**Section-End Project - Lesson 2**

# **Using the JPA Repository**

This lab walks you through the process of building an application that uses Spring Data JPA with the JPARepository interface to store and retrieve data in a relational database.

## 1. Introduction

In this lab, we shall see how to leverage the powerful [Spring Data JPA](https://projects.spring.io/spring-data-jpa/" \t "https://www.javacodegeeks.com/2018/03/_blank) APIs to interact with the database, in-memory [H2 database](http://www.h2database.com/html/main.html" \t "https://www.javacodegeeks.com/2018/03/_blank).

Spring Data JPA offers a set of very powerful and highly-abstracted interfaces which are used to interact with any underlying database. Databases can be MySQL, MongoDB, Elasticsearch or any other supported database. Other advantages for Spring Data JPA include:

* Support to build extended repositories based on JPA Convention
* In-built pagination support and dynamic query execution
* Support for XML based entity mapping

We will make use of H2 in-memory database. The choice for the database should not affect the Spring Data definitions we will construct as this is the main advantage Spring Data JPA offers. It enables us to completely separate the Database queries from the application logic.

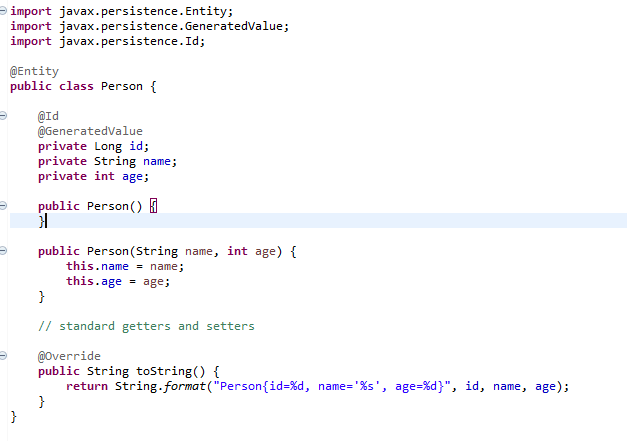
## Maven Dependencies

To start with, we need to add appropriate Maven dependencies to our project. Create a Spring Boot project as in Lab 1 with the same dependencies.

Note that we have should have added the H2 database dependency here as well with its scope as runtime as the H2 data is washed away as soon as the application has stopped.

## Defining the Model

We will start by adding a very simple model in our project, a Person. Its definition will be very standard, like:



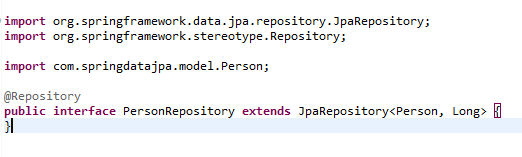
Getters ans Setters are necessary to be made as Jackson uses them during [Serialization and Deserialization](https://www.javacodegeeks.com/2013/03/serialization-in-java.html" \t "https://www.javacodegeeks.com/2018/03/_blank) of an Object.

The @Entity annotation marks this POJO as an object which will be managed by the Spring Data APIs and its fields will be treated as table columns (unless marked [transient](https://examples.javacodegeeks.com/core-java/transient-variables-in-java/" \t "https://www.javacodegeeks.com/2018/03/_blank)).

Finally, we added a custom implementation for the toString() method so that we can print related data when we test our application.

## Defining JPA Repository

JPA provides us with a very simple way of defining a JPA Repository interface.

Before getting to know how to define a JPA Repository, we need to remember that each JPA interface is only made to interact with a single Entity of Database Table when JPA-related functionality is leveraged. We will understand this deeply if we have a look at the interface definition:

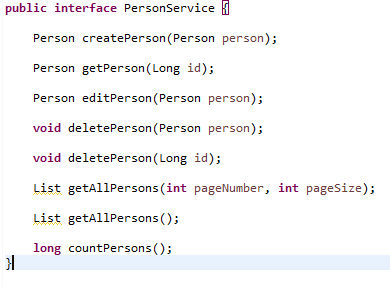
Although above interface definition is empty, we still have some points which we need to understand:

* @Repository annotation marks this interface as a Spring Bean which is initialised on application startup. With this annotation, Spring takes care of managing exceptions database interaction throws gracefuly.
* We used Person as a parameter to signify that this JPA interface will manage the Person Entity
* Finally, we also passed the data type Long as a parameter. This signifies that the Person Entity contains a unique identifier which is of the type Long

## 5. Making the Service interface

In this section, we will define a service interface which will act as a contract for the implementation and represnet all the actions our Service must support. These actions will be related to making a new user and getting information related to the objects in database.

Here is the contract definition we will be using:



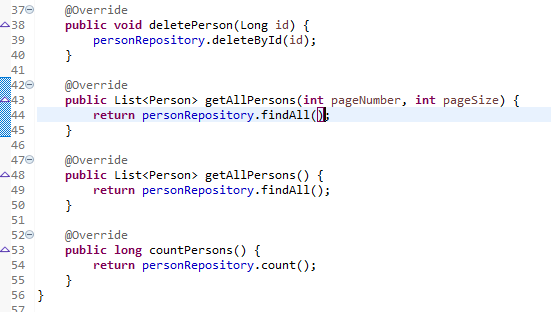
## Providing Service implementation

We will use the above interface definition to provide its implementation so that we can perform CRUD operations related to the Person Entity we defined earlier. We will do it here:

It is positively surprising that all the method implementations are only one line of code. This shows the level of abstraction JPA Repositories provide to us.

Most of the operations above are simple to understand. Main thing to notice is that we never provided any methods in the Repository we made like getAllPersons() etc! Then how did these methods appeared altogether? The answer, again, lies in the abstraction JPA Repositories provide to us. All of the methods like findAll(), delete(), save(...) etc. are in-built into the JpaRepository we extended in our repository interface definition.





## Using the CommandLineRunner

To test all the code we have written till now, along with the database interaction part, we will be using the CommandLineRunner in our main class of our Spring Boot application. A CommandLineRunner runs right before the main() method for the Spring Boot application is called and so, it is an ideal space to perform any init steps or testing code.

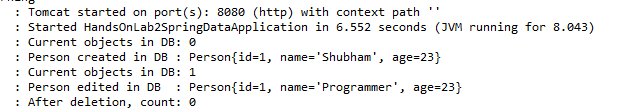
To test the application, we will use a service bean to perform database operations in our class:



In above code, we just made simple calls to some important methods we created in our service like creating some data and accessing it later method calls.

## Running the Application

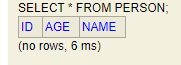
The initial output should be as follows:



## H2 Database Settings

1. Now, enable the H2 console by setting the appropriate property in application.properties.
2. Also,set the property required to access the H2 console at <http://localhost:8080/h2.>
3. Change the database name to springdatajpa.

Login to the console , double click on the Customer table and run the query that appears. Your output should be as follows:



Now, insert more records into the database after the call to deletePerson in the CommandLineRunner and see the output.

Try out the other calls available in your Service class.

In this Lab Exercise, you have written a simple application that uses Spring Data JPA to save objects to and fetch them from a databasewhile leveraging the JPARepository interface.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*