

Spring Boot REST with JPA Data

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: You will build a Spring Boot application that lets you create and retrieve Person objects stored in a database by using Spring Data REST. Spring Data REST takes the features of Spring HATEOAS and Spring Data JPA and automatically combines them together. This project is based on [Accessing JPA Data with REST](#) and just expands on the screenshots and step by step instructions.

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	3
Generate Spring Boot Download	3
Import the Spring Boot Download	3
Import the project: “accessing-data-rest”	3
Demo Project accessing-data-rest Discussion	5
Class: AccessingDataRestApplication.....	5
Create Domain Object Model and Repository classes.....	6
Create Domain Model Class: Person.....	6
Update Domain Model class: Person.....	6
Copy and paste source code for class: Person.....	8
Create Repository Interface: PersonRepository	9
Update repository interface: PersonRepository	10
Add RepositoryRestController annotation	10
Copy and paste source code for class: PersonRepository	11
Build and Run the Project	12
Test the Project Using Postman	13

Create a Postman Collection: Spring Boot (accessing-data-rest)	13
Create Postman HTTP Requests for Project (accessing-data-rest)	14
GET Request: top level service	14
GET Request: custom queries	15
GET Request: all people records	15
POST Request: create a person	16
Run GET Request: all people records	17
GET Request: custom query find by last name	18
GET Request: find by record id	19
PUT Request: update a people record	20
PATCH Request: update a people record field	20
DELETE Request: delete a people record	21
Additional Testing Using the Postman (accessing-data-rest) Requests	23
Create People Records	23
Retrieve All Records	24
Retrieve Records by Last Name	25

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”.

Generate Spring Boot Download

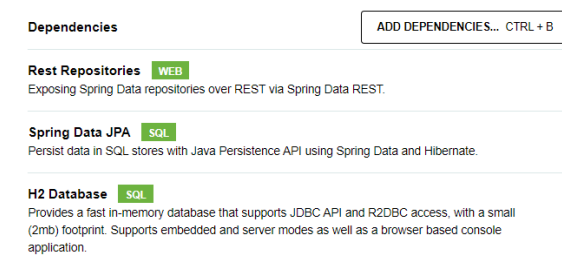
Follow the instructions in the document title “Appendix 02 Spring Initializr” to generate a spring boot download for this project.

When the document talks about adding dependencies add only these:

Rest Repositories

Spring Data JPA

H2 Database



When the document talks about the items in the “**Project Metadata**” use the values shown below:

“**Group**” use “**com.example**”

“**Artifact**” use “**accessing-data-rest**”

“**Name**” use “**accessing-data-rest**”

“**Package name**” use “**com.example.accessing-data-rest**”

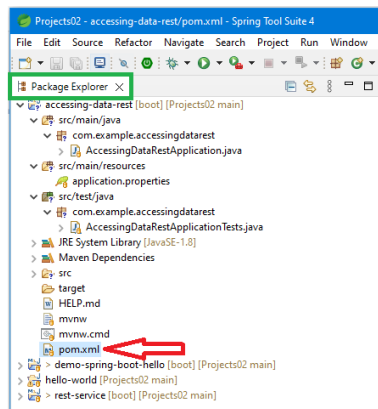
For other items in the “**Project Metadata**” use the defaults. Follow the instructions to extract the files from the zip file into the Sprint Tool Suite 4 workspace.

Import the Spring Boot Download

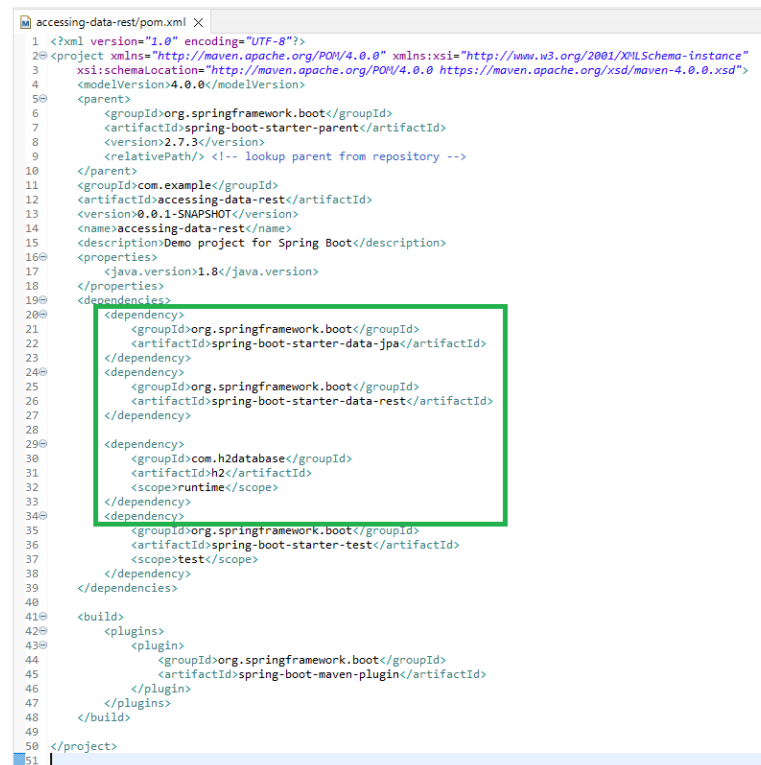
Open the Spring Tool Suite 4 IDE.

