Part 1:

1. What is Unit Testing?

- A unit test is a way to check if a specific piece of code (usually a single method) works as expected.

- Unit tests ensure that individual parts of your code perform correctly in isolation, independent of other parts.

In Java, JUnit is the most popular framework for unit testing, and Spring Boot provides excellent integration with it.

2. Setting Up Your Environment

- Tools needed: Java, Spring Boot, JUnit, and Maven/Gradle as your build tool.

- If you're using Spring Initializr (https://start.spring.io/), create a new project with:

  - Project type: Maven

  - Dependencies: Spring Web, Spring Boot DevTools, Spring Boot Starter Test (for unit testing)

3. Understanding the Structure of a Unit Test

- A typical unit test in Spring Boot includes:

  - Test class: Located in the `src/test/java` directory, with a `@Test`-annotated method for each test case.

  - Assertions: Used to check expected outcomes (e.g., `assertEquals(expected, actual)`).

  - Mocks: If your method relies on other components (like repositories), you use mocks to simulate their behavior.

4. Creating Your First Unit Test

Let's assume we're testing a simple service class:

example -> java

// src/main/java/com/example/demo/service/CalculatorService.java

public class CalculatorService {

```
    public int add(int a, int b) {
        return a + b;
    }
}
example ->
```

To test this, create a unit test in `src/test/java/com/example/demo/service`:

```
example -> java
// src/test/java/com/example/demo/service/CalculatorServiceTest.java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorServiceTest {

    @Test
    public void testAdd() {
        CalculatorService calculator = new CalculatorService();
        int result = calculator.add(2, 3);
        assertEquals(5, result);
    }
}
example ->
```

- Explanation:
  - `@Test`: Marks this method as a test case.
  - `assertEquals(5, result)`: Verifies that the output is as expected.

5. Testing Components with Dependencies

}

- In real-world applications, methods often depend on other components (e.g., repositories or services). Here, mocks are used to simulate these dependencies.
- Mockito is a popular library to create mocks in unit tests.

Let’s enhance the `CalculatorService` to include a repository dependency.

example -> java

```
// src/main/java/com/example/demo/service/CalculatorService.java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class CalculatorService {

  private final CalculatorRepository repository;

  @Autowired
  public CalculatorService(CalculatorRepository repository) {
    this.repository = repository;
  }

  public int add(int a, int b) {
    return a + b;
  }

  public int findAndAdd(int a, int b) {
    int storedValue = repository.findStoredValue();
    return a + b + storedValue;
  }
}
example ->
```

```
example -> java
// src/main/java/com/example/demo/repository/CalculatorRepository.java
public interface CalculatorRepository {
    int findStoredValue();
}
example ->
```

6. Writing a Unit Test with Mocks

In the test, we'll use Mockito to mock `CalculatorRepository`.

```
example -> java
// src/test/java/com/example/demo/service/CalculatorServiceTest.java
import com.example.demo.repository.CalculatorRepository;
import com.example.demo.service.CalculatorService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorServiceTest {

    @Mock
    private CalculatorRepository repository;
    @InjectMocks
    private CalculatorService calculatorService;

    @BeforeEach
    public void setUp() {
      MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testAdd() {
      int result = calculatorService.add(2, 3);
      assertEquals(5, result);
    }

    @Test
    public void testFindAndAdd() {
      when(repository.findStoredValue()).thenReturn(10);
      int result = calculatorService.findAndAdd(2, 3);
      assertEquals(15, result);
    }
  }
```

example ->

- Explanation:
  - `@Mock`: Creates a mock version of `CalculatorRepository`.
  - `@InjectMocks`: Injects the mock repository into the `CalculatorService`.
  - `when(repository.findStoredValue()).thenReturn(10);`: Configures the mock to return `10` when `findStoredValue()` is called.

- This allows us to test `findAndAdd` without depending on the actual database or repository implementation.

7. Moving to Spring Boot MVC Testing

When testing MVC components, we often need to test controllers. Spring Boot's `@WebMvcTest` is ideal for this, as it sets up only the web layer (controllers) without loading the full application context.

