Spring Data JPA

SPRING DATA JPA

Spring Data JPA (Java Persistence API), it is part of the Spring Data family, aims to provide the JPA based repositories that aims to simplify the implementation of data access layer using JPA.

Too much boiler plate code has to be written to execute simple queries as well as perform pagination and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that's needed. As a developer you write your repository interfaces, including custom finder methods and Spring will provide the implementation automatically.

Features

1. Create and support repositories created with Spring and JPA.
2. Support QueryDSL and JPA queries.
3. Audit of domain classes.
4. Support for batch loading, sorting, dynamical queries.
5. Supports XML mapping for entities.
6. Reduce code size for generic CRUD operations by using CrudRepository.

Project Work:

1. In Spring boot document we have learned how to create a Maven Project and developed an Employee Management application using Spring Boot RESTful web service.
2. In this, document we are going to develop an Employee Management application using Spring Data JPA.

Dependency and It’s Use

Spring Boot Starter Data JPA dependency is used to provide database access to Spring and JPA. Its code is shown below:

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

MySQL Connector Java dependency is used to store the details in the database. Its code is shown below.

```
<dependency>
        <groupId>mysql</groupId>
       <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Project Work:

1. We have already created a Maven project in STS for Employee Management.
2. Now add the above two dependencies in the pom.xml.

Properties File

Properties files are used to keep 'N' number of properties in a single file to run the application in a different environment. The application.properties file contains the configuration that is used to configure our example application. We can create this properties file by following these steps:

- Configure the database connection to our application. We need to configure the name of the MySQL driver class, the MySQL url, the username name of the database user and the password of the database user.
- Configure the used database dialect. Ensure the Hibernate creates the database when our application is started.

The application.properties file looks like:

```
# Set here configurations for the database connection

#Set the server port to run the application
server.port=8080

# Hibernate ddl auto (create, create-drop, update): with "create" the database
# schema will be automatically created accordingly to java entities found in
# the project
spring.jpa.hibernate.ddl-auto=create

# Connection url for the database
spring.datasource.url=jdbc:mysql://localhost:3306/employee

# Username and password
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
```

Project Work

1. We have an existing application.properties file.
2. Add server.port and the other properties which we require to establish the connection with database.

POJO Classes

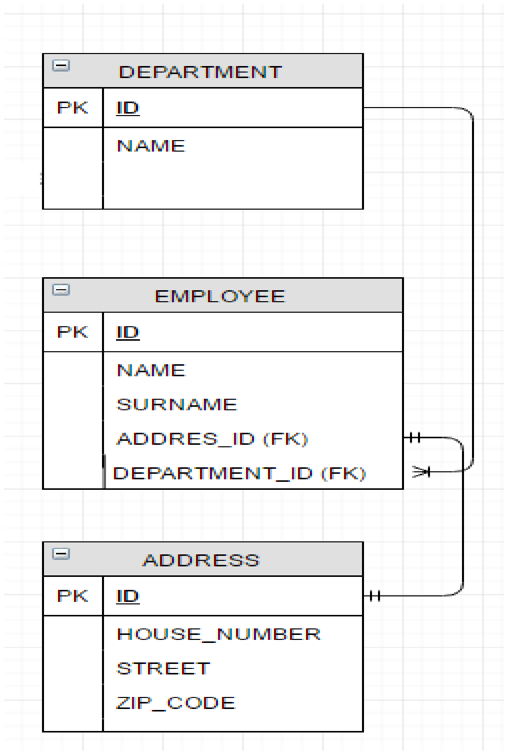
Java POJO classes can be made entity using @Entity annotation.

Requirements for Entity Classes:

- The class must be annotated with the javax.persistence.Entity annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.

Note: As there are two ways to configure entities, either in XML or with annotations, to keep contents simple and manageable, we are focusing only on annotations to configure entity classes.

db_schema



Entity Annotations

```
@Entity
@Table(name="employee_details")
@Getter
@Setter
public class Employee {
            @Id
            @GeneratedValue(strategy = GenerationType.AUTO)
            @Column(name = "ID", updatable = false, nullable = false)
            private int empId;

           @Column(name = "NAME", nullable = false)
            private String name;

            @Column(name = "SURNAME", nullable = false)
             private String surname;

            @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
            @JoinColumn(name = "addressId")
             private Address address;