Import the project: “accessing-data-rest”

Follow the instructions in the document title “Appendix 03 Import Spring Tool Suite Project” and import the “**accessing-data-rest**” project that was created with Spring Initializr. After importing the project, it should look like the following using the IDE “Package Explorer”.



Look at the pom.xml and find the three dependencies for this project.

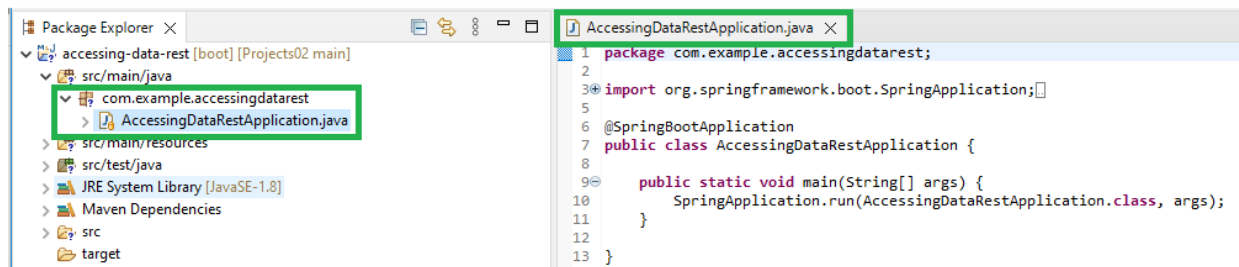


Demo Project accessing-data-rest Discussion

This project uses the generated application to start the applications. Two other classes are created one to represent an entity a model class and a controller class.

Class: AccessingDataRestApplication

The AccessingDataRestApplication class created when using the Spring Initializr is used without any additional modification from what was generated. This class contains the “**public static void** main(String[] args) {}” method that starts the application listening when the internal web server is started.



Create Domain Object Model and Repository classes

Create a Domain Object also known as a Model class with database annotation. The database annotation is what will create the Domain Object structure as a database table with columns. Since this project is using a H2 Database, the structure will only exist in memory with no physical representation.

Create a repository class which provide various operations involving the Domain Object.

Create Domain Model Class: Person

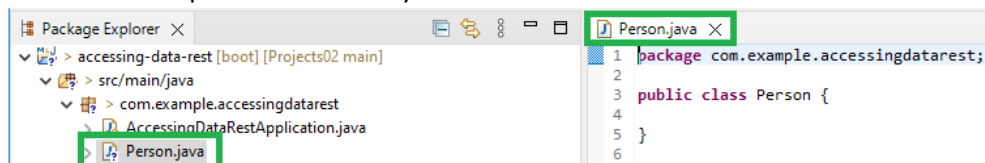
Right click on package "com.example.accessingdatarest"

Right highlight "New" → select "Class"

Enter class "Name:" "Person"

Click "Finish"

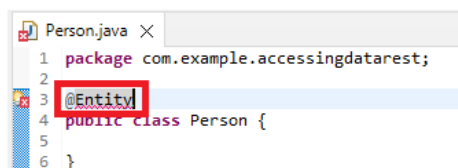
The new class opens automatically in an edit window.



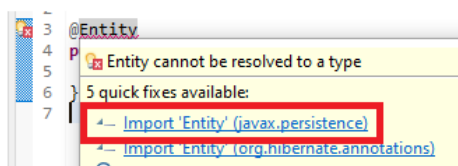
Update Domain Model class: Person

Add database entity annotation. Specifies that the class is an entity. This annotation is applied to the entity class.

```
@Entity
```



Resolve the error.



Add private class variable with database annotation.

@Id - Specifies the primary key of an entity.

@GeneratedValue - Provides for the specification of generation strategies for the values of primary keys.

GenerationType - Defines the types of primary key generation strategies.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private long id;
```

```

6 public class Person {
7     @Id
8     @GeneratedValue(strategy = GenerationType.AUTO)
9     private long id;
10 }

```

Resolve the errors.

```

7     @Id
8     @Id cannot be resolved to a type
9
10 4 quick fixes available:
11  - Import 'Id' (javax.persistence)
12  - Import 'Id' (org.springframework.data.annotation)

```

```

9     @GeneratedValue(strategy = GenerationType.AUTO)
10    @GeneratedValue cannot be resolved to a type
11
12 5 quick fixes available:
13  - Import 'GeneratedValue' (javax.persistence)
14  - Create annotation 'GeneratedValue'
15  - Change to 'Generated' (javax.annotation)

```

```

10    @GeneratedValue(strategy = GenerationType.AUTO)
11    private long id;
12
13 GenerationType cannot be resolved to a variable
14
15 8 quick fixes available:
16  - Import 'GenerationType' (javax.persistence)
17  - Create class 'GenerationType'

```

Add the rest of the method code.

