployeeTest() {
    Employee response = employeeRepository.save(employee1);
    assertNotNull(response.getId(), message: "The saved employee's ID should not be null");
    assertEquals( expected: "Bijoy Ahmed", response.getName());
    assertEquals( expected: "bijoy@gmail.com", response.getEmail());
    assertEquals( expected: "01711289334", response.getMobile());
    System.out.println(response);
}
```

Output:

Employee(id=3052, name=Bijoy Ahmed, email=bijoy@gmail.com, designation=Associate Engineer, department=JAVA DEV. PLATFORMS, division=SOFTWARE ENGINEERING, mobile=01711289334, salary=60000, address=Rajshahi, bloodGroup=B+)

TC 2: findByIdTest

Purpose: verify that an Employee object can be correctly retrieved from the database by its ID.

```
@Test
void findByIdTest() {
    employeeRepository.save(employee2);
    Employee foundEmployee = employeeRepository.findById(employee2.getId()).orElse( other: null);
    assertNotNull(foundEmployee, message: "The employee should be found by ID");
    assertEquals( expected: "Wakil Ahmed", foundEmployee.getName());
    assertNotNull(foundEmployee.getId());
    System.out.println("Find by ID test successful");
    System.out.println(foundEmployee);
}
```

Output:

Find by ID test successful

Employee(id=3102, name=Wakil Ahmed, email=wakil@gmail.com, designation=Data Analyst, department=Analytics, division=Research, mobile=01722334445, salary=50000, address=Dhaka, bloodGroup=O+)

Note: It returns an Optional<Employee>, which is a container that may or may not contain a non-null Employee object. **.orElse(null)** this part of the code handles the case where the findById method does not find an Employee with the given ID

TC 3: findByIdNotFound

Purpose : Verify that the findById method returns an empty Optional when no Employee exists with the specified ID.

```
@Test
public void findByIdNotFound() {
    Optional<Employee> foundEmployee = employeeRepository.findById(999L);
    assertFalse(foundEmployee.isPresent());
}
```

Output: No employee found

TC 4: findAllTest

Purpose: Verify that the **findAll** method retrieves all employee records and returns a non-empty list with the correct number of employees.

```java
@Test
void findAllTest() {
    List<Employee> employees = employeeRepository.findAll();
    assertFalse(employees.isEmpty(), message: "The employee list should not be empty");
    assertTrue( condition: employees.size()>5, message: "The employee list should contain at least 2 employees");
    assertEquals(employees.size(), actual: 6);
    System.out.println(employees.size());
    System.out.println("Find all employees test successful");
}
```

Output:

```
6
Find all employees test successful
```

TC 5: findByEmailTest

Purpose: This test case verifies that an Employee object can be correctly retrieved from the database using its email address.

```java
@Test
void findByEmailTest() {
    Employee foundEmployee = employeeRepository.findByEmail("bijoy@gmail.com");
    assertNotNull(foundEmployee);
    assertNotNull(foundEmployee.getId());
    assertEquals( expected: "Bijoy Ahmed", foundEmployee.getName());
    assertEquals( expected: "Rajshahi", foundEmployee.getAddress());
    System.out.println("Find by email test successful");
    System.out.println(foundEmployee);
}
```

Output:
Find by email test successful

Employee(id=3052, name=Bijoy Ahmed, email=bijoy@gmail.com, designation=Associate Engineer, department=JAVA DEV. PLATFORMS, division=SOFTWARE ENGINEERING, mobile=01711289334, salary=60000, address=Rajshahi, bloodGroup=B+)

TC 6: findByEmailNotFound

Purpose: Verify that when an Employee object with a non-existent email address is queried, the result should be null.

```java
@Test public void findByEmailNotFound() {
    Employee foundEmployee = employeeRepository.findByEmail("none@gmail.com");
    assertNull(foundEmployee);
}
```

Output: Null

TC 7: findByMobileTest

Purpose: Verify that an Employee object can be correctly retrieved from the database using a mobile number.

```
@Test
void findByMobileTest() {
    Employee foundEmployee = employeeRepository.findByMobile("01722334445");
    assertNotNull(foundEmployee);
    assertNotNull(foundEmployee.getId());
    assertEquals( expected: "Wakil Ahmed", foundEmployee.getName());
    assertEquals(employee2.getDepartment(), foundEmployee.getDepartment());
    System.out.println("Find by mobile test successful");
    System.out.println(foundEmployee);
}
```

Output:

Find by mobile test successful

Employee(id=152, name=Wakil Ahmed, email=wakil@email.com, designation=Data Analyst, department=Analytics, division=Research, mobile=01722334445, salary=50000, address=Dhaka, bloodGroup=O+)

TC 8: findByMobileNotFound

Purpose: This test case verifies that no `Employee` object is returned when querying the database with a mobile number that does not exist.

```
@Test public void findByMobileNotFound() {
    Employee foundEmployee = employeeRepository.findByMobile("0000000000");
    assertNull(foundEmployee);
}
```

Output: Null

TC 9: findAllByDesignationTest

Purpose: Verify that the findAllByDesignation method correctly retrieves all Employee objects with a specified designation.

```
@Test
void findAllByDesignationTest() {
    List<Employee> softwareEngineers = employeeRepository.findAllByDesignation("Associate Engineer");
    assertFalse(softwareEngineers.isEmpty());
    assertEquals( expected: 1, softwareEngineers.size(), message: "Expected 1 Software Engineer");
    assertEquals( expected: "Bijoy Ahmed", softwareEngineers.get(0).getName());
    assertEquals(employee1.getEmail(), softwareEngineers.get(0).getEmail() );
    System.out.println("Find by designation test successful");
}
```

Output:

Find by designation test successful

TC 10: findAllByDepartmentTest

Purpose: Verify that the findAllByDepartment method correctly retrieves all Employee objects for a specified department.

```
@Test
void findAllByDepartmentTest() {
    List<Employee> responses = employeeRepository.findAllByDepartment("JAVA DEV. PLATFORMS");
    assertFalse(responses.isEmpty());
    assertEquals( expected: 1, responses.size());
    //There are one employee in the table of IT department
    assertEquals( expected: "Bijoy Ahmed", responses.get(0).getName());
    assertEquals(employee1.getEmail(),responses.get(0).getEmail());
    System.out.println(responses.get(0));
    System.out.println("Find by department test successful");
}
```

Output:

Employee(id=3052, name=Bijoy Ahmed, email=bijoy@gmail.com, designation=Associate Engineer, department=JAVA DEV. PLATFORMS, division=SOFTWARE ENGINEERING, mobile=01711289334, salary=60000, address=Rajshahi, bloodGroup=B+)

Find by department test successful

TC 11: findAllByDivisionTest

Purpose: Verify the findAllByDivision method retrieves all Employee objects correctly for the specified division, "SOFTWARE ENGINEERING".

```
@Test
void findAllByDivisionTest() {
    List<Employee> responses = employeeRepository.findAllByDivision("SOFTWARE ENGINEERING");
    assertFalse(responses.isEmpty());
    assertEquals( expected: 1, responses.size());
    assertEquals( expected: "Bijoy Ahmed", responses.get(0).getName());
    assertEquals(employee1.getMobile(), responses.get(0).getMobile());
    assertEquals(employee1.getSalary(), responses.get(0).getSalary());
}
```

Output : No Output

TC 12: findAllByBloodGroupTest

Purpose: Verify that the findAllByBloodGroup method retrieves all Employee objects with the specified blood group, "B+".

```
@Test
void findAllByBloodGroupTest() {
    List<Employee> response= employeeRepository.findAllByBloodGroup("B+");
    assertFalse(response.isEmpty());
    assertEquals( expected: 1, response.size());
    System.out.println(response);
}
```

Output:

[Employee(id=3052, name=Bijoy Ahmed, email=bijoy@gmail.com, designation=Associate Engineer, department=JAVA DEV. PLATFORMS, division=SOFTWARE ENGINEERING, mobile=01711289334, salary=60000, address=Rajshahi, bloodGroup=B+)]

TC 13: findByDivisionDepartmentTest

Purpose: This test case verifies that the findByDivisionDepartment method retrieves all Employee objects that match the specified division and department criteria, in this case, "IT" and "Development".

```
@Test
void findByDivisionDepartmentTest() {
    List<Employee> employees = employeeRepository.findByDivisionDepartment( division: "Research",  department: "Analytics");
    assertFalse(employees.isEmpty());
    System.out.println(employees);

}
```

Output:

[Employee(id=3102, name=Wakil Ahmed, email=wakil@gmail.com, designation=Data Analyst, department=Analytics, division=Research, mobile=01722334445, salary=50000, address=Dhaka, bloodGroup=O+)]

6.2.2 Testing Table-1

Test Case ID	Test Case Title	Input Data	Expected Output	Actual Output	Satus	Remarks
TC 1	saveEmployeeTest	Employee object	Employee object	Same as Expected	**Pass**	Successfully saved and printed employee details.
TC 2	findByIdTest	ID	Statement and employee object	Same as Expected	**Pass**	Successfully retrieved employee by ID.
TC 3	findByIdNotFound	Non-existing ID	No employee found	Same as Expected	**Pass**	Employee not found by ID.
TC 4	findAllTest	None	Employee list size, statements	Same as Expected	**Pass**	Successfully retrieved all employees.
TC 5	findByEmailTest	Email	Statement and employee object	Same as Expected	**Pass**	Successfully retrieved employee by email.
TC 6	findByEmailNotFound	Non-existing email	Null	Same as Expected	**Pass**	Employee not found by email.
TC 7	findByMobileTest	Mobile number	Statement and employee object	Same as Expected	**Pass**	Successfully retrieved employee by mobile number.
TC 8	findByMobileNotFound	Non-existing mobile number	No employee found	Same as Expected	**Pass**	Employee not found by mobile number.
TC 9	findAllByDesignationTest	Designation	Statements	Same as Expected	**Pass**	Successfully retrieved employees by designation.
TC 10	findAllByDepartmentTest	Department	One Employee objects, statements	Same as Expected	**Pass**	Successfully retrieved employees by department.
TC 11	findAllByDivisionTest	Division	No output	Same as Expected	**Pass**	Successfully retrieved employees by division.
TC 12	findAllByBloodGroup Test	Blood Group	One Employee	Same as Expected	**Pass**	Successfully retrieved

			objects			employees by blood group.
TC 13	findAllByBloodGroup Test	Division and Department	One Employee objects	Same as Expected	**Pass**	Successfully retrieved employees by department and division.

6.3 Test using @DataJpaTest

The @DataJpaTest annotation is a specialized Spring Boot test annotation designed for testing JPA repositories. It sets up a minimal Spring application context that focuses solely on the JPA layer, providing an efficient and isolated environment for testing persistence-related components.

Key Features

1. **Focused Context**: @DataJpaTest limits the Spring context to JPA components only, ensuring that tests are faster and more focused by avoiding the overhead of unnecessary beans and configurations.
2. **In-Memory Database**: By default, @DataJpaTest uses an in-memory database (such as H2) to ensure that any data operations are temporary and isolated from the real database. This prevents any changes to the actual database and ensures a clean slate for each test.
3. **Automatic Transaction Management**: Each test method runs within a transaction that is automatically rolled back after the test completes. This ensures that the test environment remains clean and consistent, with no residual data affecting other tests.
4. **Pre-Configured Entities**: The annotation provides pre-configured EntityManager and TestEntityManager instances, simplifying interactions with JPA entities and allowing for straightforward repository testing.

Benefits

1. **Efficient Testing**: By focusing only on JPA components, @DataJpaTest speeds up test execution and enhances test accuracy.
2. **Isolated Testing Environment**: Utilizes an in-memory database to avoid altering the actual database and to ensure that each test runs in isolation.
3. **Consistent State**: Automatic rollback of transactions after each test method guarantees that the test environment remains consistent and unaffected by prior tests.

How It Works

- **In-Memory Database**: Automatically configures an in-memory database (e.g., H2) for testing, ensuring that all database interactions are temporary and do not impact the real database.

- **Transaction Rollback**: Runs each test method within a transaction that is rolled back after completion, ensuring that no changes persist beyond the scope of the test.
- **Reduced Application Context**: Loads only JPA-related components, providing a streamlined and efficient testing environment compared to the broader context loaded with @SpringBootTest.

Let's start with test class

In the previous section, where I tested the repository layer using @SpringBootTest, I explained each test case individually. Now, I will include the entire code for testing with @DataJpaTest as the test cases are almost identical. This approach avoids redundant explanations and provides a comprehensive view of the test implementation using @DataJpaTest..

