**FSD - Final Project - Group I**

**Spring Boot MVC RESTful API with Spring Data JPA**

[Overview of Spring Boot REST + JPA Evaluation Project 1](#_Toc28731)

[Create a Spring initializr project 2](#_Toc4736)

[The Spring Boot RESTful Application 3](#_Toc14760)

[Project Structure 4](#_Toc21945)

[Creating the Customer Controller 4](#_Toc23198)

[Time to build out our services. 8](#_Toc22418)

[Setup the Repository 9](#_Toc14583)

[Add the Domain Object 9](#_Toc23158)

[Configuring application.properties 10](#_Toc31137)

[Running the application for the first time 12](#_Toc21273)

[Summary 15](#_Toc7159)

Spring Boot makes developing RESTful services ridiculously easy .

For your Sprint 1 evaluation, you will build a Spring Boot Rest CRUD API project with Maven that use Spring Data JPA to interact with H2 database.

You will need to demonstrate a knowledge of:

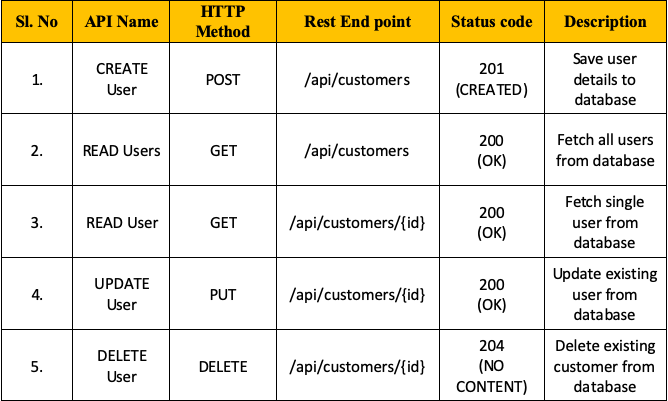
* How to configure Spring Data, JPA, Hibernate to work with an H2 Database
* How to define Data Models and Repository interfaces
* How to create a Spring Rest Controller to process HTTP requests
* How to use Spring Data JPA to interact with an H2 Database

## **Overview of Spring Boot REST + JPA Evaluation Project**

Build Spring Boot Rest Apis using Spring Data JPA with H2 Database for a Customer Relationship Management (CRM) API, that has the following features –

* Create customer
* Read single customer
* Read all customers
* Update customer
* Delete customer

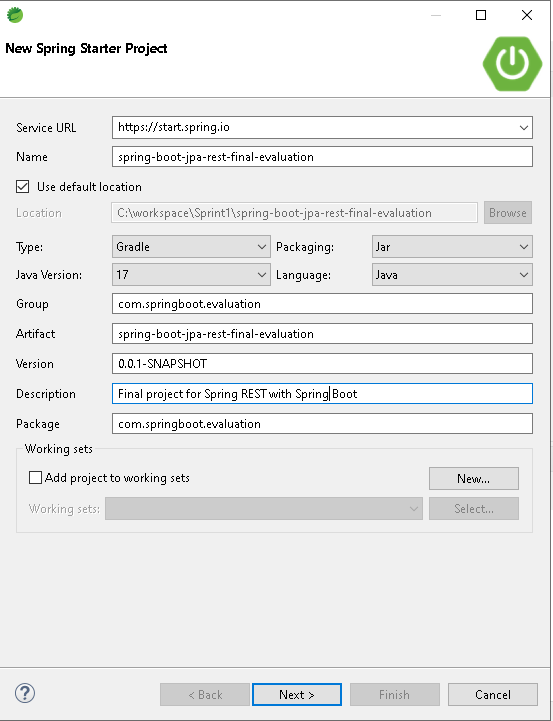
These are the APIs that you need to provide:



– Make CRUD operations & finder methods with Spring Data JPA’s JpaRepository.  
– The database will be H2 Database (in memory or on disk) by configuring project dependency & datasource.

## **Create a Spring initializr project**

Create a Spring initializr project with the following entries:



Choose the following dependencies :

– Web

– H2

- Data JPA

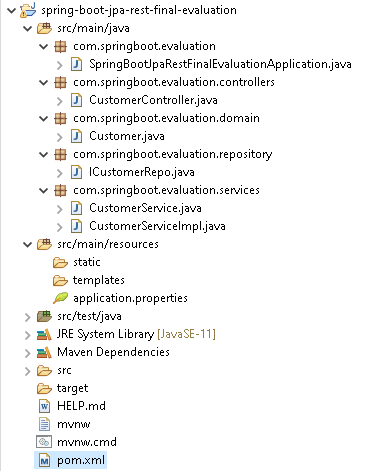
## **The Spring Boot RESTful Application**

Our application implements a set of REST endpoints to manage customers. We have a Customer JPA entity and a repository named ICustomerRepo that extends JpaRepository to perform CRUD operations on products against an in-memory H2 database.

