Spring Data JPA

# ORM Basics

## ORM

* ORM stands for object relational mapping it is the process of mapping a Java class to a database table and its fields or members to the database table columns.
* Once we do that mapping we can synchronized our class objects into database rows.
* We as Object Oriented developers will deal with objects instead of writing sql and invoke methods like SAVE Update Delete on the objects and automatically the sql statements will be generated for us.
* And using JDBC internally these ORM Tools will do the insertion updation deletion etc. so we no longer need to deal with sql directly and also we no longer have to deal with low level API's like JDBC.

## JPA

* JPA stands for Java Persistence API and it is a **standard** from Oracle to perform object relational mapping in Java EE applications.
* Like any other standard from Oracle JPA it comes with **specification** and an **API**.

The **specification** is for the JPA vendors or the providers. The **API** is for us the developers or the programmers to master it.

* There are several popular JPA **providers like Hibernate, OpenJPA, Eclipse Link,** etc. which implement the JPA API by following the specification.
* The **specification** is nothing but a set of rules written in plain English so that these vendors can implement the API easily and consistently.
* **Before JPA** came into existence, we as developers had to learn Hibernate API, Open JPA API or Eclipse link API, depending on which of these ORM tools we are using for our application. But now we learn one single API (JPA) and all these vendors/providers implement this API and we can switch from one vendor to another without making any changes to our application. That is the power of JPA.
* **Hibernate** is the most popular JPA provider and used in most of the Java EE applications.
* The EntityManagerFactory and EntityManager are specific to JPA. Spring Data will even hide this from us. We need not deal with this.

## What and Why Spring Data

* Developers at spring have a habit of making things easier for us and remove a lot of boiler plate coding that we usually do spring data is one such framework which removes a lot of configuration and coding hassle when we deal with data access layer for about applications.
* Two Simple Steps
  + All you need to do as a developer to come up with your data access layer is define your domain object or JPA entity with all the fields on it and map it to the database table using JPA annotations.
  + Once you do it you need not write any DAO classes or interfaces, simply define one single interface. It is not even a class, this interface that we create will extend a CRUDRepository interface from spring that is all that is required.
  + Once we do that at runtime we'll be able to access this employee repository in our services layer or in your presentation layer or wherever we want to perform database operations. Spring will dynamically implement a class for us at runtime out of the repository interface that we created.
* Internally spring data will use JPA and the power of ORM tools like Hibernate and it will use entity managers, session, session factory, etc. All that will happen auto magically behind the scenes. But in our application will have only two classes the entity classes and then the repository interfaces and will get everything for free at runtime.
* Spring data offers **finder methods** – so without writing any sql code, we can load data from the database by following simple method convention that it gives us.
* It makes it a lot easier to do **paging and sorting**.
* We can also use **JPQL** (JPA Query Language) and even **Native SQL Queries**.
* Whatever we can do in JPA while using Hibernate, spring data will make it a lot simpler for the developers and it offers several other features.

# Simple CRUD Operations

## Entity

* @Entity – This is mandatory. It indicates that the class is a JPA Entity.
* @Table – This is optional. We need this only if the database table name is different from the class name.
* @Id – to annotate a field which is a primary key in database table.
* @Column – is optional. We need not mark each of these fields with @Column. We only need it if the database column name is different from our field name.

## Repository

* Create any repository interface that will extend the CRUDRepository interface from spring data.
* The CRUDRpository expects 2 parameters in the form of Generics. One – for which entity we are creating this repository. And other is – what is the type of @Id field in that entity class.

## Configure Data Source

* Now we need to configure the data source information that is the JDBC connection URL, database user name and password.
* Thanks to spring boot, we can configure the data source information in the /src/main/resources/application.properties file.
* Some of the properties are

spring.datasource.url – JDBC URL of the database.

spring.datasource.username – Login username of the database.

spring.datasource.password – Login password of the database.

