JAVA BACKEND DEVELOPMENT PROGRAM

Testing

OUTLINE

- Testing
- JUnit
- Mockito
- Test Driven Development
- Code Coverage

TESTING

- What is testing?
  - An important part during development
    - Help us to check if code logic as expected
- Two major types:
  - Unit Test - focus on functional groups inside an application
  - Integration Test - these tests usually involve multiple services

JUNIT

- Unit testing is one of important part during software development
    - Agile Development — When you add more and more features to a software, you sometimes need to change old design and code
    - Documentation — Unit test is another place where the new hires can learn the logic of the application
    - Quality of Code — When we have good coverage of unit test case for application, the quality of code is higher.
    - TDD(Test Driven Development) — When writing cases first, it forces developer to think through the business logic clearly and come up with different corner cases

JUNIT

- JUnit 4 and JUnit 5
    - JUnit 4 was divided into modules that comprise JUnit 5
        - JUnit Platform – this module scopes all the extension frameworks we might be interested in test execution, discovery, and reporting
        - JUnit Vintage – this module allows backward compatibility with JUnit 4 or even JUnit 3
- JUnit 5 support Java 8 features
- Differences in annotations:
    - *@Before* annotation is renamed to *@BeforeEach*
    - *@After* annotation is renamed to *@AfterEach*
    - *@BeforeClass* annotation is renamed to *@BeforeAll*
    - *@AfterClass* annotation is renamed to *@AfterAll*
    - *@Ignore* annotation is renamed to *@Disabled*

JUNIT

- Basic Annotations
  - @Test
  - @BeforeAll and @BeforeEach

```java
@BeforeAll
static void setup() {
    log.info("@BeforeAll - executes once before all test methods in this class");
}

@BeforeEach
void init() {
    log.info("@BeforeEach - executes before each test method in this class");
}
```

- Important to note is that the method with *@BeforeAll* annotation needs to be static, otherwise the code will not compile.

JUNIT

- @DisplayName and @Disabled

```java
@DisplayName("Single test successful")
@Test
void testSingleSuccessTest() {
    log.info("Success");
}

@Test
@Disabled("Not implemented yet")
void testShowSomething() {
}
```

- @AfterEach and @AfterAll

```java
@AfterEach
void tearDown() {
    log.info("@AfterEach - executed after each test method.");
}

@AfterAll
static void done() {
    log.info("@AfterAll - executed after all test methods.");
}
```

JUNIT

- Assertion — Assertion is the key to validate if the output from the code is expected
    - Here is the full list of supported assertion statement: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html
- From JUnit 5, we can use lambdas in assertion
- From JUnit 5, It is also now possible to group assertions with *assertAll()* which will report any failed assertions within the group with a *MultipleFailuresError*

```
@Test
void groupAssertions() {
    int[] numbers = {0, 1, 2, 3, 4};
    assertAll("numbers",
        () -> assertEquals(numbers[0], 1),
        () -> assertEquals(numbers[3], 3),
        () -> assertEquals(numbers[4], 1)
    );
}
```

- Before JUnit 5, it will stop processing the rest of assertions if one of them failed — It creates a problem where we might have to run a test cases multiple times to fix the issues

LET'S TAKE A LOOK AT AN EXAMPLE

MOCKITO

- There is a fundamental problem in writing unit test cases where there are external dependencies.
    - For example, if we want to write test cases for our service classes, we have to mock proper DAO injection (Thanks to DI)
    - We can't reply on external system such as database calls — There might be no network connection during the build phase of the application
- In real-world applications, where components often depend on accessing external systems, it is important to provide proper test isolation so that we can focus on testing the functionality of a given unit without having to involve the whole class hierarchy for each test
    - Injecting a mock is a clean way to introduce such isolation.
        - Mockito provides us an easier way to mock the dependencies we need.

MOCKITO

- To get started, we have to include the dependency

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.21.0</version>
</dependency>
```

- Basic Annotations:
  - *@ExtendWith(MockitoExtension.class)* — annotate the JUnit test with a MockitoExtension
  - @Mock — create and inject mocked instances
  - *@Spy* — create and inject partially mocked instances (parameterized constructor)
  - *@InjectMocks* — inject mock fields into the tested object automatically (The class needs to be tested)
  - *verify* — provide more ways to validate the business logic

JACOCO

- Code coverage is a software metric used to measure how many lines of our code are executed during automated tests

- Jacoco is good tool to check the testing code coverage.

  - Maven Plugin for Jacoco

LET'S TAKE A LOOK AT AN EXAMPLE