What Unit testing and Integration testing

By Ramesh Fadatare (Java Guides)

What is Unit Testing

Unit testing involves the testing of each unit or an individual component of the software application.

The purpose is to validate that each unit of the software code performs as expected

Unit testing is done during development (coding phase) of an application by the **developers**

Unit may be a an individual function, method, procedure, module and object

What is Unit Testing

In Java, JUnit framework is used for unit testing Java applications

Most of the times one component will depend on other component(s), so while implementing unit tests we should mock the dependencies with the desired behaviour using frameworks like **Mockito**.

Integration Testing

As the name suggests, integration tests focus on integrating different layers of the application. That also means no mocking is involved.

Basically, we write integration tests for testing a feature which may involve interaction with multiple components.

Examples:
Employee Management Feature (EmployeeRepository, EmployeeService, EmployeeController).
User Management Feature (UserController, UserService, and UserRepository).
Login Feature (LoginRespository, LoginController, Login Service) etc

Unit Test Naming Conventions

By Ramesh Fadatare (Java Guides)

Unit Test Naming Conventions

1. **MethodName_StateUnderTest_ExpectedBehavior:**

```
isAdult_AgeLessThan18_False

withdrawMoney_InvalidAccount_ExceptionThrown

admitStudent_MissingMandatoryFields_FailToAdmit
```

2. **MethodName_ExpectedBehavior_StateUnderTest**

```
isAdult_False_AgeLessThan18

withdrawMoney_ThrowsException_IfAccountIsInvalid

admitStudent_FailToAdmit_IfMandatoryFieldsAreMissing
```

Unit Test Naming Conventions

3. test[MethodName]:

```
testSaveEmployee

testGetEmployeeById

testUpdateEmployee
```

4. test[Feature being tested]

```
testFailToWithdrawMoneyIfAccountIsInvalid

testStudentIsNotAdmittedIfMandatoryFieldsAreMissing
```

Unit Test Naming Conventions

5. Should_ExpectedBehavior_When_StateUnderTest:

```
Should_ThrowException_When_AgeLessThan18

Should_FailToWithdrawMoney_ForInvalidAccount

Should_FailToAdmit_IfMandatoryFieldsAreMissing
```

6. When_StateUnderTest_Expect_ExpectedBehavior:

```
When_AgeLessThan18_Expect_isAdultAsFalse

When_InvalidAccount_Expect_WithdrawMoneyToFail

When_MandatoryFieldsAreMissing_Expect_StudentAdmissionToFail
```

Unit Test Naming Conventions

7. givenPreconditions_whenStateUnderTest_thenExpectedBehavior:

givenEmployeeObject_whenSaveEmployee_thenReturnSavedEmployee

givenEmployeesList_whenFindAll_thenReturnListOfEmployees

givenEmployeeObject_whenUpdateEmployee_thenReturnUpdatedEmployee

givenEmployeeEmail_whenFindByEmail_thenReturnEmployeeObject

givenInValidEmployeeEmail_whenFindByEmail_thenReturnEmployeeObject

In this course, we use this naming Convention - given/when/then

Understanding Spring boot starter test dependency

Spring boot starter test dependency

The **Spring Boot Starter Test** dependency is a primary dependency for testing the Spring Boot Applications. It holds all the necessary elements required for the testing.

This starter includes:

1. Spring-specific dependencies
2. Dependencies for auto-configuration
3. Set of testing libraries - **JUnit**, **Mockito**, **Hamcrest**, **AssertJ**, **JSONassert**, and **JsonPath**.

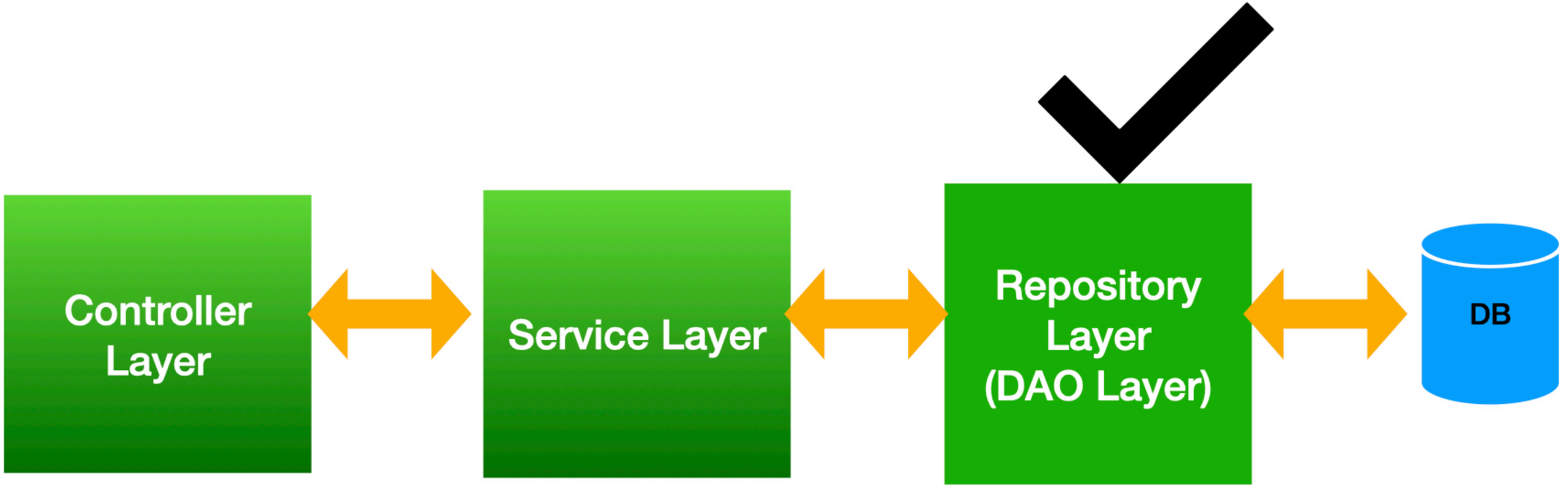
Spring boot starter test dependency

When using this starter, we don't need to update the versions of all the dependencies (**JUnit**, **Mockito**, **Hamcrest**, **AssertJ**, **JSONassert**, and **JsonPath**) manually. The Spring Boot parent POM handles all dependency versions, and the Spring Boot team ensures the different testing dependencies work properly together.

