Best Practices For Unit Testing In Java

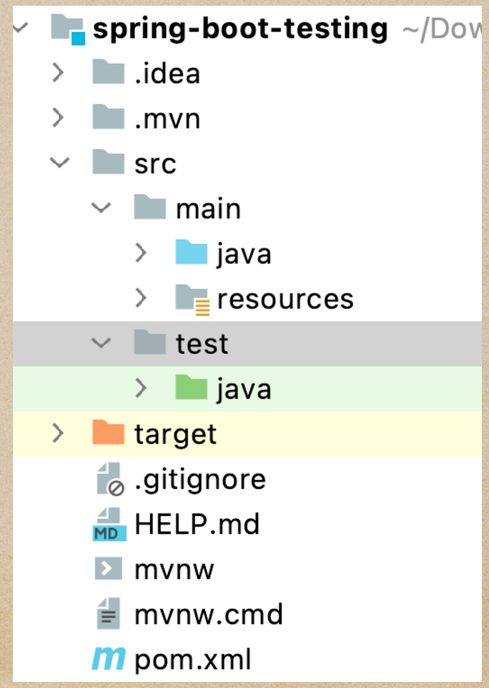
By Ramesh Fadatare (Java Guides)

Source Code Structure

It's a good idea to keep the test classes separate from the main source code. So, they are developed, executed, and maintained separately from the production code.

Also, it avoids any possibility of running test code in the production environment.

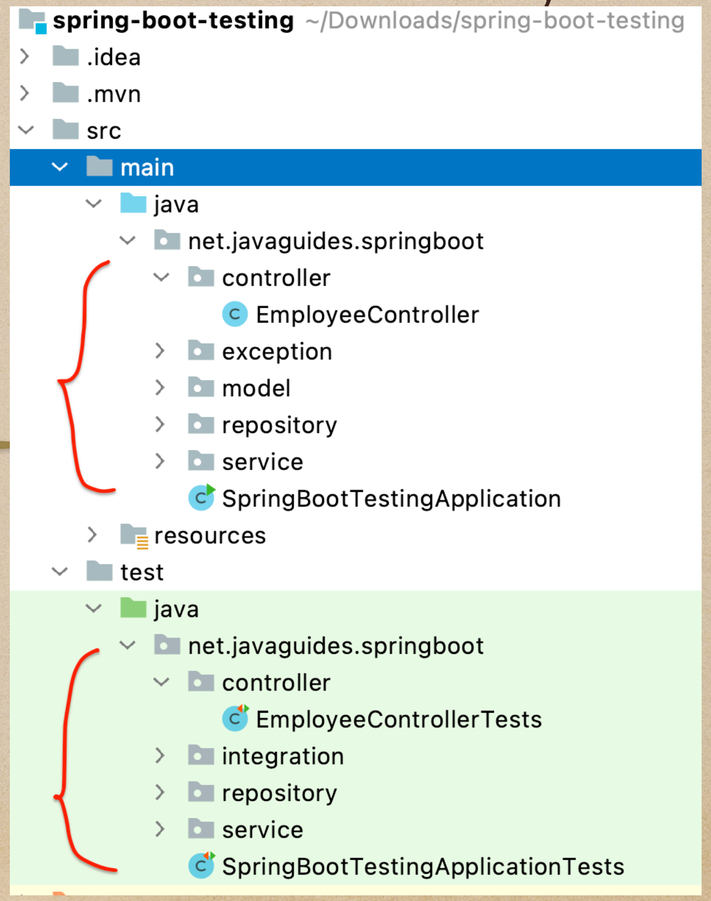
We can follow the steps of the **build tools like Maven and Gradle that look for *src/test* directory for test implementations**.



Package Naming Convention

We should create a similar package structure in the *src/test* directory for test classes. Thus, improving the readability and maintainability of the test code.

The **package of the test class should match the package of the source class** whose unit of source code it'll test.



Unit Test Case Naming Convention

The test names should be insightful, and users should understand the behavior and expectation of the test by just glancing name itself.

Given/when/then BDD style

givenEmployeeObject_whenSaveEmployee_thenReturnSavedEmployee

givenEmployeesList_whenFindAll_thenReturnListOfEmployees

givenEmployeeObject_whenUpdateEmployee_thenReturnUpdatedEmployee

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

Appropriate Assertions

Always **use proper assertions to verify the expected vs. actual results**. We should use various methods available in the *Assert* class of JUnit or similar frameworks like AssertJ.

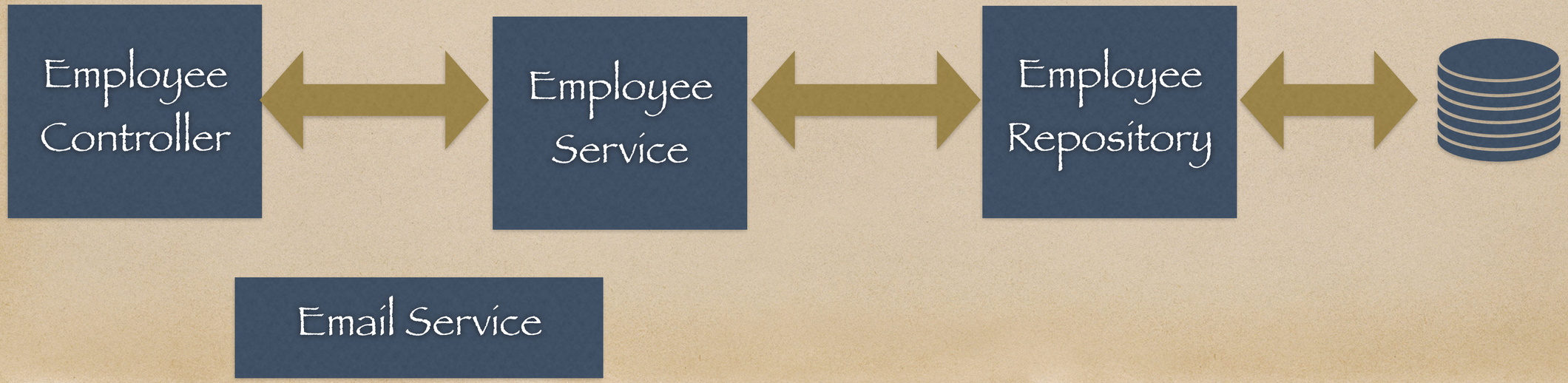
```java
@DisplayName("JUnit test for getAllEmployees method")
@Test
public void givenEmployeesList_whenGetAllEmployees_thenReturnEmployeesList(){
    // given - precondition or setup

    Employee employee1 = Employee.builder()
            .id(2L)
            .firstName("Tony")
            .lastName("Stark")
            .email("tony@gmail.com")
            .build();

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

    // when -  action or the behaviour that we are going test
    List<Employee> employeeList = employeeService.getAllEmployees();

    // then - verify the output
    assertThat(employeeList).isNotNull();
    assertThat(employeeList.size()).isEqualTo(2);
}
```

Mock External Services

Although unit tests concentrate on specific and smaller pieces of code, there is a chance that the code is dependent on external services for some logic.

Therefore, we should mock the external services and merely test the logic and execution of our code for varying scenarios.

We can use various frameworks like Mockito, EasyMock, and JMockit for mocking external services.



Specific Unit Tests

Instead of adding multiple assertions to the same unit test, we should create separate test cases.

Of course, it's sometimes tempting to verify multiple scenarios in the same test, but it's a good idea to keep them separate. Then, in the case of test failures, it'll be easier to determine which specific scenario failed and, likewise, simpler to fix the code.

Therefore, always write a unit test to test a single specific scenario.