Here's an example:

example -> java

```
// src/main/java/com/example/demo/controller/CalculatorController.java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CalculatorController {

    private final CalculatorService calculatorService;

    @Autowired
    public CalculatorController(CalculatorService calculatorService) {
        this.calculatorService = calculatorService;
    }

    @GetMapping("/add")
    public int add(@RequestParam int a, @RequestParam int b) {
        return calculatorService.add(a, b);
    }
}
```

example ->

8. Testing the Controller

example -> java

```
// src/test/java/com/example/demo/controller/CalculatorControllerTest.java
import com.example.demo.service.CalculatorService;
import org.junit.jupiter.api.Test;
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.test.web.servlet.MockMvc;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

@WebMvcTest(CalculatorController.class)
public class CalculatorControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private CalculatorService calculatorService;

    @Test
    public void testAdd() throws Exception {
        when(calculatorService.add(2, 3)).thenReturn(5);


        mockMvc.perform(get("/add")
                .param("a", "2")
                .param("b", "3"))
                .andExpect(status().isOk())
                .andExpect(content().string("5"));
    }
}
```

example ->

- Explanation:
  - `@WebMvcTest`: Sets up a Spring MVC context for `CalculatorController`.
  - `mockMvc.perform(get("/add")...)`: Simulates a GET request to `/add` with parameters `a=2` and `b=3`.
  - `andExpect(status().isOk())`: Asserts the HTTP status is OK (200).
  - `andExpect(content().string("5"))`: Asserts the response content is `5`.

Part 2:

Let’s go through step-by-step unit testing using a simple Employee CRUD project with a standard architecture: a DAO interface, a service layer, and a controller layer. We’ll cover how to create tests for each layer, starting with simple examples and then gradually introducing more complex test scenarios.

Project Structure

Our project will look like this:

- `EmployeeDao` - Data access interface for CRUD operations.
- `EmployeeService` - Service layer where business logic resides.
- `EmployeeController` - Controller layer handling HTTP requests.

Assume this setup connects to a database but we’ll focus on testing without a real database connection.

---

1. Setting Up the Environment

   - Use Spring Boot with the following dependencies: Spring Web, Spring Data JPA, and Spring Boot Starter Test.

   - Use Spring Initializr (https://start.spring.io/) to create the project or set up the dependencies manually.

2. Employee Entity

The `Employee` entity represents the data model for employees.

example -> java

```
// src/main/java/com/example/demo/model/Employee.java

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Employee {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String name;
  private String position;

  // Getters and setters
}
```

example ->

---

3. DAO Layer

The `EmployeeDao` interface uses Spring Data JPA for database communication.

example -> java

```
// src/main/java/com/example/demo/dao/EmployeeDao.java
import com.example.demo.model.Employee;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface EmployeeDao extends JpaRepository<Employee, Long> {
}
```

example ->

---

4. Service Layer

The `EmployeeService` provides the business logic, using `EmployeeDao` for CRUD operations.

example -> java

```java
// src/main/java/com/example/demo/service/EmployeeService.java
import com.example.demo.dao.EmployeeDao;
import com.example.demo.model.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;

@Service
public class EmployeeService {

  private final EmployeeDao employeeDao;

  @Autowired
  public EmployeeService(EmployeeDao employeeDao) {
    this.employeeDao = employeeDao;
  }

  public List<Employee> getAllEmployees() {
    return employeeDao.findAll();
  }
  public Optional<Employee> getEmployeeById(Long id) {
    return employeeDao.findById(id);
  }

  public Employee saveEmployee(Employee employee) {
    return employeeDao.save(employee);
  }

  public void deleteEmployee(Long id) {
    employeeDao.deleteById(id);
  }
}
```

example ->

---

5. Controller Layer

The `EmployeeController` exposes the CRUD endpoints.

example -> java

```java
// src/main/java/com/example/demo/controller/EmployeeController.java
import com.example.demo.model.Employee;
import com.example.demo.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;
```

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {

  private final EmployeeService employeeService;

  @Autowired
  public EmployeeController(EmployeeService employeeService) {
    this.employeeService = employeeService;
  }

  @GetMapping
  public List<Employee> getAllEmployees() {
    return employeeService.getAllEmployees();
  }

  @GetMapping("/{id}")
  public Optional<Employee> getEmployeeById(@PathVariable Long id) {
    return employeeService.getEmployeeById(id);
  }