@DataJpaTest annotation

By Ramesh Fadatare (Java Guides)

JUnit Tests for Spring Data JPA



@DataJpaTest annotation

Spring Boot provides the @DataJpaTest annotation to test the persistence layer components that will autoconfigure in-memory embedded database for testing purposes.

The @DataJpaTest annotation doesn’t load other Spring beans (@Components, @Controller, @Service, and annotated beans) into ApplicationContext.

By default, it scans for @Entity classes and configures Spring Data JPA repositories annotated with @Repository annotation

By default, tests annotated with @DataJpaTest are transactional and roll back at the end of each test.

@DataJpaTest annotation

To test Spring Data JPA repositories or any other JPA-related components for that matter, Spring Boot provides the @DataJpaTest annotation.

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
class UserEntityRepositoryTest {

    @Autowired private DataSource dataSource;
    @Autowired private JdbcTemplate jdbcTemplate;
    @Autowired private EntityManager entityManager;
    @Autowired private UserRepository userRepository;

    @Test
    void injectedComponentsAreNotNull(){
        assertThat(dataSource).isNotNull();
        assertThat(jdbcTemplate).isNotNull();
        assertThat(entityManager).isNotNull();
        assertThat(userRepository).isNotNull();
    }
}
```

AssertJ Overview

By Ramesh Fadatare (Java Guides)

AssertJ Overview

AssertJ is a Java library that provides a rich set of assertions and truly helpful error messages, improves test code readability, and is designed to be super easy to use within your favorite IDE.

When we want to write assertions with AssertJ, we have to use the static assertThat() method of the org.assertj.core.api.Assertions class.

Spring boot starter test dependency internally provides assertj-core dependency so we don't have to add assertj-core dependency manually in our Spring boot project

Steps to use AssertJ

import static org.assertj.core.api.Assertions.assertThat;

```
// basic assertions
assertThat(frodo.getName()).isEqualTo("Frodo");
assertThat(frodo).isNotEqualTo(sauron);

// chaining string specific assertions
assertThat(frodo.getName()).startsWith("Fro")
                           .endsWith("do")
                           .isEqualToIgnoringCase("frodo");

// collection specific assertions (there are plenty more)
// in the examples below fellowshipOfTheRing is a List<TolkienCharacter>
assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
                               .doesNotContain(sauron);

