Published in Dev Genius

Md Sajedul Karim Follow
Apr 6, 2021 · 11 min read · · Listen

Save

Controller, Service, and Repository Layer Unit Testing using JUnit and Mockito

The principal objective of software testing is to give confidence in the software.— Anonymous.

Photo by freestocks on Unsplash

UNIT TESTING is a type of software testing where individual units or components of the software are tested. The purpose is to validate that each unit of the software code performs as expected.

Unit Testing is done during the development (coding phase) of an application by the developers.

Today I will cover up bellow topics

1. Importance of software testing, Test Driven Development (TDD) basics, and aspects of test cases
2. Details of terms related to test cases: unit testing, integration testing, Mocking, Spying, Stubbing
3. In spring boot apps controller, service, and repository layer unit testing
4. Tips for writing testable code
5. Share the codebase and related files

Prerequisite

Before starting this tutorial, you have to know details about spring boot and JPA. To learn this you may read my below medium article.

Deep Dive on Spring Boot and JPA Implementation: A to Z

Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA-based repositories.

mesukcse08.medium.com

. . .

Importance of software testing, Test Driven Development (TDD) basics, and aspects of test cases

Test Case/s is a specific set of instructions that the tester is expected to follow to achieve a specific output. Test cases are documented keeping in mind the requirements provided by the client. The key purpose of a test case is to ensure if different features within an application are working as expected. It helps the tester, validate if the software is free of defects and if it is working as per the expectations of the end-users. Other benefits of test cases include:

- Test cases ensure good test coverage
- Help improve the quality of software
- Decreases the maintenance and software support costs
- Help verify that the software meets the end-user requirements
- Unit tests are a kind of living documentation of the product. To learn what functionality is provided by one module or another, developers can refer to unit tests to get a basic picture of the logic of the module and the system as a whole
- Allows the tester to think thoroughly and approach the tests from as many angles as possible
- Test cases are reusable for the future — anyone can reference them and execute the test.
- Best practices suggest that developers first run all unit tests or a group of tests locally to make sure any coding changes don't disrupt the existing code.
- However, consider the human factor: A developer might forget to run unit tests after making changes and submit potentially non-working code to a common branch. To avoid this, many companies apply a continuous development approach. Tools for continuous integration are used for this, allowing developers to run unit tests automatically. Thus, any unwanted changes in the code will be detected by a cold, logical machine.

Test-driven development (TDD)

Test-Driven Development (TDD) is a software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each functionality are created and tested first and if the test fails then the new code is written in order to pass the test and make the code simple and bug-free.

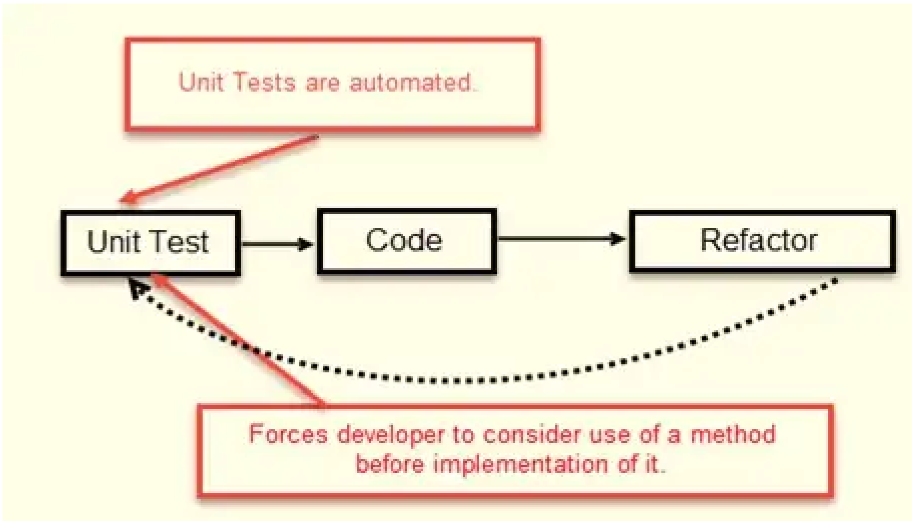

Unit Tests are automated.
Unit Test
Code
Refactor
Forces developer to consider use of a method before implementation of it.