```
package com.example.mPack.employee.repository;
import com.example.mPack.employee.entity.Employee;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import java.util.List;
import java.util.Optional;
import static org.junit.jupiter.api.Assertions.*;

@DataJpaTest
public class EmployeeRepositoryTestDataJpa {
    @Autowired
    EmployeeRepository employeeRepository;
    private Employee employee1;
    private Employee employee2;
    @BeforeEach
    public void setup() {
        employee1 = Employee.builder()
                .name("Bijoy Ahmed")
                .email("bijoy@gmail.com")
                .designation("Associate Engineer")
                .department("JAVA DEV. PLATFORMS")
                .division("SOFTWARE ENGINEERING")
                .mobile("01711289334")
                .salary(60000L)
                .address("Rajshahi")
                .bloodGroup("B+")
                .build();
        employee2 = Employee.builder()
                .name("Wakil Ahmed")
                .email("wakil@gmail.com")
                .designation("Data Analyst")
                .department("Analytics")
                .division("Research")
                .mobile("01722334445")
                .salary(50000L)
                .address("Dhaka")
                .bloodGroup("O+")
                .build();
        employeeRepository.save(employee1);
```

```
        employeeRepository.save(employee2);
    }
```

All remaining aspects should be consistent with what we discussed previously regarding testing the repository layer using @SpringBootTest.

6.4 Test using Mockito

Mocking is a testing technique where mock objects, created by frameworks like Mockito or EasyMock, simulate real objects. These mocks provide predefined outputs for specific inputs, allowing us to test components in isolation by replacing real dependencies with controlled, dummy behavior.

Mockito is a Java-based framework for unit testing that simplifies creating mock objects. It uses the Java Reflection API to generate mocks for testing purposes. An open-source tool under the MIT License, Mockito helps isolate tests by mocking external dependencies, making test code cleaner and more maintainable. It integrates seamlessly with JUnit and TestNG.**Mocking**

Mocking involves creating objects that simulate real objects for testing purposes. It uses mock objects to provide predefined outputs for specific inputs, helping isolate and test components without relying on actual implementations.

Necessary Annotations

@Mock: This annotation is used when we have to create and inject mock objects. It is used in testing frameworks like Mockito. Whenever a real object is to be stimulated, a mock object is used because it does so in a controlled manner. To create the mock instance we use this annotation.

@ExtendWith(MockitoExtension.class): This annotation integrates Mockito with JUnit 5, enabling features like mocking and dependency injection. It processes Mockito annotations such as @Mock, @InjectMocks, and @Spy, ensuring proper initialization of mocks before tests run. This reduces boilerplate code, simplifies setup, and supports advanced Mockito features for clearer and more reliable test configurations.

Benefits of Mockito

1. **Automatic Mock Creation**: Mockito eliminates the need for manually creating mock objects, streamlining test development.
2. **Safe Refactoring**: Tests remain unaffected by changes in method names or parameter lists, as mocks are dynamically created at runtime.

3. **Exception Handling**: Mockito provides robust exception handling by utilizing stack traces to pinpoint the root cause of issues.
4. **Annotation Support**: It simplifies mock creation through annotations such as @Mock, reducing boilerplate code.
5. **Call Order Verification**: Mockito allows for the verification of method call order, ensuring proper sequence in interactions.

Let's start with test class

Mocking with Mockito

When using Mockito to mock repository methods, the **when** and **thenReturn** methods are essential for defining mock behavior:

- **when**: This method specifies which method call and arguments are being mocked. It sets up the conditions under which the mock will respond.
- **thenReturn**: This method defines the value that the mock should return when the specified method (as defined in when) is invoked. It allows you to simulate various test scenarios.

By using these methods, we can control and simulate different conditions in our tests, ensuring that our code behaves as expected without relying on the actual database.

Following the entire test class using Mockito,

```
package com.example.mPack.employee.repository;
import com.example.mPack.employee.entity.Employee;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import java.util.List;
import java.util.Optional;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
public class EmployeeRepositoryTestMockito {
    26 usages
    @Mock
    EmployeeRepository employeeRepository;
    14 usages
    private Employee employee1;
    5 usages
    private Employee employee2;
```

```java
@BeforeEach
public void setup() {
    employee1 = Employee.builder()
            .id(1L).name("Bijoy Ahmed")
            .email("bijoy@gmail.com")
            .designation("Associate Engineer")
            .department("JAVA DEV. PLATFORMS")
            .division("SOFTWARE ENGINEERING")
            .mobile("01711289334")
            .salary(60000L)
            .address("Rajshahi")
            .bloodGroup("B+")
            .build();
    employee2 = Employee.builder()
            .id(2L).name("Wakil Ahmed")
            .email("wakil@gmail.com")
            .designation("Data Analyst")
            .department("Analytics")
            .division("Research")
            .mobile("01722334445")
            .salary(50000L)
            .address("Dhaka")
            .bloodGroup("O+")
            .build();
}
```

```java
@Test
void saveEmployeeTest() {
    when(employeeRepository.save(employee1)).thenReturn(employee1);

    Employee response = employeeRepository.save(employee1);
    assertNotNull(response, message: "The saved employee should not be null");
    assertNotNull(response.getId(), message: "The saved employee's ID should not be null");
    assertEquals(employee1.getEmail(), response.getEmail());
    assertEquals(employee1.getName(), response.getName());
    assertEquals(employee1.getMobile(), response.getMobile());
}

@Test
void findByIdTest() {
    when(employeeRepository.findById(2L)).thenReturn(Optional.of(employee2));

    Employee foundEmployee = employeeRepository.findById(2L).orElse( other: null);
    assertNotNull(foundEmployee, message: "The employee should be found by ID");
    assertEquals( expected: "Wakil Ahmed", foundEmployee.getName());
}
@Test
void testFindByIdNotFound() {
    when(employeeRepository.findById(999L)).thenReturn( t: Optional.empty());
    Optional<Employee> foundEmployee = employeeRepository.findById(999L);
    assertFalse(foundEmployee.isPresent(), message: "Employee with ID 999 should not be found");
}

@Test
void findAllTest() {
    when(employeeRepository.findAll()).thenReturn(List.of(employee1, employee2));

    List<Employee> employees = employeeRepository.findAll();
    assertFalse(employees.isEmpty(), message: "The employee list should not be empty");
    assertTrue( condition: employees.size() > 1, message: "The employee list should contain at least 2 employees");
}
```

```java
@Test
void findByEmailTest() {
    when(employeeRepository.findByEmail("bijoy@gmail.com")).thenReturn(employee1);

    Employee foundEmployee = employeeRepository.findByEmail("bijoy@gmail.com");
    assertNotNull(foundEmployee, message: "Employee with the given email should be found");
    assertEquals( expected: "Bijoy Ahmed", foundEmployee.getName());
    assertEquals( expected: "Rajshahi", foundEmployee.getAddress());
}

@Test
void testFindByEmailNotFound() {
    when(employeeRepository.findByEmail("none@gmail.com")).thenReturn( t: null);

    Employee foundEmployee = employeeRepository.findByEmail("none@gmail.com");
    assertNull(foundEmployee, message: "Employee with the given email should not be found");
}
@Test
void findByMobileTest() {
    when(employeeRepository.findByMobile("01722334445")).thenReturn(employee2);

    Employee foundEmployee = employeeRepository.findByMobile("01722334445");
    assertNotNull(foundEmployee, message: "Employee with the given mobile should be found");
    assertEquals( expected: "Wakil Ahmed", foundEmployee.getName());
    assertEquals(employee2.getDepartment(), foundEmployee.getDepartment());
}

@Test
void testFindByMobileNotFound() {
    when(employeeRepository.findByMobile("0000000000")).thenReturn( t: null);

    Employee foundEmployee = employeeRepository.findByMobile("0000000000");
    assertNull(foundEmployee, message: "Employee with the given mobile should not be found");
}


@Test
void findAllByDesignationTest() {
    when(employeeRepository.findAllByDesignation("Associate Engineer")).thenReturn(List.of(employee1));

    List<Employee> associateEngineers = employeeRepository.findAllByDesignation("Associate Engineer");
    assertFalse(associateEngineers.isEmpty());
    assertEquals( expected: 1, associateEngineers.size(), message: "Expected 1 Associate Engineer");
    assertEquals( expected: "Bijoy Ahmed", associateEngineers.get(0).getName());
}

@Test
void findAllByDepartmentTest() {
    when(employeeRepository.findAllByDepartment("JAVA DEV. PLATFORMS")).thenReturn(List.of(employee1));

    List<Employee> responses = employeeRepository.findAllByDepartment("JAVA DEV. PLATFORMS");
    assertFalse(responses.isEmpty());
    assertEquals( expected: 1, responses.size());
    assertEquals( expected: "Bijoy Ahmed", responses.get(0).getName());
}
```

```java
@Test
void findAllByDivisionTest() {
    when(employeeRepository.findAllByDivision("SOFTWARE ENGINEERING")).thenReturn(List.of(employee1));