* That’s all! With this information, the database connection will be created when we run our application.
* For all other properties, refer –

<https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#common-application-properties-data>

## How Spring Boot Does Behind The Scenes

* Based on project classpath, Spring boots gets necessary information to create Data Source (using spring data jps jar, hibernate jar, mysql or any other DB driver connector jar, application properties file for configuration).
* With the relevant data source information available, Spring Boot will create EntityManagerFactory in background. And then it will create the EntityManager instance.
* And when we execute any method (save, delete, update, etc.) our custom repository which is an interface, spring internally creates an implementation or a proxy implementation of that repository interface. That on-the-fly generated implementation class will in turn invoke the methods on the EntityManager because entity manager is the one which in the end talks to database.
* So we have avoided all the boiler plate code and configuration by simply using spring data.

# ID Generators

## ID Generation Strategies

* JPA provides or support four different types of ID generation strategies –
  + Auto – GenerationType.AUTO
  + Identity – GenerationType.IDENTITY
  + Sequence – GenerationType.SEQUENCE
  + Table – GenerationType.TABLE
* We can mark our @Id field in our entity class using @GeneratedValue annotation.

E.g. @GeneratedValue(strategy = GenerationType.AUTO)

* In addition to above 4 strategies, there is one more way which is “Using custom ID generation”.

### Auto

* Auto means the persistence providers like Hibernate will have to check with the underlying database which kind of ID generations strategies (identity or sequence or table) does it support.
* Accordingly it will use one of these strategies which is supported by the underlying database to generate our primary key value. That is why it is called AUTO automatic.

### Identity

* Here the persistence provider (e.g. Hibernate) will rely on the auto increment field.
* When we create a primary key in the database, will have to also configure it to be an auto increment field (AUTO\_INCREMENT). So the persistence providers like Hibernate will use the incremented value every time when a record is inserted.

### Sequence

* In this case, you can define a custom logic to generate a value and you can then use that value as an ID.
* So the persistence provider will use a database sequence.
* When you configure your generation type as sequence, you also need to tell hibernate which sequence in the database it should use. Hibernate will run that sequence, take the results from that and use it as a primary key value when your record is saved.
* Databases like mysql do not support sequences.

### Table

* Here will be using a special table. And our persistence provider will use this special table.
* It will generate a value, put that value into this special table in a column and it will also use that value as a primary key to our (entity) table.
* The next time we are saving a record, it will look at the previous value in that special table, generate the next value and use that as a primary key for that particular record. That is the reason it is called Generation type table.
* We need to create this special table along with the entities table.

#### Steps

* Create the special table in database. E.g.

create table id\_gen(

gen\_name varchar(60) PRIMARY KEY,

gen\_val int(20)

)

* Annotate your entity class’s @Id field with one more annotation @TableGenerator and pass below info –
  + name – Since we have only one ID generation table, we need to use this ‘name’ attribute to differentiate sequences for different entities (e.g. Employee)
  + table – The special ID generation table name.
  + pkColumnName – primark key column name of the special generation table,
  + valueColumnName – the column of the special generation table which stores the value of the sequence.
  + allocationSize – specifies by how much the sequence should be incremented.
  + e.g. @TableGenerator(**name = “employee\_gen”**, table = “id\_gen”, pkColumnName = ”gen\_name”, valueColumnName=”gen\_val”, allocationSize = 1)
* Finally, mark the @Id field with table type generation strategy and also provide generator as –

@GeneratedValue(strategy = GenerationType.**TABLE**, **generator = “employee\_gen”**)

The generator attribute here is set equal to what is defined at @TableGenerator annotation.

### Custom ID Generation

* In addition to above 4 strategies, there is one more way which is “Using custom ID generation”.
* Create a class which implements Hibernate’s IdentityGenerator interface and override the only one method called generate().
* One of the parameters to this generate() method is the Entity object and this method should return generated ID. So inside this method, we should implement our custom ID generation logic.
* Now for your entity class, make these changes
  + Add annotation @GenericGenerator to the @Id field of the entity class like this –

@GenericGenerator(name = “**emp\_id**”, strategy = “com.sameer.generator.CustomIdGenerator”)

So for this strategy attribute, pass the name of your custom ID generation class.