// as() is used to describe the test and will be shown before the error message
assertThat(frodo.getAge()).as("check %s's age", frodo.getName()).isEqualTo(33);
```

Why I use AssertJ?

- Easy to learn: Quick start
- Easy to use: you just need to add a dependency and static import in your test class to start using AssertJ.
- Fluent APIs: AssertJ helps you to diversify your assertions.
- More readable code auto-completion: AssertJ provides auto-completion in IDEs. So you don't need to remember all method names.

You can use JUnit 5 assertions

JUnit 5 assertAll
JUnit 5 assertArrayEquals
JUnit 5 assertEquals
JUnit 5 assertNotEquals
JUnit 5 assertNotSame
JUnit 5 assertNull and assertNotNull
JUnit 5 assertSame
JUnit 5 assertThrows
JUnit 5 assertTimeout
JUnit 5 assertFalse
JUnit 5 assertTrue

Develop JUnit Tests in BDD Style using BBDMockito class

By Ramesh Fadatare (Java Guides)

Behavior-Driven development (BDD)

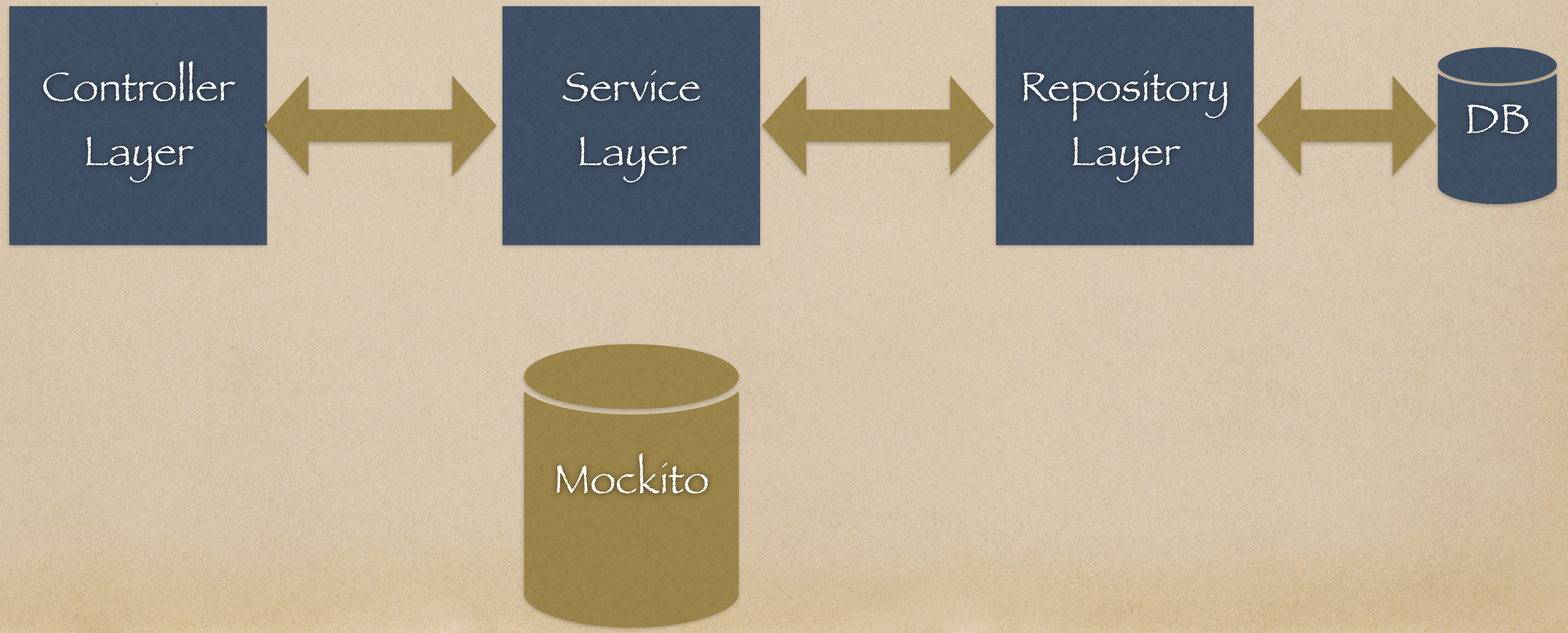
BDD encourages writing tests in a natural, human-readable language that focuses on the behavior of the application.

We write Unit Tests using a Behavior-Driven Development style (BDD) to increase the test readability (a lot).

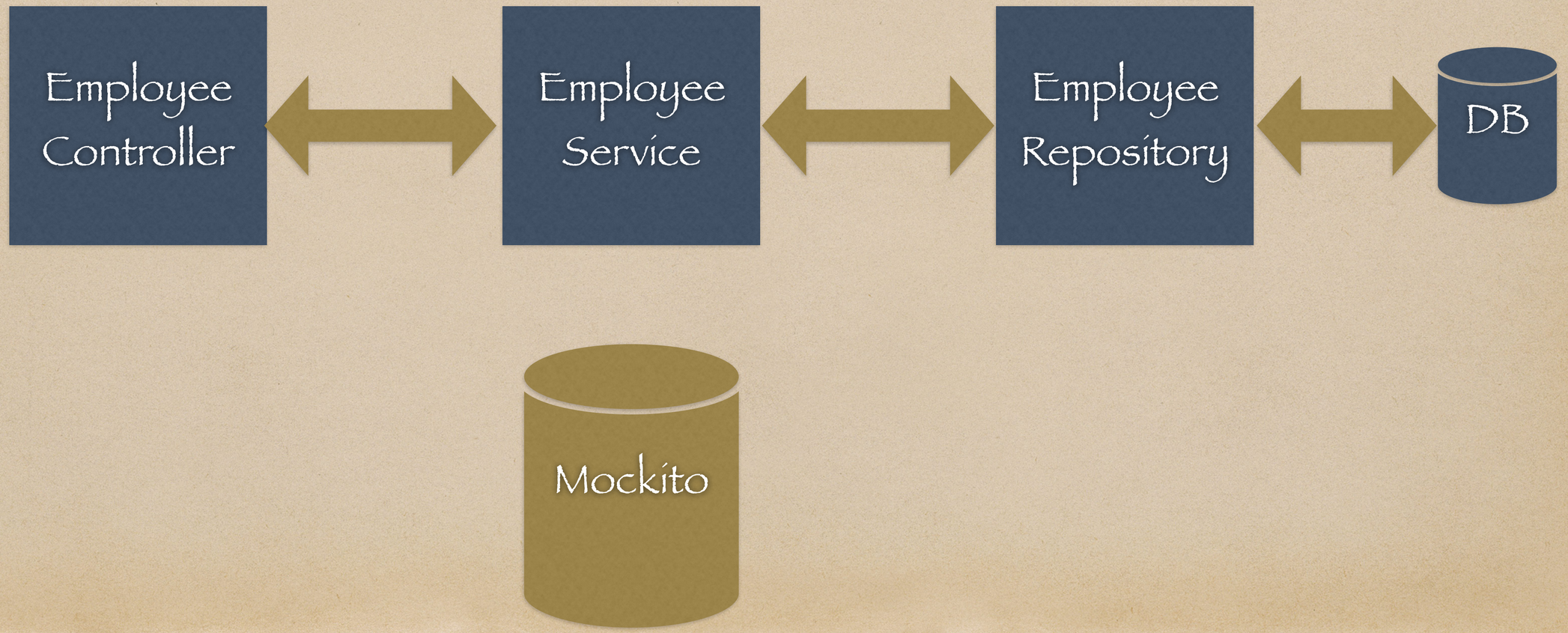
It defines a clearly structured way of writing tests following three sections (Arrange, Act, Assert):

- **given** some preconditions (Arrange)
- **when** an action occurs (Act)
- **then** verify the output (Assert)

Spring Boot Application
Service Layer Testing



Spring Boot Application
Service Layer Testing



Mocking Dependencies using Mockito

Mockito mock() method - We can use Mockito class mock() method to create a mock object of a given class or interface. This is the simplest way to mock an object.

Mockito @Mock Annotation - We can mock an object using @Mock annotation too. It's useful when we want to use the mocked object at multiple places because we avoid calling mock() method multiple times. The code becomes more readable and we can specify mock object name that will be useful in case of errors.

Mockito @InjectMocks Annotation

When we want to inject a mocked object into another mocked object, we can use @InjectMocks annotation. @InjectMock creates the mock object of the class and injects the mocks that are marked with the annotations @Mock into it.

