Spring Data JPA

The relationship between Spring Data JPA, JPA, and Hibernate/EclipseLink/Kodo n /JDBC

JPA is a part of Java EE/Jakarta EE specification that defines an API for ORM and for managing persistent objects. Hibernate and EclipseLink are popular implementations of this specification.

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.

If the JPA specification and its implementations provide most of the features that you use with Spring Data JPA, do you really need the additional layer? Can't you just use the Hibernate directly ?

You can, of course, do that. That's what a lot of Java applications do. Spring ORM provides a good integration for JPA (eg : Spring native Hibernate Integration or Spring JPA)

But the Spring Data team took the extra step to make your job a little bit easier. The additional layer on top of JPA enables them to integrate JPA into the Spring stack easily.
They also provide a lot of functionality that you otherwise would need to implement yourself.

Why Spring Data JPA

1. No-code Repositories
The repository pattern is one of the most popular persistence-related patterns. It hides the DB specific implementation details and enables you to implement your business logic with higher abstraction level.

eg : For Author Entity
How ?
to persist, update and remove one or multiple Author entities,
to find one or more Authors by their primary keys,
to count, get and remove all Authors and
to check if an Author with a given primary key exists.

All you need to do is :
```
public interface AuthorRepository extends JpaRepository<Author, Integer> {}
```

---

2. Reduced boilerplate code
To make it even easier, Spring Data JPA provides a default implementation for each method defined by one of its repository interfaces. You don't need to implement these operations.

---

3. Generated queries
Another cool feature of Spring Data JPA is the generation of database queries based on method names.(finder method pattern)

eg : Write a method that gets a Book entity with a given title. Internally, Spring generates a JPQL query based on the method name, sets the provided method parameters as bind parameter values, executes the query and returns the result.

```
//JpaRepository<T,ID>
//T : entity type
//ID : Data type of id property(PK)
public interface BookRepository extends JpaRepository<Book, Integer> {

    Optional<Book> findByTitle(String title123);
}
```

Assumption : title : property of Book POJO

Using Spring Data JPA with Spring Boot

You only need to add the spring-boot-starter-data-jpa and your JDBC driver to your maven build. The Spring Boot Starter includes all required dependencies and activates the default configuration.
Add DB config properties in application.properties file

By default, Spring Boot expects that all repositories are located in a sub-packages of the class annotated with @SpringBootApplication.
If your application doesn’t follow this default, you need to configure the packages of your repositories using an @EnableJpaRepositories annotation.

Repositories(API) in Spring Data JPA
package : o.s.data.repository
CrudRepository
PagingAndSortingRepository
JpaRepository
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.
The JpaRepository adds JPA-specific methods, like flush() to trigger a flush on the persistence context or findAll(Example<S> example) to find entities

Defining an entity-specific repository
eg :
Book entity is a normal JPA entity with a generated primary key of type Long, a title and a many-to-many association to the Author entity.

```
@Entity
@Table(name="books")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;

    @Version
    private int version; //for optimistic locking

    private String title;

    @ManyToMany
    @JoinTable(name = "book_author",
                joinColumns = { @JoinColumn(name = "fk_book") },
                inverseJoinColumns = { @JoinColumn(name = "fk_author") })
    private Set<Author> authors = new HashSet<>();

    ...
}
```

If you want to define a JPA repository for this entity, you need to extend Spring Data JpaRepository interface and type it to Book and Long.

```
public interface BookRepository extends JpaRepository<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 entire web application ,the DAO layer usually consists of a lot of boilerplate code that can be simplified.

Benefits of simplification
1. Decrease in the number of layers that we need to define and maintain
2. Consistency of data access patterns and consistency of configuration.

Spring Data JPA framework 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.

For this , a DAO interface needs to extend the JPA specific Repository interface – JpaRepository or its super i/f CrudRepository. This will enable Spring Data to find this interface and automatically create an implementation for it.

1. Inherited API
API

1.o.s.data.repository.Repository<T,ID> : a marker i/f

---

2. Sub i/f
o.s.data.repository.CrudRepository<T,ID>

By extending from this interface we get the most required CRUD methods in a standard DAO.

eg : CRUDRepository methods
long   count()
Returns the number of entities available.
void   delete(T entity)
Deletes a given entity.
void   deleteAll()
Deletes all entities managed by the repository.
void   deleteAll(Iterable<? extends T> entities)
Deletes the given entities.
void   deleteById(ID id)
Deletes the entity with the given id.
boolean   existsById(ID id)
Returns whether an entity with the given id exists.
Iterable<T>   findAll()
Returns all instances of the type.
Iterable<T>   findAllById(Iterable<ID> ids)
Returns all instances of the type with the given IDs.
Optional<T>   findById(ID id)
Retrieves an entity by its id.
<S extends T>
S   save(S entity)
Saves or updates a given entity.
<S extends T>
Iterable<S>   saveAll(Iterable<S> entities)
Saves all given entities.

---

3. Sub i/f
o.s.data.repository.PagingAndSortingRepository<T, ID>
Methods
Iterable<T> findAll(Sort sort);
Page<T> findAll(Pageable pageable);
Used for sorting n pagination

---

4. Sub i/f
org.springframework.data.jpa.repository.JpaRepository<T, ID>

Method of JpaRepository
void   deleteAllInBatch()
Deletes all entities in a batch call.
void   deleteInBatch(Iterable<T> entities)
Deletes the given entities in a batch which means it will create a single Query.
List<T>   findAll()

```
<S extends T>
List<S>         findAll(Example<S> example)
<S extends T>
List<S>         findAll(Example<S> example, Sort sort)
List<T>         findAll(Sort sort)
List<T>         findAllById(Iterable<ID> ids)
void    flush()
Flushes all pending changes to the database.
T       getOne(ID id)
Returns a reference to the entity with the given identifier.
<S extends T>
List<S>         saveAll(Iterable<S> entities)
```

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

1. simply define a new method in the interface
provide the actual JPQL query by using the @Query annotation
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.
eg :

```
   Customer findByName(String name);
   List<Person> findByAddressAndLastname(String address, String lastname);

  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String
firstname);

  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);

  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

List<Person> findByAddressZipCode(String zipCode);
```

Assuming a Person p has an Address with a String zipCode. In that case, the method creates the property traversal p.address.zipCode.

Limiting the result size of a query with Top and First

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);
```

```
List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

---

2. Derived Query methods(Finder methods typically !)

By extending one of the Repository interfaces, the DAO will already have some basic CRUD methods (and queries) defined and implemented.

3

Custom query methods

One can add directly in DAO i/f, using JPQL

eg :

```
@Query("select u from User u where u.emailAddress = :em")
  User fetchUserByEmailAddress(@Param("em")String emailAddress);

@Query("SELECT p FROM Person p WHERE LOWER(p.name) = LOWER(:nm)")
Foo retrieveByName(@Param("nm") String name);

@Modifying
  @Query("delete from User u where u.role.id = ?1")
  void deleteInBulkByRoleId(long roleId);

@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);
```

---

3. Transaction Configuration

The actual implementation of the Spring Data managed DAO is hidden since we don’t work with it directly. It's implemented by – the SimpleJpaRepository – which defines default transaction mechanism using annotations.

These can be easily overridden manually per method.

---

4. Exception Translation is still supplied

Exception translation is still enabled by the use of the @Repository annotation internally applied on the DAO implementation class.