The service layer is composed of a CustomerService interface and a CustomerServiceImpl implementation class.

## **Project Structure**

The project structure is as below:



## **Creating the Customer Controller**

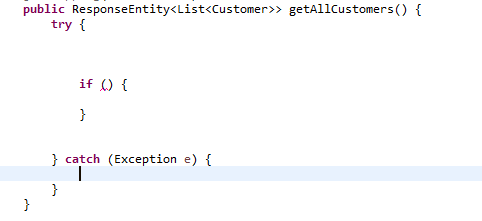
The controller of the application, CustomerController, defines the REST API endpoints.

Update the CustomerController as follows:

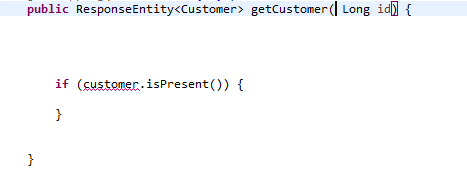
* At the class level, add the required annotation to designate this class as a Rest Controller. Remember that this annotation works as a convenience annotation to annotate the class with the @Controller and @ResponseBody.
* At the class level, add the required annotation for mapping web requests from "/api" onto methods in this class
* Autowire the CustomerService.
* Add the appropriate @RequestMapping annotations to each method so as to corresopond to the operations displayed in the table on page 2.

**Note**: The  class-level annotation maps requests to /api onto the CustomerController class. The method-level annotations map web requests to the handler methods of the controller.

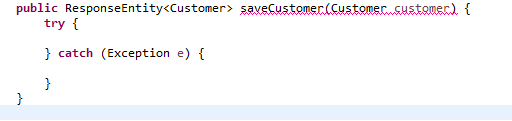
Implement the following for each handler method. Refer to the table on page 2 for the appropriate URLs. Note that ResponseEntity has a number of overloaded constructors. For code examples of the usage of ResponseEntity, refer to *Hands On Demos - Day 21.docx*



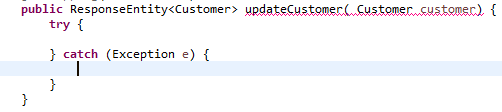
* Map the appropriate HTTP request onto this method. Add the appropriate method level annotation.
* In the try block, retrieve the list of customers by calling the appropriate method on the Customer Service.
* If the list is empty or the list size is zero, return a ResponseEntity with an HttpStatus thaat translates to “204 No Content”.
* Else return a ResponseEntity that contains the list and an HttpStatus that translates to “200 OK”.
* In the catch block, return a ResponseEntity with headers and a status code. Set null for the headers and an HttpStatus thaat translates to “500 Internal Server Error.”



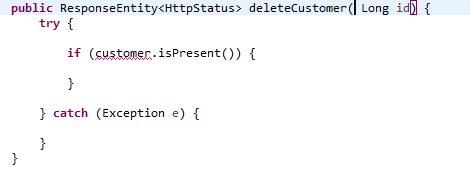
* Map the appropriate HTTP request onto this method. Add the appropriate method level annotation.
* In the try block, retrieve the customer by calling the appropriate method on the Customer Service.
* If the customer exists, return a ResponseEntity that contains the customer and an HttpStatus that translates to “200 OK”.
* Else return a ResponseEntity that contains an HttpStatus that translates to “404 Not Found.”
* Add an annotation to map the method parameter “id” to the corresponding URI parameter.



* Map the appropriate HTTP request onto this method. Add the appropriate method level annotation.
* In the try block, persist the customer object by calling the appropriate method on the Customer Service. This should return an updated Customer object with the id field.
* Return a ResponseEntity that contains the updated customer and an HttpStatus that translates to “201 CREATED”.
* In the catch block, return a ResponseEntity with an HttpStatus that translates to “500 Internal Server Error.”



