**Spring Data JPA Unit Test: Repository Layer**

https://medium.com/@husnapoyraz88/spring-data-jpa-unit-test-repository-layer-9e875390645e

Hello, I will explain how to write unit tests for the repository layer in a simple Spring Boot project based on the following topic headings.

1. What is Unit Testing?
2. Why Write Unit Tests for the Repository Layer?
3. How to Write Unit Tests for the Repository Layer?
4. Repository Layer Unit Testing with a Demo Projec

**1) What is Unit Testing?**

Unit testing is the process of testing the behavior of the smallest units or components of software during the development phase. These units are typically functions, methods, or classes. Unit tests are used to ensure that the code functions correctly and produces the expected results. However, to isolate units and ensure they are independent, external dependencies (such as databases, network services) are mocked or fake databases are used.

**2) Why Write Unit Tests for the Repository Layer?**

The repository layer is typically responsible for database operations. Database operations can lead to errors or unexpected results. Therefore, writing unit tests for the repository layer is important to verify that database operations are performed correctly and that the expected data is retrieved from the database.

These tests may include adding, updating, querying, and deleting data from the database. Since the repository layer is often heavily utilized by other layers of the application, ensuring the proper functioning of this layer is critical for the overall health of the application.

**3) How to Write Unit Tests for the Repository Layer?**

To write unit tests for the repository layer, we can follow the steps below.

1. **Adding Dependencies:** Test frameworks and database dependencies should be added, and database configuration should be done.
2. **Creating the Test Class:** Create a test class for the methods to be tested.
3. **Creating Test Methods:**Write test methods to test the functionality in the repository layer. These tests should cover operations such as data insertion, updating, querying, and deletion.
4. **Preparing Test Data:** Before each test, prepare the necessary data for the operation to be tested. This step is usually done under annotations like @BeforeEach.
5. **Calling the Tested Operation and Checking the Result**
6. **Running Tests and Evaluating Results:** Run the tests and evaluate the results. If all tests pass, it confirms that the repository layer is functioning correctly.

**4) Repository Layer Unit Testing with a Demo Project**

Let’s examine the steps to write unit tests for the repository layer in a Spring Boot project with a demo.

**1. Adding Dependencies**

Firstly, necessary dependencies should be added to the build.gradle or pom.xml file. These dependencies typically include test frameworks and libraries required for database operations.

**1.a. Test Framework Dependencies:** For writing unit tests for the repository layer, JUnit and Spring Test frameworks are commonly used. These frameworks are used to define test classes, run tests, and evaluate results.

For a Maven project, JUnit and Spring Test dependencies can be added in the pom.xml file:

<dependencies>  
 <!-- Spring Boot Starter Test -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
  
 <!-- JUnit Jupiter Engine -->  
 <dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-engine</artifactId>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

**1.b. Database Dependencies:** In Spring Boot, it is common to create and use an in-memory test database (such as H2 database) for unit tests. This isolates from external dependencies (e.g., databases). However, in some cases, a real database may need to be used. In such cases, test databases are generally used, and tests run on a real database.

For a Maven project, H2 database dependency can be added in the pom.xml file:

<!-- H2 Veritabanı -->  
<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>test</scope>  
</dependency>

**1.b.1. Database Configuration:** Configuration of the database to be used for tests should be done. This is usually done in the application.properties or application.yml file. Separate database configuration is often done for tests.

For H2 database, an application-test.yml file can be used:

spring:  
 datasource:  
 url: jdbc:h2:mem:test\_db;DB\_CLOSE\_DELAY=-1;DB\_CLOSE\_ON\_EXIT=FALSE  
 username: sa  
 password: password  
 driver-class-name: org.h2.Driver  
 jpa:  
 hibernate:  
 ddl-auto: create-drop  
 database-platform: org.hibernate.dialect.H2Dialect  
 show-sql: true

* DB\_CLOSE\_DELAY: This parameter specifies how long (in milliseconds) the database connection will not be closed. By default, the DB\_CLOSE\_DELAY parameter is zero, which means the database connection will be closed immediately after each use. However, it is often preferred to set this parameter to -1 for in-memory databases, so that the database is not automatically deleted when the application shuts down.
* DB\_CLOSE\_ON\_EXIT: This parameter specifies whether the database should be automatically closed when the JVM exits. By default, the DB\_CLOSE\_ON\_EXIT parameter is TRUE. When set to DB\_CLOSE\_ON\_EXIT=FALSE, the database will not be automatically closed when the JVM exits.

**2. Creating the Test Class**

In this example, let’s assume we have a User class and a UserRepository interface as follows:

// User.java  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
  
@Entity  
@Getter  
@Setter  
@AllArgsArgsConstructor  
@NoArgsConstructor  
public class User {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
 private String username;  
 @Column(unique = true, nullable = false, updatable = false)  
 private String email;  
}

// UserRepository.java  
import org.springframework.data.jpa.repository.JpaRepository;  
  
@Repository  
public interface IUserRepository extends JpaRepository<User, Long> {  
 User findByUsername(String username);  
}

* Let’s create a test class to test our repository layer. This class should be annotated with one of the Spring Boot test annotations (@DataJpaTest, @SpringBootTest, etc.) to indicate that it is a test class recognized by Spring Boot.
* @DataJpaTest annotation provides a specialized test environment for testing only JPA components. It is commonly used to test JPA entities and repository classes. It creates a database connection and provides a limited container, allowing tests to run faster. Therefore, we annotate our class with this annotation.