```
@ExtendWith(MockitoExtension.class)
public class EmployeeServiceTest {

    @Mock
    private EmployeeRepository employeeRepository;

    @InjectMocks
    private EmployeeServiceImpl employeeService;
```

BDDMockito Class

The Mockito library is shipped with a BDDMockito class which introduces BDD-friendly APIs.

Example: BDD style writing tests uses **//given //when //then** comments

```
@Test
public void givenEmployeesList_whenGetAllEmployees_thenGetEmployeesList(){

    // given (precondition)
    given(employeeRepository.findAll()).willReturn(List.of(employee, employee1));

    // when (action occurs)
    List<Employee> employees = employeeService.getAllEmployees();

    // then (verify the output)
    assertThat(employees.size()).isEqualTo(2);
}
```

Mockito vs. BDDMockito

The traditional mocking in Mockito is performed using **when(obj).thenReturn()** in the Arrange step.

```
// given
Mockito.when(employeeRepository.findAll()).thenReturn(List.of(employee, employee1));
```

This BDDMockito allows us to take a more BDD friendly approach arranging our tests using given().willReturn().

```
// given
given(employeeRepository.findAll()).willReturn(List.of(employee, employee1));
```

Steps to use BDDMockito API

1. Add static import statement

```
import static org.mockito.BDDMockito.given;
```

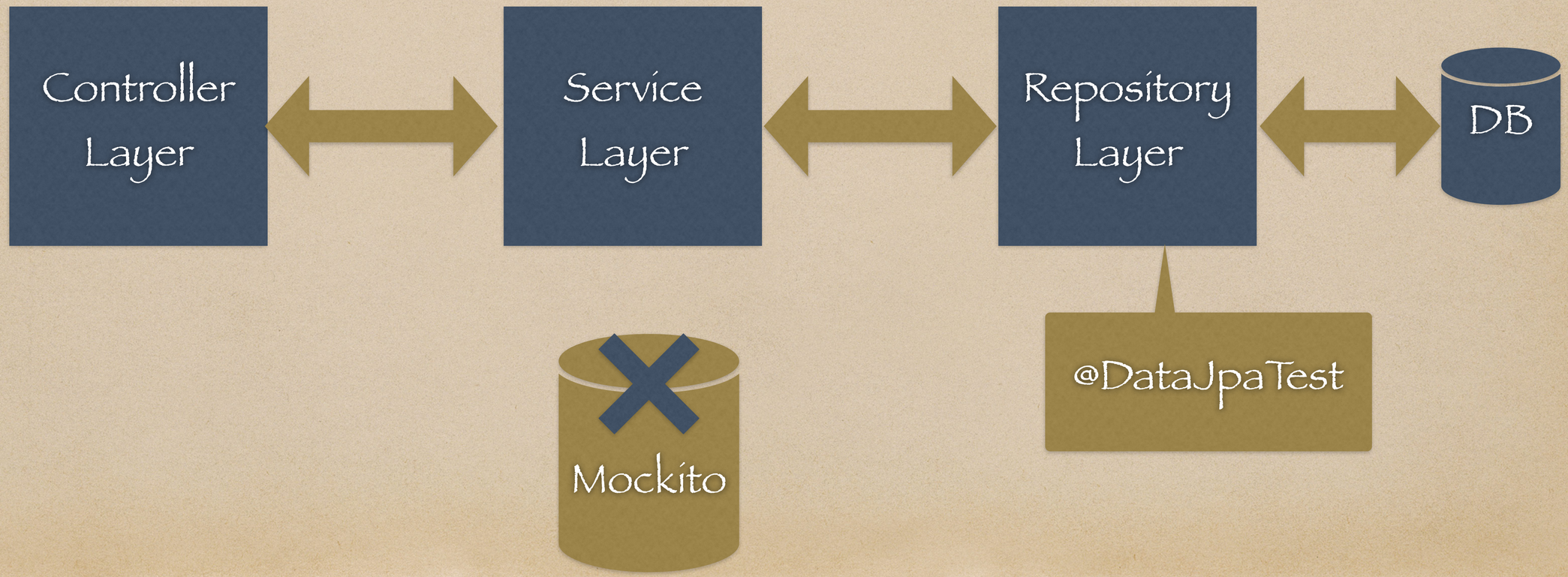
2. Use given().willReturn() method

```
// given
given(employeeRepository.findAll()).willReturn(List.of(employee, employee1));
```

Spring Boot Application Repository Layer Testing

By Ramesh Fadatare (Java Guides)

Spring Boot Application
Repository Layer Testing



Repository Testing

1. Overview of @DataJpaTest annotation
2. Unit test for save employee operation
3. Unit test for get all employees operation
4. Unit test for get employee by id operation
5. Unit test for get employee by email operation (custom query)
6. Unit test for update employee operation
7. Unit test for delete employee operation
8. Unit test for custom query using JPQL with index parameters (using @Query annotation)
9. Unit test for custom query using JPQL with named parameters (using @Query annotation)
10. Unit test for custom Native query with index parameters (using @Query annotation)
11. Unit test for custom Native query with Named parameters (using @Query annotation)
12. Refactoring JUnit tests to use @BeforeEach annotation

JUnit tests in BDD Style

Syntax