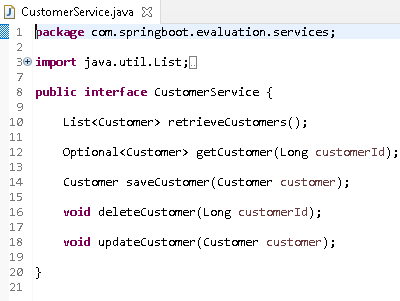
* Map the appropriate HTTP request onto this method. Add the appropriate method level annotation.
* You will need to attach an annotation to the “Customer” parameter. This annotation indicates a method parameter should be bound to the body of the web request.
* In the try block, update the customer object by calling the appropriate method on the Customer Service. This should return an updated Customer object.
* Return a ResponseEntity that contains the updated customer and an HttpStatus that translates to “200 OK”.
* In the catch block, return a ResponseEntity with an HttpStatus that translates to “500 Internal Server Error.”



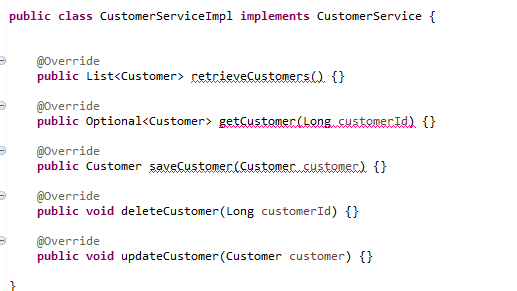
* Map the appropriate HTTP request onto this method. Add the appropriate method level annotation.
* In the try block, check if the customer exists by calling the appropriate method on the Customer Service.
* If the customer is found, delete the customer by calling the appropriate method on the Customer Service.
* If the customer is not found, return a ResponseEntity that contains an HttpStatus that translates to “204 No Content.”.
* In the catch block, return a ResponseEntity with headers and a status code. Set null for the headers and an HttpStatus thaat translates to “500 Internal Server Error.”
* Add an annotation to map the method parameter “id” to the corresponding URI parameter

## **Time to build out our services.**

Below is the code for the CustomerService iterface.

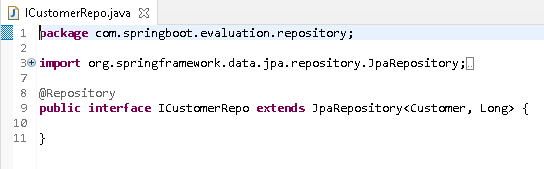


* Implement the interface methods in the CustomerServiceImpl class by calling the Customer Repository methods. This translates into methods that are called on the JpaRepository that the Customer Repository extends.
* Autowire the Customer Repository.
* Annotate the class appropriately to indicate that it is a “service”.



## **Setup the Repository**

The Repository tier is simple. We will leverage the functionality provided by Spring Data JPA.



* For this to function as a Repository, you need a class level annotation. Add the appropriate annotation.

## **Add the Domain Object**

Create a class Customer.java in the appropriate package.

* Add the following fields.

**private** Long id;

**private** String name;

**private** Long age;

**private** String location;

**private** Date createdAt;

**private** Date updatedAt;

* Annotate Customer.java at the class level to designate it as a persistent class with a database table name of “tbl\_customer”.
* Annotate the id variable appropriately to designate it as the primary key. The Generation Type can be set to IDENTITY.
* The variable *createdAt* should have an annotation which marks the property as the creation timestamp of the containing entity ie; the timestamp when the Customer instance was created.
* *createdAt* should also have the Column annotation. Set the database column name as “created\_at”. Provide attributes so that null values are not sent to the database and column is ***not*** included in SQL UPDATE statements generated by the persistence provider.
* The variable *updatedAt* should have an annotation which marks the property as the update timestamp of the containing entity ie; the timestamp when the Customer instance was updated. The property value will be set to the current VM date whenever the owning entity is updated.
* Set the database column name as “updated\_at”.
* Add the appropriate Getters and Setters.

## **Configuring application.properties**

In application.properties, configure the H2 database:

* Enable the H2 console.
* Configure the console path as “/h2”.
* Set “jdbc:h2:mem:crm” as the URL of the data source.
* Add a property to show the SQL generated.
* Set the dialect to Hibernate.
* The property for “ddl-auto” should be set to “update”.

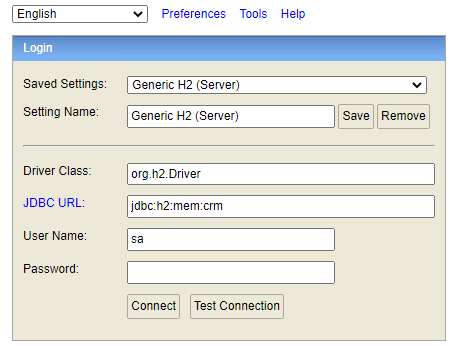
The default username for the h2 database is sa and there is no password for the database. Make sure to add the database name as crm

**Note:** We configure the H2 database with Spring boot to create and use an in-memory database in runtime, generally for unit testing or POC purposes. Remember an in-memory database is created/initialized when an application starts up; and destroyed when the application shuts down.