```

private String firstName;
private String lastName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

```

```

Person.java X
1 package com.example.accessingdatarest;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.GenerationType;
6 import javax.persistence.Id;
7
8 @Entity
9 public class Person {
10     @Id
11     @GeneratedValue(strategy = GenerationType.AUTO)
12     private long id;
13
14     private String firstName;
15     private String lastName;
16
17     public String getFirstName() {
18         return firstName;
19     }
20
21     public void setFirstName(String firstName) {
22         this.firstName = firstName;
23     }
24
25     public String getLastName() {
26         return lastName;
27     }
28
29     public void setLastName(String lastName) {
30         this.lastName = lastName;
31     }
32 }
33

```

Copy and paste source code for class: Person

As an alternative you can copy the code below and replace all code in the class if you used the same package name.

```
package com.example.accessingdatarest;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

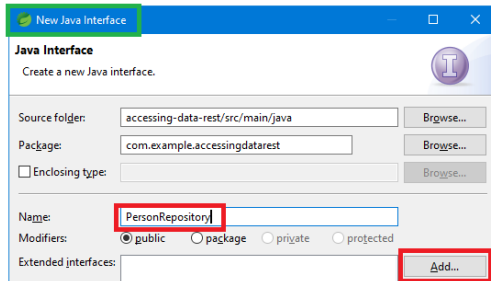

Create Repository Interface: PersonRepository

Right click on package “com.example.accessingdatarest”

Right highlight “New” → select “Interface”

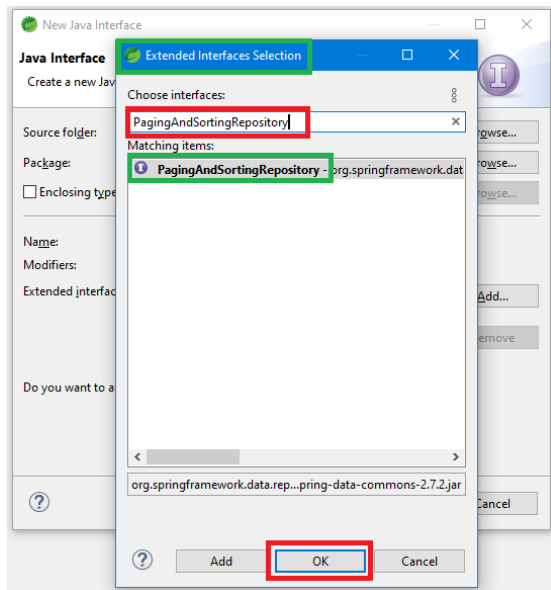
Enter interface “Name:” “PersonRepository”

Click “Extended interfaces:” “Add...” button

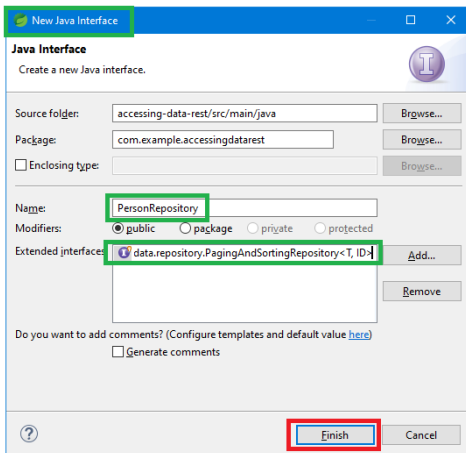


Enter “Chooses interface” : “PagingAndSortingRepository”

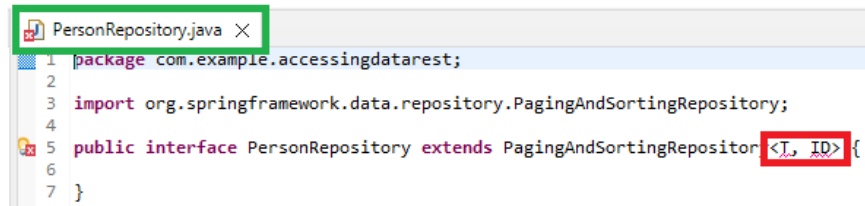
Click “OK”



Click “Finish”



The interface opens automatically in an edit window with errors.

