테스트 케이스 작성: Java, JUnit 5, Mockito

단위 테스트(Unit Test)

JUnit 5 기본 설정

1. **라이브러리 추가**

- Maven

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

- Gradle

```
testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.7.0'
```

2. **테스트 클래스 생성**

- 테스트 대상 클래스의 이름에 "Test"를 붙입니다(예: `CalculatorTest`).
- 테스트 메서드에 `@Test` 애너테이션을 사용합니다.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class CalculatorTest {

    @Test
    void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result);
    }
}
```

Mockito 사용

1. **라이브러리 추가**

- Maven

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.5.13</version>
    <scope>test</scope>
</dependency>
```

- Gradle

```
testImplementation 'org.mockito:mockito-core:3.5.13'
```

2. **테스트 클래스에서 모의 객체 사용**

- 의존성을 가진 클래스를 테스트할 때 `@Mock` 애너테이션을 사용하여 모의 객체로 대체합니다.
- `@InjectMocks` 애너테이션을 사용하여 테스트 대상 클래스에 모의 객체를 주입합니다.

```
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

class UserServiceTest {

    @Mock
    UserRepository userRepository;

    @InjectMocks
    UserService userService;

    @Test
    void testAddUser() {
        User user = new User("John", "john@example.com");
        when(userRepository.save(user)).thenReturn(user);

        userService.addUser(user);

        verify(userRepository, times(1)).save(user);
    }
}
```

통합 테스트(Integration Test)

통합 테스트는 여러 컴포넌트를 함께 테스트하여 시스템이 전체적으로 올바르게 작동하는지 확인합니다. 일반적으로 데이터베이스, 외부 서비스 등 실제 의존성을 사용합니다.

Spring Boot 통합 테스트 설정

1. **라이브러리 추가**

- Maven

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

- Gradle

```
testImplementation 'org.springframework.boot:spring-boot-starter-test'
```

2. **테스트 클래스 생성**

- `@SpringBootTest` 애너테이션을 사용하여 스프링 부트 통합 테스트를 설정합니다.
- 필요한 경우, `@AutoConfigureMockMvc` 애너테이션을 사용하여 컨트롤러 테스트를 설정하고, `MockMvc` 객체를 주입받습니다.

```
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockM
vc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.web.servlet.MockMvc;
import org.junit.jupiter.api.Test;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
class UserControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testGetUser() throws Exception {
        mockMvc.perform(get("/users/1"))
                .andExpect(status().isOk());
    }
}
```

테스트용 데이터베이스 설정

통합 테스트 시 테스트용 데이터베이스를 사용하는 것이 좋습니다. 예를 들어, H2 데이터베이스를 사용하면 테스트 환경을 격리하고 테스트 속도를 높일 수 있습니다.

1. **라이브러리 추가**

- Maven

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

- Gradle

```
testImplementation 'com.h2database:h2'
```

2. **테스트 환경 설정**

- `src/test/resources` 폴더에 `application-test.yml` 파일을 생성하여 테스트용 데이터베이스 설정을 추가합니다.

```
spring:
  datasource:
    url: jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
    username: sa
    password:
  jpa:
    hibernate:
      ddl-auto: create-drop
```

- 테스트 클래스에서 `@ActiveProfiles("test")` 애너테이션을 사용하여 테스트 프로파일을 활성화합니다.

```
@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("test")
class UserControllerIntegrationTest {
    // ...
}
```

이러한 방식으로 단위 테스트와 통합 테스트를 작성하고 실행할 수 있습니다. 테스트 케이스를 추가하고 수정하여 프로젝트의 품질을 향상시키고, 리팩토링을 안전하게 진행할 수 있습니다.

단위 테스트에서 주로 사용되는 메소드와 설명, 동작 방식, 예시를 정리하겠습니다.

1. assertEquals

- 설명: 두 객체의 값이 동일한지 확인합니다.
- 동작 방식: 첫 번째 인자(expected)와 두 번째 인자(actual)의 값이 동일한지 확인하고, 동일하지 않으면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testAddition() {
    int a = 5;
    int b = 3;
    int result = a + b;
    assertEquals(8, result);
}
```

2. assertNotEquals

- 설명: 두 객체의 값이 다른지 확인합니다.
- 동작 방식: 첫 번째 인자(unexpected)와 두 번째 인자(actual)의 값이 다른지 확인하고, 동일하면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testSubtraction() {
    int a = 5;
    int b = 3;
    int result = a - b;
    assertNotEquals(5, result);
}
```

