# Spring Boot Accessing Data with JPA / REST

**Description:** Spring Boot is an open source, micro service-based Java web framework. The Spring Boot framework creates a fully production-ready environment that is completely configurable using its prebuilt code within its own codebase.

**Project:** Modify the application built using document title "Spring Boot 104a Access Data JPA" that is based on the website Accessing Data with JPA. This project modifies the project built with the references just mentioned to use RESTful APIs that can test the application externally through an application like Postman. At the end the modified project should work like the project built with the document title "Spring Boot 103 REST JPA Data".

**Technology:** This project uses the following technology:

Integrated Development Environment (IDE):

Spring Tool Suite 4 (Version: 4.15.0.RELEASE)

Java Development Kit (JDK):

Oracle's JDK 8 (1.8)

Other tools:

Postman – a web and desktop application used for API testing.

# Table of Contents

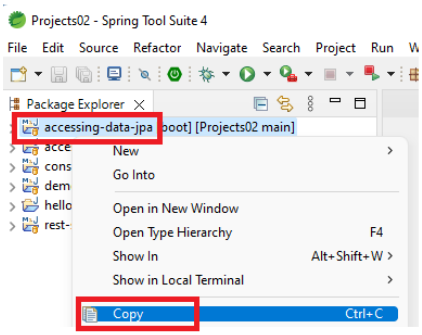
## Glossary of Terminology

For a list of key terms and definitions used throughout this and various Spring Boot demo documents see the document titled “Appendix 01 Glossary”.

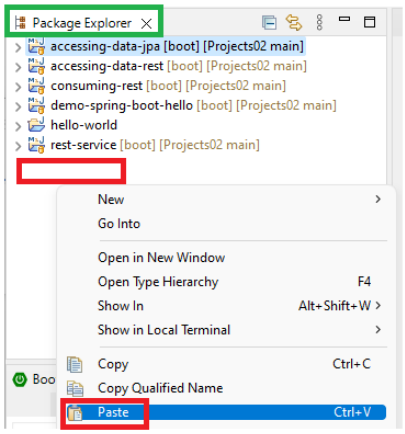
## Copy Existing Project: accessing-data-jpa-rest

This project builds on the project created using “Spring Boot 104a Access Data JPA” and on the website Accessing Data with JPA. The accessing-data-jpa-rest project must exist to following the details in this document.

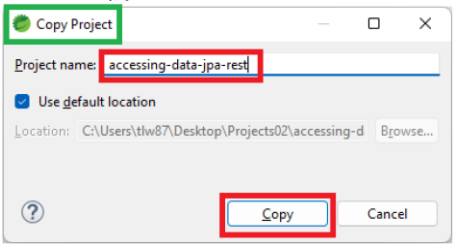
Right click on the project ➔ select “Copy”



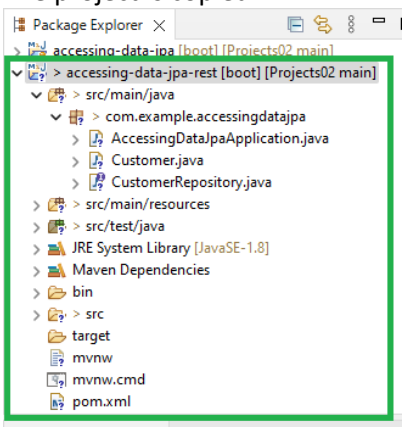

Right click in the “Package Explorer” ➔ select “Paste”




Enter “Project name:” accessing-data-jpa-rest
Click “Copy”



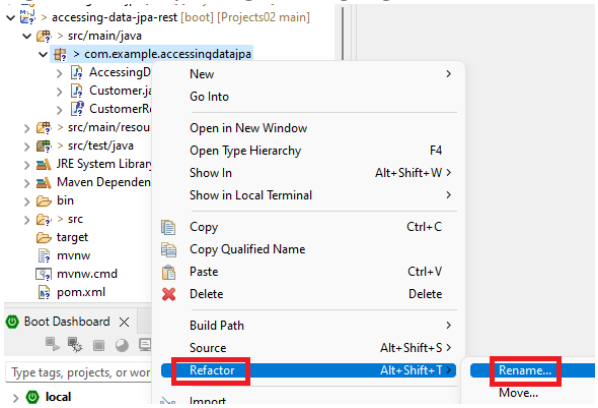
The project is copied.



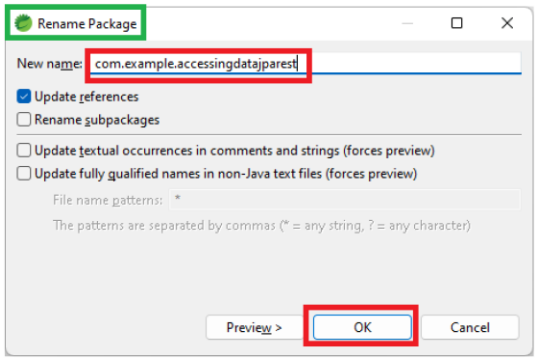

## Refactor the Copied Project

Refactor the elements in this section just makes the project consisted with naming convention if the project was built from scratch. Refactoring automatically takes care of reference made to the refactored element in other project files.