```
@Test
public void given_when_then() {
    // given - precondition or setup

    // when - action or the behaviour we're testing

    // then - verify the output
}
```

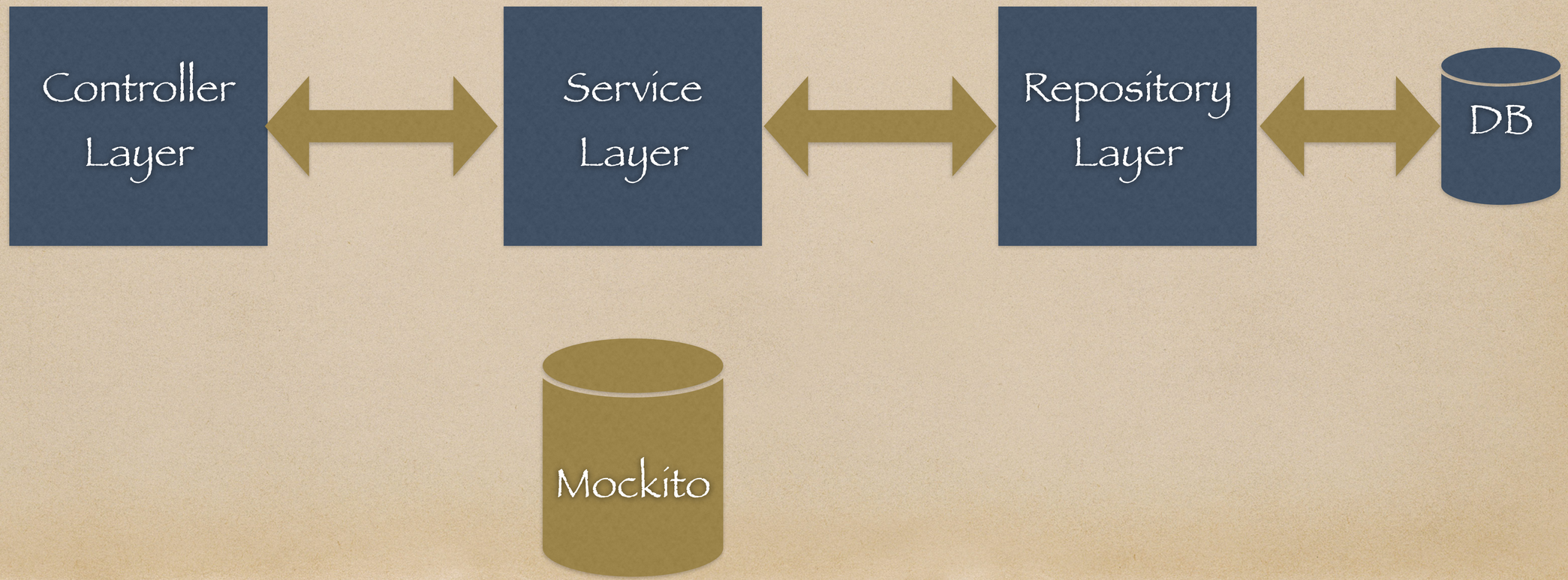
Example

```
@Test
public void givenEmployeeObject_whenSave_thenReturnSavedEmployee(

    //given - precondition or setup
    Employee employee = Employee.builder()
            .firstName("Ramesh")
            .lastName("Ramesh")
            .email("ramesh@gmail,com")
            .build();
    // when - action or the behaviour that we are going test
    Employee savedEmployee = employeeRepository.save(employee);

    // then - verify the output
    assertThat(savedEmployee).isNotNull();
    assertThat(savedEmployee.getId()).isGreaterThan(0);
}
```

Spring Boot Application Service Layer Testing

By Ramesh Fadatare (Java Guides)

Spring Boot Application
Service Layer Testing



Service Layer Testing

1. **Create EmployeeService with saveEmployee method**
2. **Quick Recap of Mockito basics (before writing Unit tests)**
3. **Unit test for EmployeeService saveEmployee method**
4. **Using @Mock and @InjectMocks annotations to mock the object**
5. **Unit test for saveEmployee method which throws Exception**
6. **Unit test for EmployeeService getAllEmployees method - Positive Scenario**
7. **Unit test for EmployeeService getAllEmployees method - Negative Scenario**
8. **Unit test for EmployeeService getEmployeeById method**
9. **Unit test for EmployeeService updateEmployee method**
10. **Unit test for EmployeeService deleteEmployee method**

JUnit tests in BDD Style

Syntax

```
@Test
public void given_when_then() {
    // given - precondition or setup

    // when - action or the behaviour we're testing

    // then - verify the output
}
```

Example

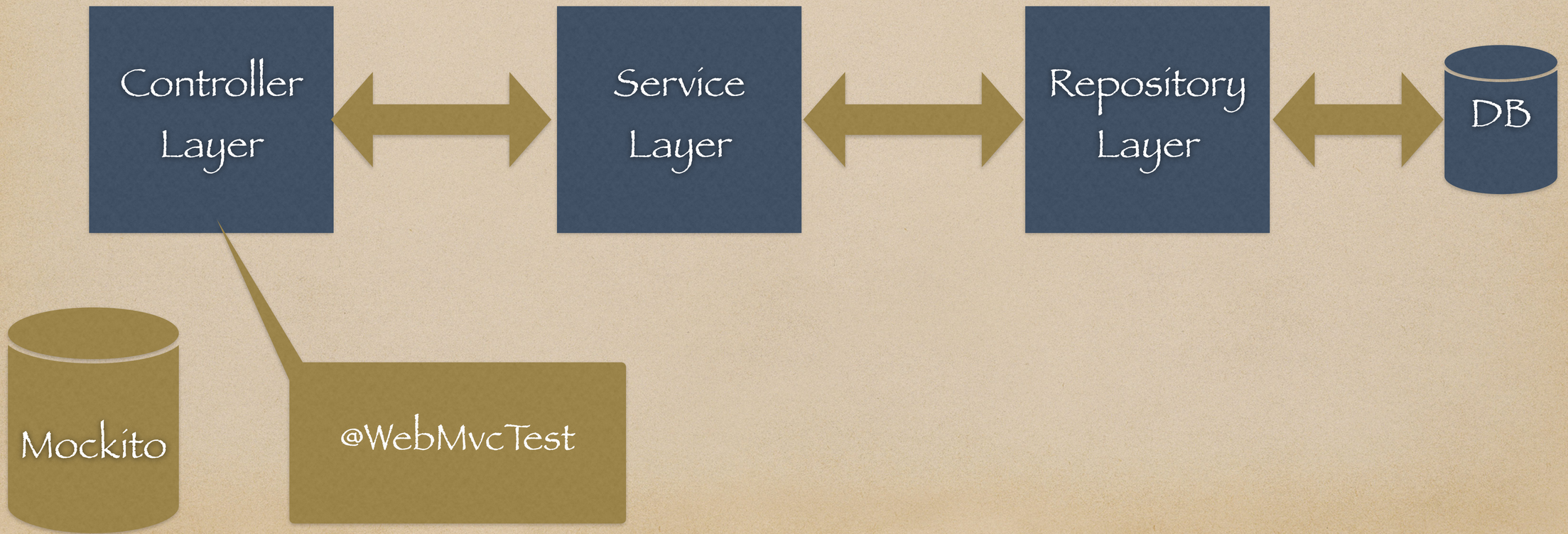
```
// JUnit test for saveEmployee method
@DisplayName("JUnit test for saveEmployee method which throws exception")
@Test
public void givenExistingEmail_whenSaveEmployee_thenThrowsException(){
    // given - precondition or setup
    given(employeeRepository.findByEmail(employee.getEmail()))
            .willReturn(Optional.of(emplo

