What is Spring Data JPA?

When you implement a new application, you should focus on the business logic instead of technical complexity and boilerplate code. That's why the Java Persistence API (JPA) specification and Spring Data JPA are extremely popular. JPA handles most of the complexity of JDBC-based database access and object-relational mappings. On top of that, Spring Data JPA reduces the amount of boilerplate code required by JPA. That makes the implementation of your persistence layer easier and faster.

Spring Data JPA vs JPA vs Hibernate/EclipseLink

JPA is a specification that defines an API for object-relational mappings and for managing persistent objects. Hibernate and EclipseLink are 2 popular implementations of this specification (more details).

Spring Data JPA adds a layer on top of JPA. That means it uses all features defined by the JPA specification, especially the entity and association mappings, the entity lifecycle management, and JPA's query capabilities. On top of that, Spring Data JPA adds its own features like a no-code implementation of the repository pattern and the creation of database queries from method names.

3 reasons to use Spring Data JPA

1. No-code Repositories

Implementing the repository pattern isn't too complicated but writing the standard CRUD operations for each entity creates a lot of repetitive code. Spring Data JPA provides you a set of repository interfaces which you only need to extend to define a specific repository for one of your entities.

2. Reduced boilerplate code

Spring Data JPA provides a default implementation for each method defined by one of its repository interfaces.

3. Generated queries

Another comfortable feature of Spring Data JPA is the generation of database queries based on method names. As long as your query isn't too complex, you just need to define a method on your repository interface with a name that starts with find...By. Spring then parses the method name and creates a query for it.

Using Spring Data JPA with Spring Boot

If youusing Spring Boot and structure your application in the right way, you only need to add the spring-boot-starter-data-jpa artifact, and your JDBC driver to your maven build. The Spring Boot Starter includes all required dependencies and activates the default configuration.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>test</scope>
</dependency>
```

In the next step, you can configure your database connection in the application.properties or application.yml file. If you use JPA outside of Spring, you need to configure this and a few other things in the persistence.xml. Spring Boot and Spring Data JPA handle the default configuration for you, so that you only need to override the parameters you want to change.

```
spring.datasource.url = jdbc:postgresql://localhost:5432/recipes
spring.datasource.username = postgres
spring.datasource.password = postgres
```

Repositories in Spring Data JPA

After setting everything up, it's time to take a closer look at repositories. There are 3 repository interfaces that you should know when you use Spring Data JPA:

- CrudRepository
- PagingAndSortingRepository
- JpaRepository

As you might guess from its name, the CrudRepository interface defines a repository that offers standard create, read, update and delete operations. The PagingAndSortingRepository extends the CrudRepository and adds findAll methods that enable you to sort the result and to retrieve it in a paginated way. Both interface are also supported by other Spring Data projects, so that you can apply the same concepts to different datastores. The JpaRepository adds JPA-specific methods, like flush() to trigger a flush on the persistence context or findAll(Example<S> example) to find entities by example, to the PagingAndSortingRepository.

Defining an entity-specific repository

You can use any of the standard interfaces to define your own repository definition. You, therefore, need to extend one of Spring Data JPA's interface, e.g. the CrudRepository interfaces and type it to the entity class and its primary key class. You can also add custom query methods to your repository.

```
public interface BookRepository extends CrudRepository<Book, Long> {


    Book findByTitle(String title);
}
```

Working with Repositories

After you defined your repository interface, you can use the @Autowired annotation to inject it into your service implementation. Spring Data will then provide you with a proxy implementation of your repository interface. This proxy provides default implementations for all methods defined in the interface.

In your business code, you can then use the injected repository to read entities from the database and to persist new or changed entities.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class GettingStartedApplicationTests {


    @Autowired
    private BookRepository bookRepository;



    @Test
    @Transactional
    public void testByTitle() {
        Book b = bookRepository.findByTitle("Hibernate Tips");
        Assert.assertEquals(new Long(1), b.getId());
    }
}
```