    List<Employee> responses = employeeRepository.findAllByDivision("SOFTWARE ENGINEERING");
    assertFalse(responses.isEmpty());
    assertEquals( expected: 1, responses.size());
    assertEquals( expected: "Bijoy Ahmed", responses.get(0).getName());
}

@Test
void findAllByBloodGroupTest() {
    when(employeeRepository.findAllByBloodGroup("B+")).thenReturn(List.of(employee1));

    List<Employee> response = employeeRepository.findAllByBloodGroup("B+");
    assertFalse(response.isEmpty());
    assertEquals( expected: 1, response.size());
    assertEquals( expected: "Bijoy Ahmed", response.get(0).getName());
}
```

```java
@Test
void findByDivisionDepartmentTest() {
    when(employeeRepository.findByDivisionDepartment( division: "SOFTWARE ENGINEERING",  department: "JAVA DEV. PLATFORMS")).
            thenReturn(List.of(employee1));
    List<Employee> employees = employeeRepository.findByDivisionDepartment( division: "SOFTWARE ENGINEERING",  department: "JAVA DEV. PLATFORMS");
    assertFalse(employees.isEmpty());
    assertEquals( expected: 1, employees.size());
    assertEquals( expected: "Bijoy Ahmed", employees.get(0).getName());
}
```

}

6. Testing on Service Layer

7.1 Service Layer

The service layer is central to business logic in an application. It bridges the controller layer, which handles HTTP requests, and the repository layer, which interacts with the database. This layer defines core functionality, such as data validation, transaction management, and complex calculations. It processes and applies business rules to data, ensuring that operations are executed consistently and correctly.

Testing the service layer verifies that the business logic is correctly implemented and that the service methods interact appropriately with the repository and other components.

Key Components:

- **Service Interface:** Specifies the methods for performing domain-specific operations, defining the expected behavior and interactions without implementing the logic.
- **ServiceImpl Class:** Provides the concrete implementation of the service interface. It contains the business logic, executes the defined operations, and manages interactions with the repository layer for data handling.

7.2 Test using Mockito

Mockito is a popular mocking framework used for unit testing in Java. Using mock objects in unit testing allows us to replace real dependencies (like services or repositories) with controlled, simulated versions. This means we can dictate exactly how these mock dependencies should behave, such as returning specific values or throwing exceptions when their methods are called. By doing this, we can isolate the part of our code we're testing from the rest of the system, ensuring that your tests focus solely on the code under test and not on its dependencies. This approach helps us test different scenarios more easily and makes our tests more reliable and focused.

Necessary Annotations

@ExtendWith(MockitoExtension.class) This annotation is used in JUnit 5 tests to automatically initialize Mockito mocks and handle dependency injection, which streamlines test setup by eliminating the need for manual mock initialization and configuration. If we don't this annotation then we need to initialize Mockito mocks manually in the @BeforeEach annoted method

MockitoAnnotations.openMocks(this);

@MockBean : This annotation is used in Spring Boot tests to swap out a real bean with a mock version. It's helpful when we want to replace certain dependencies of a Spring-managed bean with mock objects while running tests with @SpringBootTest. This lets us simulate how the bean interacts with its dependencies without using the real implementations, making it easier to test specific parts of our application in isolation.

@Mock : This annotation is used to create a mock instance of a class or interface using Mockito. It is used in unit tests to define and manage mock objects, which simulate the behavior of real objects. @Mock allows us to set up specific behaviors and responses for these mock objects, enabling us to test how the class under test interacts with its dependencies without involving the actual implementations of those dependencies.

@InjectMocks : This annotation is used to automatically inject mock instances (created with @Mock or Mockito.mock()) into the class under test. Mockito handles the injection of these mocks into the class by matching the types and field names of the mocks with the corresponding

fields in the class under test. This annotation simplifies the process of injecting dependencies into the class under test, making it easier to test the class in isolation with its dependencies mocked.

@ActiveProfiles("test") : This is an annotation in Spring Boot that activates the specified profile ("test") when running the application or tests. This allows you to load specific configurations, beans, and properties tailored for the "test" environment. It's commonly used in testing to isolate the test environment from production settings.

In Spring Boot applications, testing the service layer can be approached in different ways depending on the level of integration and the specific context of the test. The two primary methods for testing with Mockito are:

1. Using @SpringBootTest and @MockBean

The approach for the service layer is primarily designed for integration testing. It loads the entire Spring Boot application context, ensuring that all configurations and beans are fully initialized and properly wired. This means that our tests run within the complete environment of our application, which is beneficial for verifying how different layers of our application work together. However, when used for unit testing, this approach might not be ideal. Unit tests typically focus on testing a single class or method in isolation, without the need to load the entire application context. Loading the full context can make unit tests slower and more complex.

Each approach has its own set of advantages and disadvantages.

Advantages:

- **Context Loading:** Loads the entire Spring application context, which includes all bean configurations and dependencies. This is beneficial for verifying the service's integration with other components and configurations.
- **Real Environment:** Tests how the service layer operates with actual Spring components and real configurations, providing a realistic testing scenario.
- **Integration Testing:** Validates the service layer in the context of the full Spring Boot application, ensuring that all Spring components are correctly configured and interact as expected.

Disadvantages:

- **Slower Execution:** Due to the need to load the full application context, tests can be slower compared to unit tests.
- **Complexity:** May be considered overkill for simple unit testing scenarios where the goal is to test a single service in isolation without the full application context.

Note: When we use this approach we can use either the interface (EmployeeService) or the concrete implementation class (EmployeeServiceImpl).

2. Using @Mock and @InjectMocks

This approach is primarily intended for unit testing, where we focus on testing a single class (such as a service) in isolation. In this approach, we do not load the entire Spring application context, which helps to keep the tests fast and straightforward. Instead, we create mock versions of the class's dependencies using @Mock, and we inject these mocks into the class being tested using @InjectMocks. This way, we can test the class's behavior without relying on its actual dependencies, ensuring that the test is isolated and focused solely on the logic of the class itself.

Advantages:

- **Unit Testing:** Allows for focused testing of a specific class in isolation from the rest of the application. It is useful for testing the internal logic of the class without the need for a full application context.
- **Faster Execution:** Since it avoids loading the entire Spring context, tests run faster compared to integration tests.
- **Isolation:** Ensures that the test focuses solely on the class under test, using mocks to simulate interactions with its dependencies.

Disadvantages:

- **No Application Context:** Does not verify how the service interacts with other Spring components or the application context. It only tests the logic within the class itself.

Note: In this approach, we directly mock the dependencies of EmployeeServiceImpl using @Mock and inject these mocks into the EmployeeServiceImpl instance using @InjectMocks. We do not use the EmployeeService interface in this setup. Instead, we work with the concrete implementation (EmployeeServiceImpl) to test its behavior with the mocked dependencies

ServiceImpl class of mPack:

```
@Service
public class EmployeeServiceImpl implements EmployeeService {
    17 usages
    private EmployeeRepository employeeRepository;
    3 usages
    private ModelMapper mapper;

    public EmployeeServiceImpl(EmployeeRepository employeeRepository, ModelMapper mapper) {
        this.employeeRepository = employeeRepository;
        this.mapper=mapper;
    }
    5 usages
    @Override
    public CustomizeEmpDto createEmployee(EmployeeDto employeeDto) {
        Employee employee=employeeRepository.save(mapToEntity(employeeDto));
        CustomizeEmpDto customizeEmpDto=new CustomizeEmpDto(mapToDto(employee));
        return customizeEmpDto;
    }
```

```
public List<EmployeeDto> getAllEmployee() {                                    ⚠11 ⚠2
    List<Employee> employees= employeeRepository.findAll();  //collect all the employees
    return employees.stream().map(employee -> mapToDto(employee)).collect(Collectors.toList());
}
7 usages
@Override
public EmployeeDto getEmployeeById(Long id) {
    Employee employee=employeeRepository.findById(id).orElseThrow(()-> new ResourceNotFoundException("employee", "id",id));
    return mapToDto(employee);
}
5 usages
@Override
public EmployeeDto updateEmployee(EmployeeDto employeeDto, Long id) {
    Employee employee= employeeRepository.findById(id).orElseThrow(()-> new ResourceNotFoundException("employee", "id",id));

    employee.setName(employeeDto.getName());
    employee.setAddress(employeeDto.getAddress());
    employee.setDesignation(employeeDto.getDesignation());
    employee.setDepartment(employeeDto.getDepartment());
    employee.setDivision(employeeDto.getDivision());
    employee.setEmail(employeeDto.getEmail());
    employee.setSalary(employeeDto.getSalary());
    employee.setBloodGroup(employeeDto.getBloodGroup());
    employee.setMobile(employeeDto.getMobile());

    Employee updatedEmployee=employeeRepository.save(employee);
    return mapToDto(updatedEmployee);
}
```

```java
public ResponseEmpDto updateEmployeeByPatch(EmployeeDto employeeDto, Long id) {
    Employee employee= employeeRepository.findById(id).orElseThrow(()-> new ResourceNotFoundException("employee", "id",id));
    ResponseEmpDto responseEmpDto=new ResponseEmpDto();
    employee.setName(employeeDto.getName());
    employee.setAddress(employeeDto.getAddress());
    employee.setDesignation(employeeDto.getDesignation());
    employee.setDepartment(employeeDto.getDepartment());
    employee.setDivision(employeeDto.getDivision());
    employee.setEmail(employeeDto.getEmail());
    employee.setSalary(employeeDto.getSalary());
    employee.setBloodGroup(employeeDto.getBloodGroup());
    employee.setMobile(employeeDto.getMobile());
    try {
        Employee updateEmployee = employeeRepository.save(employee);
        responseEmpDto.setResponseCode(2);
        responseEmpDto.setResponseMessage("Updated successfully");
    }
    catch (Exception exce){
        exce.printStackTrace();
    }
    return responseEmpDto;
}
```

```java
@Override
public ResponseEmpDto deleteEmployee(Long id) {
    Employee employee= employeeRepository.findById(id).orElseThrow(()-> new ResourceNotFoundException("employee", "id",id));
    ResponseEmpDto responseEmpDto=new ResponseEmpDto();
    try {
        employeeRepository.delete(employee);
        responseEmpDto.setResponseCode(3);
        responseEmpDto.setResponseMessage("Deleted successfully");
    } catch (Exception exce){
        exce.printStackTrace();
    }
    return responseEmpDto;}
@Override
public EmployeeDto getByEmail(String email) { return mapToDto(employeeRepository.findByEmail(email)); }

@Override
public EmployeeDto getByMobile(String mobile) {
    return mapToDto(employeeRepository.findByMobile(mobile));
}
@Override
public List<EmployeeDto> getByDesignation(String designation) {
    List<Employee> employees=employeeRepository.findAllByDesignation(designation);
    List<EmployeeDto> employeeDtos= employees.stream().map(employee -> mapToDto(employee)).collect(Collectors.toList());
    return employeeDtos;
}
5 usages
@Override
public List<EmployeeDto> getByDivision(String division) {
    List<Employee> employees =employeeRepository.findAllByDivision(division);
    return employees.stream().map(employee -> mapToDto(employee)).collect(Collectors.toList());
}
5 usages
@Override
public List<EmployeeDto> getByDepartment(String department) {
    List<Employee> employees=employeeRepository.findAllByDepartment(department);
    return employees.stream().map(employee -> mapToDto(employee)).collect(Collectors.toList());
}
```

```
public List<CustomizeEmpDto> getByBloodGroup(String bloodGroup) {
    List<Employee> employees =employeeRepository.findAllByBloodGroup(bloodGroup);
    return employees.stream()
            .map(employee -> new CustomizeEmpDto(
                    employee.getId(),employee.getName(),
                    employee.getEmail(),
                    employee.getMobile())).
            collect(Collectors.toList()); }
5 usages
@Override
public List<EmployeeDto> getByDepartmentDivision(String department, String division) {
    List<Employee> responses=employeeRepository.findByDivisionDepartment(department,division);
    return responses.stream().map(this::mapToDto).collect(Collectors.toList());
}
// Convert Entity into DTO
10 usages
private EmployeeDto mapToDto(Employee employee){
    return mapper.map(employee, EmployeeDto.class);
}
//Convert DTO into Entity
1 usage
private Employee mapToEntity(EmployeeDto employeeDto){
    return mapper.map(employeeDto, Employee.class);
}
```

}

Let's start with test class

In this context, we are utilizing the second approach for testing the service layer.