             @ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
             @JoinColumn(name = "departmentId")
             private Department department;
}
```

Snippet: **1**

@Entity: It specifies that the class is an entity.

@Table: It specifies the table in the database with which this entity is mapped. In the above example the data will be stores in the “employee_details” table. Name attribute of @Table annotation is used to specify the table name.

@Id: This annotation specifies the primary key of the entity. It will automatically determine the most appropriate primary key generation strategy to use – you can override this by applying the @GeneratedValue annotation. This takes a pair of attributes: strategy and generator.

@GeneratedValue: This annotation specifies the generation strategies for the values of primary keys. The strategy attribute must be a value from the GenerationType enumeration, which defines four types of strategy constants.

1. **AUTO: (Default)** JPA decides which generator type to use, based on the database's support for primary key generation.
2. **IDENTITY:** The database is responsible for determining and assigning the next primary key.
3. **SEQUENCE:** Some database support a SEQUENCE column type.
4. **TABLE:** This type keeps a separate table with the primary key values.

Note: Identity strategy depends upon database.

@Column: This annotation is used to specify column or attribute for persistence property.

@JoinColumn: This annotation is used to specify an entity association or entity collection. This is used in many- to-one and one-to-many associations.

@OneToOne: This annotation is used to define a one-to-one relationship between the join Tables.

@ManyToOne: This annotation is used to define a many-to-one relationship between the join Tables.

Project Work

1. Create a POJO class and add the required annotations as mentioned in the above snippet1.
2. As you already know how to create a controller and service classes.
3. Now, let's create a repository interface.

Repositories

Spring data provides the abstract repositories that are implemented at run-time by the spring container and perform the CRUD operations.

The following are the three base interfaces defined in the **spring data commons** project.

Repository:

It is the central interface in the spring data repository abstraction. This is a marker interface. If you are extending this interface, you have to declare your own methods and the implementations will be provided by the spring run-time. For this interface also we have to pass two parameters: type of the entity and type of the entity's id field. This is the super interface for CrudRepository.

Look at example:

```
public interface EmployeeRepository extends Repository<Employee, Integer>{
}
```

CRUDRepository:

CrudRepository provides methods for the CRUD operations. This interface extends the Repository interface. When you define CrudRepository, you have to pass the two parameters: type of the entity and type of the entity's id field**.** If you are extending the Crud Repository, there is no need for implementing your own methods. Just extend this interface and leave it as blank. Required implementations are provide at run time.

Look at example:

```
public interface EmployeeRepository extends CrudRepository<Employee, Integer>{
}
```

JpaRepository:

JpaRepository which extends PagingAndSortingRepository and in turn the CrudRepository. It provides JPA related methods such as flushing the persistence context and delete records in a batch.

Look at example:

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer>{
}
```

PagingAndSortingRepository:

The PagingAndSortingRepository provides additional methods to retrieve entities using pagination and sorting. This is extension of CrudRepository.

Pagination:

Pagination is often helpful when we have a large dataset and we want to present it to the user in smaller chunks.

Once we have our repository extending from PagingAndSortingRepository, we just need to:

1. Create or obtain a PageRequest object, which is an implementation of the Pageable interface.
2. Create a **PageRequest** Object which is an implementation of **Pageable** interface.
3. This PageRequest Object takes page number, limit of the page (page size) and sorts direction and sort field.
4. By passing requested page number and page limit you can get the data for this page.

Look at example:

```
public interface EmployeeRepository extends PagingAndSortingRepository<Employee, Integer>
{
}
```