* + Also add another annotation @GeneratedValue like this –

@GeneratedValue(generator = “**emp\_id**”)

# Spring Data Finder Methods

## Introduction

* Finder methods is a very powerful feature that spring data offers where without writing any code or even a SQL query, just by following some naming conventions, we can load data from the database.
* Refer this link to see naming conventions for different ways of fetching data.

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

# Paging and Sorting

## Introduction

* To enable paging, instead of extending the CRUDRepository, our repository interface should extend PagingAndSortingRepository which is a child interface of the CRUDRepository.
* Once you do this, you will get paging and sorting on the findAll method inside your repository and also it is easy to implement paging and sorting on your custom finder methods (e.g. findByName(), etc.).
* To implement paging and sorting on your custom finder methods, simply add an additional parameter called Pagable as last parameter. E.g. findByName(String name, Pageable pg). Now you can pass this method a PageRequest object and depending on how you want the page and sort, it will work for you automatically.
* Note – You need not extend PagingAndSortingRepository if you want to. If you want your findAll() method to support paging and sorting that is when you extend PagingAndSortingRepository. Otherwise you can extend CRUDRepository and you can pass an additional parameter called Pagable to your custom finder methods and that method will then start supporting all the paging and sorting logic.
* The key interfaces when you do **paging**, you use Pagable and PageRequest is an implementation of that interface.
* So once we enable paging, to use paging in our services layer or any other layers of our application where we are retrieving the database record using finder methods, we use Pagable and PageRequest.
* For **sorting,** the key classes are Sort, Direction and Order from spring data. Order represents a direction and property.
* So all of these are not a part of JPA. They are part of spring data which makes our life a lot easier to implement paging and sorting.
* You can only Paging or only Sorting or both Paging and Sorting depending on your requirement.

# JPQL

## Introduction

* JPQL stands for a Java Persistence Query Language.
* It is a standard from JPA to perform queries against objects and domain classes. So instead of writing SQL queries against database tables and columns, we write JP queries against our objects and their fields.
* These JP queries are internally converted to native SQL queries by the ORM tools like Hibernate.
* Most of what is there in sql is also provided in JPQL keywords and syntax.
* JPQL is **case sensitive** when it comes to the domain class names and its field names but it is **not case sensitive** to when it comes to keywords in the language syntax itself like ‘select’, ‘count’, etc.

## Using JPQL

* Mark the method in your repository interface with @Query annotation. And pass your JP query to it. You can also use alias for your entity names in the JPQ like native queries.
* E.g.   
  Assuming Entity class name as Student and underlying table name as student\_t.  
  @Query(“select \* from **S**tudent”)  
  List<Student> findAllStudents();

@Query(“from **S**tudent”)  
List<Student> findAllStudents();

* Note – @Query takes one optional boolean parameter called nativeQuery. If set to true, you can pass native SQL query. Otherwise it is treated as JPQ.

E.g.  
@Query(value = "select \* from **student\_t**", **nativeQuery = true**)

List<Student> findAllStudentNQ();

* Note –
  + If you are fetching all columns, then writing ‘select \*’ is optional.
  + If you are fetching only specific columns, the return type of your method should be List<Object[]>. And each element in the list is an array of Object type. So to access each columns, use indexes.

E.g.

@Query(“select firstName, lastName from Student”)  
List<Object[]> findAllStudentsPartialData();

When you loop over this returned List, you can access firstName as obj[0], lastName as obj[1].

