**Hands On Lab 2**

# **Accessing Data with JPA**

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

## What You Will build

You will build an application that stores Customer POJOs (Plain Old Java Objects) in a memory-based database.

## Starting with Spring Initializr

You can use this [pre-initialized project](https://start.spring.io/" \l "!type=maven-project&language=java&platformVersion=2.5.5&packaging=jar&jvmVersion=11&groupId=com.example&artifactId=accessing-data-jpa&name=accessing-data-jpa&description=Demo project for Spring Boot&packageName=com.example.accessing-data-jpa&dependencies=data-jpa,h2) and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

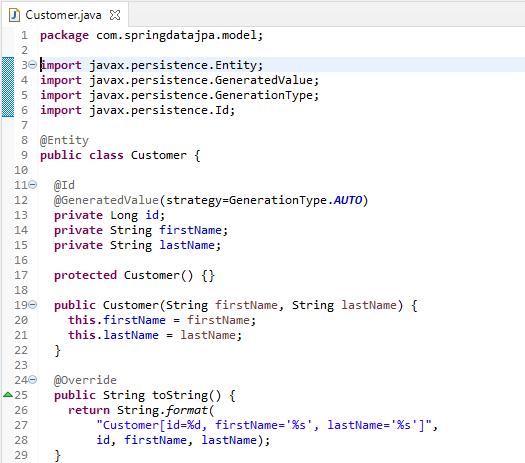
To manually initialize the project:

1. Navigate to [https://start.spring.io](https://start.spring.io/). or use your IDE. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Maven and the language you want to use. This lab assumes that you chose Java.
3. Click **Dependencies** and select **Spring Web, Spring Data JPA** and then **H2 Database**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

Create appropriate packages for your classes.

## Define a Simple Entity

In this example, you store Customer objects, each annotated as a JPA entity. Add getters and setters.



Here you have a Customer class with three attributes: id, firstName, and lastName. You also have two constructors. The default constructor exists only for the sake of JPA. You do not use it directly, so it is designated as protected. The other constructor is the one you use to create instances of Customer to be saved to the database.

The Customer class is annotated with @Entity, indicating that it is a JPA entity. (Because no @Table annotation exists, it is assumed that this entity is mapped to a table named Customer.)

The Customer object’s id property is annotated with @Id so that JPA recognizes it as the object’s ID. The id property is also annotated with @GeneratedValue to indicate that the ID should be generated automatically.

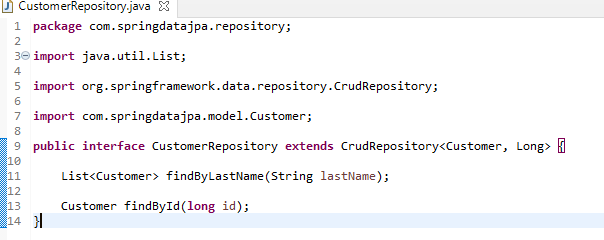
The other two properties, firstName and lastName, are left unannotated. It is assumed that they are mapped to columns that share the same names as the properties themselves.

The convenient toString() method print outs the customer’s properties.

## Create Simple Queries

Spring Data JPA focuses on using JPA to store data in a relational database. Its most compelling feature is the ability to create repository implementations automatically, at runtime, from a repository interface.

To see how this works, create a repository interface that works with Customer entities as the following listing shows:



CustomerRepository extends the CrudRepository interface. The type of entity and ID that it works with, Customer and Long, are specified in the generic parameters on CrudRepository. By extending CrudRepository, CustomerRepository inherits several methods for working with Customer persistence, including methods for saving, deleting, and finding Customer entities.

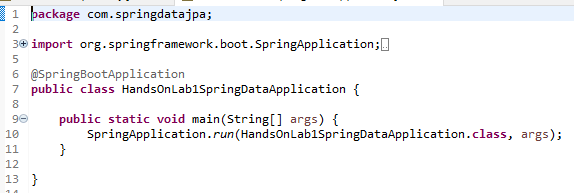
Spring Data JPA also lets you define other query methods by declaring their method signature. For example, CustomerRepository includes the findByLastName() method.