    System.out.println(employeeRepository);
    System.out.println(employeeService);

    // when -  action or the behaviour that we are going test
    org.junit.jupiter.api.Assertions.assertThrows(ResourceNotFoundException.class, () -> {
        employeeService.saveEmployee(employee);
    });

    // then
    verify(employeeRepository, never()).save(any(Employee.class));
}
```

```
net.javaguides.springboot.service.EmployeeServiceTes
private Employee employee
```

Spring Boot Application Controller Layer Testing

By Ramesh Fadatare (Java Guides)

Spring Boot Application
Controller Layer Testing



Controller Layer Testing

1. **@MockMvcTest Annotation**
2. **Build createEmployee REST API**
3. **Unit test createEmployee REST API**
4. **Build GetAllEmployees REST API**
5. **Unit test GetAllEmployees REST API**
6. **Refactoring JUnit test to use static imports**
7. **Build getEmployeeById REST API**
8. **Unit test getEmployeeById REST API - Positive Scenario**
9. **Unit test getEmployeeById REST API - Negative Scenario**
10. **Build updateEmployee REST API**
11. **Unit test updateEmployee REST API - Positive Scenario**
12. **Unit test updateEmployee REST API - Negative Scenario**
13. **Build deleteEmployee REST API**
14. **Unit test deleteEmployee REST API**

Hamcrest Library

Hamcrest is the well-known framework used for unit testing in the Java ecosystem. It's bundled in JUnit and simply put, it uses existing predicates – called matcher classes – for making assertions.

Hamcrest is commonly used with *JUnit* and other testing frameworks for making assertions. Specifically, instead of using *JUnit's* numerous *assert* methods, we only use the API's single *assertThat* statement with appropriate matchers.

Hamcrest is() method: If we want to verify that the expected value (or object) is equal to the actual value (or object), we have to create our Hamcrest matcher by invoking the *is()* method of the Matchers class.
Syntax:
assertThat(ACTUAL, is(EXPECTED));

JsonPath Library

A Java DSL for reading JSON documents.

JsonPath expressions always refer to a JSON structure in the same way as XPath expression are used in combination with an XML document. The "root member object" in JsonPath is always referred to as $ regardless if it is an object or array.

JSON

```
{
   "firstName": "Ramesh",
   “lastName": “Fadatare",
    “email": "ramesh@gmail.com"
}
```

JsonPath Expressions

$ - root member of a JSON structure whether it is an object or array.

$.firstName = “Ramesh”

$.lastName = “Fadatare”

$.email = “ramesg@gmail.com”

JUnit tests in BDD Style

Syntax

```
@Test
public void given_when_then() {
    // given - precondition or setup

    // when - action or the behaviour we're testing

    // then - verify the output
}
```

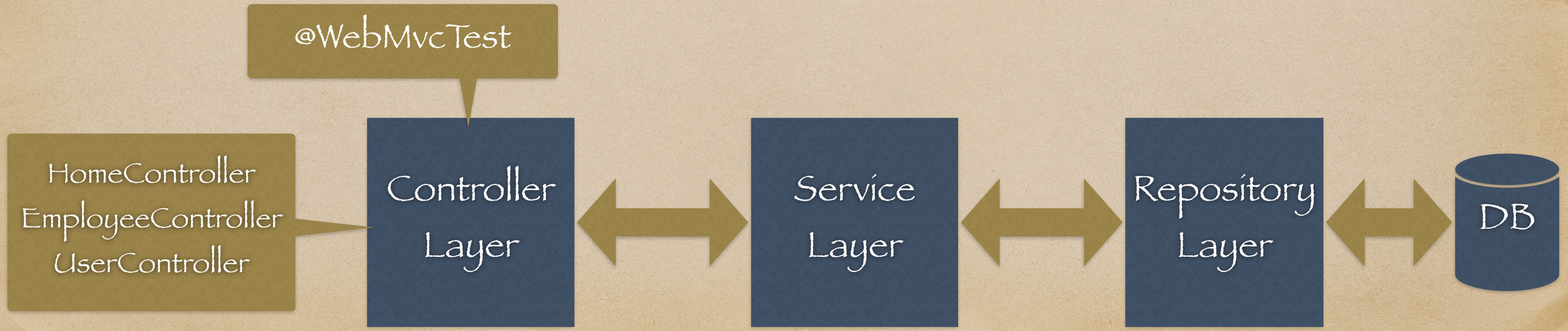
Example

```
// positive scenario - valid employee id
// JUnit test for GET employee by id REST API
@Test
public void givenEmployeeId_whenGetEmployeeById_thenReturnEmployeeObject() throws Exception{
    // given - precondition or setup
    long employeeId = 1L;
    Employee employee = Employee.builder()
            .firstName("Ramesh")
            .lastName("Fadatare")
            .email("ramesh@gmail.com")
            .build();
    given(employeeService.getEmployeeById(employeeId)).willReturn(Optional.of(employee));