### Using Named Query Parameters

* Using named query parameters, we can dynamically pass in parameters to the repository function and then set those into the query mentioned in @Query.
* To bind the correct method argument to the named parameter in the JPQ, we need to use @Param annotation.
* E.g. 1

@Query("from Student where firstName=:**firstName**")

List<Student> findAllStudentsByFirstName(**@Param**("**firstName**") String firstName);

* E.g. 2

@Query("from Student where score>**:min** and score<**:max**")

List<Student> findStudentsForGivenScores(@Param("**min**") int min, @Param("**max**") int max);

### Non Select (DML) Operations

* JPQL not only supports select operations but also DML statements like insert, update and delete.
* It is almost the same as writing select/find queries. i.e. it also uses the same @Query, @Param annocatations.
* In addition to this, we also need to annotate the method with **@Modifying** annotation. The reason being spring data by default assumes that all queries in your repository are read queries. So with @Modifying, we tell Soring data that this query is going to modify data (insert, update or delete).
* E.g.

@Modifying

@Query("delete from Student where firstName = :firstName")

void deleteStudentsByFirstName(@Param("firstName") String firstName);

* So if you try to perform a non-select operation without using @Modifying annotation, you will get an exception.
* Side Note –
  + If you are writing a test, you need to mark that test as **@Transactional** since we are performing a DML operation.
  + A Junit test will rollback the transaction even if you annotate it with @Transactional. This is the default behavior for Tests.
  + To avoid this default behavior of rollback in Tests, we need to annotate the test method with **@Rollback(false)** annotation. Use it with caution.

## Paging and Sorting with JQPL

* Using paging and sorting with JPQL is no different.

It is described here – [Paging and Sorting](#_Paging_and_Sorting)

* E.g.

@Query("from Student")

List<Student> findAllStudents(**Pageable** pageable);

@Test

public void testFindAllStudents() {

System.out.println(repository.findAllStudents(new **PageRequest**(0, 5, Direction.DESC, "id")));

}

# Native SQL Query

## Introduction

* Native SQL queries means the queries which database directly understands.
* When our JPQL query is **complex** involving multiple joins, etc. or if we have a **database** **expert** who is really good at writing Native SQL queries, then we go for Native SQL queries.
* In addition to the DML and DQL, native queries also support DDL queries. DDL queries are NOT supported in JPQL.
* Link JPQL, native queries also support named parameters.
* If you can’t do something using JPQL or if it is getting too complex, use Native SQL queries as the backup.

## Using Native Queries

* Mark the method in your repository interface with @Query annotation, pass your native query to its value attribute and also set **nativeQuery** parameter to **true**. If nativeQuery is not set to true, Spring data will assume it as a JPQL.
* You can also use alias for your native query table names.
* E.g.   
  Assuming Entity class name as Student and underlying table name as student\_t.  
  @Query(**value** = "select \* from **student\_t**", **nativeQuery = true**)

List<Student> findAllStudentNQ();

* E.g. Native Query **with Named Parameters**

@Query(value = "select \* from student where fname=**:firstName**", **nativeQuery = true**)

List<Student> findByFirstNQ**(@Param("firstName")**String firstName);

# Inheritance Mapping

# Tips and Tricks

* Spring tool suite (STS) is a very powerful IDE that helps us build spring based projects very easily. It is very similar to eclipse with support for special support for spring based projects.
* spring-boot-starter-parent is a **BOM**, which is a special type of pom. BOM stand for **Bill of Materials** within which all the versions of various libraries required for our projects are defined.
* @RunWith(SpringRunner.class) – with this we tell Spring Boot to use SpringRunner instead of default JUnitRunner class to run our Test classes.
* @SpringBootTest - This annotation tells spring boot to search for a class that is marked with spring Boot application and use that class to create a spring application context, a container with all the beans in that application so that we can start testing those beans in that application.
* Typically, your repository interface extends Repository, CrudRepository, or PagingAndSortingRepository. Alternatively, if you do not want to extend Spring Data interfaces, you can also annotate your repository interface with @RepositoryDefinition. Extending CrudRepository exposes a complete set of methods to manipulate your entities. If you prefer to be selective about the methods being exposed, copy the methods you want to expose from CrudRepository into your domain repository.