```
@ExtendWith(MockitoExtension.class)
class EmployeeServiceImplTestMockito2 {
    @Mock
    private EmployeeRepository employeeRepository;
    @Mock
    private ModelMapper mapper;
    @InjectMocks
    private EmployeeServiceImpl employeeService;
    private Employee employee1;
    private Employee employee2;
    private EmployeeDto employeeDto1;
    private EmployeeDto employeeDto2;
    private List<Employee> employeeList;
    private List<EmployeeDto> employeeDtoList;

    @BeforeEach
    void setUp() {
        // Initialize employees
        employee1 = new Employee();
        employee1.setId(101L);
        employee1.setName("Mamun Hossain");
        employee1.setEmail("mamun@gmail.com");
        employee1.setDesignation("Intern");
        employee1.setDepartment("iStelar");
        employee1.setDivision("Banking");
        employee1.setMobile("01820702329");
        employee1.setAddress("Dhaka");
        employee1.setSalary(10000L);
        employee1.setBloodGroup("A+");
```

```
        employee2 = new Employee();
        employee2.setId(102L);
        employee2.setName("Nuh Salman");
        employee2.setEmail("nuhsalman@gmail.com");
        employee2.setDesignation("Intern");
        employee2.setDepartment("iStelar");
        employee2.setDivision("Banking");
        employee2.setMobile("01820702330");
        employee2.setAddress("Dhaka");
        employee2.setSalary(10000L);
        employee2.setBloodGroup("B+");

        employeeDto1 = new EmployeeDto();
        employeeDto1.setId(employee1.getId());
        employeeDto1.setName(employee1.getName());
        employeeDto1.setEmail(employee1.getEmail());
        employeeDto1.setDesignation(employee1.getDesignation());
        employeeDto1.setDepartment(employee1.getDepartment());
        employeeDto1.setDivision(employee1.getDivision());
        employeeDto1.setMobile(employee1.getMobile());
        employeeDto1.setAddress(employee1.getAddress());
        employeeDto1.setSalary(employee1.getSalary());
        employeeDto1.setBloodGroup(employee1.getBloodGroup());

        employeeDto2 = new EmployeeDto();
        employeeDto2.setId(employee2.getId());
        employeeDto2.setName(employee2.getName());
        employeeDto2.setEmail(employee2.getEmail());
        employeeDto2.setDesignation(employee2.getDesignation());
        employeeDto2.setDepartment(employee2.getDepartment());
        employeeDto2.setDivision(employee2.getDivision());
        employeeDto2.setMobile(employee2.getMobile());
        employeeDto2.setAddress(employee2.getAddress());
        employeeDto2.setSalary(employee2.getSalary());
        employeeDto2.setBloodGroup(employee2.getBloodGroup());

        employeeList = Arrays.asList(employee1, employee2);
        employeeDtoList = Arrays.asList(employeeDto1, employeeDto2);
    }
```

7.2.1 Test cases

TC 1: createEmployeeTest

Purpose: To verify that the createEmployee method of the EmployeeServiceImpl class correctly converts an EmployeeDto to an Employee, saves it to the repository, and returns a CustomizeEmpDto with the expected values.

```
@Test
void createEmployeeTest() {
    when(mapper.map(any(EmployeeDto.class), eq(Employee.class))).thenReturn(employee1);
    when(employeeRepository.save(any(Employee.class))).thenReturn(employee1);
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class))).thenReturn(employeeDto1);

    CustomizeEmpDto result = employeeService.createEmployee(employeeDto1);
    assertNotNull(result);
    assertEquals( expected: "Mamun Hossain", result.getName());
    assertEquals( expected: "mamun@gmail.com", result.getEmail());
    System.out.println(result);;
}
```

Mock Behavior

- When the mapper.map method is called with any EmployeeDto and Employee.class, it returns employee1.
- When the employeeRepository.save method is called with any Employee, it returns employee1.
- When the mapper.map method is called with any Employee and EmployeeDto.class, it returns employeeDto1.

Output:

CustomizeEmpDto(id=101, name=Mamun Hossain, email=mamun@gmail.com, mobile=01820702329)

To provide a clear understanding, I'll describe a when…then scenario once.

when(mapper.map(any(EmployeeDto.class),eq(Employee.class))).thenReturn(employee1);

when(...): Specifies what should be done when a certain method is called on the mock object.

mapper.map(...): The method being mocked. In this case, it's the map method of the mapper object.

any(EmployeeDto.class): A Mockito matcher that indicates any EmployeeDto object can be passed as the first argument.

eq(Employee.class): A Mockito matcher that specifies the exact class type for the second argument should be Employee.class.

thenReturn(employee1): Defines that when the map method is called with any EmployeeDto and the Employee.class type, it should return the employee1 object.

TC 2: getAllEmployeeTest

Purpose: Verifies that the service method getAllEmployee() returns a list of all employees correctly mapped to EmployeeDto objects.

```
@Test
void getAllEmployeeTest() {
    when(employeeRepository.findAll()).thenReturn(employeeList);
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class)))
            .thenReturn(employeeDto1, employeeDto2);
    List<EmployeeDto> result = employeeService.getAllEmployee();
    assertEquals( expected: 2, result.size());
    assertEquals( expected: "nuhsalman@gmail.com",result.get(1).getEmail());
    assertEquals( expected: "01820702329", result.get(0).getMobile());
    System.out.println(result);
}
```

Mock Behavior:

- When employeeRepository.findAll() is called, it returns employeeList.
- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns employeeDto1 for the first employee and employeeDto2 for the second employee.

Output: No output

TC 3: getEmployeeByIdTest

Purpose: Ensures that the getEmployeeById() method in the service returns the correct EmployeeDto when provided with a specific employee ID.

```
@Test
void getEmployeeByIdTest() {
    when(employeeRepository.findById(101L)).thenReturn(Optional.of(employee1));
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class))).thenReturn(employeeDto1);
    EmployeeDto result = employeeService.getEmployeeById(101L);
    assertNotNull(result);
    assertEquals(employeeDto1.getName(), result.getName());
    System.out.println(result.getDesignation());
}
```

Mock Behavior:

- When employeeRepository.findById(101L) is called, it returns Optional.of(employee1).
- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns employeeDto1.

Output: Intern

TC 4: getByEmailTest

Purpose: Verifies that the getByEmail() method in the service correctly returns the EmployeeDto for a given email address.

```
@Test
void getByEmailTest() {
    when(employeeRepository.findByEmail("mamun@gmail.com")).thenReturn(employee1);
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class))).thenReturn(employeeDto1);
    EmployeeDto result = employeeService.getByEmail("mamun@gmail.com");
    assertNotNull(result);
    assertEquals( expected: "mamun@gmail.com", result.getEmail());
}
```

Mock Behavior:

- When employeeRepository.findByEmail("mamun@gmail.com") is called, it returns employee1.
- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns employeeDto1.

Output: No Output

TC 5: getByMobileTest

Purpose: Verifies that the getByMobile() method in the service correctly returns the EmployeeDto for a given mobile number.

```
@Test
void getByMobileTest() {
    when(employeeRepository.findByEmail("01820702330")).thenReturn(employee2);
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class))).thenReturn(employeeDto2);
    EmployeeDto result = employeeService.getByEmail("01820702330");
    assertNotNull(result);
    assertEquals( expected: "nuhsalman@gmail.com", result.getEmail());
}
```

Mock Behavior:

- When employeeRepository.findByMobile("01820702330") is called, it returns employee2.
- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns employeeDto2.

Output: No Output

TC 6: updateEmployeeTest

Purpose : Verifies that the updateEmployee() method in the service correctly updates and returns the EmployeeDto for a given employee ID.

```
@Test
void updateEmployeeTest() {
    when(employeeRepository.findById(101L)).thenReturn(Optional.of(employee1));
    when(employeeRepository.save(any(Employee.class))).thenReturn(employee1);
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class))).thenReturn(employeeDto1);
    EmployeeDto result = employeeService.updateEmployee(employeeDto1, id: 101L);
    assertNotNull(result);
    assertEquals(employeeDto1.getName(), result.getName());
    System.out.println(result);
}
```

Mock Behavior:

- When employeeRepository.findById(101L) is called, it returns Optional.of(employee1).
- When employeeRepository.save(any(Employee.class)) is called, it returns employee1.
- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns employeeDto1.

Output:

EmployeeDto(id=101, name=Mamun Hossain, email=mamun@gmail.com, designation=Intern, department=iStelar, division=Banking, mobile=01820702329, salary=10000, address=Dhaka, bloodGroup=A+)

TC 7: deleteEmployeeTest

Purpose: Verifies that the deleteEmployee() method in the service correctly deletes an employee and returns the expected response.

```
@Test
void deleteEmployeeTest() {
    when(employeeRepository.findById(101L)).thenReturn(Optional.of(employee1));
    ResponseEmpDto result = employeeService.deleteEmployee( id: 101L);
    verify(employeeRepository, times( wantedNumberOfInvocations: 1)).delete(employee1);
    assertEquals( expected: 3, result.getResponseCode());
    assertEquals( expected: "Deleted successfully", result.getResponseMessage());
}
```

Mock Behavior:

- When employeeRepository.findById(101L) is called, it returns Optional.of(employee1).
- employeeRepository.delete(employee1) is verified to be called once.

Output: No Output

TC 8: getByDesignationTest

Purpose: To verify that the service correctly retrieves and maps employees by their designation.

