Spring Data Annotations

**1. Introduction**

Spring Data provides an abstraction over data storage technologies. Therefore, our business logic code can be much more independent of the underlying persistence implementation. Also, Spring simplifies the handling of implementation-dependent details of data storage.

In this tutorial, we’ll see the most common annotations of the Spring Data, Spring Data JPA, and Spring Data MongoDB projects.

**2. Common Spring Data Annotations**

**2.1. *@Transactional***

When we want to **configure the transactional behavior of a method**, we can do it with:

|  |  |
| --- | --- |
| 1  2 | @Transactional  void pay() {} |

If we apply this annotation on class level, then it works on all methods inside the class. However, we can override its effects by applying it to a specific method.

It has many configuration options, which can be found in [this article](https://www.baeldung.com/transaction-configuration-with-jpa-and-spring).

**2.2. *@NoRepositoryBean***

**Sometimes we want to create repository interfaces with the only goal of providing common methods for the child repositories**.

Of course, we don’t want Spring to create a bean of these repositories since we won’t inject them anywhere. *@NoRepositoryBean*does exactly this: when we mark a child interface of *org.springframework.data.repository.Repository*, Spring won’t create a bean out of it.

For example, if we want an *Optional<T> findById(ID id)*method in all of our repositories, we can create a base repository:

|  |  |
| --- | --- |
| 1  2  3  4 | @NoRepositoryBean  interface MyUtilityRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {      Optional<T> findById(ID id);  } |

This annotation doesn’t affect the child interfaces; hence Spring will create a bean for the following repository interface:

|  |  |
| --- | --- |
| 1  2 | @Repository  interface PersonRepository extends MyUtilityRepository<Person, Long> {} |

Note, that the example above isn’t necessary since Spring Data version 2 which includes this method replacing the older *T findOne(ID id)*.

**2.3. *@Param***

We can pass named parameters to our queries using *@Param*:

|  |  |
| --- | --- |
| 1  2 | @Query("FROM Person p WHERE p.name = :name")  Person findByName(@Param("name") String name); |

Note, that we refer to the parameter with the *:name*syntax.

For further examples, please visit [this article](https://www.baeldung.com/spring-data-jpa-query).

**2.4. *@Id***

*@Id*marks a field in a model class as the primary key:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Person {        @Id      Long id;        // ...    } |

Since it’s implementation-independent, it makes a model class easy to use with multiple data store engines.

**2.5. *@Transient***

We can use this annotation to mark a field in a model class as transient. Hence the data store engine won’t read or write this field’s value:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person {        // ...        @Transient      int age;        // ...    } |

Like *@Id*, *@Transient*is also implementation-independent, which makes it convenient to use with multiple data store implementations.

**2.6. *@CreatedBy*, *@LastModifiedBy*, *@CreatedDate*, *@LastModifiedDate***

With these annotations, we can audit our model classes: Spring automatically populates the annotated fields with the principal who created the object, last modified it, and the date of creation, and last modification:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class Person {        // ...        @CreatedBy      User creator;        @LastModifiedBy      User modifier;        @CreatedDate      Date createdAt;        @LastModifiedDate      Date modifiedAt;        // ...    } |

Note, that if we want Spring to populate the principals, we need to use Spring Security as well.

For a more thorough description, please visit [this article](https://www.baeldung.com/database-auditing-jpa).

**3. Spring Data JPA Annotations**

**3.1. *@Query***

With *@Query*, we can provide a JPQL implementation for a repository method:

|  |  |
| --- | --- |
| 1  2 | @Query("SELECT COUNT(\*) FROM Person p")  long getPersonCount(); |

Also, we can use named parameters:

|  |  |
| --- | --- |
| 1  2 | @Query("FROM Person p WHERE p.name = :name")  Person findByName(@Param("name") String name); |

Besides, we can use native SQL queries, if we set the *nativeQuery*argument to *true*:

|  |  |
| --- | --- |
| 1  2 | @Query(value = "SELECT AVG(p.age) FROM person p", nativeQuery = true)  int getAverageAge(); |

For more information, please visit [this article](https://www.baeldung.com/spring-data-jpa-query).