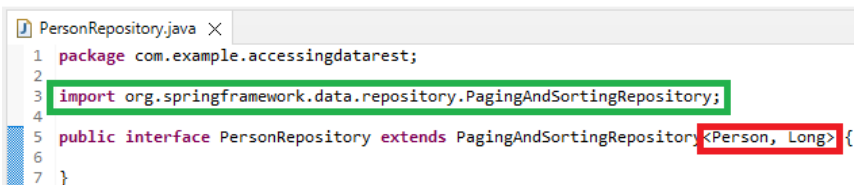


```
1 package com.example.accessingdatarest;
2
3 import org.springframework.data.repository.PagingAndSortingRepository;
4
5 public interface PersonRepository extends PagingAndSortingRepository<T, ID> {
6
7 }
```

Update repository interface: PersonRepository

Fix the initial error, update the types in the extends class <diamond>.

```
<Person, Long>
```

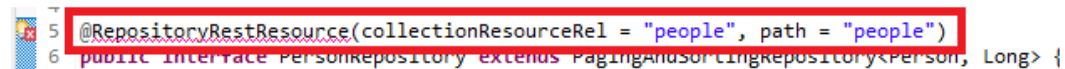


```
1 package com.example.accessingdatarest;
2
3 import org.springframework.data.repository.PagingAndSortingRepository;
4
5 public interface PersonRepository extends PagingAndSortingRepository<Person, Long> {
6
7 }
```

Add RepositoryRestController annotation

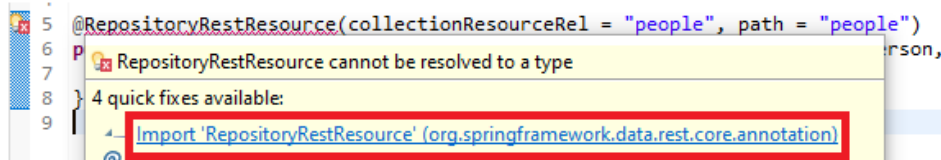
RepositoryRestController - used to set options on the public Repository interface - it will automatically create endpoints as appropriate based on the type of Repository that is being extended.

```
@RepositoryRestController(collectionResourceRel = "people", path = "people")
```



```
5 @RepositoryRestController(collectionResourceRel = "people", path = "people")
6 public interface PersonRepository extends PagingAndSortingRepository<Person, Long> {
```

Resolve the error.

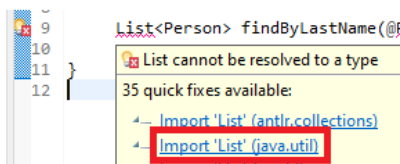


```
5 @RepositoryRestController(collectionResourceRel = "people", path = "people")
6
7 RepositoryRestController cannot be resolved to a type
8 4 quick fixes available:
9 - Import 'RepositoryRestController' (org.springframework.data.rest.core.annotation)
```

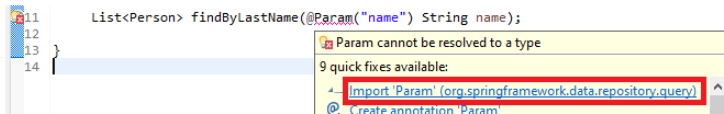
Add a class method returning a List of Person.

```
List<Person> findByLastName(@Param("name") String name);
```

Resolve the errors.



```
9 List<Person> findByLastName(@Param("name") String name);
10
11 List cannot be resolved to a type
12 35 quick fixes available:
13 - Import 'List' (antlr.collections)
14 - Import 'List' (java.util)
```



The completed interface.

```
PersonRepository.java X
1 package com.example.accessingdatarest;
2
3 import java.util.List;
4
5 import org.springframework.data.repository.PagingAndSortingRepository;
6 import org.springframework.data.repository.query.Param;
7 import org.springframework.data.rest.core.annotation.RepositoryRestResource;
8
9 @RepositoryRestResource(collectionResourceRel = "people", path = "people")
10 public interface PersonRepository extends PagingAndSortingRepository<Person, Long> {
11
12     List<Person> findByLastName(@Param("name") String name);
13
14 }
```

Copy and paste source code for class: PersonRepository

As an alternative you can copy the code below and replace all code in the class if you used the same package name.

```
package com.example.accessingdatarest;

import java.util.List;

import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(collectionResourceRel = "people", path = "people")
public interface PersonRepository extends PagingAndSortingRepository<Person, Long> {

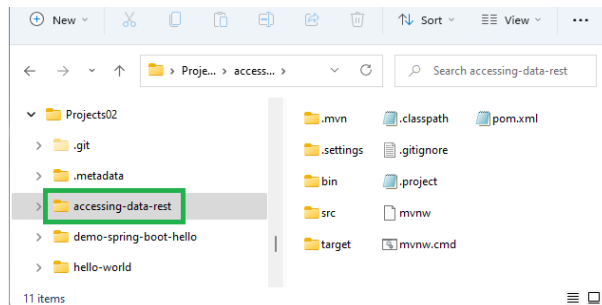
    List<Person> findByLastName(@Param("name") String name);

}
```

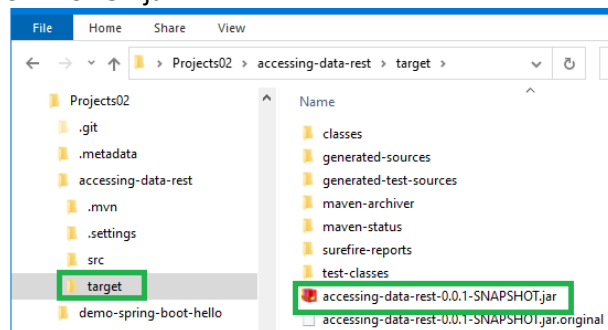
Build and Run the Project

The website “[Accessing JPA Data with REST](#)” this project is based on has testing uses an executable Java Archive (JAR) file as the way to run and test the program. Alternatively testing can be performed using the IDE running the main class “AccessingDataRestApplication” as “Sprint Boot App” inside the IDE.

