##### **Testing Spring Data Repositories**

https://courses.baeldung.com/courses/1295711/lectures/30127904

**1. Goals**

In this lesson, we're going to learn how to test Spring Data repositories’ methods using a convenience annotation provided by Spring Boot.

**2. Lesson Notes**

The relevant module you need to import when you're starting with this lesson is: [testing-spring-data-repositories-start](https://github.com/Baeldung/learn-spring-data/tree/module2/testing-spring-data-repositories-start)

If you want to have a look at the fully implemented lesson, as a reference, feel free to import: [testing-spring-data-repositories-end](https://github.com/Baeldung/learn-spring-data/tree/module2/testing-spring-data-repositories-end)

**2.1. Unit Test or Integration Test**

Let’s quickly analyze the kinds of tests we can perform in order to check the correct behavior of the repositories’ methods.

If we want to create Unit Tests, then we have to mock several JPA components that Spring loads. Naturally, this would affect the functionality of the repository. Therefore, Unit Tests don't make much sense in this case.

On the other hand, when we test Spring Data repositories, we usually want to verify that our queries retrieve the expected data from a database when running a SELECT operation. When executing UPDATE, DELETE, and INSERT operations, we want to verify that our queries get executed successfully, maintaining the consistent state of our database.

Consequently, **we need integration tests, as the validation exceeds the bare Java code we actually develop.**

**2.2. Auto-configured Data JPA Tests**

**Spring Boot provides the *@DataJpaTest* annotation** to test our repositories. By default, this annotation invokes several Spring Boot auto-configurations that are used in our application. In this way, it **closely replicates the real behavior of our application**. Of course, it also allows us to customize these configurations if necessary.

Some of the features that *@DataJpaTest*provides for the tests include enabling the scanning of our *@Entity* classes, and configuring the Spring Data JPA repositories.Additionally, it **enables the logging of the database queries**that are executed, so that we can clearly see what's going on in the tests.

Naturally, this annotation doesn't configure all the layers when setting up the application context. Instead, **it configures only those that are suitable for data access tests**.

The *data.sql* initialization script is also executed for our tests, but we won't use these records in our test examples. Instead, we'll opt for a different approach, using one more feature provided by the annotation we're analyzing here. We'll explore this in detail in the next section.

With this annotation, we don’t need to declare transaction scopes in a test.**By default, all tests decorated with the *@DataJpaTest* annotation become transactional**. Moreover, the transaction rolls back at the end of each test, so as not to affect the initial database state of other tests.

There are use cases, however, in which we might not want to follow this approach. We’ll leave aside the analysis of the possible pitfalls of this approach, but it’s worth knowing that we can disable the default transaction management by adding an extra annotation:

java

@Transactional(propagation = Propagation.NOT\_SUPPORTED)

**2.3. Injecting a*TestEntityManager***

When working with Spring Data JPA, we don’t usually talk to the *EntityManager* directly. Our repositories will use a configured *EntityManager* to interact with the database.

However,**in order to populate the database with the test data or verify the test results, we can also inject a *TestEntityManager*** **bean in our tests**, which as you can guess, is configured when we add the *@DataJpaTest*annotation.

Alternatively, we can use the repositories' methods for this purpose, but we recommend using a *TestEntityManager,* as sometimes the repositories offer a limited set of operations. One of the possible scenarios might be a situation where we need to test a read-only repository, but we need to add records in the database in order to execute the tests successfully.

Moreover, we usually want to test each repository feature separately; therefore, using some of its functions as helper methods would affect the scope of the test. For instance, we may want to test a query method, but we would also be indirectly testing the persistence functionality. As a result, an error in this test could be misleading.

**2.4. Creating a Test Class**

Now let's see the *@DataJpaTest* annotation and *TestEntityManager* class in action. We'll test our *CampaignRepository* repository by creating a test class in the *com.baeldung.lsd.persistence.repository* package:

java

@DataJpaTest

class CampaignRepositoryIntegrationTest {

@Autowired

CampaignRepository campaignRepository;

@Autowired

TestEntityManager entityManager;

// ...

}

Note that we annotated the class, and injected the entity manager and repository we'll be testing.