```
@Test
void getByDesignationTest() {
    when(employeeRepository.findAllByDesignation("Intern")).thenReturn(employeeList);
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class)))
            .thenReturn(employeeDto1, employeeDto2);
    List<EmployeeDto> result = employeeService.getByDesignation("Intern");
    assertEquals( expected: 2, result.size());
    assertEquals(result.get(0).getEmail(),  actual: "mamun@gmail.com");
    assertEquals(result.get(1).getEmail(),  actual: "nuhsalman@gmail.com");
}
```

Mock Behavior:

- employeeRepository.findAllByDesignation("Intern") returns the employeeList.
- mapper.map(any(Employee.class), eq(EmployeeDto.class)) returns employeeDto1 for the first call and employeeDto2 for the second call.

Output: No Output

TC 9: getByDivisionTest

Purpose: To verify that the getByDivision method in the EmployeeService correctly retrieves employees by their division and maps them to EmployeeDto objects.

```
@Test
void getByDivisionTest() {
    when(employeeRepository.findAllByDivision("Banking")).thenReturn(Collections.singletonList(employee1));
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class)))
            .thenReturn(employeeDto1);
    List<EmployeeDto> result = employeeService.getByDivision("Banking");
    assertEquals( expected: 1, result.size());
    assertEquals(result.get(0).getName(), employee1.getName());
    System.out.println(result.get(0).getId());
}
```

Mock Behavior:

- When employeeRepository.findAllByDivision("Banking") is called, it returns a list containing a single employee1 instance.

- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns employeeDto1.

Output: 101

TC 10: getByDepartmentTest

Purpose: To validate that the getByDepartment method in the EmployeeService correctly retrieves employees by their department and maps them to EmployeeDto objects.

```
@Test
void getByDepartmentTest() {
    when(employeeRepository.findAllByDepartment("Java Development")).thenReturn(Collections.singletonList(employee2));
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class)))
            .thenReturn( employeeDto2);
    List<EmployeeDto> result = employeeService.getByDepartment("Java Development");
    assertEquals( expected: 1, result.size());
    System.out.println(result.get(0).getName());
    System.out.println(result.get(0).getId());
}
```

Mock Behavior:

- When employeeRepository.findAllByDepartment("Java Development") is called, it returns a list containing a single employee2 instance.
- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns employeeDto2.

Output: Nuh Salman

102

TC 11: getByBloodGroupTest

Purpose: To verify that the getByBloodGroup method in the EmployeeServiceImpl class correctly retrieves and transforms employee data based on blood group criteria.

```
@Test
void getByBloodGroupTest() {
        when(employeeRepository.findAllByBloodGroup("A+")).thenReturn(employeeList);
        lenient().when(mapper.map(any(Employee.class), eq(EmployeeDto.class)))
                .thenReturn(employeeDto1, employeeDto2);
        List<CustomizeEmpDto> result = employeeService.getByBloodGroup("A+");
        assertEquals( expected: 2, result.size());
        System.out.println(result.get(0).getEmail());
        System.out.println(result.get(1).getMobile());
}
```

Mock Behavior:

- When employeeRepository.findAllByBloodGroup("A+") is called, it returns a predefined list of employees (employeeList).
- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns a sequence of predefined employee DTOs (employeeDto1, employeeDto2).

Output:

mamun@gmail.com

01820702330

TC 12: getByDepartmentDivisionTest

Purpose: To verify that the getByDepartmentDivision method in the EmployeeService correctly retrieves employees based on both department and division criteria and maps them to EmployeeDto objects.

```
@Test
void getByDepartmentDivisionTest() {
    when(employeeRepository.findByDivisionDepartment( department: "iStelar",  division: "Banking")).
            thenReturn(Collections.singletonList(employee1));
    when(mapper.map(any(Employee.class), eq(EmployeeDto.class)))
            .thenReturn(employeeDto1);
    List<EmployeeDto> result = employeeService.getByDepartmentDivision( department: "iStelar",  division: "Banking");
    assertEquals( expected: 1, result.size());
    System.out.println(result.get(0).getDepartment());
    System.out.println(result.get(0).getDivision());
}
```

Mock Behavior:

- When employeeRepository.findByDivisionDepartment("iStelar", "Banking") is called, it returns a list containing a single employee1 instance.
- When mapper.map(any(Employee.class), eq(EmployeeDto.class)) is called, it returns employeeDto1.

Output:

iStelar

Banking

7.2.2 Test Table -2

Test Case ID	Test Case Title	Input Data	Expected Output	Actual Output	Satus	Remarks
TC 1	createEmployeeTest	employeeDto1	name: "Mamun Hossain", email: "mamun@gmail.com"	Same as Expected	**Pass**	Verifies correct employee creation.
TC 2	getAllEmployeeTest	None (fetches all employees)	List of 2 employees: emails "nuhsalman@gmail.com", "01820702329"	Same as Expected	**Pass**	Confirms retrieval of all employees.
TC 3	getEmployeeByIdTest	101L	name: "Mamun Hossain", designation: printed	Same as Expected	**Pass**	Checks employee details by ID.
TC 4	getByEmailTest	"mamun@gmail.com"	email: "mamun@gmail.com"	Same as Expected	**Pass**	Ensures correct employee retrieval by email.
TC 5	getByMobileTest	"01820702330"	email: "nuhsalman@gmail.com"	Same as Expected	**Pass**	Verifies retrieval by mobile number.
TC 6	updateEmployeeTest	employeeDto1, 101L	name: "Mamun Hossain"	Same as Expected	**Pass**	Validates employee update functionality.
TC 7	deleteEmployeeTest	101L	responseCode: 3, responseMessage: "Deleted successfully"	Same as Expected	**Pass**	Confirms successful employee deletion.
TC 8	getByDesignationTest	"Intern"	List of 2 employees: emails "mamun@gmail.com", "nuhsalman@gmail.com"	Same as Expected	**Pass**	Verifies employee retrieval by designation.
TC 9	getByDivisionTest	"Banking"	name: "Mamun Hossain"	Same as Expected	**Pass**	Ensures correct retrieval by division.
TC 10	getByDepartmentTest	"Java Development"	name: "Nuh Salman"	Same as Expected	**Pass**	Checks employee retrieval by department.
TC 11	getByBloodGroupTest	"A+"	List of 2 employees: email "mamun@gmail.	Same as Expected	**Pass**	Verifies retrieval of employees by

			com", mobile "01820702329"			blood group.
TC 12	getByDepartmentDivisionTest	"iStelar", "Banking"	department: "iStelar", division: "Banking"	Same as Expected	**Pass**	Ensures correct retrieval by department and division.

7.2.3 Using @SpringBootTest and @MockBean

Let's start with test class

```
@SpringBootTest
@ActiveProfiles("test")
class EmployeeServiceImplTestMockito {
    @MockBean
    private EmployeeRepository employeeRepository;
    @Autowired
    private ModelMapper mapper;
    @Autowired
    private EmployeeServiceImpl employeeService;  // If we use
EmployeeService interface it will work fine
    private Employee employee1;
    private Employee employee2;
    private EmployeeDto employeeDto1;
    private EmployeeDto employeeDto2;
```

Questions: Why is ModelMapper not mocked in some thenReturn conditions in this approach?

In the EmployeeServiceImplTestMockito test class, ModelMapper is not mocked in some thenReturn conditions because:

1. **Spring Boot Test Context**:

   The @SpringBootTest annotation loads the full application context, including the real ModelMapper bean. This allows the tests to leverage the actual ModelMapper instance rather than a mock, providing a more realistic test environment.

2. **Direct Return Values:**

   Some tests focus on verifying interactions with the EmployeeRepository and returning static values or lists directly from the repository mock (e.g., when(employeeRepository.findByEmail(employee1.getEmail())).thenReturn(employee1);). Since the mapping logic is not the focus of these tests, ModelMapper is not involved in these thenReturn conditions.

Example of a method:

```
@Test
void getEmployeeById() {
    when(employeeRepository.findById(101L)).thenReturn(Optional.of(employee1));