    // when -  action or the behaviour that we are going test
    ResultActions response = mockMvc.perform(get( urlTemplate: "/api/employees/{id}", employeeId));

    // then - verify the output
    response.andExpect(status().isOk())
            .andDo(print())
            .andExpect(jsonPath( expression: "$.firstName", is(employee.getFirstName())))
            .andExpect(jsonPath( expression: "$.lastName", is(employee.getLastName())))
            .andExpect(jsonPath( expression: "$.email", is(employee.getEmail())));

}
```

@WebMvcTest Annotation

SpringBoot provides **@WebMvcTest** annotation to test Spring MVC Controllers.
Also, **@WebMvcTest** based tests runs faster as it will load only the specified controller and its dependencies only without loading the entire application.

Spring Boot instantiates only the web layer rather than the whole application context. In an application with multiple controllers, you can even ask for only one to be instantiated by using, for example, **@WebMvcTest(HomeController.class).**



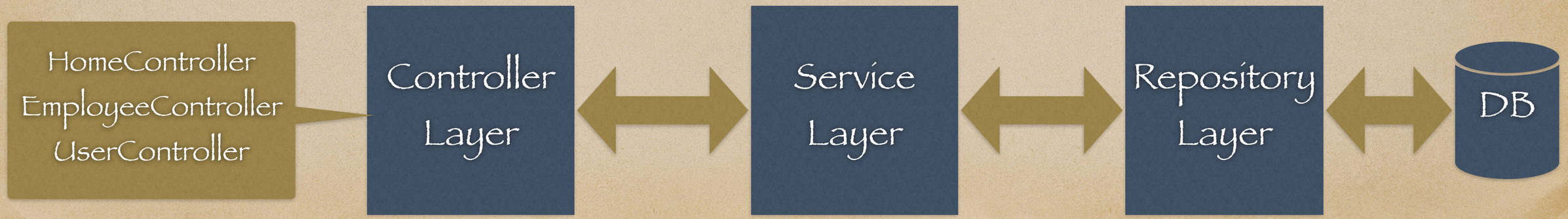
@WebMvcTest vs @SpringBootTest

Spring Boot provides @WebMvcTest annotation to test Spring MVC controllers. This annotation creates an application context that contains all the beans necessary for testing a Spring web controller.

Spring Boot provides @SpringBootTest annotation for Integration testing. This annotation creates an application context and loads full application context.

Unit testing - @WebMvcTest annotation

Integration testing - @SpringBootTest



Spring Boot Application Integration Testing

By Ramesh Fadatare (Java Guides)

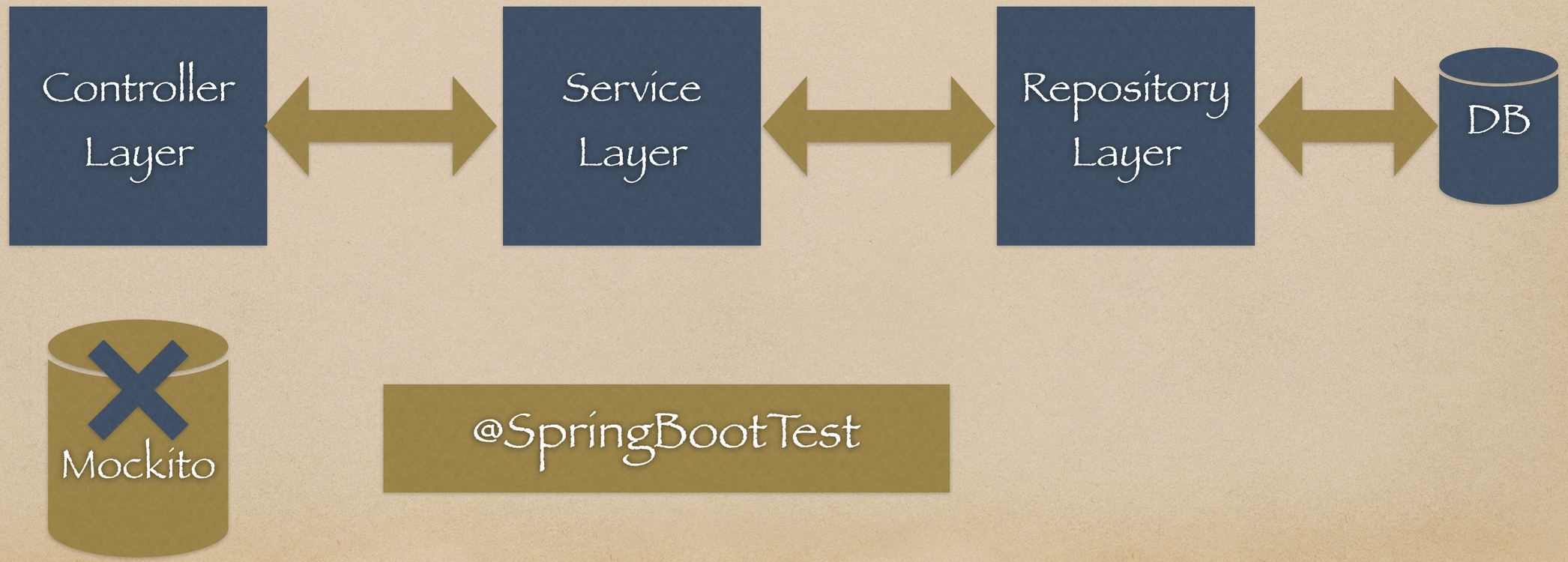
Integration Testing

As the name suggests, integration tests focus on integrating different layers of the application. That also means no mocking is involved.

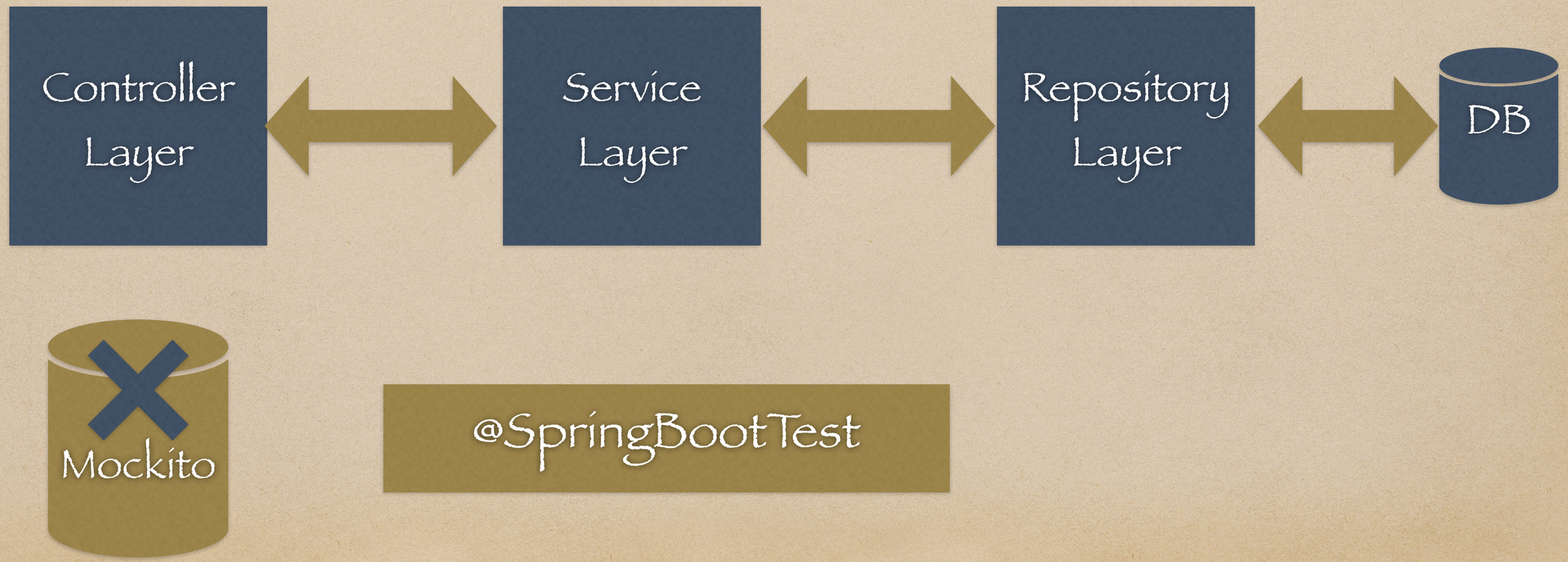
Basically, we write integration tests for testing a feature which may involve interaction with multiple components.

Examples:
Employee Management Feature (EmployeeRepository, EmployeeService, EmployeeController).
