JPA Object Relational Mapping

Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column. It is capable to handle various database operations easily such as inserting, updating, deleting etc.

 

ORM Frameworks

Following are the various frameworks that function on ORM mechanism: -

* Hibernate
* TopLink
* ORMLite
* iBATIS
* JPOX

Mapping Directions

Mapping Directions are divided into two parts: -

* **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.
* **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

Types of Mapping

Following are the various ORM mappings: -

* **One-to-one -** This association is represented by @OneToOne annotation. Here, instance of each entity is related to a single instance of another entity.
* **One-to-many -** This association is represented by @OneToMany annotation. In this relationship, an instance of one entity can be related to more than one instance of another entity.
* **Many-to-one -** This mapping is defined by @ManyToOne annotation. In this relationship, multiple instances of an entity can be related to single instance of another entity.
* **Many-to-many -** This association is represented by @ManyToMany annotation. Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side.

We will learn about each mapping separately in later section of this tutorial.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Method name** | **Return** | **Description** | **Issued SQL statement** |
| 1 | beginTransaction() | Transaction | Creates a Transaction object or returns an existing one, for working under context of a transaction. |  |
| 2 | getTransaction() | Transaction | Returns the current transaction. |  |
| 3 | get(Class class, Serializable id) | Object | Loads a persistent instance of the given class with the given id, into the session. | SELECT |
| 4 | load(Class class, Serializable id) | Object | Does same thing as get() method, but throws an ObjectNotFound error if no row with the given id exists. | SELECT |
| 5 | persist(Object) | void | saves a mapped object as a row in database | INSERT |
| 6 | save(Object) | Serializable | Does same thing as persist()method, plus returning a generated identifier. | INSERT |
| 7 | update(Object) | void | Updates a detached instance of the given object and the underlying row in database. | UPDATE |
| 8 | saveOrUpdate(Object) | void | Saves the given object if it does not exist, otherwise updates it. | INSERT or UPDATE |
| 9 | delete(Object) | void | Removes a persistent object and the underlying row in database. | DELETE |
| 10 | close() | void | Ends the current session. |  |
| 11 | flush() | void | Flushes the current session. This method should be called before committing the transaction and closing the session. |  |
| 12 | disconnect() | void | Disconnects the session from current JDBC connection. |  |

## ****1. Overview****

This article will focus on **introducing Spring Data JPA into a Spring project** and fully configuring the persistence layer. For a step by step introduction about setting up the Spring context using Java based configuration and the basic Maven pom for the project, see [this article](https://www.baeldung.com/bootstraping-a-web-application-with-spring-and-java-based-configuration).

## ****2. The Spring Data generated DAO – No More DAO Implementations****

As we discussed in an earlier article, [the DAO layer](https://www.baeldung.com/simplifying-the-data-access-layer-with-spring-and-java-generics) usually consists of a lot of boilerplate code that can and should be simplified. The advantages of such a simplification are many: a decrease in the number of artifacts that we need to define and maintain, consistency of data access patterns and consistency of configuration.

Spring Data takes this simplification one step forward and **makes it possible to remove the DAO implementations entirely**. The interface of the DAO is now the only artifact that we need to explicitly define.

In order to start leveraging the Spring Data programming model with JPA, a DAO interface needs to extend the JPA specific Repository interface – JpaRepository. This will enable Spring Data to find this interface and automatically create an implementation for it.

By extending the interface we get the most relevant CRUD methods for standard data access available in a standard DAO.

## ****3. Custom Access Method and Queries****

As discussed, **by implementing one of the Repository interfaces, the DAO will already have some basic CRUD methods (and queries) defined and implemented**.

To define more specific access methods, Spring JPA supports quite a few options:

* simply**define a new method** in the interface
* provide the actual **JPQ query** by using the @Query annotation
* use the more advanced **Specification and Querydsl support** in Spring Data
* define **custom queries** via JPA Named Queries

The [third option](https://spring.io/blog/2011/04/26/advanced-spring-data-jpa-specifications-and-querydsl/) – the Specifications and Querydsl support – is similar to JPA Criteria but using a more flexible and convenient API. This makes the whole operation much more readable and reusable. The advantages of this API will become more pronounced when dealing with a large number of fixed queries, as we could potentially express these more concisely through a smaller number of reusable blocks.

This last option has the disadvantage that it either involves XML or burdening the domain class with the queries.

### **3.1. Automatic Custom Queries**

When Spring Data creates a new Repository implementation, it analyses all the methods defined by the interfaces and tries to **automatically generate queries from the method names**. While this has some limitations, it's a very powerful and elegant way of defining new custom access methods with very little effort.

Let's look at an example: if the entity has a name field (and the Java Bean standard getName and setName methods), **we'll define the findByName method in the DAO interface**; this will automatically generate the correct query:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IFooDAO extends JpaRepository<Foo, Long> {       Foo findByName(String name);    } |

This is a relatively simple example. The query creation mechanism supports [a much larger set of keywords](https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#jpa.query-methods.query-creation).