In the “Appendix 04 Run Spring Initializr Project” document follow the instructions in section title “Executable JAR File Lifecycle”. Follow the instructions in sub sections title “Build the Executable JAR” and “Run the Executable JAR File” and stop there. Makes sure to build in this project’s folder.



After the build the executable JAR file is found in the target folder “accessing-data-rest-0.0.1-SNAPSHOT.jar”.



After starting the project’s executable JAR file test the application. Alternatively, right click inside the main class “AccessingDataRestApplication” select “Run As” ➔ “Sprint Boot App”.

Note: Any time the internal Spring Boot server is restarted the memory database is cleared. All data created during a testing session is lost.

Test the Project Using Postman

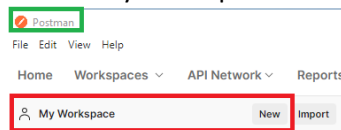
This project's website [Accessing JPA Data with REST](#) presents testing using the executable JAR file by showing examples using the [nix tool curl](#). If you are familiar with the [nix tool curl](#) you can follow the examples presented on the project's website [Accessing JPA Data with REST](#).

This section uses Postman in creating the examples from the website [Accessing JPA Data with REST](#). This section adds additional tests gleamed from the responses from the initial testing. 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.

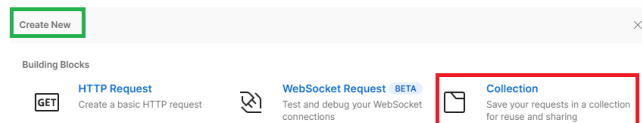
The Postman desktop application is used since we are testing "localhost" URLs. In a version update of Postman, it started restricting using the web-base version for "localhost" testing.

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

Select "My Workspace" → "New"

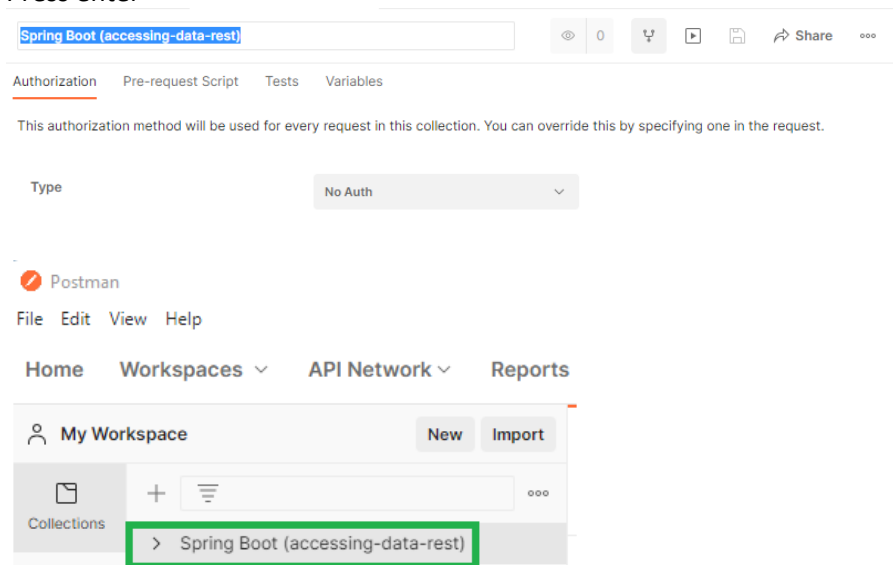


Select "Collection"



Enter "Name": "Spring Boot (accessing-data-rest)"

Press enter



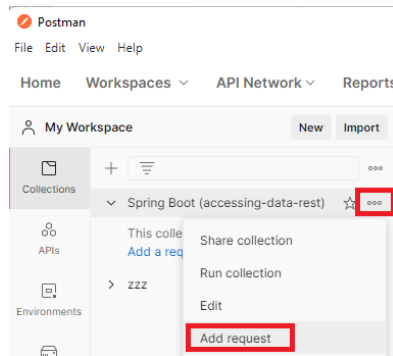
Create Postman HTTP Requests for Project (accessing-data-rest)

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.

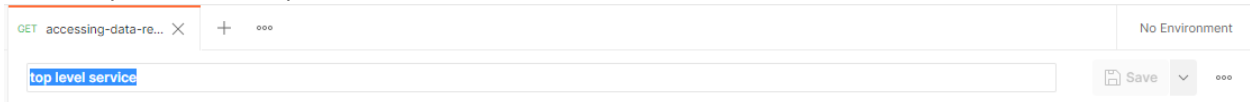
GET Request: top level service

Request to get the top-level service.