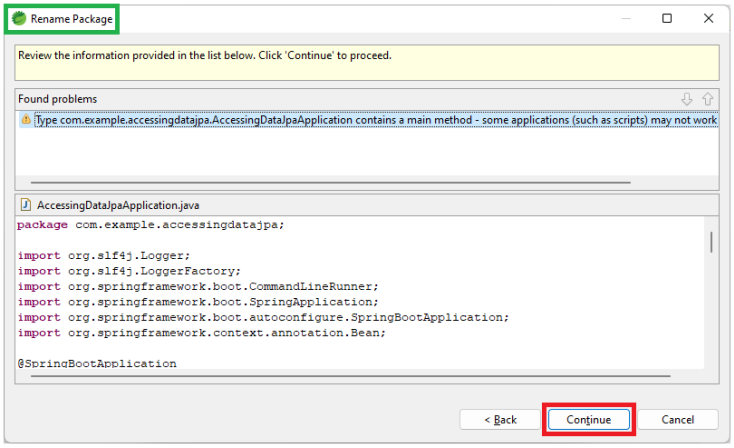
Refactor the package name
Right click the package ➔ highlight “Refactor” ➔ select “Rename”




Enter “New name:” com.example.accessingdatajparest
Click “OK”



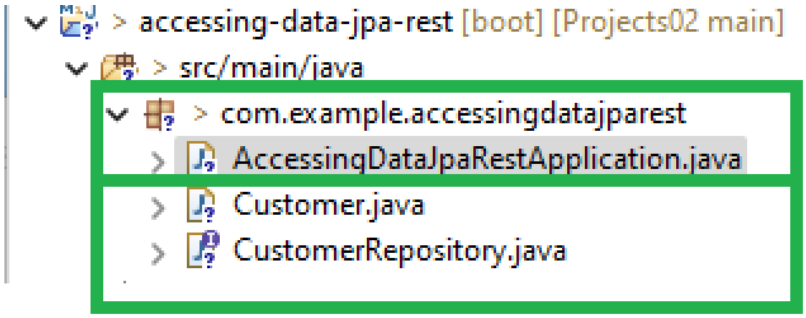
Click “Continue”




The refactoring will update the package name in each of the classes impacted.

Refactor the main class name
Right click the main class “AccessingDataRestApplication” ➔ highlight “Refactor” ➔ select “Rename”
Enter “New name:” “AccessingDataJpaRestApplication”
Click “Finish”
Click “Finish” in the next window.

The refactoring items are renamed, and references updated.



# Project accessing-data-jpa-rest Discussion

This project requires a new dependency in the pom.xml file and updates to the main class AccessingDataJpaRestApplication. The two other classes one to represent an entity a model class remains unchanged and a repository class require updates.

# Update pom.xml File

Update the “artifactId” and “name” tag values.

```
<artifactId>accessing-data-jpa-rest</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>accessing-data-jpa-rest</name>
```

```
12  <artifactId>accessing-data-jpa-rest</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>accessing-data-jpa-rest</name>
```

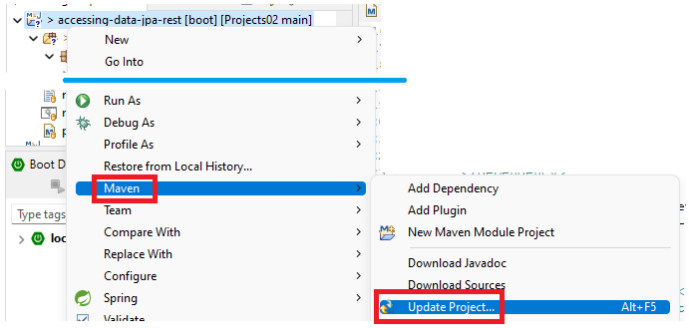
Add the rest dependency.

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

```
20      <dependency>
21          <groupId>org.springframework.boot</groupId>
22          <artifactId>spring-boot-starter-data-jpa</artifactId>
23      </dependency>
24      <dependency>
25          <groupId>org.springframework.boot</groupId>
26          <artifactId>spring-boot-starter-data-rest</artifactId>
27      </dependency>
28
29      <dependency>
30          <groupId>com.h2database</groupId>
31          <artifactId>h2</artifactId>
32          <scope>runtime</scope>
33      </dependency>
```

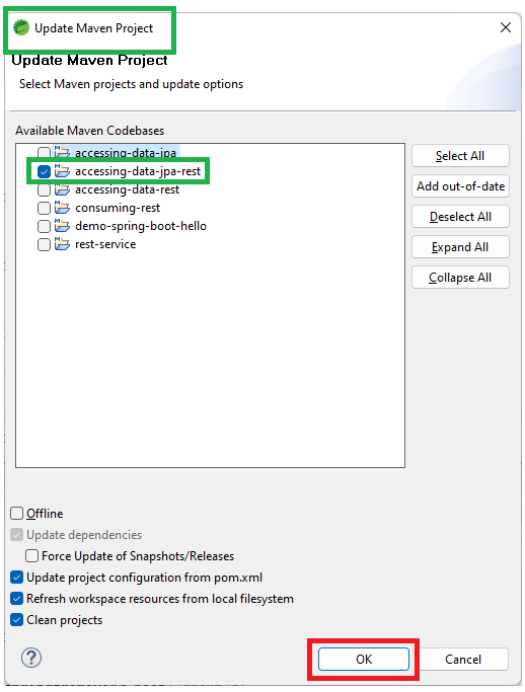
Save pom.xml and rebuild the Maven project. The project should pull in the new dependency when the pom.xml is saved. It has been seen where the dependency is not pulled so a good practice is to rebuild the Maven project.

Right click the project ➔ highlight “Maven” ➔ select “Update Project…”




Portions of the menu not shown to save space.

Click “OK”




## Update Main Repository classes

Update the repository class which provide various operations involving the Domain Object.
Update the main application class.

### Update Repository Query Interface: CustomerRepository

The updates to this interface include add the RepositoryRestResource annotation and changes the parent class.

Existing interface.