    EmployeeDto response = employeeService.getEmployeeById(101L);
    assertNotNull(response);
    assertEquals( expected: "Mamun Hossain", response.getName());
    assertEquals( expected: 101, response.getId());
    assertEquals( expected: "01820702329", response.getMobile());
    assertEquals( expected: "Dhaka", response.getAddress());
    assertEquals( expected: "Intern", response.getDesignation());
}
```

7.3 Test without Mockito

We test the EmployeeServiceImpl class by using real components instead of mocking dependencies. This allows us to perform testing that interacts directly with the database and other related components.

Necessary Annotations

@SpringBootTest : This annotation is used to bootstrap the entire container, making it a good choice for integration testing where the full Spring context is needed.

@ActiveProfiles("test"): Activates the test profile, allowing us to use specific configurations suited for testing (e.g., in-memory databases).

@Transactional: The @Transactional annotation ensures that the methods within the annotated class or method are executed within a transaction. If any exception occurs, the transaction will be rolled back, preventing changes from being committed to the database.

It is commonly used in service layer tests or integration tests where database operations are performed.

Key Features of @Transactional

- **Rollback by Default:** In tests, @Transactional will roll back transactions after each test method, ensuring that the database remains in a consistent state and does not retain any data from previous tests.
- **Transactional Context:** It ensures that all the database operations within the annotated method/class are part of the same transaction.

When we use @Transactional, several things happen. These are following

Any employee is not saved in the real database permanently due to the use of the @Transactional annotation in your test class.

Because this annotation ensures that each test method runs within its own transaction. By default, Spring will roll back the transaction at the end of the test method. This means any changes made to the database during the test, including saving new entities, will be undone after the test method finishes.

During the test method's execution, these employees will be accessible, but once the test ends, they will be removed from the database due to the transaction rollback.

If we want these records to be stored in the real database after the test method finishes, we would need to disable the rollback behavior by using the **@Rollback(false)** annotation on specific test methods.

When we run your test cases

1. **During the Test Execution**

   - The records (like employee1 and employee2 in our setup() method) are saved into the database, and we can interact with them as if they were stored in the database.

2. **After the Test Case Finishes**

   - The @Transactional annotation ensures that once the test case finishes, any changes made to the database, including the records you saved, are automatically rolled back.
   - This rollback happens at the end of each test method, effectively removing the records from the database and returning it to its previous state before the test started.

So, while the test is running, the records exist in the database, but they are automatically removed when the test method finishes due to the rollback of the transaction. This behavior ensures that our tests do not leave behind any data in the database, keeping your testing environment clean and consistent.

7 Test the Controller layer

7.1 Controller Layer

The controller layer in a Spring Boot application serves as a crucial component within the overall architecture, primarily responsible for managing incoming HTTP requests and generating appropriate responses. This layer acts as an intermediary between the user interface (e.g. web browsers, mobile applications) and the service layer where the core business logic resides.

This layer is responsible for:

1. **Handling Requests**:

   - Receives HTTP requests from clients and maps them to corresponding methods within the controller.
   - Uses annotations such as @RequestMapping, @GetMapping, @PostMapping, etc., to define the endpoints and HTTP methods that the controller can handle.

2. **Processing Requests**:

   - Executes business logic by invoking methods from the service layer. This may involve manipulating data, performing calculations, or coordinating other operations.
   - Validates and processes the input data, which may be included in the request body or URL parameters.

3. **Returning Responses**:

   - Prepares and sends HTTP responses back to the client. This may involve returning data in formats like JSON or XML, or rendering view templates.
   - Utilizes annotations such as @ResponseBody to indicate that the return value of a method should be bound to the web response body.

Verification in the Controller Layer

In the controller layer of Spring Boot applications, verification is crucial to ensure that the controller methods behave as expected. Following some techniques used for verification in the controller layer:

1. **MockMvc:** is used to perform HTTP requests and validate responses in a test environment. It allows you to simulate web requests and check how the controller handles them.

- **Perform Requests**: Use MockMvc to send GET, POST, PUT, DELETE, and other HTTP requests to your controller endpoints.

- **Validate Responses**: Assert the status codes, response bodies, headers, and other aspects of the HTTP response.

2. MockMvcResultMatchers: It provides a set of matchers to validate various aspects of the HTTP response.

- **Status Codes**: Check if the HTTP status code is as expected (e.g., isOk(), isCreated()).
- **Response Content**: Verify the content of the response (e.g., JSON path, string content).
- **Headers**: Assert the presence and values of HTTP headers.

3. ObjectMapper: It is used to convert Java objects to JSON and vice versa. In tests, it helps in verifying the content of JSON responses by converting them into Java objects.

- **Convert Response**: Convert the JSON response into a Java object for further assertions.
- **Serialize Requests**: Convert Java objects into JSON format for sending requests.

4. Assertions: These are used to validate the expected values against actual values. In tests, you use assertions to ensure that the response from the controller meets your expectations.

- **Assert Values**: Check if the actual response values match the expected values using assertion libraries like AssertJ, JUnit, or Hamcrest.
- **Verify Data**: Ensure the correctness of the data returned by the controller.

7.2 Test using Mockito

Mockito

It is a popular testing framework for Java that is used for creating mock objects in unit tests. It provides a simple way to isolate the component under test by replacing its dependencies with mock implementations. This allows you to focus on testing the logic of the component itself without being affected by the behavior of its dependencies.

Key Features:

- **Mock Creation:** Allows you to create mock objects for dependencies, specifying their behavior and interactions.
- **Stubbing:** Enables you to define what should be returned when specific methods are called on the mock objects.
- **Verification:** Provides mechanisms to verify that certain methods were called on the mock objects with expected arguments.

This is commonly used in unit tests to replace dependencies of the class under test with mock objects, ensuring that the test is focused on the behavior of the class itself and not on its interactions with external components.

MockMvc

MockMvc is a tool in Spring's testing framework that lets you test Spring MVC controllers by simulating HTTP requests and responses. It's part of the Spring Test module, used for unit and integration testing of controllers without needing to start a full web server.

MockMvc is not related to Mockito; instead, it’s provided by Spring Test specifically for testing MVC controllers. When using @WebMvcTest, the MockMvc object is auto-configured, allowing you to send mock HTTP requests in a simulated MVC environment, which enables efficient testing without launching an application server.

Here are some key benefits and features of using MockMvc:

Benefits

- **Lightweight and Fast**: MockMvc is exceptionally fast as it doesn’t start a full servlet container like Tomcat, Jetty, or Undertow. This makes it a go-to choice for rapid testing.
- **Seamless Integration with Spring Security**: It easily integrates with Spring Security, allowing you to bypass or test various authentication and authorization setups.
- **Effective for REST Endpoints and Views**: MockMvc integrates seamlessly with REST endpoints and views.
- **Fluent Checks**: Offers numerous fluent checks for asserting the responses.
- **Mocked Servlet Environment**: It's important to note that MockMvc creates a mocked servlet environment, meaning there's no real server and no port involved; interactions are done directly in-memory.

Key Features and Uses of MockMvc

1. **Simulating HTTP Requests**

   MockMvc allows you to perform various HTTP requests (GET, POST, PUT, DELETE, etc.) and interact with your controllers as if they were being called by an actual client, such as a browser or a REST client.

2. **Testing Controller Logic**:

   You can test the behavior of your controllers in isolation, ensuring that they handle requests and responses correctly. This includes testing URL mappings, request parameters, headers, content negotiation, and more.

3. **Verifying Responses**:

MockMvc provides a fluent API to assert the expected response status, headers, and body. You can verify that the response contains the expected data, status codes (e.g., 200 OK, 404 Not Found), and even specific JSON elements.

4. **Integration with Mockito**:

You can use MockMvc in combination with Mockito to mock service layer dependencies. This allows you to focus on testing the controller layer without worrying about the actual implementation of services.

@WebMvcTest

The @WebMvcTest annotation is a Spring Boot test annotation that focuses on testing the web layer of an application. It is specifically designed for testing controllers and related components in isolation from other parts of the application.

Key Features:

1. **Controller Focus:** Loads only the controller beans and related components (such as @ControllerAdvice, @JsonComponent, etc.), excluding other parts of the application like services and repositories.
2. **Mocking Dependencies:** Automatically configures MockMvc, allowing you to perform and test HTTP requests and responses. It also provides the ability to mock service layer dependencies using @MockBean.
3. **Limited Context:** Loads a minimal Spring application context, which speeds up tests by focusing only on the web layer.

This annnotation used to write unit tests for controllers, allowing you to test request handling, response generation, and interactions with mocked service dependencies. It ensures that your tests are focused on the web layer's behavior without involving the full application context.

ObjectMapper

ObjectMapper is a class in the Jackson library used for converting Java objects to JSON and vice versa. It's a versatile and powerful tool for handling JSON data in Java applications.

1. **Serialization**: Converts Java objects into JSON.

```
ObjectMapper objectMapper = new ObjectMapper();

String jsonString = objectMapper.writeValueAsString(Object);
```

2. **Deserialization**: Converts JSON into Java objects.

```
YourClass yourObject = objectMapper.readValue(jsonString, YourClass.class);
```

.contentType(MediaType.APPLICATION_JSON): This line specifies that the content of the request body is in JSON format. This is typically used in POST, PUT, or PATCH requests where the request body contains data that needs to be processed or stored by the server. For instance, when you are creating or updating a resource and sending a DTO as JSON in the request body, setting the content type to application/json ensures that the server treats the body as JSON.

.content(objectMapper.writeValueAsString(object)): This line converts the object into a JSON string using ObjectMapper and then sets this JSON string as the body of the HTTP request.

Controller layer of mPack:

```
package com.example.mPack.employee.controller;
import com.example.mPack.employee.dto.CustomizeEmpDto;
import com.example.mPack.employee.dto.EmployeeDto;
import com.example.mPack.employee.dto.ResponseEmpDto;
import com.example.mPack.employee.service.EmployeeService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;
1 usage
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {
    14 usages
    EmployeeService employeeService;
    EmployeeController(EmployeeService employeeService) { this.employeeService = employeeService; }

    @PostMapping
    public ResponseEntity<CustomizeEmpDto> createEmployee (@RequestBody EmployeeDto employeeDto) {
        CustomizeEmpDto employeeResponse = employeeService.createEmployee(employeeDto);
        return new ResponseEntity<>(employeeResponse, HttpStatus.CREATED);
    }


@GetMapping
public List<EmployeeDto> getAllEmployee (){
    return employeeService.getAllEmployee();
}

@GetMapping("/{id}")
public ResponseEntity<EmployeeDto> getEmployeeById (@PathVariable (name="id")Long id){
    return ResponseEntity.ok(employeeService.getEmployeeById(id));
}
@PutMapping("/{id}")
public ResponseEntity <EmployeeDto>  updateEmployee(@RequestBody EmployeeDto employeeDto,
                                                    @PathVariable (name="id")Long id ){
    System.out.println("employeeDto = " + employeeDto);
    EmployeeDto employeeResponse=  employeeService.updateEmployee(employeeDto, id);
    return new ResponseEntity<>(employeeResponse, HttpStatus.OK);
}
```

```
@PatchMapping("/{id}")
public ResponseEntity <ResponseEmpDto>  updateEmployee2(@RequestBody EmployeeDto employeeDto,
                                                        @PathVariable (name="id")Long id ){
    ResponseEmpDto employeeResponse=  employeeService.updateEmployeeByPatch(employeeDto, id);
    return new ResponseEntity<>(employeeResponse, HttpStatus.OK);
}
@DeleteMapping("/{id}")
public ResponseEntity<ResponseEmpDto> deleteEmployee (@PathVariable (name="id")Long id ){
    ResponseEmpDto employeeResponse  =employeeService.deleteEmployee(id);
    return new ResponseEntity<>(employeeResponse, HttpStatus.OK);
}

@GetMapping("/email/{email}")
public ResponseEntity<EmployeeDto> getByEmail (@PathVariable (name="email")String email){
 return new ResponseEntity<>(employeeService.getByEmail(email), HttpStatus.OK);
}

@GetMapping("/mobile/{mobile}")
public ResponseEntity<EmployeeDto> getByMobile (@PathVariable (name="mobile")String mobile){
    return new ResponseEntity<>(employeeService.getByMobile(mobile), HttpStatus.OK);
}

@GetMapping("/department/{department}")
public ResponseEntity<List<EmployeeDto>> getByDepartment (@PathVariable (name = "department")String department){
    List<EmployeeDto> responses=employeeService.getByDepartment(department);
    return ResponseEntity.ok(responses);
}
@GetMapping("/designation/{designation}")
public ResponseEntity<List<EmployeeDto>> getByDesignation (@PathVariable (name = "designation")String designation){
    return new ResponseEntity<>(employeeService.getByDesignation(designation),HttpStatus.OK);
}

@GetMapping("/division/{division}")
public ResponseEntity<List<EmployeeDto>> getByDivision (@PathVariable (name = "division")String division){
    List<EmployeeDto> responses=employeeService.getByDivision(division);
    return new ResponseEntity<>(responses,HttpStatus.OK);
}

@GetMapping("/bloodGroup/{bloodGroup}")
public ResponseEntity<List<CustomizeEmpDto>> getByBloogGroup (@PathVariable (name = "bloodGroup")String bloodGroup){
    List<CustomizeEmpDto> responses=employeeService.getByBloogGroup(bloodGroup);
    return new ResponseEntity<>(responses, HttpStatus.OK);
}