Photo from: https://www.guru99.com/test-driven-development.html

Test-Driven Development starts with designing and developing tests for every small functionality of an application. TDD instructs developers to write new code only if an automated test has failed. This avoids duplication of code. The full form of TDD is Test-driven development.

The simple concept of TDD is to write and correct the failed tests before writing new code (before development). This helps to avoid duplication of code as we write a small amount of code at a time in order to pass tests. (Tests are nothing but requirement conditions that we need to test to fulfill them).

What aspects does a test case contain?

The test cases contain mainly the below sections

- **Objective**
  Here the tester mentions what he plans to achieve with that particular test case. The tester must have end-to-end business logic of that unit of code.
- **Pre-Conditions**
  Any assumptions that apply to the test and any preconditions that must be met prior to the test being executed should be listed here. Any dependency's, any pre initialization must stay here.
- **Test Case Steps**
  Here the tester mentions the steps that need to be followed to achieve the objective. The test steps should include the necessary data and information on how to execute the test. The steps should be clear and brief, without leaving out essential facts.

- **Test data**
  It's important to select a proper data set that gives sufficient coverage. The selected data set must cover all possible positive and negative scenarios/cases. Increase the number of test cases to ensure more software stability.

- **Expected and actual Result**
  The tester must have a clear knowledge of business logic and data behavior. The expected results tell the tester what they should experience as a result of the test steps. They also specify how the application actually behaved while test cases were being executed.

- **Pass/ Fail**
  If the tester fails to achieve the 'Expected Output' by following the steps then he will mention 'Fail' against that particular test case. Similarly, if the tester is able to achieve the 'Expected Output' then he will mention 'Pass' against the test case.

2. Details of terms related to test cases: unit testing, integration testing, Mocking, Spying, Stubbing

- **Unit testing**

1. UNIT TESTING is a type of software testing where individual units or components of the software are tested.
2. The purpose is to validate that each unit of the software code performs as expected.
3. Unit Testing is done during the development (coding phase) of an application by the developers.
4. Unit Tests isolate a section of code and verify its correctness.
5. A unit may be an individual function, method, procedure, module, or object.

- **Integration testing**

1. **INTEGRATION TESTING** is defined as a type of testing where software modules are integrated logically and tested as a group
2. The purpose of this level of testing is to expose defects in the interaction between these software modules when they are integrated

3. Integration Testing focuses on checking data communication amongst these modules.
4. It deals with the verification of the high and low-level software requirements specified in the Software Requirements Specification/Data and the Software Design Document.

- **jUnit**

1. JUnit is an open-source Unit Testing Framework for JAVA. It is useful for Java Developers to write and run repeatable tests.
2. As the name implies, it is used for Unit Testing of a small chunk of code.
3. **TestNG** is also a good alternative to **jUnit** in java

- **Mocking**

1. Mocking is the act of removing external dependencies from a unit test in order to create a controlled environment around it.
2. Mocking is a process used in unit testing when the unit being tested has external dependencies.
3. The purpose of mocking is to isolate and focus on the code being tested and not on the behavior or state of external dependencies.
4. In a mocking, the dependencies are replaced by closely controlled replacements objects that simulate the behavior of the real ones.

Typically, we mock all other classes that interact with the class that we want to test. Common targets for mocking are:

- Database connections,
- Web services,
- Classes that are slow,
- Classes with side effects, and
- Classes with non-deterministic behavior.

- Class/method those have legacy code and those are not testable yet
- **Stubbing**
  Mocks and stubs are fake Java classes that replace these external dependencies. These fake classes are then instructed before the test starts to behave as you expect.