Select the menu icon “...” ➔ select “Add request”



Enter request name “top level service”

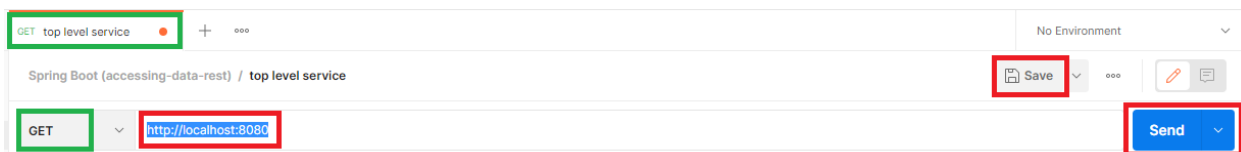


Leave the default request type: “GET”

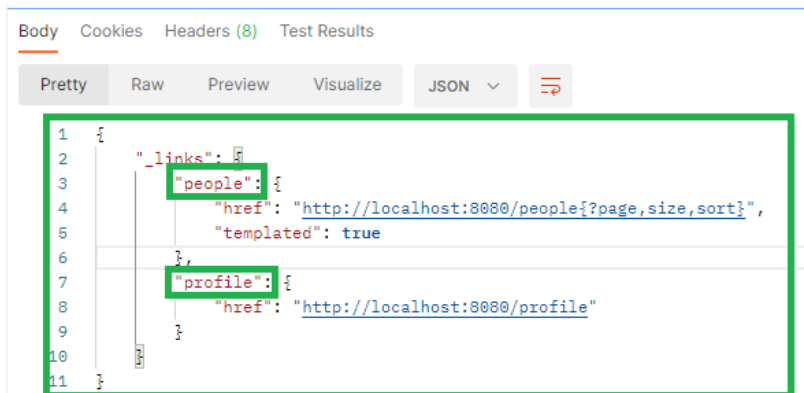
Enter URL: “http://localhost:8080”

Click “Save”

Click “Send”



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.



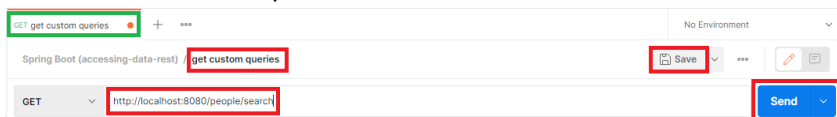
GET Request: custom queries

Request to find all the custom queries defined in the program. In this program the class “PersonRepository” defines a “findByLastName()” query. As mentioned before until now all other requests in this document used the CRUD methods inherited by the interface in the repository class “PagingAndSortingRepository<>”. This custom request still uses CRUD methods and returns a list of values.

Add new GET request name “get custom queries”

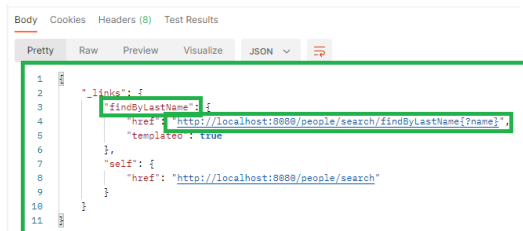
Enter URL: “http://localhost:8080/people/search”

Save then Send the request.



Examine the response:

The response shows the one custom query “findByLastName()” with the URL and expected parameter defined in this program.



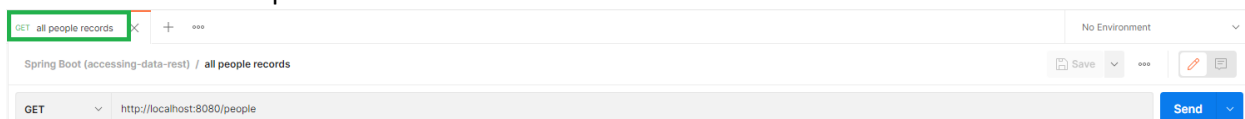
GET Request: all people records

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

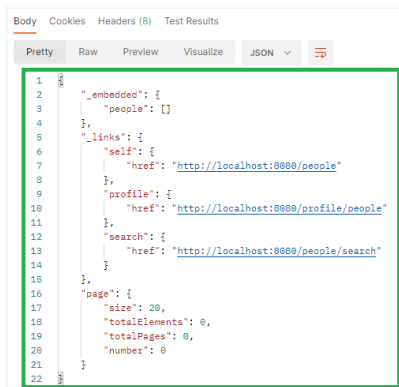
Add new GET request name “all people records”

Enter URL: “http://localhost:8080/people”

Save then Send the request.