CustomerRepository.java

```
1  package com.example.accessingdatajparest;
2
3  import java.util.List;
6
7  public interface CustomerRepository extends CrudRepository<Customer, Long> {
8
9        List<Customer> findByLastName(String lastName);
10
11       Customer findById(long id);
12 }
13
```

Add annotation and update extends class.

```
@RepositoryRestResource(collectionResourceRel = "customer", path = "customer")
public interface CustomerRepository extends PagingAndSortingRepository<Customer, Long> {
```

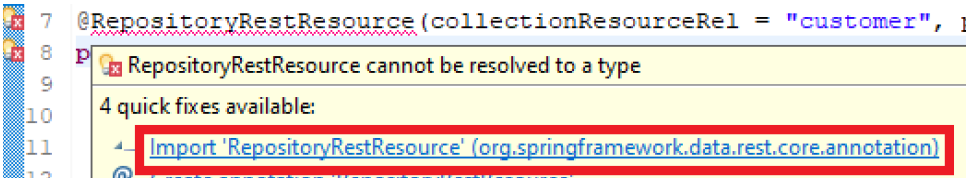
```java
package com.example.accessingdatajparest;

import java.util.List;

@RepositoryRestResource(collectionResourceRel = "customer", path = "customer")
public interface CustomerRepository extends PagingAndSortingRepository<Customer, Long> {

	List<Customer> findByLastName(String lastName);

	Customer findById(long id);
}
```

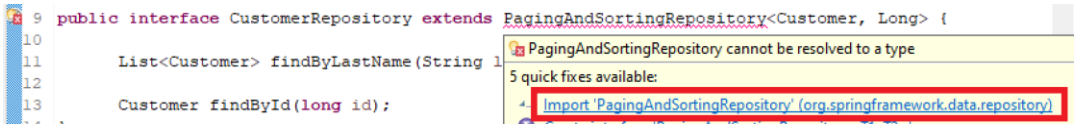
**Correct the import errors:**

Use the IDE Wizard to help correct the errors.

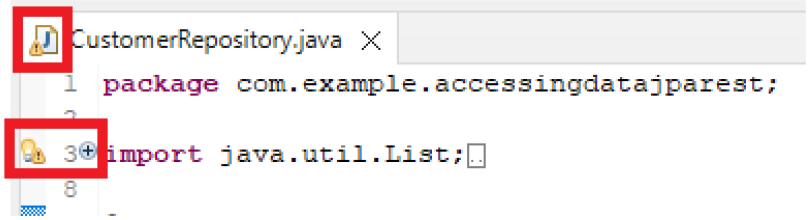
Hover the mouse over the error.

Select the import link shown below.

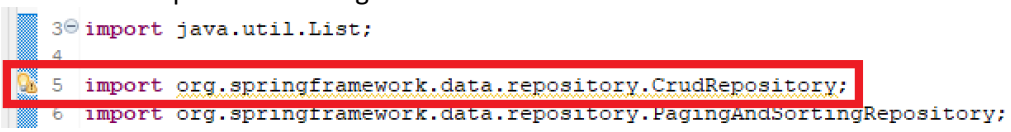



Expand the import section to see what import is no longer used and is causing the warning.



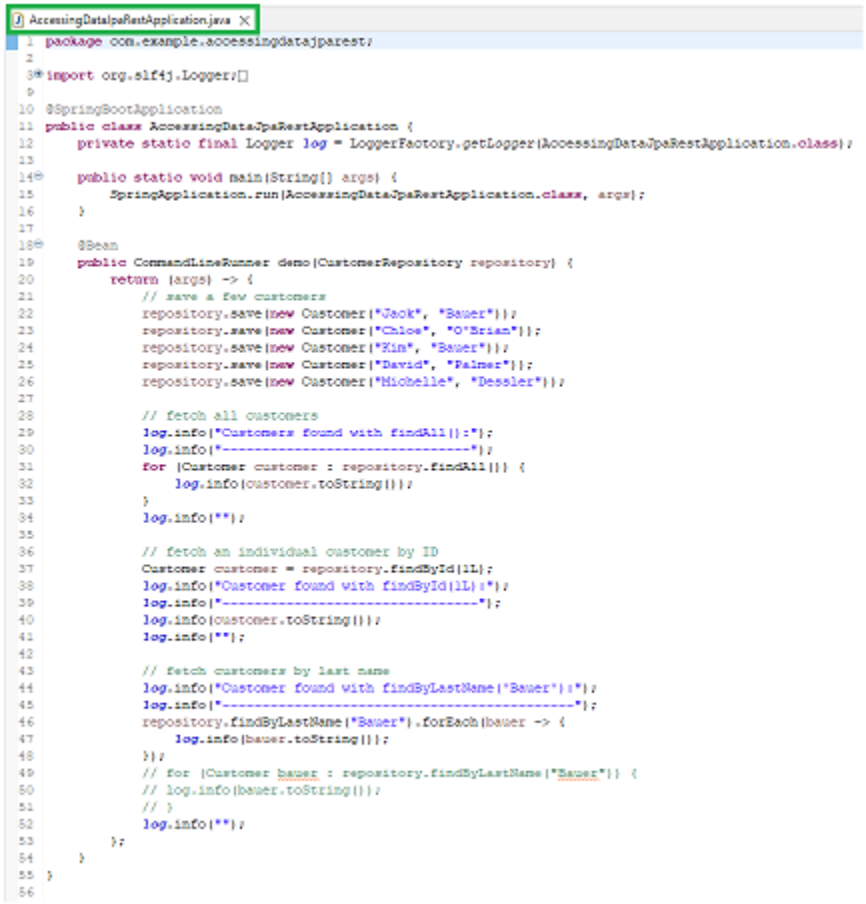

Remove the import line no longer used.



## Update Main Class: AccessingDataJpaRestApplication

The updates to the main class returns it to the original state when first generated. The class variable and bean method are removed.

Existing main class.




To make this update simply copy the code below and replace all the code in the main class.

```
package com.example.accessingdatajparest;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AccessingDataJpaRestApplication {

        public static void main(String[] args) {
                SpringApplication.run(AccessingDataJpaRestApplication.class, args);
        }

}
```

## Test the Project Spring Boot Tomcat Server

An external build of an executable JAR file is not necessary to test this project. Use the Tomcat server bundled with a Spring Boot project built with the "Spring Initializr".