3. assertTrue

- 설명: 조건이 참인지 확인합니다.
- 동작 방식: 인자로 전달된 조건이 참인지 확인하고, 거짓이면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testIsEven() {
    int a = 4;
    assertTrue(a % 2 == 0);
}
```

4. assertFalse

- 설명: 조건이 거짓인지 확인합니다.
- 동작 방식: 인자로 전달된 조건이 거짓인지 확인하고, 참이면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testIsOdd() {
    int a = 3;
    assertFalse(a % 2 == 0);
}
```

5. assertNull

- 설명: 객체가 null인지 확인합니다.
- 동작 방식: 인자로 전달된 객체가 null인지 확인하고, null이 아니면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testIsNull() {
    Object object = null;
    assertNull(object);
}
```

6. assertNotNull

- 설명: 객체가 null이 아닌지 확인합니다.
- 동작 방식: 인자로 전달된 객체가 null이 아닌지 확인하고, null이면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testIsNotNull() {
    Object object = new Object();
    assertNotNull(object);
}
```

죄송합니다. 짤린 부분부터 다시 작성하겠습니다.

7. assertSame

- 설명: 두 객체의 참조가 동일한지 확인합니다.
- 동작 방식: 첫 번째 인자(expected)와 두 번째 인자(actual)의 참조가 동일한지 확인하고, 동일하지 않으면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testSingleton() {
    Singleton singleton1 = Singleton.getInstance();
    Singleton singleton2 = Singleton.getInstance();
    assertSame(singleton1, singleton2);
}
```

8. assertNotSame

- 설명: 두 객체의 참조가 다른지 확인합니다.
- 동작 방식: 첫 번째 인자(unexpected)와 두 번째 인자(actual)의 참조가 다른지 확인하고, 동일하면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testDifferentInstances() {
    Object object1 = new Object();
    Object object2 = new Object();
    assertNotSame(object1, object2);
}
```

9. assertArrayEquals

- 설명: 두 배열의 값이 동일한지 확인합니다.
- 동작 방식: 첫 번째 인자(expecteds)와 두 번째 인자(actuals)의 배열 값이 동일한지 확인하고, 동일하지 않으면 테스트 실패로 표시합니다.
- 예시:

```
@Test
public void testArrayEquality() {
    int[] expected = new int[]{1, 2, 3};
    int[] actual = new int[]{1, 2, 3};
    assertArrayEquals(expected, actual);
}
```

이러한 메소드들은 주로 JUnit과 함께 사용되며, 단위 테스트를 작성할 때 사용되는 주요 메소드들입니다. 이 메소드들을 사용하여 코드의 정확성과 기능을 검증할 수 있습니다.

통합테스트에서는 여러 컴포넌트나 서비스들 간의 상호작용을 테스트하기 때문에, 단위테스트와는 달리 더 복잡한 환경에서 수행됩니다. 이를 위해 JUnit에서는 `@SpringBootTest` 어

노테이션을 제공하며, Spring Boot를 사용하여 애플리케이션 컨텍스트를 빌드하고 테스트할 수 있습니다.

통합테스트에서 주로 사용되는 메소드들은 다음과 같습니다.

@SpringBootTest

- Spring Boot를 사용하여 테스트를 위한 애플리케이션 컨텍스트를 구성합니다.
- `classes` 속성을 사용하여 테스트할 구성 클래스를 지정할 수 있습니다.

```
@SpringBootTest(classes = {MyTestConfig.class})
class MyIntegrationTest {
    // ...
}
```

@WebMvcTest

- Spring MVC 컨트롤러 테스트에 특화된 슬라이스 테스트를 수행합니다.
- `controllers` 속성으로 테스트할 컨트롤러를 지정할 수 있습니다.
- `AutoConfigureMockMvc` 어노테이션이 함께 선언되어야 `MockMvc` 객체를 사용할 수 있습니다.

```
@WebMvcTest(MyController.class)
class MyIntegrationTest {
    // ...
}
```

@DataJpaTest

- JPA 레파지토리 테스트에 특화된 슬라이스 테스트를 수행합니다.
- 인메모리 데이터베이스를 사용하여 테스트합니다.

```
@DataJpaTest
class MyIntegrationTest {
    // ...
}
```

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)

- 실제 서버를 띄우고 테스트하는 것으로, `@AutoConfigureMockMvc` 어노테이션과 달리 실제 웹 환경에서 테스트를 수행합니다.
- `WebEnvironment` 속성으로 서버를 구동할 때 사용할 포트나 프로토콜 등을 설정할 수 있습니다.

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MyIntegrationTest {
    // ...
}
```

위와 같은 어노테이션 외에도, `@Transactional`, `@Commit`, `@Rollback` 등의 어노테이션을 사용하여 데이터베이스 트랜잭션과 관련된 테스트를 수행할 수 있습니다. 또한 `Mockito`와 같은 라이브러리를 사용하여 통합테스트에서도 단위테스트와 유사한 방식으로 모의 객체(Mock)를 사용할 수 있습니다.