  @PostMapping
  public Employee createEmployee(@RequestBody Employee employee) {
    return employeeService.saveEmployee(employee);
  }

  @DeleteMapping("/{id}")
  public void deleteEmployee(@PathVariable Long id) {
        employeeService.deleteEmployee(id);
    }
}
```

example ->

---

6. Writing Unit Tests

#Service Layer Testing

We’ll start by testing the `EmployeeService` layer, mocking the `EmployeeDao` dependency.

example -> java

```
// src/test/java/com/example/demo/service/EmployeeServiceTest.java
import com.example.demo.dao.EmployeeDao;
import com.example.demo.model.Employee;
import com.example.demo.service.EmployeeService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import java.util.Arrays;
import java.util.Optional;
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class EmployeeServiceTest {
```

```
@Mock
private EmployeeDao employeeDao;

@InjectMocks
private EmployeeService employeeService;

@BeforeEach
public void setUp() {
  MockitoAnnotations.openMocks(this);
}

@Test
public void testGetAllEmployees() {
  Employee employee1 = new Employee();
  employee1.setName("Alice");
  employee1.setPosition("Developer");

  Employee employee2 = new Employee();
  employee2.setName("Bob");
  employee2.setPosition("Manager");

  when(employeeDao.findAll()).thenReturn(Arrays.asList(employee1, employee2));

  assertEquals(2, employeeService.getAllEmployees().size());
}

@Test
public void testGetEmployeeById() {
  Employee employee = new Employee();
```

```
        employee.setId(1L);
        employee.setName("Alice");

        when(employeeDao.findById(1L)).thenReturn(Optional.of(employee));

        Optional<Employee> result = employeeService.getEmployeeById(1L);
        assertEquals("Alice", result.get().getName());
    }
}
```

example ->

- Explanation:
  - `@Mock` creates a mock of `EmployeeDao`.
  - `@InjectMocks` injects the mocked `EmployeeDao` into `EmployeeService`.
  - `when(employeeDao.findAll()).thenReturn(...)` and `when(employeeDao.findById(1L)).thenReturn(...)`: Define mock behavior.

#Controller Layer Testing

Next, we'll test the `EmployeeController` layer using Spring's `@WebMvcTest`, which focuses on web layer components.

example -> java

```
// src/test/java/com/example/demo/controller/EmployeeControllerTest.java
import com.example.demo.model.Employee;
import com.example.demo.service.EmployeeService;
import org.junit.jupiter.api.Test;
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.test.web.servlet.MockMvc;
import java.util.Arrays;
import static org.mockito.Mockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.hamcrest.Matchers.*;

@WebMvcTest(EmployeeController.class)
public class EmployeeControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private EmployeeService employeeService;

  @Test
  public void testGetAllEmployees() throws Exception {
    Employee employee1 = new Employee();
    employee1.setName("Alice");

    Employee employee2 = new Employee();
    employee2.setName("Bob");

    when(employeeService.getAllEmployees()).thenReturn(Arrays.asList(employee1, employee2));

    mockMvc.perform(get("/employees"))
        .andExpect(status().isOk())
```

```
            .andExpect(jsonPath("$", hasSize(2)))
            .andExpect(jsonPath("$[0].name", is("Alice")))
            .andExpect(jsonPath("$[1].name", is("Bob")));
    }

    @Test
    public void testGetEmployeeById() throws Exception {
        Employee employee = new Employee();
        employee.setId(1L);
        employee.setName("Alice");

        when(employeeService.getEmployeeById(1L)).thenReturn(Optional.of(employee));

        mockMvc.perform(get("/employees/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.name", is("Alice")));
    }
}
```
example ->

- Explanation:
  - `@WebMvcTest(EmployeeController.class)`: Configures the test for `EmployeeController` only.
  - `MockMvc mockMvc`: Used to simulate HTTP requests to the controller.
  - `jsonPath("$", hasSize(2))`: Checks the response array size.
  - `jsonPath("$.name", is("Alice"))`: Verifies specific JSON fields.

Summary

We’ve covered:

1. Setting up a basic project with entity, DAO, service, and controller layers.

2. Unit testing the service layer with mocked dependencies.

3. Testing the controller layer’s endpoints using `@WebMvcTest`.