1. A stub is a fake class that comes with preprogrammed return values.
2. **Stubbing** means replacing a method, function, or an entire object with a version that produces hard-coded responses.
3. This is typically used to isolate components from each other, and your code from the outside world.
4. It's injected into the class under test to give you absolute control over what's being tested as input.
5. A typical stub is a database connection that allows you to mimic any scenario without having a real database.
6. For example, stubbing is often used to decouple tests from storage systems and to hard-code the result of HTTP requests to test code that relies on data from the internet.

- **Spying**

1. Annotation that can be used to apply Mockito spies to a Spring ApplicationContext.
2. a spy *wraps* around an *existing* object of your class under test.
3. It contains mock and stubs both advantages
4. Meaning: when you create a spy, you can decide if method calls going to the spy should be "intercepted" (then you are using the spy as if it would be a mock); or be "passed through" to the actual object the spy wraps around.

. . .

3. Practical implementation of each term on controller, service, and repository layer

In a typical project follow the MVC pattern, where there are some controllers, some services, and some repository for data access. In this section, we will write all layers.

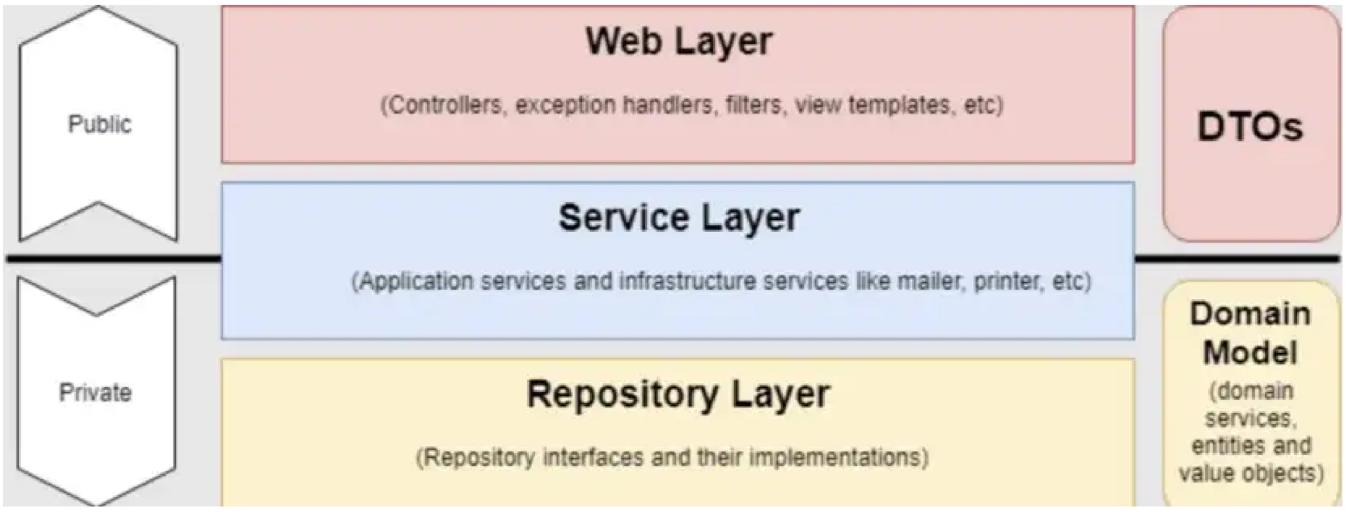
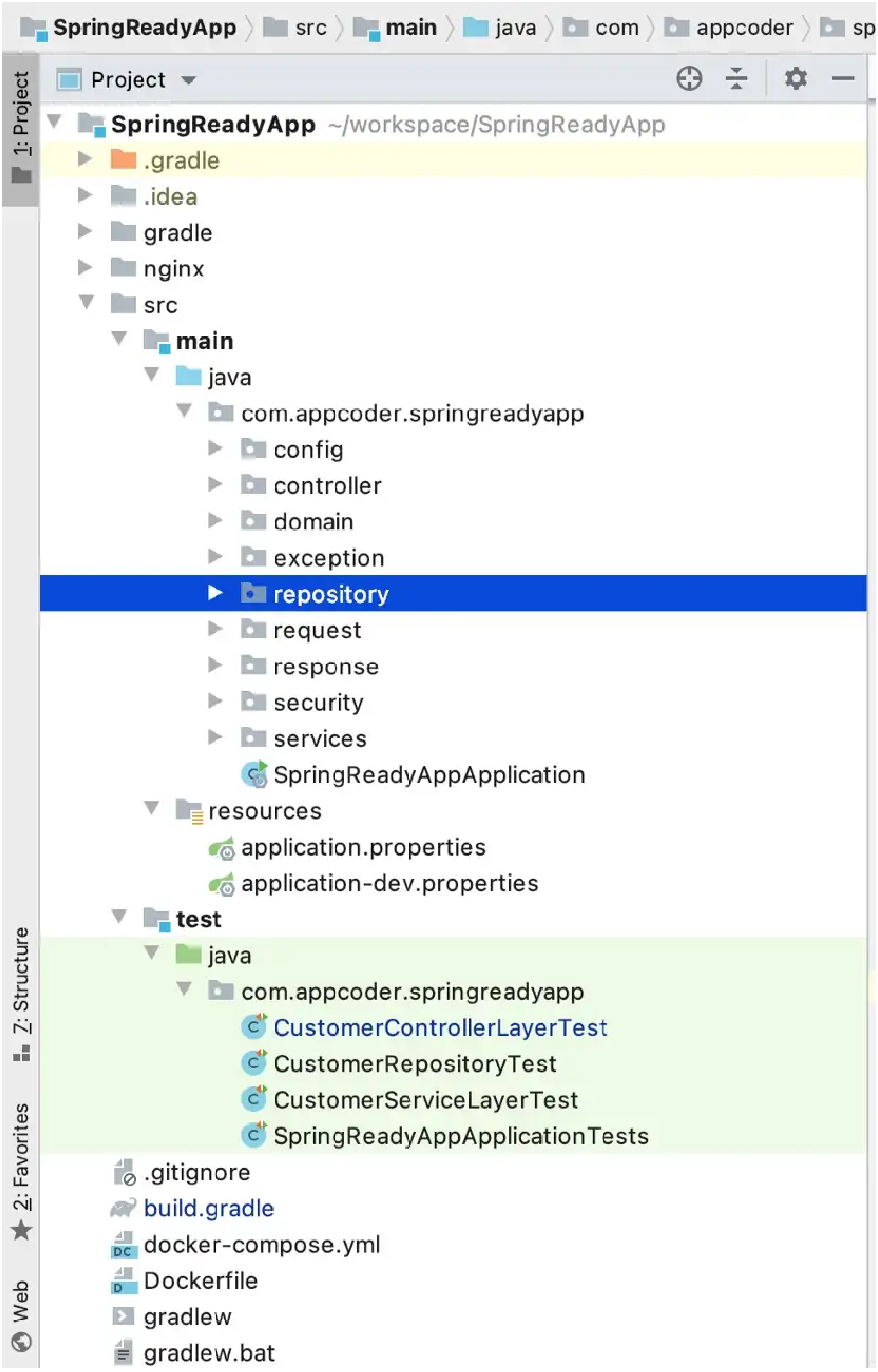

Public
Private
Web Layer
(Controllers, exception handlers, filters, view templates, etc)
Service Layer
(Application services and infrastructure services like mailer, printer, etc)
Repository Layer
(Repository interfaces and their implementations)
DTOs
Domain Model
(domain services, entities and value objects)

Image from www.mitrais.com

Setup the environment

We will work on Gradle based project. We require only three dependencies for cover the testing