In a typical Java application, you might expect to write a class that implements CustomerRepository. However, that is what makes Spring Data JPA so powerful: You need not write an implementation of the repository interface. Spring Data JPA creates an implementation when you run the application.

Now you can wire up this example and see what it looks like!

## Create an Application Class

Spring Initializr creates a simple class for the application. The following listing shows the class that Initializr created :



@SpringBootApplication is a convenience annotation that adds all of the following:

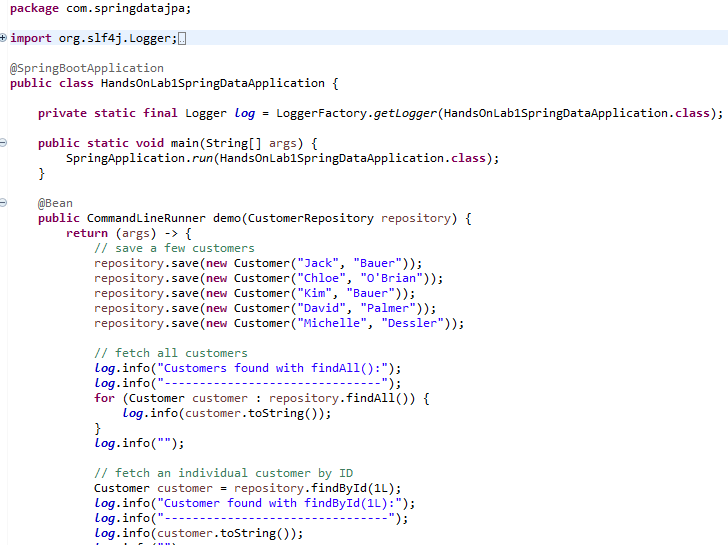
@Configuration: Tags the class as a source of bean definitions for the application context.

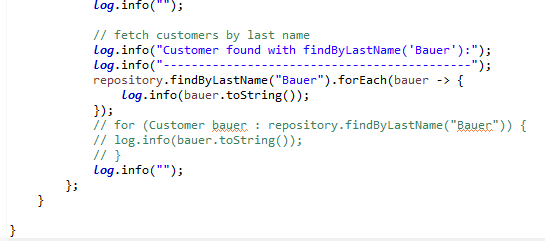
@EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.

@ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application. Did you notice that there was not a single line of XML? There is no web.xml file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

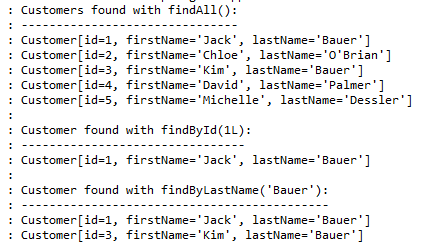
Now you need to modify the simple class that the Initializr created for you. To get output (to the console, in this example), you need to set up a logger. Then you need to set up some data and use it to generate output. The following listing shows the finished HandsOnLab1SpringDataApplication class:





The HandsOnLab1SpringDataApplication class includes a demo() method that puts the CustomerRepository through a few tests. First, it fetches the CustomerRepository from the Spring application context. Then it saves a handful of Customer objects, demonstrating the save() method and setting up some data to work with. Next, it calls findAll() to fetch all Customer objects from the database. Then it calls findById() to fetch a single Customer by its ID. Finally, it calls findByLastName() to find all customers whose last name is "Bauer". The demo() method returns a CommandLineRunner bean that automatically runs the code when the application launches.

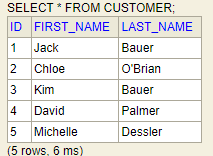
When you run your application, you should see output similar to the following:



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 datajpa.

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



Congratulations! You have written a simple application that uses Spring Data JPA to save objects to and fetch them from a database, all without writing a concrete repository implementation.

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