# @DataJpaTest and Repository Class in JUnit

https://www.baeldung.com/junit-datajpatest-repository

## 1. Introduction

When working with Spring Boot applications that utilize [Spring Data JPA](https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa) for data persistence, it’s crucial to test the repositories that interact with the database. In this tutorial, we’ll explore how to effectively test Spring Data JPA repositories using the @DataJpaTest annotation provided by Spring Boot along with [JUnit](https://www.baeldung.com/junit-5).

## 2. Understanding @DataJpaTest and Repository Class

In this section, we’ll delve into the interaction between @DataJpaTest and class repositories within the context of Spring Boot applications.

### 2.1. @DataJpaTest

The @DataJpaTest annotation is used to test JPA repositories in Spring Boot applications.**It’s a specialized test annotation that provides a minimal Spring context for testing the persistence layer.** This annotation can be used in conjunction with other testing annotations like @RunWith and [@SpringBootTest](https://www.baeldung.com/spring-boot-testing#integration-testing-with-springboottest).

**In addition, the scope of @DataJpaTest is limited to the JPA repository layer of the application.** It doesn’t load the entire application context, which can make testing faster and more focused. This annotation also provides a pre-configured EntityManager and TestEntityManager for testing JPA entities.

### 2.2. Repository Class

In Spring Data JPA, repositories serve as an abstraction layer on top of JPA entities. **It provides a set of methods for performing CRUD (Create, Read, Update, Delete) operations and executing custom queries.** These repositories typically extend from interfaces like [JpaRepository](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html) and are responsible for handling database interactions related to specific entity types.

## 3. Optional Parameters

@DataJpaTest does have some optional parameters we can use to customize the test environment.

### 3.1. properties

This parameter allows us to specify Spring Boot configuration properties that will be applied to our test context. This can be **useful for adjusting settings like database connection details, transaction behavior, or other application properties** relevant to our testing needs:

@DataJpaTest(properties = {

"spring.datasource.url=jdbc:h2:mem:testdb",

"spring.jpa.hibernate.ddl-auto=create-drop"

})

**public** **class** **UserRepositoryTest** {

// ... test methods

}Copy

### 3.2. showSql

This enables SQL logging for our tests and allows us to see the actual SQL queries executed by the repository methods. Moreover, **this can help debug or understand how the JPA queries are translate**d. By default, the SQL logging is enabled. We can turn it off by setting the value to false:

@DataJpaTest(showSql = false)

**public** **class** **UserRepositoryTest** {

// ... test methods

}Copy

### 3.3. includeFilters and excludeFilters

These parameters enable us to include or exclude specific components during component scanning. We can **use them to narrow down the scanning scope and optimize test performance** by focusing only on the relevant components:

@DataJpaTest(includeFilters = @ComponentScan.Filter(

type = FilterType.ASSIGNABLE\_TYPE,

classes = UserRepository.class),

excludeFilters = @ComponentScan.Filter(

type = FilterType.ASSIGNABLE\_TYPE,

classes = SomeIrrelevantRepository.class))

**public** **class** **UserRepositoryTest** {

// ... test methods

}Copy

## 4. Key Features

When it comes to testing JPA repositories in Spring Boot applications, the @DataJpaTest annotation can be a handy tool. Let’s explore its key features and benefits in detail.

### 4.1. Test Environment Configuration

Setting up a proper test environment for JPA repositories can be time-consuming and tricky. **@DataJpaTest provides a ready-made testing environment that includes essential components for testing JPA repositories, such as the EntityManager and DataSource.**

This environment is specifically designed for testing JPA repositories. It ensures that our repository methods run within the context of a test transaction, interacting with a safe, in-memory database like H2 instead of the production database.

### 4.2. Dependency Injection

**@DataJpaTest simplifies the process of dependency injection within our test classes.** Repositories, along with other essential beans, are automatically injected into the test context. This seamless integration enables developers to focus on writing concise and effective test cases without the hassle of explicit bean wiring.

### 4.3. Rollback by Default

Moreover, keeping tests independent and reliable is crucial. **By default, each test method annotated with @DataJpaTest runs within a transactional boundary.** This ensures that changes made to the database are automatically rolled back at the end of the test, leaving a clean slate for the next test.

## 5. Configuration and Setup

To use @DataJpaTest, we need to add the [spring-boot-starter-test](https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-test)dependency to our project with scope “test“. This lightweight dependency includes essential testing libraries like JUnit for testing, ensuring it’s not included in our production build.

### 5.1. Adding Dependency to pom.xml

Let’s add the following dependency to the pom.xml:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-test</**artifactId**>

<**scope**>test</**scope**>

</**dependency**>Copy

Once we’ve added the dependency, we can use the @DataJpaTest annotation in our tests. This annotation sets up an in-memory H2 database and configure Spring Data JPA for us, allowing us to write tests that interact with our repository classes.

### 5.2. Creating the Entity Class

Now, let’s create the User entity class, which will represent user data:

@Entity

**public** **class** **User** {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

**private** Long id;

**private** String username;

**private** String password;

// getters and setters

}Copy

### 5.3. Creating the Repository Interface

Next, we define the UserRepository, a Spring Data JPA repository interface for managing User entities:

**public** **interface** **UserRepository** **extends** **JpaRepository**<User, Long> {

// Add custom methods if needed

}

Copy

By extending JpaRepository<User, Long>, our UserRepository gains access to standard CRUD operations provided by Spring Data JPA out-of-the-box.