We used **spring-boot-starter-test** for testing framework, **mockito** and **Jupiter.** We also used **spring-boot-starter-data-jpa** for JPA and **h2database** for the in-memory databases. The project structure is given bellow



```
SpringReadyApp ~/workspace/SpringReadyApp
  .gradle
  .idea
  gradle
  nginx
  src
    main
      java
        com.appcoder.springreadyapp
          config
          controller
          domain
          exception
          repository
          request
          response
          security
          services
          SpringReadyAppApplication
      resources
        application.properties
        application-dev.properties
    test
      java
        com.appcoder.springreadyapp
          CustomerControllerLayerTest
          CustomerRepositoryTest
          CustomerServiceLayerTest
          SpringReadyAppApplicationTests
  .gitignore
  build.gradle
  docker-compose.yml
  Dockerfile
  gradlew
  gradlew.bat
```

README.md

services in the service package, and repositories in the repository package.

1. Testing repository layer

Here, the repository is based on spring **@Repository** annotation. Using repository we execute some simple CRUD operations on our database. Our domain class is given below.

In this entity class, we have mainly firstName, lastName, and mobileNumber fields. To access data from this table, our repository class is like bellow

From this repository class, we can execute CRUD operation on Entity **Customer**

Our test code is as bellow.

From our test case class, we have to know some keywords.

1. **@ExtendWith(MockitoExtension.class):** enabling the mockito extension .
2. **@DataJpaTest:** Annotation that will prepare spring data JPA. It will enable entity-based save, fetch, and other environments.
3. **@BeforeEach**: Here before execution started, we can initialize some tasks. Here, we are saving a default customer.
4. **@AfterEach**: AFter execution of all cases, we are doing some tasks here. Here we are removing all our changed data.

Save data on the table

Save a list of entity code is bellow

```
<S extends Customer> Iterable<S> saveAll(Iterable<S> entities);
```

Look at the test case

```
void saveAll_success() {
    List<Customer> customers = Arrays.asList(
            new Customer("sajedul", "karim", "01737186095"),
            new Customer("nafis", "khan", "01737186096"),
            new Customer("aayan", "karim", "01737186097")
    );
    Iterable<Customer> allCustomer = 
customerRepository.saveAll(customers);

    AtomicInteger validIdFound = new AtomicInteger();
    allCustomer.forEach(customer -> {
        if(customer.getId()>0){
            validIdFound.getAndIncrement();
        }
    });

    assertThat(validIdFound.intValue()).isEqualTo(3);
}
```

Here, we are building 3 customer objects and savings using the customer repository. After saving data we are checking the size of saved data. Here, validity is each entity id must be greater than 0 after savings new data. Here real database transaction is occurring.

Fetch and match data

The repository fetch method is given bellow

```
@Override
List<Customer> findAll();
```

And our test code is given bellow

```
@Test
void findAll_success() {
    List<Customer> allCustomer = customerRepository.findAll();
    assertThat(allCustomer.size()).isGreaterThanOrEqualTo(1);
}
```

Here, we are fetching data and checking size 1. Our approximate max customer data size is 1 to 4. Because for default case execute of test cases don't maintain any order. In the @**BeforeEach** command, we are inserting only one customer. So we are checking **isGreaterThanOrEqualTo(1).**

2. Testing Service Layer

Here, the service is based on spring **@Service** annotation**.** Using service we check some business logic and save and fetch data to/from the database and return to our controllers.

Notes: We are testing the service layer. So we don't require a database layer actual operation. So we will make the database layer mock.

Our service layer code is given bellow

Here, we have few methods like save or update, fetch data. We will test all of these methods using unit testing.

Our unit testing code is given bellow

In this code, we are mocking the database layer by annotation @**Mock CustomerRepository**. Here, mocking means this is a dummy layer, no actual operation will happen during database save or fetch. For mocking the database layer we used some **stubbing**

Save or update
Our save or update test code is given bellow

```
@Test
public void savedCustomer_Success() {
    Customer customer = new Customer();
    customer.setFirstName("sajedul");
    customer.setLastName("karim");
    customer.setMobileNumber("01737186095");

    // providing knowledge

when(customerRepository.save(any(Customer.class))).thenReturn(custom
er);

    Customer savedCustomer = customerRepository.save(customer);
    assertThat(savedCustomer.getFirstName()).isNotNull();
}
```

Here, we are building a customer object. Here, the repository save object returns an object after successfully saved into the database. So we are telling mockito that if we try to save any customer object then you return us our provided customer object. Finally, we are checking this returned object. By stubbing we convert database operation with our predefined operation.