Difference between JpaRepository and CrudRepository

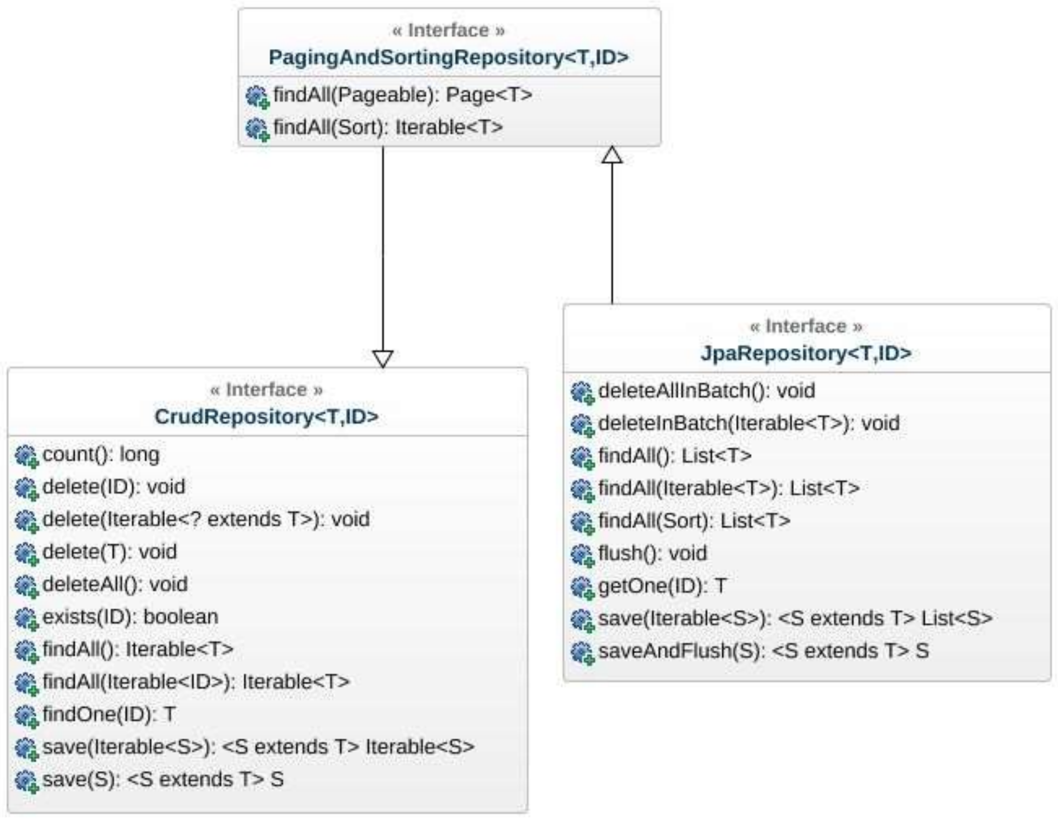
Both, JpaRepository and CrudRepository are base interfaces in Spring Data. Application developer has to choose any of the base interfaces for your own repository. It has two purposes,

- One is to allow spring data to create proxy instance for your repository interface.
- Second one is to inherit the maximum default functionality from the base interfaces without declaring your own methods.

When we look at both interfaces, the clear differences are:

- CrudRepository is part of Spring Data Commons project and declared under the package org.springframework.data.repository. Where as JpaRepository is part of store specific implementation and declared under the package org.springframework.data.jpa.repository.
- CrudRepository extends from Repository interface. Where as JpaRepository extends from PagingAndSortingRepository which in turn extends from the CrudRepository.
- It is general recommendation that not to use the store specific base interfaces as they expose the underlying persistence technology and tighten the coupling between them and the repository. It is always good idea to choose any of the generic base interfaces like CrudRepository or PagingAndSortingRepository.

Repository class diagram is shown below:


« Interface »
PagingAndSortingRepository<T,ID>
findAll(Pageable): Page<T>
findAll(Sort): Iterable<T>
« Interface »
CrudRepository<T,ID>
count(): long
delete(ID): void
delete(Iterable<? extends T>): void
delete(T): void
deleteAll(): void
exists(ID): boolean
findAll(): Iterable<T>
findAll(Iterable<ID>): Iterable<T>
findOne(ID): T
save(Iterable<S>): <S extends T> Iterable<S>
save(S): <S extends T> S
« Interface »
JpaRepository<T,ID>
deleteAllInBatch(): void
deleteInBatch(Iterable<T>): void
findAll(): List<T>
findAll(Iterable<T>): List<T>
findAll(Sort): List<T>
flush(): void
getOne(ID): T
save(Iterable<S>): <S extends T> List<S>
saveAndFlush(S): <S extends T> S

Project Work

1. Create a repository interface under the repositories package.
2. Use any one of the above mentioned repositories to fetch the information from the database.
3. Now let’s create Query Methods and @Query annotation in the repository interface.
4. And @NamedQuery annotation in the POJO class.

Query Methods

- Query methods are used for finding the information from database and that are declared in the repository interface.
- Query methods can return one result or more than one result.
- This also supports passing parameter to database queries by sending through parameter binding or named parameters.
- If you want to implement query methods, implement Repository interface and write your own methods.