One other property is required here before we run the app. In Spring applications AntPathMatcher is used to identify classpath, file system, remote, and other resources in Spring configuration. It has also been used in Spring MVC to match URL paths.

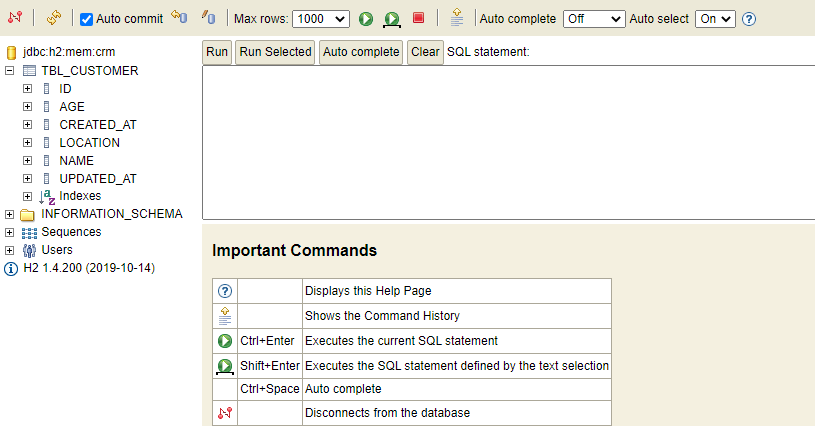
Add the required property for the AntPathMatcher.

You have added a property to enable the H2 console which tells Spring to start H2 database administration inside the browser. We can access it by visiting this url <http://localhost:8080/h2> **after**  running the application.



Click on the Connect button and it will connect you to the H2 database. and you can see that the TBL\_CUSTOMER table has been created, but there is no data in this table, which is as we expected.

Youc can check this by executing “SELECT \* FROM TBL\_CUSTOMER”.

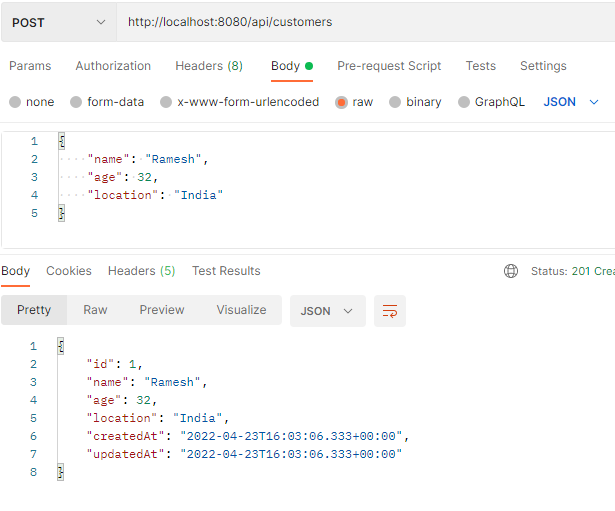


## **Running the application for the first time**

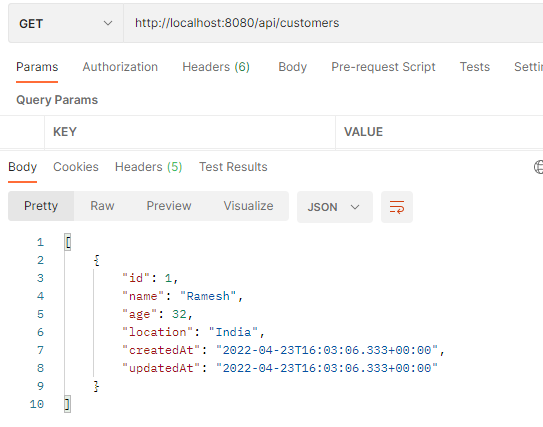
At this point, you should be able to test the configuration in Postman .

For the Sprint 1 project evaluation, you will need to demonstrate correct responses from the following APIs. You are free to use any data in the body of the POST and PUT requests.