Run the application within the IDE.
Right click inside the main class "AccessingDataJpaRestApplication".
Highlight "Run As" ➔ select "3 Spring Boot App"

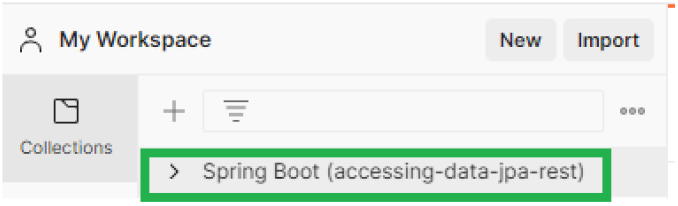
## Test the Project Using Postman

This section uses Postman to test this modified project. The testing follow the Postman approach described in the document titled "Spring Boot 103 REST JPA Data". Postman requests from the document are modeled and modify to test this project. Using an existing Postman account or creating a new account by using the document title "Environment Setup 05 Postman Setup" is required for following the test in this section. A basic understanding of using Postman is assumed.

The Postman desktop application is used since we are testing "localhost" URLs. In a version update of Postman, it started restricting using the web-base version for "localhost" testing.

### Create a Postman Collection: Spring Boot (accessing-data-jpa-rest)

Select "My Workspace" ➔ "New"
Select "Collection"
Enter "Name": "Spring Boot (accessing-data-jpa-rest)"
Press enter



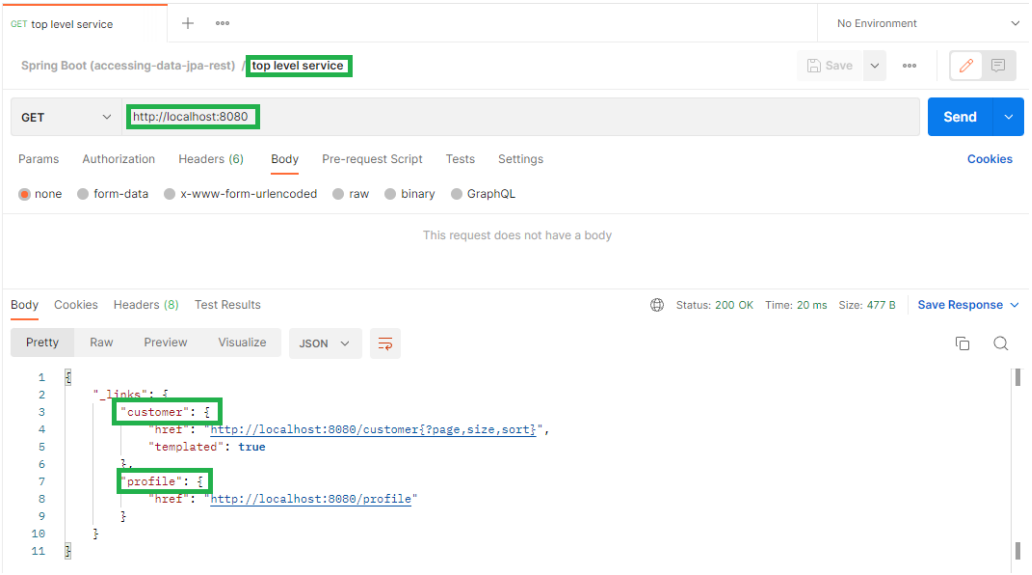

### Create Postman HTTP Requests for Project (accessing-data-jpa-rest)

This section describes a collection of various HTTP request for testing this project. While creating the requests in this section keep in mind only one request is defined in the repository class that extends the interface "PagingAndSortingRepository<>". Nowhere in the code is defined the other data Create, Read, Update, and Delete (CRUD) methods. All methods including the one defined in the code will use the default CRUD methods from the extended interface.

## GET Request: top level service

Request to get the top-level service.

Create GET Request named “top level service” with URL: “http://localhost:8080”.
Send the request.
Examine the response: as the name implies the response shows top level data for requests “people” and “profile”. Request will be developed based on the top-level response shown.

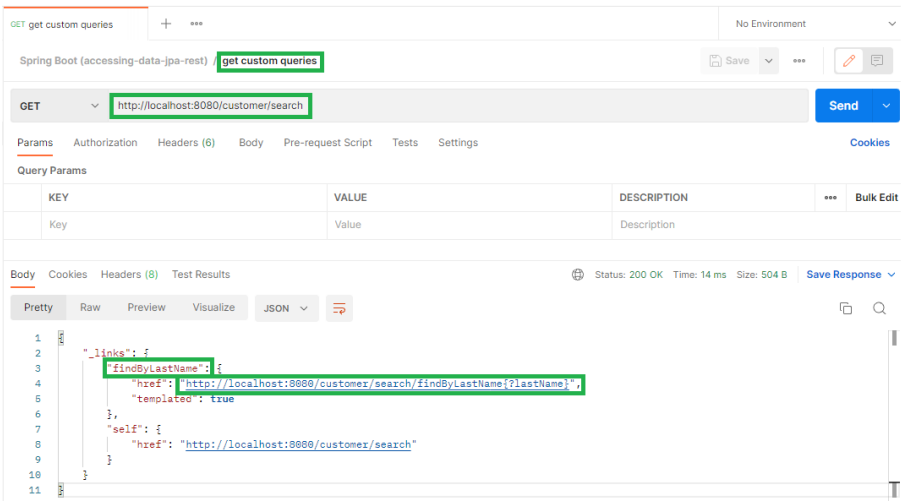



## GET Request: custom queries

Request to find all the custom queries defined in the program. In this program the class “CustomerRepository” defines “findByLastName()” and “findById()” queries. As mentioned before until now all other requests in this document used the CRUD methods inherited by the interface in the repository class “PagingAndSortingRepository<>”. Custom queries still uses CRUD methods and returns the type defined by the method.