Here is a simple example of how to write Query Methods:

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer>{
        List<Employee> findByempId(int empId);
}
```

Snippet: **2**

@Query Annotation

- When you cannot use the query methods to perform database operations, @Query could be used to write the more flexible query to fetch data.
- @Query annotation supports both JPQL and native SQL queries.
- By default, @Query annotation will be using JPQL to execute the queries. If you try to use the normal SQL queries, you would get the query syntax error exceptions.
- If you want to write native SQL queries, annotate the query method with the @Query annotation and specify the invoked query by setting it as the value of the @Query annotation’s value attribute. Set the nativeQuery flag to true.

The source code of repository interface looks as follow:

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer>{
        @Query(value="SELECT * FROM Employee e WHERE e.name = ?1", nativeQuery =
true)
        List<Employee> findByName();
}
```

Snippet: **3**

@NamedQuery Annotation

- Named queries are a way of organizing your static queries in a manner that is more readable, maintainable.
- The named queries are defined in the single place at entity class itself with each query has its unique name. That is the reason it is known as named query.
- @NamedQuery annotation can be applied only at the class level.
- Note that named queries have the global scope (can be accessed throughout the persistence unit), so every query should be uniquely identified even if you define queries for different entities.

The example code is given below:

```
@Entity
@NamedQuery(name = "Employee.findById", query = "SELECT e from Employee e
where e.id=?1")
@Table(name="employee_details")
public class Employee {
    }
```

Snippet: **4**

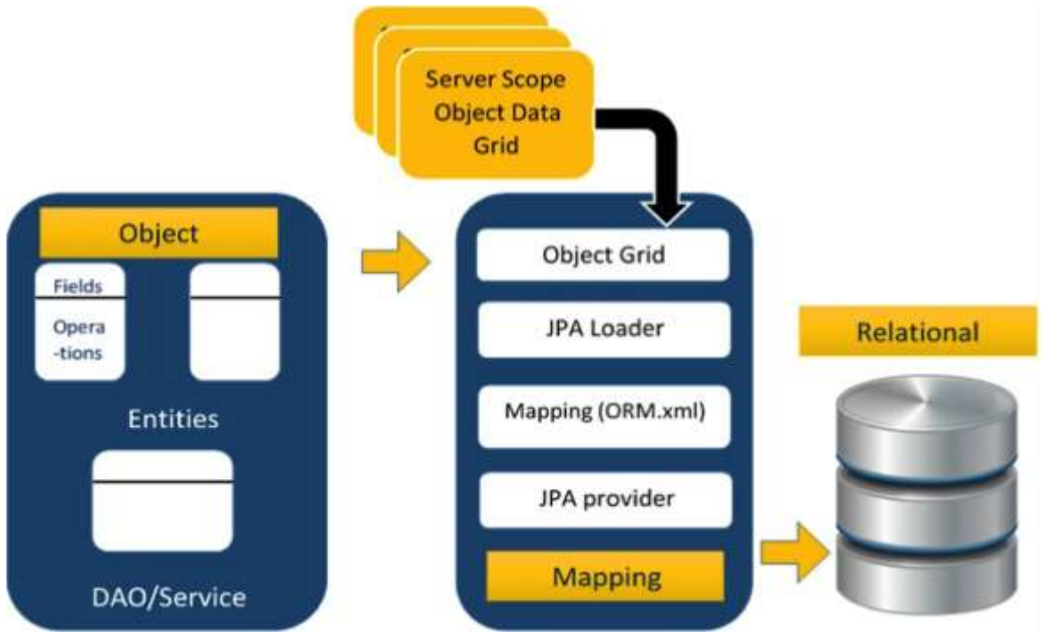
Project Work

1. Create a Query Methods to get the information from the database as mentioned in the snippet 2.
2. We can use @Query annotation to write the query as mentioned in the snippet 3.
3. @NamedQuery annotation is written in POJO class as mentioned in the snippet 4.

What is ORM?

ORM stands for **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc. The main feature of ORM is mapping or binding an object to its data in the database.

ORM Architecture



The above architecture explains how object data is stored into relational database in three phases.

Phase1:

The first phase, named as the **Object data** phase contains POJO classes, service interfaces and classes. It is the main business component layer, which has business logic operations and attributes.

Phase2:

The second phase named as **mapping** or **persistence** phase which contains JPA provider, mapping file (ORM.xml), JPA Loader, and Object Grid.