@GetMapping("twoProperties/{department},{division}")
public ResponseEntity<List<EmployeeDto>> getByDepartmentDivision (@PathVariable (name = "department")String department,
                                                                  @PathVariable (name = "division") String division){
    List<EmployeeDto> responses=employeeService.getByDepartmentDivision(department, division);
    return new ResponseEntity<>(responses, HttpStatus.OK);
}

}
```

Let's start with test class

```
package com.example.mPack.employee.controller;

import com.example.mPack.employee.dto.CustomizeEmpDto;
import com.example.mPack.employee.dto.EmployeeDto;
import com.example.mPack.employee.dto.ResponseEmpDto;
import com.example.mPack.employee.service.EmployeeService;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;
import java.util.Arrays;
import java.util.List;

import static org.mockito.Mockito.when;

@WebMvcTest(EmployeeController.class)
public class EmployeeControllerTestMockito {
    @MockBean
    private EmployeeService employeeService;
    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private ObjectMapper objectMapper;
    private EmployeeDto employeeDto1;
    private EmployeeDto employeeDto2;
    private CustomizeEmpDto customizeEmpDto1;
    private CustomizeEmpDto customizeEmpDto2;
    private ResponseEmpDto responseEmpDto;
    private List<EmployeeDto> employeeDtoList;


BeforeEach
void setup() {
    employeeDto1 = new EmployeeDto();
    employeeDto1.setId(1L);
    employeeDto1.setName("Nuh Salman");
    employeeDto1.setEmail("nuhsalman@gmail.com");
    employeeDto1.setDesignation("Manager, Sales");
    employeeDto1.setDepartment("Marketing Dept.");
    employeeDto1.setDivision("Marketing Division");
    employeeDto1.setMobile("01820702329");
    employeeDto1.setSalary(50000L);
    employeeDto1.setAddress("Dhaka");
    employeeDto1.setBloodGroup("O+");
```

```
    employeeDto2 = new EmployeeDto();
    employeeDto2.setId(2L);
    employeeDto2.setName("Hud Salman");
    employeeDto2.setEmail("hudsalman@gmail.com");
    employeeDto2.setDesignation("Senior Software Developer");
    employeeDto2.setDepartment("Development");
    employeeDto2.setDivision("Engineering");
    employeeDto2.setMobile("01333722677");
    employeeDto2.setSalary(70000L);
    employeeDto2.setAddress("Cumilla");
    employeeDto2.setBloodGroup("A+");

    customizeEmpDto1 = new CustomizeEmpDto(employeeDto1);
    customizeEmpDto2 = new CustomizeEmpDto(employeeDto2);

    responseEmpDto = new ResponseEmpDto();
    responseEmpDto.setResponseCode(1);
    responseEmpDto.setResponseMessage("Operation Successful");
    employeeDtoList = Arrays.asList(employeeDto1, employeeDto2);
}
```

7.2.1 Test Cases

TC 1: createEmployeeTest()

Purpose: Tests the **POST /api/employees** endpoint for creating a new employee.

```
@Test
void createEmployeeTest() throws Exception {
    when(employeeService.createEmployee(Mockito.any(EmployeeDto.class))).thenReturn(customizeEmpDto1);
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/api/employees")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(employeeDto1)))
            .andExpect(MockMvcResultMatchers.status().isCreated())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.id").value(employeeDto1.getId()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.name").value(employeeDto1.getName()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.email").value(employeeDto1.getEmail()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.mobile").value(employeeDto1.getMobile()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
    System.out.println(MockMvcResultMatchers.jsonPath( expression: "$.name"));
}
```

Setup: Mocks the EmployeeService.createEmployee() method to return a CustomizeEmpDto object when given an EmployeeDto object.

Execution: Sends a POST request to the /api/employees endpoint with the EmployeeDto object as the request body.

Assertions:

- Verifies that the HTTP response status is 201 Created.
- Checks that the JSON response contains the expected id, name, email, and mobile values from the EmployeeDto object.

Output:

{"id":1,"name":"Nuh Salman","email":"nuhsalman@gmail.com","mobile":"01820702329"}

TC 2: getAllEmployeeTest()

Purpose: Tests the GET **/api/employees** endpoint to retrieve all employees.

```
@Test
void getAllEmployeeTest() throws Exception {

    when(employeeService.getAllEmployee()).thenReturn(employeeDtoList);
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees"))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[0].name").value(employeeDto1.getName()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[0].email").value(employeeDto1.getEmail()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[1].name").value(employeeDto2.getName()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[1].email").value(employeeDto2.getEmail()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.getAllEmployee() method to return a list of EmployeeDto objects.

Execution: Sends a GET request to the /api/employees endpoint.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains the expected name and email values for some employees among all employees.

Output:

[{"id":1,"name":"Nuh Salman","email":"nuhsalman@gmail.com","designation":"Manager, Sales","department":"Marketing Dept.","division":"Marketing Division","mobile":"01820702329","salary":50000,"address":"Dhaka","bloodGroup":"O+"},{"id":2,"name":"Hud Salman","email":"hudsalman@gmail.com","designation":"Senior Software Developer","department":"Development","division":"Engineering","mobile":"01333722677","salary":70000,"address":"Cumilla","bloodGroup":"A+"}].

TC 3: getEmployeeByIdTest()

Purpose: Tests the **GET /api/employees/{id}** endpoint to retrieve an employee by ID.

```
@Test
void getEmployeeByIdTest() throws Exception {
    when(employeeService.getEmployeeById(1L)).thenReturn(employeeDto1);
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees/{id}",  ...uriVariables: 1L)
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.name").value(employeeDto1.getName()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.email").value(employeeDto1.getEmail()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.mobile").value(employeeDto1.getMobile()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.address").value(employeeDto1.getAddress()))
            .andReturn();

    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.getEmployeeById() method to return an EmployeeDto object for a given ID.

Execution: Sends a GET request to the /api/employees/{id} endpoint with the specified ID.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains the expected name, email, mobile, and address values.

Output:

{"id":1,"name":"Nuh Salman","email":"nuhsalman@gmail.com","designation":"Manager, Sales","department":"Marketing Dept.","division":"Marketing Division","mobile":"01820702329","salary":50000,"address":"Dhaka","bloodGroup":"O+"}.

TC 4: updateEmployeeTest()

Purpose: Tests the **PUT /api/employees/{id}** endpoint for updating an employee's details.

```
@Test
void updateEmployeeTest() throws Exception {
    when(employeeService.updateEmployee(Mockito.any(EmployeeDto.class), Mockito.anyLong())).thenReturn(employeeDto1);
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.put( urlTemplate: "/api/employees/{id}",  ...uriVariables: 1L)
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(employeeDto1)))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.name").value(employeeDto1.getName()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.email").value(employeeDto1.getEmail()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.mobile").value(employeeDto1.getMobile()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.address").value(employeeDto1.getAddress()))
            .andReturn();

    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.updateEmployee() method to return an EmployeeDto object when given an EmployeeDto object and ID.

Execution: Sends a PUT request to the /api/employees/{id} endpoint with the updated EmployeeDto object.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains the expected name, email, mobile, and address values.

Output:

{"id":1,"name":"Nuh Salman","email":"nuhsalman@gmail.com","designation":"Manager, Sales","department":"Marketing Dept.","division":"Marketing Division","mobile":"01820702329","salary":50000,"address":"Dhaka","bloodGroup":"O+"}

TC 5: updateEmployeeByPatchTest()

Purpose: Tests the **PATCH /api/employees/{id}** endpoint for partially updating an employee's details.

```
@Test
void updateEmployeeByPatchTest() throws Exception {
    when(employeeService.updateEmployeeByPatch(Mockito.any(EmployeeDto.class), Mockito.anyLong())).thenReturn(responseEmpDto);
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.patch( urlTemplate: "/api/employees/{id}",  ...uriVariables: 1L)
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(employeeDto1)))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.responseCode").value(responseEmpDto.getResponseCode()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.responseMessage").value(responseEmpDto.getResponseMessage()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.updateEmployeeByPatch() method to return a ResponseEmpDto object for a given ID.

Execution: Sends a PATCH request to the /api/employees/{id} endpoint with the patch details in the request body.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains the expected responseCode and responseMessage values.

Output: {"responseCode":1,"responseMessage":"Operation Successful"}

TC 6: deleteEmployeeTest()

Purpose: Tests the **DELETE /api/employees/{id}** endpoint to delete an employee by ID.

```
@Test
void deleteEmployeeTest() throws Exception {
    when(employeeService.deleteEmployee( id: 1L)).thenReturn(responseEmpDto);

    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.delete( urlTemplate: "/api/employees/{id}",  ...uriVariables: 1L)
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.responseCode").value(responseEmpDto.getResponseCode()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.responseMessage").value(responseEmpDto.getResponseMessage()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.deleteEmployee() method to return a ResponseEmpDto object for a given ID.

Execution: Sends a DELETE request to the /api/employees/{id} endpoint.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains the expected responseCode and responseMessage values.

Output: {"responseCode":1,"responseMessage":"Operation Successful"}

TC 6: getByEmailTest()

Purpose: Tests the **GET /api/employees/email/{email}** endpoint to retrieve an employee by email.

```
@Test
void getByEmailTest() throws Exception {
    when(employeeService.getByEmail("hudsalman@gmail.com")).thenReturn(employeeDto2);
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees/email/{email}",  ...uriVariables: "hudsalman@gmail.com")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.email").value(employeeDto2.getEmail()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.mobile").value(employeeDto2.getMobile()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.getByEmail() method to return an EmployeeDto object for a given email.

Execution: Sends a GET request to the /api/employees/email/{email} endpoint with the specified email.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains the expected email, mobile, and name values.

Output:

{"id":2,"name":"Hud Salman","email":"hudsalman@gmail.com","designation":"Senior Software Developer","department":"Development","division":"Engineering","mobile":"01333722677","salary":70000,"address":"Cumilla","bloodGroup":"A+"}

TC 7: getByMobileTest()

Purpose: Tests the **GET /api/employees/mobile/{mobile}** endpoint to retrieve an employee by mobile number.

```
@Test
void getByMobileTest() throws Exception {
    when(employeeService.getByMobile("01820702329")).thenReturn(employeeDto1);

    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees/mobile/{mobile}",  ...uriVariables: "01820702329")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.email").value(employeeDto1.getEmail()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.mobile").value(employeeDto1.getMobile()))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.name").value(employeeDto1.getName()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.getByMobile() method to return an EmployeeDto object for a given mobile number.

Execution: Sends a GET request to the /api/employees/mobile/{mobile} endpoint with the specified mobile number.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains the expected email, mobile, and name values.