**2.5. Testing the Repository Methods**

A repository can offer various methods that allow us to perform different queries against our database. Let's demonstrate how we can test them.

**Testing the Insertion Operation**

As we know, the repositories’ *save()* method can handle two operations: database insert and update. First, let's see how to test the insertion:

java

@Test

void givenNewCampaign\_whenSave\_thenSuccess() {

Campaign newCampaign = new Campaign("CTEST-1", "Test Campaign 1", "Description for campaign CTEST-1");

Campaign insertedCampaign = campaignRepository.save(newCampaign);

assertThat(entityManager.find(Campaign.class, insertedCampaign.getId()) ).isEqualTo(newCampaign);

}

As we can see, we create a new transient *Campaign*object, and then we call the repository's *save()* method. After that, we can find the newly inserted object using the injected *TestEntityManager,*and we expect it to be equal to the *newCampaign*instance.

Let's run the test, and make sure that it passes. If we check the console log, among other messages we’ll see the following:

text

Hibernate: insert into campaign (id, code, description, name) values (null, ?, ?, ?)

// ...

o.s.t.c.transaction.TransactionContext : Rolled back transaction for test

We can see the SQL statements executed during the test, which can undoubtedly be helpful when we’re hunting query problems. In addition, we're informed that the test transaction will be rolled back, so that the inserted data won’t actually be saved in our database and affect other tests.

**Testing the Update Operation**

Similarly, we can test the update functionality:

java

@Test

void givenCampaignCreated\_whenUpdate\_thenSuccess() {

Campaign newCampaign = new Campaign("CTEST-1", "Test Campaign 1", "Description for campaign CTEST-1");

entityManager.persist(newCampaign);

String newName = "New Campaign 001";

newCampaign.setName(newName);

campaignRepository.save(newCampaign);

assertThat(entityManager.find(Campaign.class, newCampaign.getId()).getName()).isEqualTo(newName);

}

First, we prepare the *newCampaign* instance and persist it, this time using the *entityManager*bean. Then we change its name, and call the repository's *save()* method. Finally, we verify that the name has been updated in the database.

**Testing the *findById* Method**

Now let’s test the *findById()* method. First, we'll persist a new *Campaign*object using the *TestEntityManager* bean, like we did in the previous test. Then we’ll call the repository's *findById()* method, and verify the returned value:

java

@Test

void givenCampaignCreated\_whenFindById\_thenSuccess() {

Campaign newCampaign = new Campaign("CTEST-1", "Test Campaign 1", "Description for campaign CTEST-1");

entityManager.persist(newCampaign);

Optional<Campaign> retrievedCampaign = campaignRepository.findById(newCampaign.getId());

assertThat(retrievedCampaign).contains(newCampaign);

}

**Testing Other *findBy...* Queries**

We can easily test any other read methods defined in our repository in the way described in the previous section. For example, in our repository, we defined the *findByNameContaining()* method, which retrieves an *Iterable*collection of entities, instead of an *Optional*. We can test it as follows:

java

@Test

void givenCampaignCreated\_whenFindByNameContaining\_thenSuccess() {

Campaign newCampaign1 = new Campaign("CTEST-1", "Test Campaign 1", "Description for campaign CTEST-1");

Campaign newCampaign2 = new Campaign("CTEST-2", "Test Campaign 2", "Description for campaign CTEST-2");

entityManager.persist(newCampaign1);

entityManager.persist(newCampaign2);

Iterable<Campaign> campaigns = campaignRepository.findByNameContaining("Test");

assertThat(campaigns).contains(newCampaign1, newCampaign2);

}

**Testing the Delete Operation**

Finally, let's test the *delete()* method. Once again, we create a *Campaign*entry in the database, and then we call our repository's *delete()* method. After that, we verify if the data entry has been removed:

java

@Test

void givenCampaignCreated\_whenDelete\_thenSuccess() {

Campaign newCampaign = new Campaign("CTEST-1", "Test Campaign 1", "Description for campaign CTEST-1");

entityManager.persist(newCampaign);

campaignRepository.delete(newCampaign);

assertThat(entityManager.find(Campaign.class, newCampaign.getId())).isNull();

}

You can check the codebase to see examples of tests for our other repositories.

**3. Resources**

- [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)