Phase3:

The third phase is the Relational data phase. It contains the relational data which is logically connected to the business component.

For more information about each phase

Reference: https://www.tutorialspoint.com/jpa/jpa_orm_components.htm

Hibernate Framework

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

Reference: https://www.tutorialspoint.com/hibernate/hibernate_architecture.htm

Persistence Context

Persistence context defines a scope under which particular entity instances are created, persisted and removed.

Error occurred while running the code

An "org.springframework.http.converter.HttpMessageNotWritableException" was thrown while I am trying to get details of an Employee using GET call. This exception was occurred as the bean classes were empty. To avoid such exception disable SerializationFeature.fail on empty beans.

Include the below property in application.properties file

```
spring.jackson.serialization.fail-on-empty-beans=false
```

What is PostgreSQL:

PostgreSQL (pronounced as **post-gress-Q-L**) is an open source relational database management system (DBMS) developed by a worldwide team of volunteers. PostgreSQL is not controlled by any corporation or other private entity and the source code is available free of charge.

Difference between postgresql and mysql:

The governance model around the **MySQL** and **PostgreSQL** is one of the more significant differences between the two database technologies. **MySQL** is controlled by Oracle, whereas **Postgres** is available under an open-source license from the **PostgreSQL** Global Development Group.

CONNECT TO POSTGRESQL FROM THE COMMAND LINE

To connect to PostgreSQL from the command line:

1. Log in to your A2 Hosting account using SSH.
2. At the command line, type the following command. Replace DBNAME with the name of the database, and USERNAME with the database username
3. At the Password prompt, type the database user's password. When you type the correct password, the psql prompt appears.
4. After you access a PostgreSQL database, you can run SQL queries and more. Here are some common psql commands:

Dependency needed:

Postgresql dependency is used to store the details in the database. Its code is shown below.

```
<dependency>
      <groupId>postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>9.1-901-1.jdbc4</version>
</dependency>
```

Properties File:

```
# Set here configurations for the database connection

Set the server port to run the application
server.port=8080
#company-structure-spring-security-oauth2-authorities

# Connection url for the database
spring.datasource.url=jdbc:postgresql://localhost:5433/postgres

# Username and password
spring.datasource.username=postgres
spring.datasource.password=root

# Hibernate ddl auto (create, create-drop, update): with "create" the database
# schema will be automatically created accordingly to java entities found in
# the project
spring.jpa.hibernate.ddl-auto=create

# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.id.new_generator_mappings=true
spring.jackson.serialization.fail-on-empty-beans=false
```

Spring Profiles:

Spring Profiles provides a powerful and easy way to control code and configuration based on the environment. Using Spring Profiles it’s possible to segregate parts of our application and make it only available in certain environments.

How Do we Maintain Profiles?

- We make properties files for each environment and set the profile in the application accordingly, so it will pick the respective properties file.
- Now, we will see how to configure different databases at runtime based on the specific environment by their respective profiles.

As the DB connection is better to be kept in a property file, it remains external to an application and can be changed. But, Spring Boot — by default — provides just one property file (application.properties). So, how will we segregate the properties based on the environment?

The solution would be to create more property files and add the "profile" name as the suffix and configure Spring Boot to pick the appropriate properties based on the profile.

Then, we need to create two application.properties:

1. application-mysql.properties
2. application-postgresql.properties

The base application.properties file is shown below

```
server.port=8080
spring.profiles.active=postgresql
```

To connect to mysql database the application-mysql.properties file is shown below

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/employee
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
spring.jackson.serialization.fail-on-empty-beans=false
```

To connect to postgresql database the application-postgresql.properties file is shown below

```
spring.datasource.url=jdbc:postgresql://localhost:5433/postgres
spring.datasource.username=postgres
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.id.new_generator_mappings=true
spring.jackson.serialization.fail-on-empty-beans=false
```

We will set the spring.profiles.active=postgresql or mysql in application.properties file to let the system know that it is postgresql or mysql.

Project Work:

1. Create application.properties as mentioned.
2. To run the application go to Run $\rightarrow$ RunConfigurations $\rightarrow$ Environment $\rightarrow$ New $\rightarrow$ Name $\rightarrow$ Value and then click on Run.

Note: The Name should be SPRING_PROFILE_ACTIVE and the Value may be either mysql or postgresql that you want to run.