Create GET Request named “get custom queries” with URL: “http://localhost:8080/people/search”.
Send the request.
Examine the response:

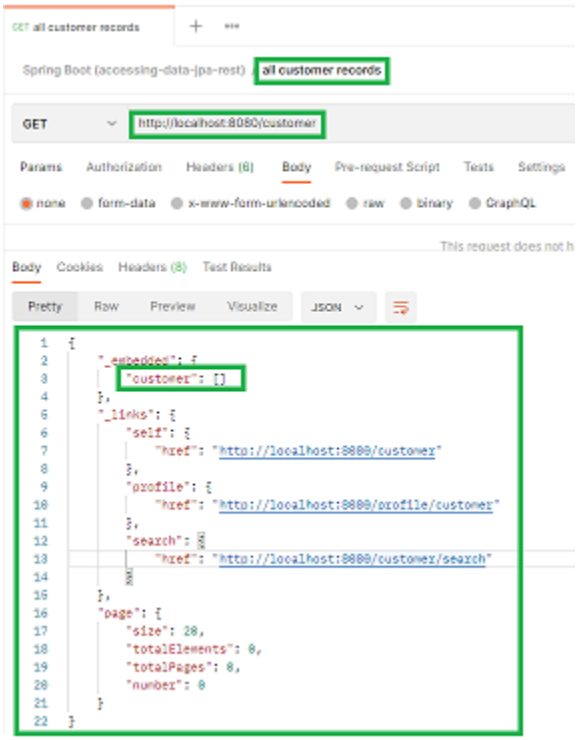
> The response shows the one custom query “findByLastName()” with the URL and expected parameter defined in this program. The response does not show the query “findById()” which returns a Customer object.




## GET Request: all customer records

Request to get the list of all customer records in the database.

Create GET Request named “all customer records” with URL: “http://localhost:8080/customer”.
Send the request.
Examine the response: since there are no records yet in the memory database, we see no customer records. We do see other URL that can be tested.



## POST Request: create a customer

Request to create a customer record in the database.

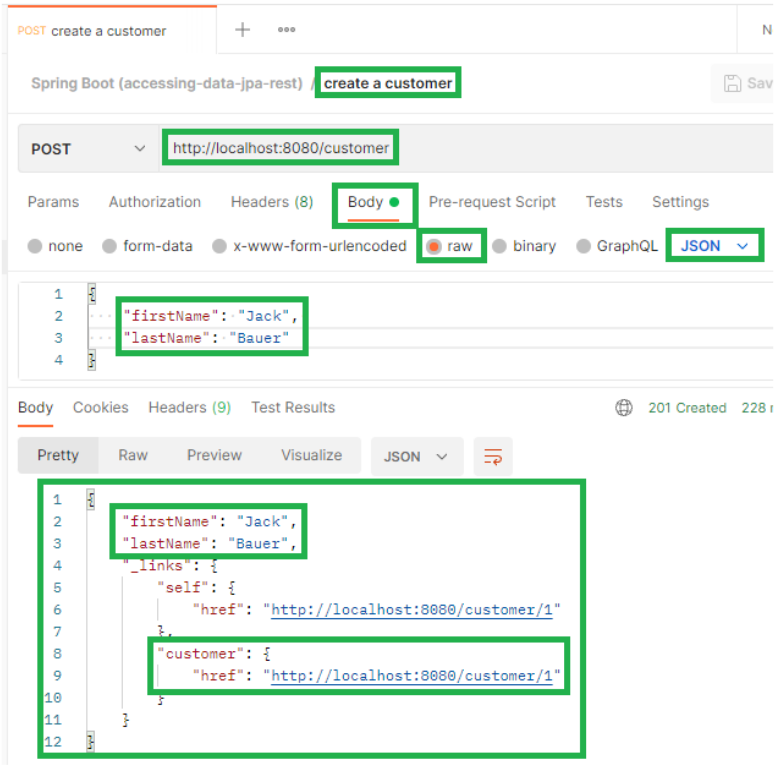
Create POST Request named “create a customer” with URL: “http://localhost:8080/customer”.
Send the request.
Create POST request body.

Add the body.

```
{
    "firstName": "Jack",
    "lastName": "Bauer"
}
```

Send the request.
Examine the response:

The response echoes back the record created showing a URL to find the record number by its record number 1 in this case.




To create additional records, update the first and last name in the request body. You can also run the same record again and have two records with the same names. There is no restriction to have duplicate records.

### *Run GET Request: all customer records*

Run the get all customer records request again.  Now the response has a customer record in the response.