In case that the parser cannot match the property with the domain object field, we'll see the following exception:

|  |  |
| --- | --- |
| 1 | java.lang.IllegalArgumentException: No property nam found for type class org.rest.model.Foo |

### **3.2. Manual Custom Queries**

Let's now look at a custom query that we'll define via the @Query annotation:

|  |  |
| --- | --- |
| 1  2 | @Query("SELECT f FROM Foo f WHERE LOWER(f.name) = LOWER(:name)")  Foo retrieveByName(@Param("name") String name); |

For even more fine-grained control over the creation of queries, such as using named parameters or modifying existing queries, [the reference](https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#jpa.named-parameters) is a good place to start.

## ****4. Transaction Configuration****

The actual implementation of the Spring Data managed DAO is indeed hidden since we don't work with it directly. However, this is a simple enough implementation – t**he SimpleJpaRepository – which defines transaction semantics using annotations**.

More explicitly, this uses a read-only @Transactional annotation at the class level, which is then overridden for the non-read-only methods. The rest of the transaction semantics are default, but these can be easily overridden manually per method.

### **4.1. Exception Translation Is Alive and Well**

The question is now – since we're not using the default Spring ORM templates (JpaTemplate, HibernateTemplate) – are we losing exception translation by using Spring Data JPA? Are we not going to get our JPA exceptions translated to Spring's DataAccessException hierarchy?

Of course not – **exception translation is still enabled by the use of the @Repository annotation on the DAO**. This annotation enables a Spring bean postprocessor to advise all @Repository beans with all the PersistenceExceptionTranslator instances found in the Container, and provide exception translation just as before.

Let's verify exception translation with an integration test:

|  |  |
| --- | --- |
| 1  2  3  4 | @Test(expected = DataIntegrityViolationException.class)  public void givenFooHasNoName\_whenInvalidEntityIsCreated\_thenDataException() {      service.create(new Foo());  } |

Keep in mind that **exception translation is done through proxies.** In order for Spring to be able to create proxies around the DAO classes, these must not be declared final.

## ****5. Spring Data Configuration****

To activate the Spring JPA repository support we can use the @EnableJpaRepositories annotation and specify the package that contains the DAO interfaces:

|  |  |
| --- | --- |
| 1  2 | @EnableJpaRepositories(basePackages = "com.baeldung.jpa.dao")  public class PersistenceConfig { ... } |

We can do the same with an XML configuration:

|  |  |
| --- | --- |
| 1 | <jpa:repositories base-package="org.rest.dao.spring" /> |

## ****6. The Spring Java or XML Configuration****

We already discussed in great detail how to [configure JPA in Spring](https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa) in a previous article. Spring Data also takes advantage of the Spring support for the JPA @PersistenceContext annotation. It uses this to wire the EntityManagerinto the Spring factory bean responsible for creating the actual DAO implementations – JpaRepositoryFactoryBean.

In addition to the already discussed configuration, we also need to include the Spring Data XML Config – if we are using XML:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Configuration  @EnableTransactionManagement  @ImportResource( "classpath\*:\*springDataConfig.xml" )  public class PersistenceJPAConfig{     ...  } |

## ****7. The Maven Dependency****

In addition to the Maven configuration for JPA-defined in a [previous article](https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa), the spring-data-jpa dependency is added:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>     <groupId>org.springframework.data</groupId>     <artifactId>spring-data-jpa</artifactId>     <version>2.1.6.RELEASE</version>  </dependency> |

## 8. Using Spring Boot

**We can also use the**[**Spring Boot Starter Data JPA**](https://search.maven.org/search?q=a:spring-boot-starter-data-jpa)**dependency that will automatically configure the DataSource for us.**

We also need to make sure that the database we want to use is present in the classpath. In our example, we've added the H2 in-memory database:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <dependency>     <groupId>org.springframework.boot</groupId>     <artifactId>spring-boot-starter-data-jpa</artifactId>     <version>2.1.3.RELEASE</version>  </dependency>  <dependency>      <groupId>com.h2database</groupId>      <artifactId>h2</artifactId>      <version>1.4.197</version>  </dependency> |

That’s it, just by doing these dependencies, our application is up and running and we can use it for other database operations.

**The explicit configuration for a standard Spring application is now included as part of Spring Boot auto-configuration.**

We can, of course, modify the auto-configuration by adding our own explicit configuration.

Spring Boot provides an easy way to do this using properties in the application.properties file:

|  |  |
| --- | --- |
| 1  2  3 | spring.datasource.url=jdbc:h2:mem:db;DB\_CLOSE\_DELAY=-1  spring.datasource.username=sa  spring.datasource.password=sa |

In this example, we've changed the connection URL and credentials.

## ****9. Conclusion****