**3.2. *@Procedure***

**With Spring Data JPA we can easily call stored procedures from repositories.**

First, we need to declare the repository on the entity class using standard JPA annotations:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @NamedStoredProcedureQueries({      @NamedStoredProcedureQuery(          name = "count\_by\_name",          procedureName = "person.count\_by\_name",          parameters = {              @StoredProcedureParameter(                  mode = ParameterMode.IN,                  name = "name",                  type = String.class),              @StoredProcedureParameter(                  mode = ParameterMode.OUT,                  name = "count",                  type = Long.class)              }      )  })    class Person {} |

After this, we can refer to it in the repository with the name we declared in the *name*argument:

|  |  |
| --- | --- |
| 1  2 | @Procedure(name = "count\_by\_name")  long getCountByName(@Param("name") String name); |

**3.3. *@Lock***

We can configure the lock mode when we execute a repository query method:

|  |  |
| --- | --- |
| 1  2  3 | @Lock(LockModeType.NONE)  @Query("SELECT COUNT(\*) FROM Person p")  long getPersonCount(); |

The available lock modes:

* *READ*
* *WRITE*
* *OPTIMISTIC*
* *OPTIMISTIC\_FORCE\_INCREMENT*
* *PESSIMISTIC\_READ*
* *PESSIMISTIC\_WRITE*
* *PESSIMISTIC\_FORCE\_INCREMENT*
* *NONE*

**3.4. *@Modifying***

We can modify data with a repository method if we annotate it with *@Modifying*:

|  |  |
| --- | --- |
| 1  2  3 | @Modifying  @Query("UPDATE Person p SET p.name = :name WHERE p.id = :id")  void changeName(@Param("id") long id, @Param("name") String name); |

For more information, please visit [this article](https://www.baeldung.com/spring-data-jpa-query).

**3.5. *@EnableJpaRepositories***

To use JPA repositories, we have to indicate it to Spring. We can do this with *@EnableJpaRepositories.*

Note, that we have to use this annotation with *@Configuration*:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableJpaRepositories  class PersistenceJPAConfig {} |

Spring will look for repositories in the sub packages of this *@Configuration*class.

We can alter this behavior with the *basePackages*argument:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableJpaRepositories(basePackages = "org.baeldung.persistence.dao")  class PersistenceJPAConfig {} |

Also note, that Spring Boot does this automatically if it finds Spring Data JPA on the classpath.

**4. Spring Data Mongo Annotations**

Spring Data makes working with MongoDB much easier. In the next sections, we’ll explore the most basic features of Spring Data MongoDB.

For more information, please visit our [article about Spring Data MongoDB](https://www.baeldung.com/spring-data-mongodb-tutorial).

**4.1. *@Document***

This annotation marks a class as being a domain object that we want to persist to the database:

|  |  |
| --- | --- |
| 1  2 | @Document  class User {} |

It also allows us to choose the name of the collection we want to use:

|  |  |
| --- | --- |
| 1  2 | @Document(collection = "user")  class User {} |

Note, that this annotation is the Mongo equivalent of *@Entity*in JPA.

**4.2. *@Field***

With *@Field*, we can configure the name of a field we want to use when MongoDB persists the document:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Document  class User {        // ...        @Field("email")      String emailAddress;        // ...    } |

Note, that this annotation is the Mongo equivalent of *@Column*in JPA.

**4.3. *@Query***

With *@Query*, we can provide a finder query on a MongoDB repository method:

|  |  |
| --- | --- |
| 1  2 | @Query("{ 'name' : ?0 }")  List<User> findUsersByName(String name); |

**4.4. *@EnableMongoRepositories***

To use MongoDB repositories, we have to indicate it to Spring. We can do this with *@EnableMongoRepositories.*

Note, that we have to use this annotation with *@Configuration*:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableMongoRepositories  class MongoConfig {} |

Spring will look for repositories in the sub packages of this *@Configuration*class. We can alter this behavior with the *basePackages*argument:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableMongoRepositories(basePackages = "org.baeldung.repository")  class MongoConfig {} |

Also note, that Spring Boot does this automatically if it finds Spring Data MongoDB on the classpath.

**5. Conclusion**

In this article, we saw which are the most important annotations we need to deal with data in general, using Spring. In addition, we looked into the most common JPA and MongoDB annotations