Examine the response: since there are no records yet in the memory database, we see no people records. We do see other URL that can be tested.



```
1 {
2   "embedded": {
3     "people": []
4   },
5   "links": {
6     "self": {
7       "href": "http://localhost:8080/people"
8     },
9     "profile": {
10      "href": "http://localhost:8080/profile/people"
11    },
12    "search": {
13      "href": "http://localhost:8080/people/search"
14    }
15  },
16  "page": {
17    "size": 20,
18    "totalElements": 0,
19    "totalPages": 0,
20    "number": 0
21  }
22 }
```

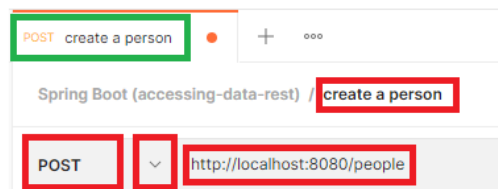
POST Request: create a person

Request to create a person record in the database.

Add new request name “create a person”

Change request type to “POST”

Enter URL: “http://localhost:8080/people”

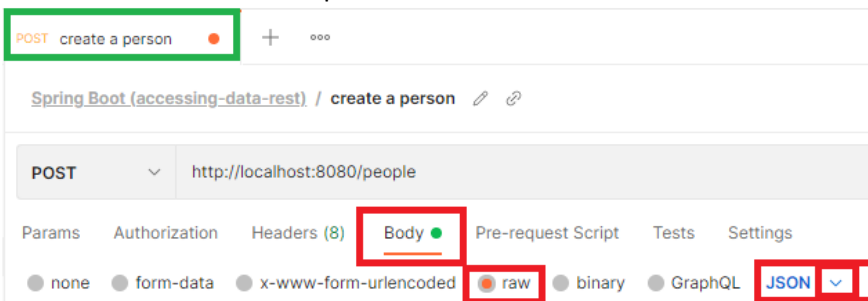


Create POST request body.

Select “Body”

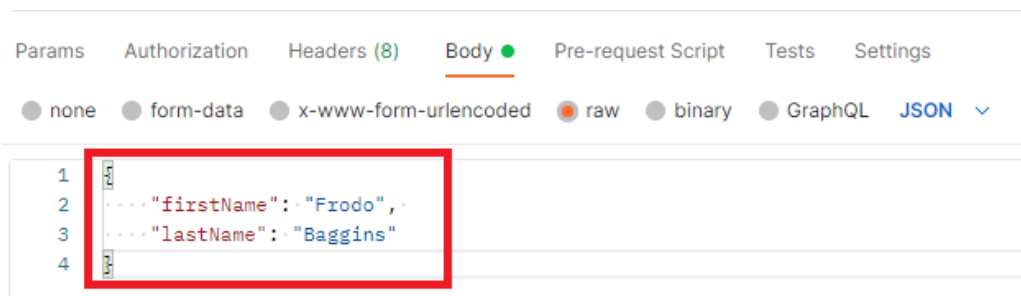
Select “raw”

Select "JSON" from the drop-down



Add the body.

```
{
  "firstName": "Frodo",
  "lastName": "Baggins"
}
```

Save then Send the request.

Examine the response:

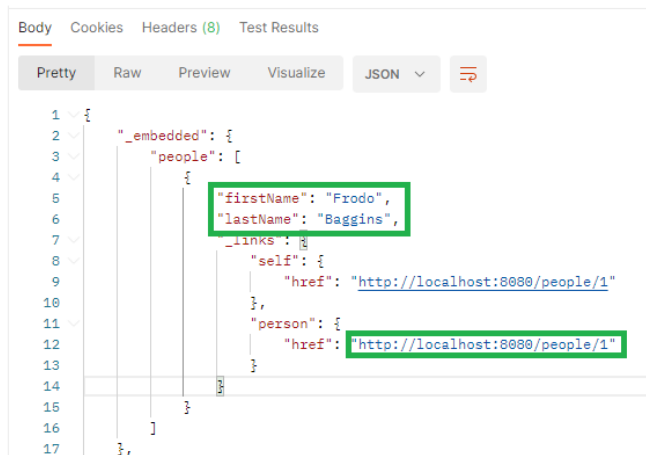
The response echoes back the record created showing a URL to find the record number by its record number 1 in this case.



To create additional records update the first and last name in the request body.

Run GET Request: all people records

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

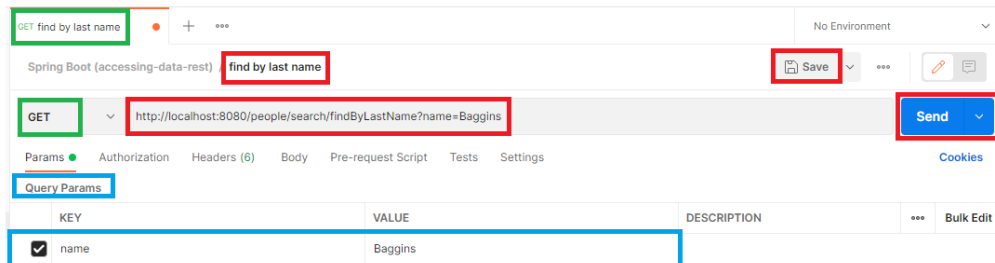


GET Request: custom query find by last name

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

Add new GET request name “find by last name”