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;  
  
@DataJpaTest  
public class UserRepositoryTest {  
 // Test methodları buraya gelecek  
}

**3. Creating Test Methods**

Let’s create unit test methods to test the findByUsername method of UserRepository.

We mark the test methods with the @Test annotation to indicate that these methods are unit tests. This allows the test environment to be prepared and the tests to be executed automatically.

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;  
  
@DataJpaTest  
public class UserRepositoryTest {  
  
 @Test  
 public void whenFindByUsername\_thenReturnUser() {  
 // Test senaryosu (best case) buraya gelecek  
 }  
  
 @Test  
 public void whenFindByNonExistingUsername\_thenReturnNull() {  
 // Test senaryosu (wrong case) buraya gelecek  
 }  
}

In the above example, more descriptive names are chosen for the test methods that will test the findByUsername method. In the first test method, named “whenFindByExistingUsername\_thenReturnUser,” we check whether a user exists in the database with a specific username. In the second test method, named “whenFindByNonExistingUsername\_thenReturnNull,” we check the scenario where a user with a non-existing username is searched in the database.

**4. Preparing Test Data**

In both test methods, we need to first create an environment that meets the requirements of the test scenario. Therefore, it is important to prepare the necessary test data before each test method. This step is usually performed before writing test methods and is often placed under annotations like @BeforeEach.

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.BeforeEach;  
  
@DataJpaTest  
public class UserRepositoryTest {  
  
 @Autowired  
 private TestEntityManager entityManager;  
  
 // Her test öncesinde kullanılacak verileri hazırlamak için kullanılır.  
 @BeforeEach  
 public void setUp() {  
 // Test verilerini hazırla  
 User user = new User();  
 user.setUsername("testuser");  
 user.setEmail("testuser@example.com");  
 entityManager.persist(user);  
 }  
  
 @Test  
 public void whenFindByUsername\_thenReturnUser() {  
 // Test senaryosu (best case) buraya gelecek  
 }  
  
 @Test  
 public void whenFindByNonExistingUsername\_thenReturnNull() {  
 // Test senaryosu (wrong case) buraya gelecek  
 }  
}

Here, we added test data to the database using the TestEntityManager. The setUp() method will run before each test method and prepare the test data.

**5. Calling the Tested Operation and Checking the Result**

Once the test data is prepared, each test method tests a specific scenario. The first test method checks whether a user exists in the database with a specified username. The second test method checks whether null is returned when a user with a non-existing username is searched in the database.

import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;  
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;  
  
import static org.assertj.core.api.Assertions.assertThat;  
import static org.junit.jupiter.api.Assertions.assertNull;  
  
@DataJpaTest  
public class UserRepositoryTest {  
  
 @Autowired  
 private UserRepository userRepository;  
  
 @Autowired  
 private TestEntityManager entityManager;  
  
  
 // Her test öncesinde kullanılacak verileri hazırlamak için kullanılır.  
 @BeforeEach  
 public void setUp() {  
 // Test verilerini hazırla  
 User user = new User();  
 user.setUsername("testuser");  
 user.setEmail("testuser@example.com");  
 entityManager.persist(user);  
 }  
  
 // Kullanıcı adına göre kullanıcı bulunduğunda doğru kullanıcıyı döndürür.  
 @Test  
 public void whenFindByUsername\_thenReturnUser() {  
 // given  
 String username = "testuser";  
 String email = "testuser@example.com";  
  
 // when  
 User found = userRepository.findByUsername(username);  
  
 // then  
 // AssertJ kütüphanesi methodu  
 assertThat(found.getUsername()).isEqualTo(username);  
 }  
  
 // Kullanıcı adına göre kullanıcı bulunamadığında null döndürür.  
 @Test  
 public void whenFindByNonExistingUsername\_thenReturnNull() {  
 // given  
 String nonexistinguser = "nonexistinguser";  
  
 // when  
 User found = userRepository.findByUsername(nonexistinguser);  
  
 // then  
 // JUnit kütüphanesi methodu  
 assertNull(found);  
 }  
}

Now, let’s explain in more detail how these test scenarios work and check the results:

1. **whenFindByUsername\_thenReturnUser Test:**

* given First, we define the username and email information required for the test.
* when: Then, we call the findByUsername method in the UserRepository to search for a user with the specified username.
* then: Finally, we compare the returned user’s name with the expected username using assertThat.

**2. whenFindByNonExistingUsername\_thenReturnNull Test:**

* given First, we define a username that does not exist in the database for the test.
* when: Then, we call the findByUsername method in the UserRepository to search for a user with the specified username that does not exist.
* then: Finally, we expect the result to be null, indicating that no user was found with the given username.

**6. Running Tests and Evaluating Results**

The results of the tests are collected by the test frameworks (JUnit, TestNG, etc.) and presented as a report. Successful tests are marked in green, while failed tests are marked in red.

The successful completion of tests provides evidence that the written code provides the expected functionality. Failed tests indicate that the code needs to be corrected or revised.

In conclusion, feedback is provided on whether the tests pass or fail, and whether the application functions as expected. This feedback is important for improving the quality of the application and detecting errors early.

I hope this explanation and examples on how to write unit tests for the repository layer are helpful to you.

Source:

* [Testing in Spring Boot — Integration Testing With @DataJpaTest](https://www.baeldung.com/spring-boot-testing#integration-testing-with-datajpatest)
* [Spring Boot — Auto-configured Data JPA Tests](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing-spring-boot-applications-testing-autoconfigured-jpa-test)