Output:

{"id":1,"name":"Nuh Salman","email":"nuhsalman@gmail.com","designation":"Manager, Sales","department":"Marketing Dept.","division":"Marketing Division","mobile":"01820702329","salary":50000,"address":"Dhaka","bloodGroup":"O+"}

TC 8: getByDepartmentTest()

Purpose: Tests the **GET /api/employees/department/{department}** endpoint to retrieve employees by department.

```
@Test
void getByDepartmentTest() throws Exception {
  //  when(employeeService.getByDepartment("Marketing Dept.")).thenReturn(employeeDtoList);
   when(employeeService.getByDepartment("Marketing Dept.")).thenAnswer(invocationOnMock -> {
        return employeeDtoList.stream().filter(emp->"Marketing Dept.".equals(emp.getDepartment())).
                collect(Collectors.toList());
    });
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees/department/{department}",
                            ...uriVariables: "Marketing Dept.")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[0].name").value(employeeDto1.getName()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.getByDepartment() method to return a list of EmployeeDto objects filtered by department.

Execution: Sends a GET request to the /api/employees/department/{department} endpoint with the specified department.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains employees from the specified department.

Output:

[{"id":1,"name":"Nuh Salman","email":"nuhsalman@gmail.com","designation":"Manager, Sales","department":"Marketing Dept.","division":"Marketing Division","mobile":"01820702329","salary":50000,"address":"Dhaka","bloodGroup":"O+"}]

TC 9 : getByDesignationTest()

Purpose: Tests the **GET /api/employees/designation/{designation}** endpoint to retrieve employees by designation.

```
@Test
void getByDesignationTest() throws Exception {
    when(employeeService.getByDesignation("Manager, Sales")).thenReturn(Arrays.asList(employeeDto1));
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees/designation/{designation}",
                            ...uriVariables: "Manager, Sales")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[0].name").value(employeeDto1.getName()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.getByDesignation() method to return a list of EmployeeDto objects filtered by designation.

Execution: Sends a GET request to the /api/employees/designation/{designation} endpoint with the specified designation.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains employees with the specified designation.

Output:

[{"id":1,"name":"Nuh Salman","email":"nuhsalman@gmail.com","designation":"Manager, Sales","department":"Marketing Dept.","division":"Marketing Division","mobile":"01820702329","salary":50000,"address":"Dhaka","bloodGroup":"O+"}]

TC 10: getByDivisionTest()

Purpose: Tests the **GET /api/employees/division/{division}** endpoint to retrieve employees by division.

```
@Test
void getByDivisionTest() throws Exception {
    //when(employeeService.getByDivision("Engineering")).thenReturn(employeeDtoList);

    when(employeeService.getByDivision("Engineering")).thenAnswer(invocationOnMock -> {
        return employeeDtoList.stream().filter(emp->"Engineering".equals(emp.getDivision())).
                collect(Collectors.toList());
    });
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees/division/{division}",
                            ...uriVariables: "Engineering")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[0].name").value(employeeDto2.getName()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.getByDivision() method to return a list of EmployeeDto objects filtered by division.

Execution: Sends a GET request to the /api/employees/division/{division} endpoint with the specified division.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains employees with the specified division.

Output:

[{"id":2,"name":"Hud Salman","email":"hudsalman@gmail.com","designation":"Senior Software Developer","department":"Development","division":"Engineering","mobile":"01333722677","salary":70000,"address":"Cumilla","bloodGroup":"A+"}]

TC 11: getByBloodGroupTest()

Purpose: Tests the **GET /api/employees/bloodGroup/{bloodGroup}** endpoint to retrieve employees by blood group.

```
@Test
void getByBloodGroupTest() throws Exception {
    when(employeeService.getByBloogGroup( bloodGroup: "O+")).thenReturn(Arrays.asList(customizeEmpDto1));
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees/bloodGroup/{bloodGroup}",  ...uriVariables: "O+")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[0].name").value(employeeDto1.getName()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());
}
```

Setup: Mocks the EmployeeService.getByBloogGroup() method to return a list of CustomizeEmpDto objects filtered by blood group.

Execution: Sends a GET request to the /api/employees/bloodGroup/{bloodGroup} endpoint with the specified blood group.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains employees with the specified blood group.

Output:

[{"id":1,"name":"Nuh Salman","email":"nuhsalman@gmail.com","mobile":"01820702329"}]

TC 12 : getByDepartmentDivisionTest()

Purpose: Tests the **GET /api/employees/twoProperties/{department},{division}** endpoint to retrieve employees by department and division.

```
@Test
void getByDepartmentDivisionTest() throws Exception {
    when(employeeService.getByDepartmentDivision( department: "Development",  division: "Engineering")).thenReturn(Arrays.asList(employeeDto2));
    MvcResult result = mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/api/employees/twoProperties/{department},{division}",
                            ...uriVariables: "Development","Engineering")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[0].name").value(employeeDto2.getName()))
            .andReturn();
    System.out.println(result.getResponse().getContentAsString());

    ObjectMapper objectMapper = new ObjectMapper();
    JsonNode jsonNode = objectMapper.readTree(result.getResponse().getContentAsString());

    String email = jsonNode.get(0).get("email").asText();
    System.out.println("Email of first employee: " + email);
}
```

Setup: Mocks the EmployeeService.getByDepartmentDivision() method to return a list of EmployeeDto objects filtered by department and division.

Execution: Sends a GET request to the /api/employees/twoProperties/{department},{division} endpoint with the specified department and division.

Assertions:

- Verifies that the HTTP response status is 200 OK.
- Checks that the JSON response contains employees with the specified department and division.

Output:

[{"id":2,"name":"Hud Salman","email":"hudsalman@gmail.com","designation":"Senior Software Developer","department":"Development","division":"Engineering","mobile":"01333722677","salary":70000,"address":"Cumilla","bloodGroup":"A+"}]

Email of first employee: hudsalman@gmail.com

7.2.2 Testing Table-3

Test Case ID	Test Case Title	Input Data	Expected Output	Actual Output	Satus	Remarks
TC 1	**createEmployeeTest**	mployeeDto object	Status 201, JSON with created employee's details.	Same as Expected	**Pass**	Verifies employee creation via POST
TC 2	**getAllEmployeeTest**	None	Statement and employee object	Same as Expected	**Pass**	Successfully found and printed employee by ID.
TC 3	**getEmployeeByIdTest**	ID 1	No employee found	Same as Expected	**Pass**	No employee found for the given ID.
TC 4	**updateEmployeeTest**	Employee Dto, ID 1	Employee list size, statements	Same as Expected	**Pass**	Successfully retrieved and printed all employees.
TC 5	**updateEmployeeByPatchTest**	Employee Dto, ID 1	Statement and employee object	Same as Expected	**Pass**	Successfully found and printed employee by email.
TC 6	**deleteEmployeeTest**	ID 1	Null	Same as Expected	**Pass**	No employee found for the given email.
TC 7	**getByEmailTest**	Email	Statement and employee object	Same as Expected	**Pass**	Successfully found and printed employee by mobile number.
TC 8	**getByMobileTest**	Mobile	No employee found	Same as Expected	**Pass**	No employee found for the given mobile number.
TC 9	**getByDepartmentTest**	Department	Statements	Same as Expected	**Pass**	Successfully retrieved employees by designation.
TC 10	**getByDesignationTest**	Designation	One Employee objects, statements	Same as Expected	**Pass**	Successfully retrieved and printed employees by department.
TC 11	**getByDivisionTest**	Division	No output	Same as Expected	**Pass**	Successfully retrieved employees by division.
TC 12	**getByBloodGroupTest**	BloodGroup	One Employee objects	Same as Expected	**Pass**	Successfully retrieved and printed employees by blood group.

TC 13	**createEmployeeTest**	Departme nt, Division	One Employee objects	Same as Expected	**Pass**	Successfully retrieved and printed employees by division and department.

7.3 Test without Mockito

@SpringBootTest

This used to create an application context for integration tests in a Spring Boot application. It allows the test class to run with a fully initialized Spring Boot application context, which is essential for testing the application's behavior as it would operate in a real environment.

Usage:

- **Full Context Initialization**: It starts the entire Spring Boot application context, which includes the configuration, beans, and other components, providing a comprehensive testing environment.
- **Configuration**: By default, it will use the same configuration as the main application. You can customize it with additional parameters if needed.
- **Integration Testing**: This annotation is commonly used when you need to test the interaction between various components of the application, such as controllers, services, and repositories, to ensure they work together correctly.

@AutoConfigureMockMvc

This used to configure MockMvc for integration tests automatically. MockMvc is a class provided by Spring Test that allows you to perform HTTP requests and validate responses in a test environment without starting a real server.

Usage:

- **MockMvc Setup**: It sets up a MockMvc instance that you can use to send HTTP requests to the application and assert responses. This is useful for testing web layers (e.g., controllers) in isolation from the rest of the application.
- **HTTP Layer Testing**: This annotation is particularly useful for testing the HTTP endpoints of your application, validating the behavior of controllers, and ensuring that requests and responses are handled correctly.
- **Automatic Configuration**: It configures MockMvc with all the necessary settings, so you don't need to manually instantiate or configure it.

Let's start with test class

```
@SpringBootTest

@AutoConfigureMockMvc
public class EmployeeControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private ObjectMapper objectMapper;
    @Autowired
    private EmployeeRepository employeeRepository;
    private EmployeeDto employeeDto1;
    private EmployeeDto employeeDto2;
    private CustomizeEmpDto customizeEmpDto1;
    private CustomizeEmpDto customizeEmpDto2;
    private ResponseEmpDto responseEmpDto;
    private List<EmployeeDto> employeeDtoList;
```

In this section, we leverage MockMvc to perform real integration tests of the controller layer. The approach to setting up tests, executing requests, and validating responses is similar to that described in the Mockito-based tests. However, the key differences include:

1. **Real Components:** Unlike the Mockito-based tests where components are mocked, here we use real instances of services and repositories. This means that tests will interact with the actual database and service implementations.
2. **Test Execution:** We use MockMvc to perform actual HTTP requests and verify responses from real controller endpoints, rather than simulating interactions with mocked components.
3. **Context and Setup:**

    - **Mockito-Based Tests:** These typically involve setting up mock beans and using Mockito to define behavior.
    - **Real Integration Tests:** We set up the test database and use the full application context provided by @SpringBootTest. This setup ensures that all components are fully initialized and interact as they would in a production environment.

Key Differences Compared to Mockito-Based Testing:

1. **Real Components vs. Mocks:**

Mockito-Based Tests: Mocked services and repositories are used, and Mockito is employed to define behavior and interactions.

Without Mockito: The tests use real implementations of services and repositories. The tests interact with the actual database and service implementations, providing a more integrated view of the application.

2. **Test Setup and Execution:**

Mockito-Based Tests: Uses @MockBean or @Mock to create mock instances and @InjectMocks to inject them into the controller.

Without Mockito: Utilizes @SpringBootTest along with MockMvc to perform integration tests. The @SpringBootTest annotation starts the full application context, and MockMvc is used to perform real HTTP requests.