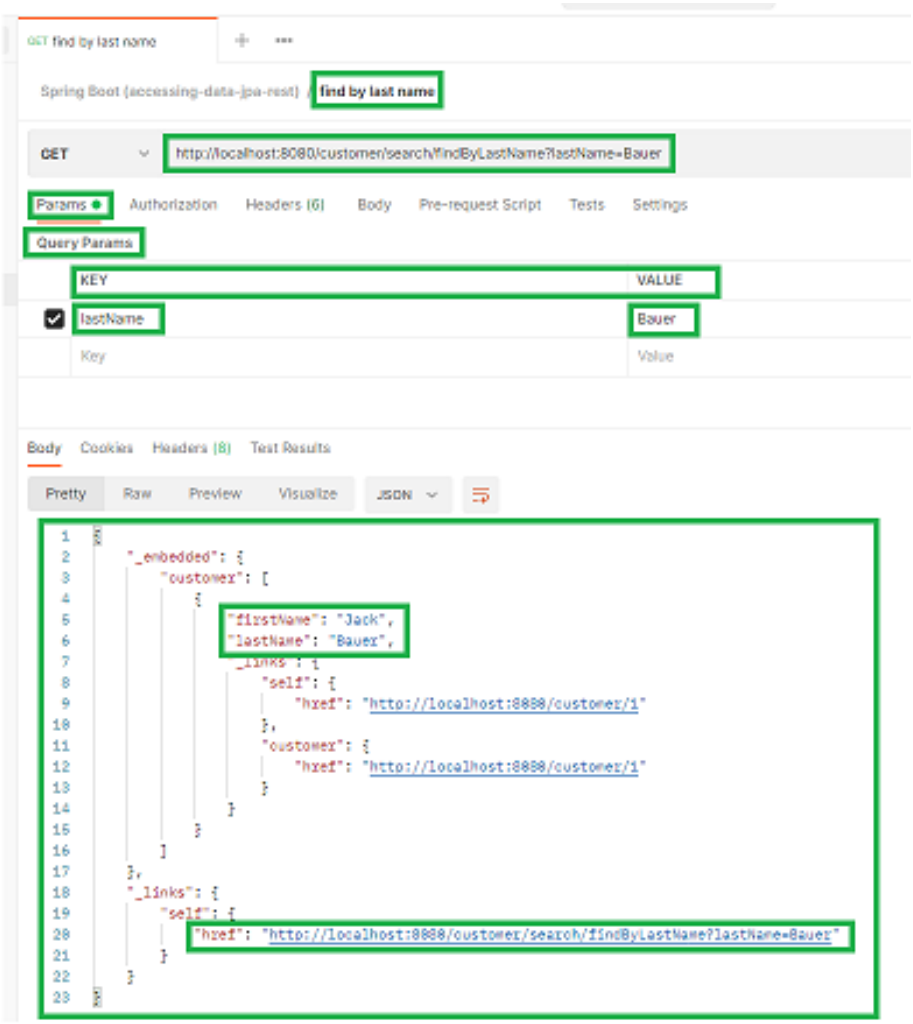
## GET Request: custom query find by last name

Custom query request to find a customer by their last name. In this request there is "?name=some_name" notation. This notation is called a query parameter.

Create GET Request named "find by last name" with URL:
"http://localhost:8080/customer/search/findByLastName?lastName=Bauer".
Send the request.

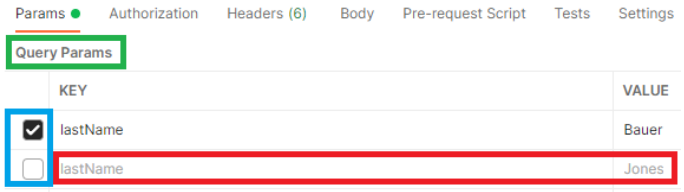
Examine the response:

The response should contain a list of all people records with the last name of "Bauer". In this case just one element is in the list. The document will create another customer with the same last name and test this again in another section. Along with other information the response shows an URL to retrieve this specific record indicated by the /1 at the end of the URL. This will become another query to get a record by its record id.

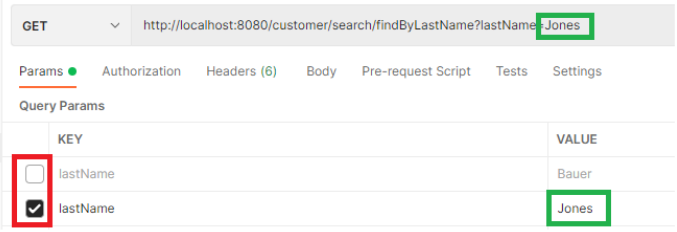



NOTE: "Query Params" with "KEY" of "name" and "VALUE" of "Bauer" is automatically created.
Send the request.

As stated above “Query Params” are automatically created.  With this you can add other “name” keys and values.




Then toggle off the current name and on the new name and the URL will change.



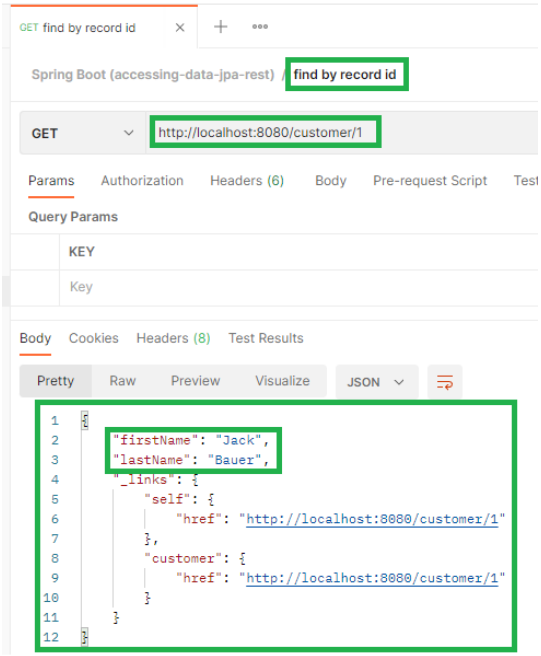

## GET Request: find by record id

Define a query request to find a people record by the unique record id.  In this request there is “/rec_id” notation.  This notation is called a path parameter.

Create GET Request named “find by record id” with URL: “http://localhost:8080/customer/1”.
Send the request.
Examine the response:

> Since the record id is a unique record id identified and generated by the annotation and statement in class Person.

The query will return one record based on the unique record id.