User Management Feature (UserController, UserService, and UserRepository).
Login Feature (LoginRespository, LoginController, Login Service) etc

Spring Boot Application Integration Testing



Spring Boot Application Integration Testing



Spring Boot Integration Testing

1. **@SpringBootTest annotation overview**
2. **Integration test save employee feature**
3. **Integration test get all employees feature**
4. **Integration test get employee by id feature**
5. **Integration test update employee feature**
6. **Integration test delete employee feature**

@SpringBootTest

Spring Boot provides **@SpringBootTest** annotation for Integration testing. This annotation creates an application context and loads full application context.

@SpringBootTest will bootstrap the full application context, which means we can **@Autowire** any bean that's picked up by component scanning into our test.

Integration testing - @SpringBootTest

@SpringBootTest

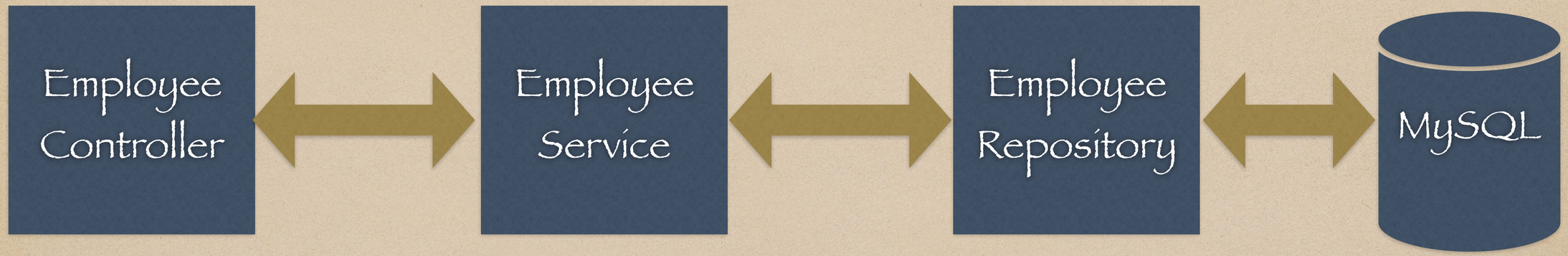
It starts the embedded server, creates a web environment and then enables **@Test** methods to do integration testing.

By default, **@SpringBootTest** does not start a server. We need to add attribute webEnvironment to further refine how your tests run. It has several options:

- ★ **MOCK(Default):** Loads a web ApplicationContext and provides a mock web environment
- ★ **RANDOM_PORT:** Loads a WebServerApplicationContext and provides a real web environment. The embedded server is started and listen on a random port. This is the one should be used for the integration test
- ★ **DEFINED_PORT:** Loads a WebServerApplicationContext and provides a real web environment.
- ★ **NONE:** Loads an ApplicationContext by using SpringApplication but does not provide any web environment

Steps for Integration Testing

1. We will add MySQL driver dependency to our Spring boot application. Of course you can use h2 in-memory database for integration testing



2. Add MySQL configuration in application.properties file

3.Write Integration tests for EmployeeController