Spring Boot Testing

1. Introduction to Testing in Spring Boot

What is SDLC (Software Development Life Cycle)?

The Software Development Life Cycle (SDLC) is a structured process for developing software applications. It consists of stages such as planning, design, development, testing, deployment, and maintenance.

Importance of Testing

Testing plays a crucial role in SDLC by ensuring the quality, reliability, and stability of software applications. The key benefits of testing include:

- Identifies Bugs Early and Saves Money: Catching bugs early in the development process reduces costs and prevents critical failures in production.
- Mitigates Deployment Risks and Ensures System Stability: Well-tested applications reduce the risk of deployment issues and enhance system stability.
- Increases Productivity and Speeds Up Development: Automated testing frameworks streamline development by reducing manual verification efforts.
- Promotes Confidence Among Developers and Builds Trust: Reliable test coverage assures developers and stakeholders that the application functions as expected.

2. Test Writing Approaches

Test-Driven Development (TDD)

TDD is a development approach where tests are written before the implementation. It follows three steps:

1. Write a failing test.
2. Write the minimum code required to pass the test.
3. Refactor the code for better design and maintainability.

Behavior-Driven Development (BDD)

BDD extends TDD by focusing on defining behavior in human-readable language. It follows these steps:

1. Define behavior in plain English (e.g., using Gherkin syntax).
2. Write scenarios to meet the defined behavior.
3. Implement code to pass the test scenarios.

Other Approaches

- Test-After Development: Writing tests after the feature has been implemented.
- Simultaneous Development: Writing code and tests in parallel.

3. Understanding JUnit Testing Framework and AssertJ Library

JUnit Annotations

JUnit is a widely used testing framework in Java. Key JUnit annotations include:

- @Test: Marks a method as a test case.
- @DisplayName: Provides a custom name for test reports.
- @Disabled: Disables a test method or class.
- @BeforeEach: Executes a method before each test case.
- @AfterEach: Executes a method after each test case.
- @BeforeAll: Runs once before all test cases (must be static).
- @AfterAll: Runs once after all test cases (must be static).

More information: JUnit Documentation

JUnit vs AssertJ

- JUnit: A testing framework that provides a standardized way to write and execute test cases.
- AssertJ: A library that enhances assertions, making test code more readable and maintainable.

Common AssertJ Methods

- **Numbers**: assertThat(5).isEqualTo(5).isGreaterThan(4);
- **Strings**: assertThat("hello").startsWith("he").contains("ell");
- **Booleans**: assertThat(true).isTrue();
- **Lists/Arrays**: assertThat(List.of("apple", "banana")).contains("apple").hasSize(2);
- **Exceptions**: assertThatThrownBy(() -> { throw new IllegalArgumentException("Invalid argument"); })
- .**isInstanceOf**(IllegalArgumentException.class)
- .**hasMessage**("Invalid argument");

More details: AssertJ Documentation

4. Unit Testing vs Integration Testing

Unit Tests

- Focus: Individual components (e.g., service methods).
- Tools: JUnit, Mockito.
- Example: Testing a business logic method.

Integration Tests

- Focus: Interaction between components (e.g., services and repositories).

- Tools: @SpringBootTest, Spring Test.
- Example: Testing the interaction between a service and a database.

Key Testing Annotations in Spring Boot

- @SpringBootTest: Loads the full application context for integration tests.
- @DataJpaTest: Configures an in-memory database to test JPA repositories.
- @TestConfiguration: Defines extra beans or configurations for tests.
- @WebMvcTest: Initializes only the web layer for controller testing.
- @AutoConfigureTestDatabase: Replaces the database with an embedded database for tests.

5. Unit Testing Persistence Layer and Setting Up TestContainers

Using @DataJpaTest

- Configures an in-memory database and scans for JPA entities.
- Rolls back transactions after each test.

Configuring Test Database

- @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE): Prevents Spring Boot from replacing the database with an in-memory version.

More info: Spring Boot Docs

TestContainers for Mocking a Real Database

- Use TestContainers to mock a real database for integration testing.
- Requires Docker to run containerized databases.

6. Understanding Mocking with the Mockito Library

Mockito Overview

Mockito is a testing framework that allows mocking dependencies. It is useful for:

- Mocking: Simulating behavior of real objects.
- Stubbing: Defining return values for method calls.
- Verification: Ensuring specific method calls occurred.

Common Mockito Methods

- Creating Mocks: @Mock, Mockito.mock(Classname.class).
- Stubbing Mocks: when(methodCall).thenReturn(value);
- Verifying Method Calls: verify(mock).methodName();
- Verification Modes:

- times(n): Ensures method was called n times.
- never(): Ensures method was never called.
- atLeastOnce(): Ensures method was called at least once.
- only(): Ensures no other method was called.

7. Unit Testing Service Layer Using Mockito

Mockito helps test service methods in isolation. Example:

```
@Mock
private EmployeeRepository employeeRepository;
@InjectMocks
private EmployeeServices employeeServices;

@BeforeEach
void setup() {
    mockEmployee = new EmployeeEntity(1, "John Doe", "john@example.com", 50000);
    when(employeeRepository.findById(1)).thenReturn(Optional.of(mockEmployee));
}

@Test
void whenGetEmpById_thenReturnEmp() {
    EmployeeDTO employeeDTO = employeeServices.getEmpById(1);
    assertThat(employeeDTO.getEmpId()).isEqualTo(1);
}
```

8. Integration Testing in Spring Boot

Importance of Integration Testing

- Detects issues in interactions between modules.
- Ensures configuration correctness.
- Prevents regression when modifying code.

Using WebTestClient

WebTestClient provides an API for making HTTP requests and validating responses in integration tests.

This document provides an in-depth understanding of Spring Boot testing, covering unit tests, integration tests, mocking with Mockito, and using TestContainers. By following these guidelines, developers can ensure robust and maintainable test coverage for their applications.

9.Test Coverage and Reports-

Generating test coverage reports. - Using tools like Jacoco for analysis.

Link → Intro to JaCoCo | Baeldung

Running the test using JUnit will automatically set in motion the JaCoCo agent. It will create a coverage report in binary format in the target directory, target/jacoco.exec.

Benefits of JaCoCo

• Visibility into Test Coverage

• Identify Dead Code

• Focus on Critical Areas: By showing which parts of your code are not covered, JaCoCo allows you to focus your testing efforts on the most critical and potentially vulnerable areas of your application.

• Better project maintenance, compliance and standards

10.Dependency Required for project →

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.36</version>
</dependency>

<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.0.0</version>
</dependency>

<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.7.0</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-testcontainers</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
```

```
    <groupId>org.testcontainers</groupId>
    <artifactId>mysql</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <scope>test</scope>
</dependency>
```

By Aron...