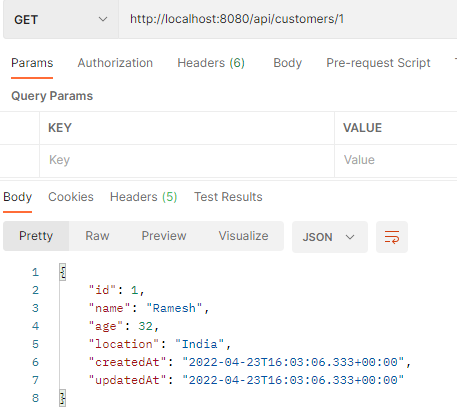
* POST /api/customers



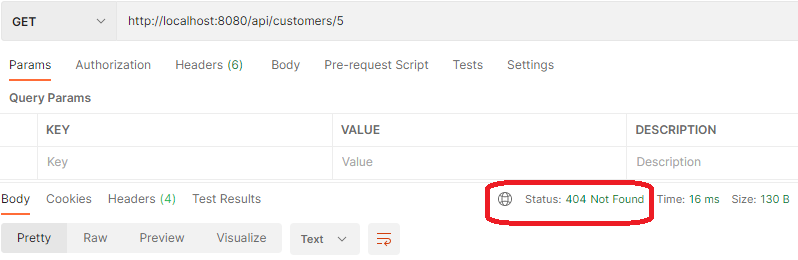
* GET /api/customers



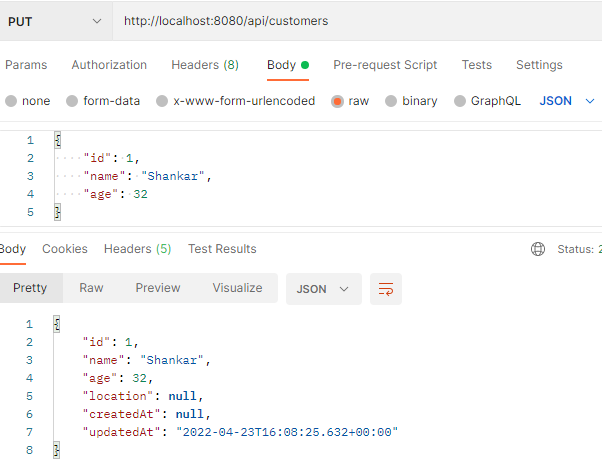
* GET /api/customers/1



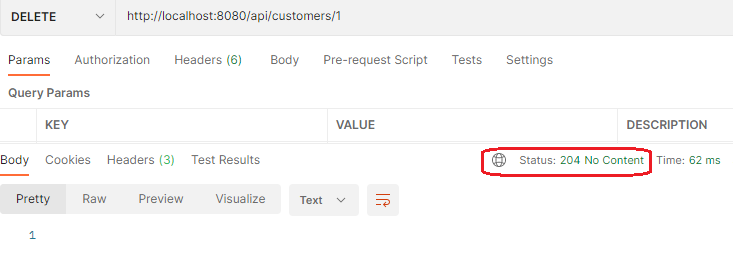
* GET /api/customers/5



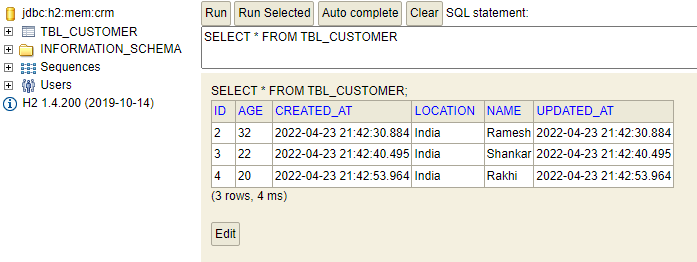
* PUT /api/customers



* DELETE /api/customers/1



****Add a few customers and the let’s verify once in the H2 Database:****



## **Summary**

The Spring framework is one of the most popular frameworks in the entire java ecosystem. More than 90% of java enterprise applications use the Spring framework in production.

Spring Boot is a tool to create Spring applications with zero configuration. Spring boot is getting popular day by day that is because you can create REST APIs and Microservices in no time.

JPA is a Java Persistence API, is the most popular data accessor API. It is also getting more popular day by day. Almost all the spring boot applications using JPA to access the database data access layer. JPA is just a standard/specification and Hibernate is the implementation.

You have  created a REST API project with Spring Boot, H2 Database, JPA, and Hibernate. These APIs are used to perform the basic CRUD (Create, Read, Update, Delete) operations on the H2 in-memory database..

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