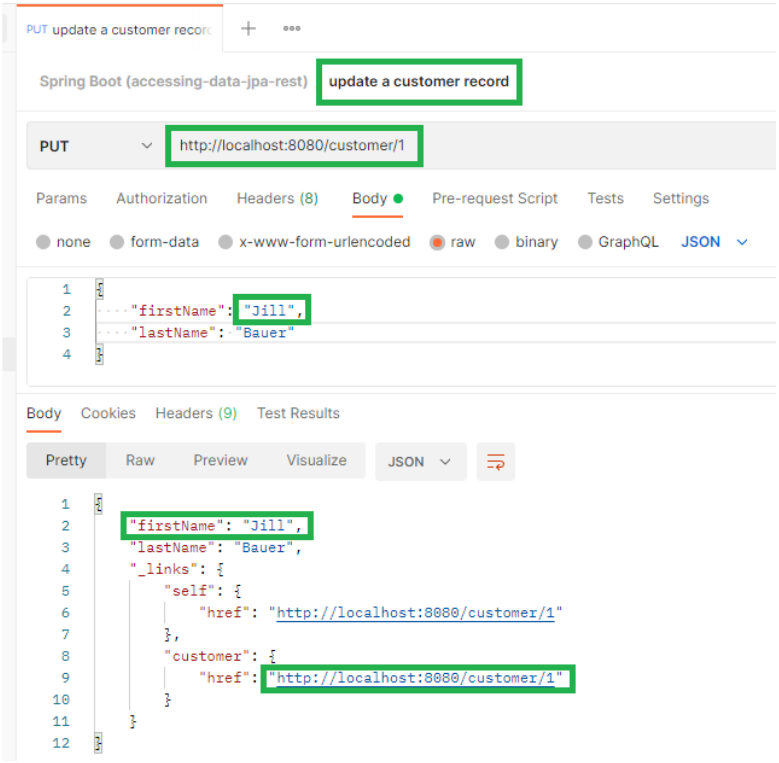


## PUT Request: update a customer record

Request to update an exist record in the database. A PUT request updates the entire record even if no data is changed. The path parameter in the URL "/1" is the unique record id to modify in the database. If the record does not exist, it will be created.

Create PUT Request named "update a customer record" with URL: "http://localhost:8080/customer/1".
Change the body's "firstName" to "Jill".
Send the request.
Examine the response:

The response echoes back the record created showing a URL to find the record number by its record number 1 in this case.




## PATCH Request: update a customer record field

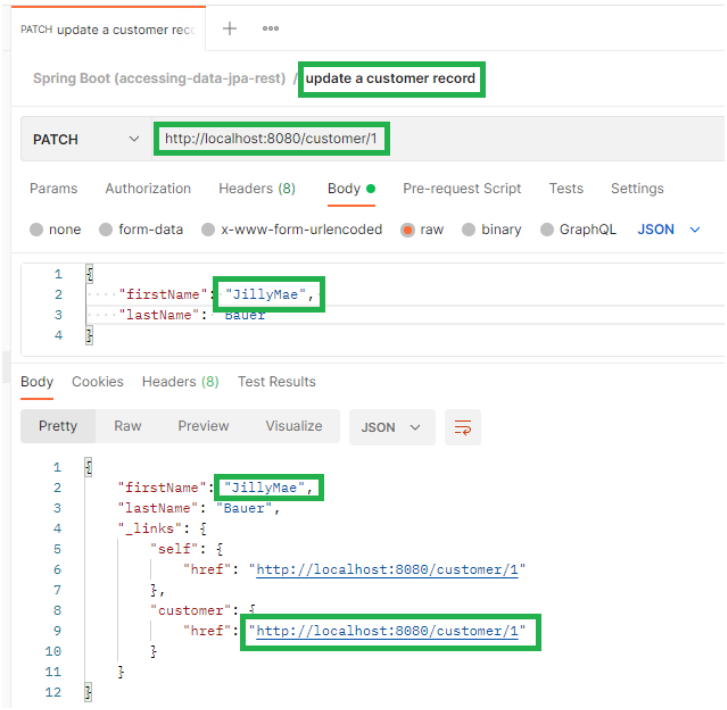
Request to update a field in an exist record in the database. A PATCH request updates only the fields that have changed in a record. The path parameter in the URL "/1" is the unique record id to modify in the database. If the record does not exist, nothing happened, and no error message is seen.

Creating this request is like the PUT. Just the request type is change and an element in the body is change.

Create PATCH Request named "update a customer record" with URL: "http://localhost:8080/customer/1".
Change the body's "firstName" to "JillyMae".
Send the request.

Examine the response:

The response echoes back the record showing the changes to the record.