Fetch data from the database
Our code for fetching data is given bellow

```
@Test
public void customer_exists_in_db_success() {
    Customer customer = new Customer();
    customer.setFirstName("sajedul");
    customer.setLastName("karim");
    customer.setMobileNumber("01737186095");
    List<Customer> customerList = new ArrayList<>();
    customerList.add(customer);

    // providing knowledge
    when(customerRepository.findAll()).thenReturn(customerList);

    List<Customer> fetchedCustomers =
customerService.fetchAllCustomer();
    assertThat(fetchedCustomers.size()).isGreaterThan(0);
}
```

Here, repository **findAll()** method return a list of customer. So we are building a list of customers and stubbing the **findAll()** method and checking the data size after the operation.

3. Testing controller layer

In the controller layer, we are mocking the service layer and testing the API. The controller layer code is given bellow

Here, we have two endpoints one for save and another for fetch data from the service layer.

Our test code is given bellow

Here, we are mocking the **CustomerService** and for testing API we are using **MockMvc.** Here, also we are stubbing the service layer methods.

During testing, we are building request and after execution checking the status

. . .

4. Tips for writing testable code

We have to follow some coding rules for testable code. Those are given bellow

- Strictly follow the SOLID principal

- Use dependency injection properly. Use constructor injection or setter injection instead of Autowired. For autowired beans hard to test.
- Separate object creation and application logic. Otherwise, you have to rewrite your code before writing a test case
- Remove global state. Global state makes code more difficult to understand, as the user of those classes might not be aware of which variables need to be instantiated.
- Avoid static methods and variables. Static methods are procedural code and should be avoided in an object-oriented paradigm, as they don't provide the seams required for unit testing.
- Follow naming conventions strictly. When writing unit tests, it is important to be able to determine which properties and methods of an object are public and which are private implementation details of the object itself. This is because unit tests should only test the publicly defined APIs, as these are the only APIs that are guaranteed to exist and produce stable results.
- Clean code and well documented. Tests are not a substitute for a clear, well-maintained codebase. In fact, in order to write accurate tests, it is necessary that code is kept clean enough that test authors and future maintainers can quickly understand the purpose of each unit of code being tested and how it fits into the overall application.
- Code must be fast. Developers write unit tests so they can repeatedly run them and check that no bugs have been introduced. If unit tests are slow, developers are more likely to skip running them on their own machines.
- Don't mix up with unit tests and integration tests. As we already discussed, unit and integration tests have different purposes. Both the unit test and the system under test should not access the network resources, databases, file system, etc., to eliminate the influence of external factors.

. . .

5. Source code details

All source code is in this GitHub repository.

Controller Name : CustomerController

Service Name: CustomerServiceImpl

Repository Name: CustomerRepository

Test Classes: CustomerRepositoryTest, CustomerServiceLayerTest, CustomerControllerLayerTest

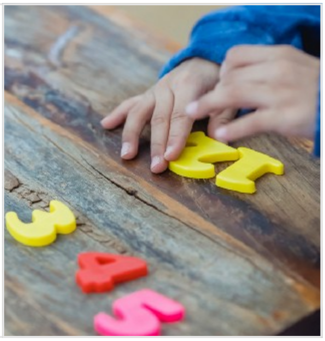
Swagger Url: http://localhost:8181/springreadyapp/swagger-ui.html#/MainController

Thanks in advance :)

Overview of SOLID Principles and its JAVA Implementations

To create understandable, readable, and testable code that many developers can collaboratively work on.

levelup.gitconnected.com



Unit Testing Spring Boot Mockito Jpa Junit

Thanks to Domenico Nicoli