Additionally, we can define custom query methods within this interface to suit specific data access retrieval needs, such as findByUsername():

**public** **interface** **UserRepository** **extends** **JpaRepository**<User, Long> {

// Custom query method to find a user by username

User **findByUsername**(String username);

}Copy

## 6. Implementing Repository Tests

To test the repository layer of our application, we’ll utilize the @DataJpaTest annotation. **By using this annotation, an in-memory H2 database will be set up, and Spring Data JPA will be configured.** This allows us to write tests that interact with our repository classes.

### 6.1. Setting up the Test Class

To begin, let’s set up the test class by annotating it with @DataJpaTest. This annotation scans for entity classes annotated with @Entity and Spring Data JPA repositories interfaces. This ensures that only relevant components are loaded for testing, improving test focus and performance:

@DataJpaTest

**public** **class** **UserRepositoryTest** {

// Add test methods here

}Copy

To create a repository test case, we first need to inject the repository that we want to test into our test class. This can be done using the @Autowired annotation:

@Autowired

**private** UserRepository userRepository;Copy

### 6.2. Test Lifecycle Management

In the context of test lifecycle management, @BeforeEach and @AfterEach annotations are used to perform setup and teardown operations before and after each test method, respectively. **This ensures that each test method runs in a clean and isolated environment, with consistent initial conditions and cleanup procedures.**

Here’s how we can incorporate test lifecycle management into our test class:

**private** User testUser;

@BeforeEach

**public** **void** **setUp**() {

// Initialize test data before each test method

testUser = **new** **User**();

testUser.setUsername("testuser");

testUser.setPassword("password");

userRepository.save(testUser);

}

@AfterEach

**public** **void** **tearDown**() {

// Release test data after each test method

userRepository.delete(testUser);

}Copy

In the setUp() method annotated with @BeforeEach, we can perform any necessary setup operations required before each test method execution. This might include initializing test data, setting up mock objects, or preparing resources needed for the test.

Conversely, in the tearDown() method annotated with @AfterEach, we can perform cleanup operations after each test method execution. This might involve resetting any changes made during the test, releasing resources, or performing any necessary cleanup tasks to restore the test environment to its original state.

### 6.3. **Testing the Insertion Operation**

Now, we can write test methods that interact with the JPA repository. For example, we might want to test that we can save a new user to the database. Since a user is automatically saved before each test, we can directly focus on testing interactions with the JPA repository:

@Test

**void** **givenUser\_whenSaved\_thenCanBeFoundById**() {

**User** savedUser = userRepository.findById(testUser.getId()).orElse(null);

assertNotNull(savedUser);

assertEquals(testUser.getUsername(), savedUser.getUsername());

assertEquals(testUser.getPassword(), savedUser.getPassword());

}Copy

If we observe the console log for the test case, we’ll notice the following logs:

Began transaction (1) **for** test context

.....

Rolled back transaction **for** test: Copy

These logs indicate that the @BeforeEach and @AfterEach methods are functioning as expected.

### 6.4. **Testing the Update Operation**

In addition, we can create a test case for testing the update operation:

@Test

**void** **givenUser\_whenUpdated\_thenCanBeFoundByIdWithUpdatedData**() {

testUser.setUsername("updatedUsername");

userRepository.save(testUser);

**User** updatedUser = userRepository.findById(testUser.getId()).orElse(null);

assertNotNull(updatedUser);

assertEquals("updatedUsername", updatedUser.getUsername());

}Copy

### 6.5. Testing the findByUsername() Method

Now, let’s test the findByUsername() custom query method we created:

@Test

**void** **givenUser\_whenFindByUsernameCalled\_thenUserIsFound**() {

**User** foundUser = userRepository.findByUsername("testuser");

assertNotNull(foundUser);

assertEquals("testuser", foundUser.getUsername());

}Copy

## 7. Transactional Behavior

By default, all tests annotated with @DataJpaTest are executed within a transaction. **This means that any changes made to the database during the test are rolled back at the end of the test, ensuring that the database is left in its original state.** This default behavior simplifies testing by preventing interference between tests and data corruption.

However, there may be cases where we need to disable transactional behavior to test certain scenarios. For instance, testing results may need to persist beyond the test.

In such a case, we can disable transactions for a specific test class using the @Transactional annotation with propagation = *[propagation.NOT\_SUPPORTED](https://www.baeldung.com/spring-transactional-propagation-isolation" \l "5notsupported-propagation)*:

@DataJpaTest

@Transactional(propagation = Propagation.NOT\_SUPPORTED)

**public** **class** **UserRepositoryIntegrationTest** {

// ... test methods

}Copy

Or we can disable transactions for an individual test method:

@Test

@Transactional(propagation = Propagation.NOT\_SUPPORTED)

**public** **void** **testMyMethodWithoutTransactions**() {

// ... code that modifies the database

}Copy

## 8. Conclusion

In this article, we learned how to use @DataJpaTest to test our JPA repository in JUnit. Overall, @DataJpaTest is a powerful annotation for testing JPA repositories in Spring Boot applications. It provides a focused testing environment and pre-configured tools for testing persistence layers. By using @DataJpaTest, we can ensure that our JPA repositories are functioning correctly without having to start up the entire Spring context.

As always, the source code for the examples is available [over on GitHub](https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-data-jpa-annotations-2).