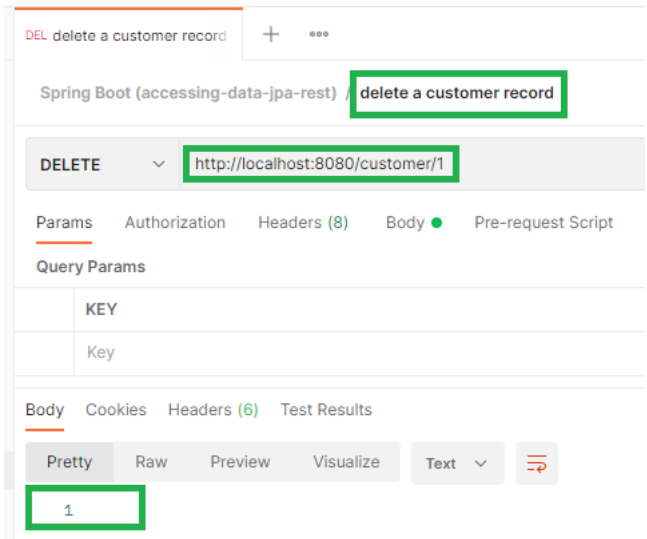



## DELETE Request: delete a customer record

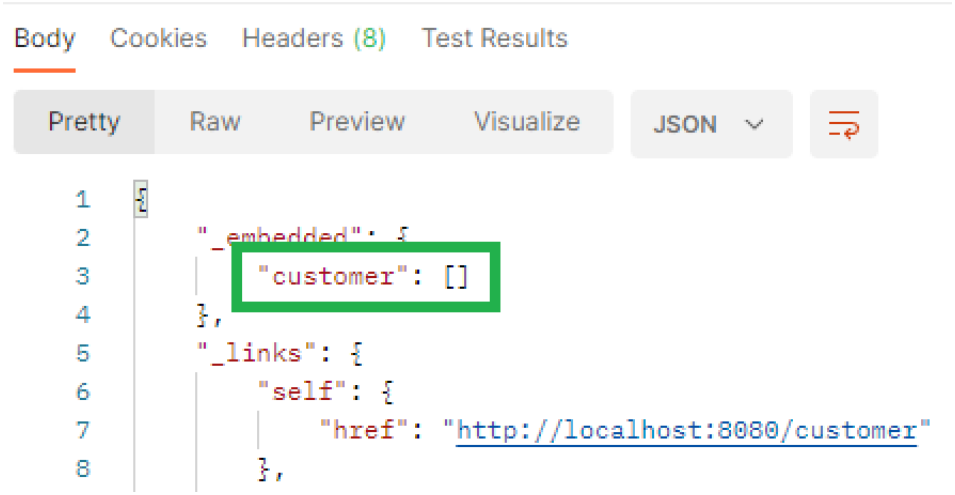
Request to delete an existing record from the database. If the record does not exist, nothing happened, and no error message is seen.

Create DELETE Request named “delete a customer record” with URL: “http://localhost:8080/customer/1”.
Send the request.
Examine the response:

The response does not give a good indication that the record was delete, it just sends a 1.



If you run the “GET all people records” request, the results show no records in the list.  This is expected since only one record was created.



## Additional Testing Using the Postman (accessing-data-jpa-rest) Requests

This section describes running Postman request in the previous section. These tests should show more results in most cases.

### Create Customer Records

Use the "POST create a customer" request with the following bodies to create multiple records for testing.

```
{
    "firstName": "Jack",
    "lastName": "Bauer"
}
```

```
{
    "firstName": "Chloe",
    "lastName": "O'Brian"
}
```

```
{
    "firstName": "Kim",
    "lastName": "Bauer"
}
```

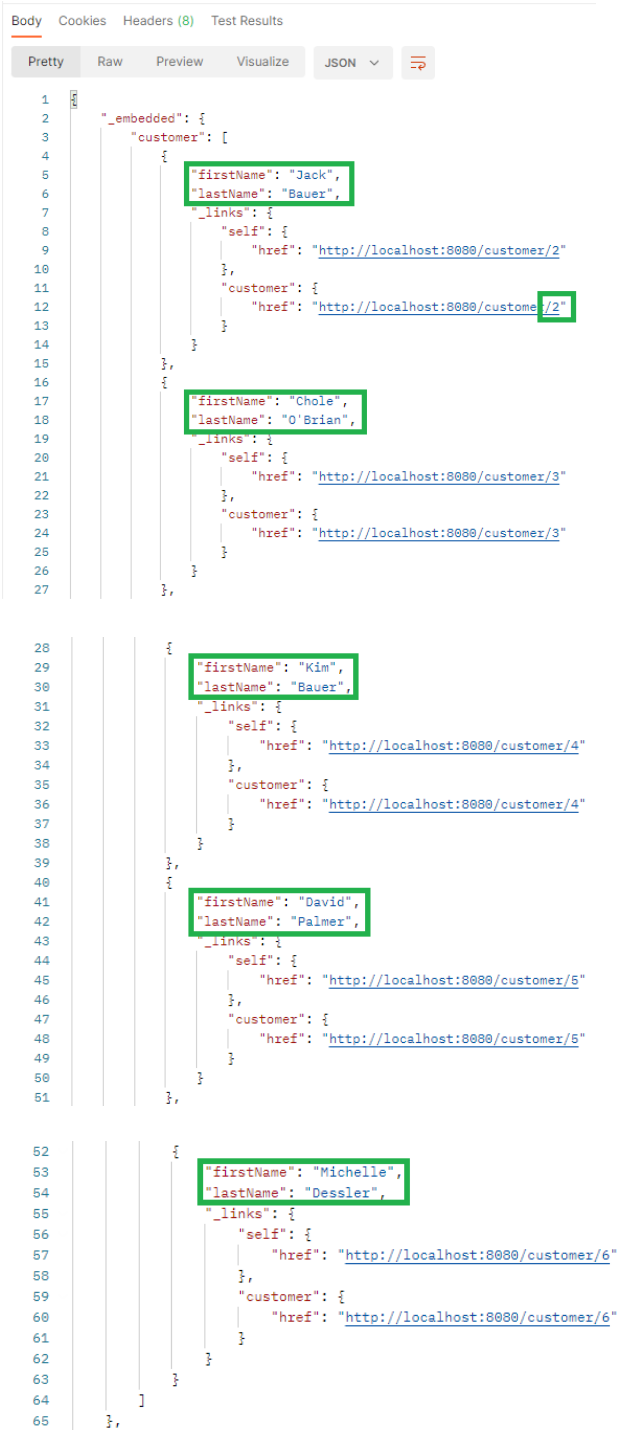
```
{
    "firstName": "David",
    "lastName": "Palmer"
}
```

```
{
    "firstName": "Michelle",
    "lastName": "Dessler"
}
```

## Retrieve All Records

Use the “GET all customer records” request to retrieve record.
Examine the response:

Five records where created. All records are retrieved. Noticed that the first record has a record id of 2 and not 1. This test was conducted in the same session where the “Jack Bauer” record was first created and deleted. Once a record is deleted from the database, the record id is not reused.



## Retrieve Records by Last Name

Use the “GET find by last name” request to retrieve record with last name of “Bauer”.
Examine the response:

Two records are retrieved. Only records with last name of “Bauer” are in the list. The records with other last names are not retrieved.


```
{
    "_embedded": {
        "customer": [
            {
                "firstName": "Jack",
                "lastName": "Bauer",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/customer/2"
                    },
                    "customer": {
                        "href": "http://localhost:8080/customer/2"
                    }
                }
            },
            {
                "firstName": "Kim",
                "lastName": "Bauer",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/customer/4"
                    },
                    "customer": {
                        "href": "http://localhost:8080/customer/4"
                    }
                }
            }
        ]
```