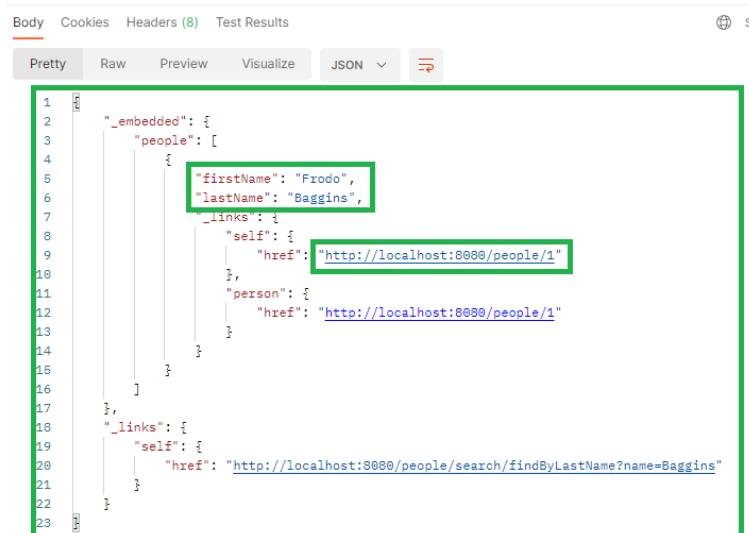
Enter URL: “http://localhost:8080/people/search/findByLastName?name=Baggins”



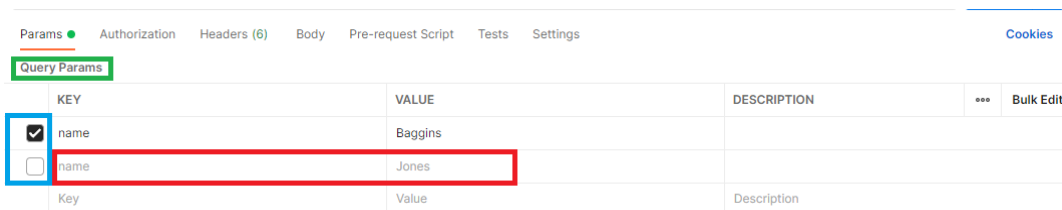
NOTE: “Query Params” with “KEY” of “name” and “VALUE” of “Baggins” is automatically created. Save then Send the request.

Examine the response:

The response should contain a list of all people records with the last name of “Baggins”. In this case just one element is in the list. The document will create another person 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.



When creating this request it was pointed out that “Query Params” were 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.

KEY	VALUE
<input type="checkbox"/> name	Baggins
<input checked="" type="checkbox"/> name	Jones

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.

Add new GET request name “find by record id”

Enter URL: “http://localhost:8080/people/1”

Save then Send the request.

GET find by record id

Spring Boot (accessing-data-rest) / find by record id

GET http://localhost:8080/people/1

Save Send

Examine the response:

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

```
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
```

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

```
1  {
2    "firstName": "Bilbo",
3    "lastName": "Baggins",
4    "links": {
5      "self": {
6        "href": "http://localhost:8080/people/1"
7      },
8      "person": {
9        "href": "http://localhost:8080/people/1"
10     }
11   }
12 }
```

PUT Request: update a people 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.

Add new request name “update a people record”

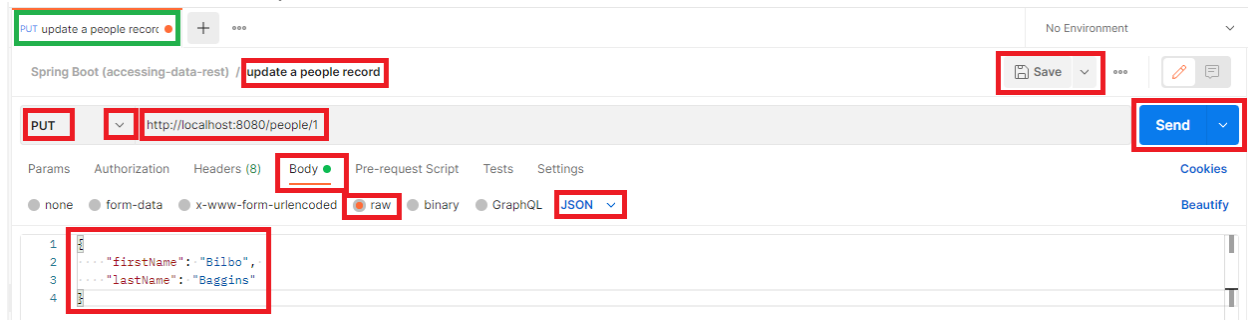
Change request type to “PUT”

Enter URL: “http://localhost:8080/people/1”

Enter request “Body”

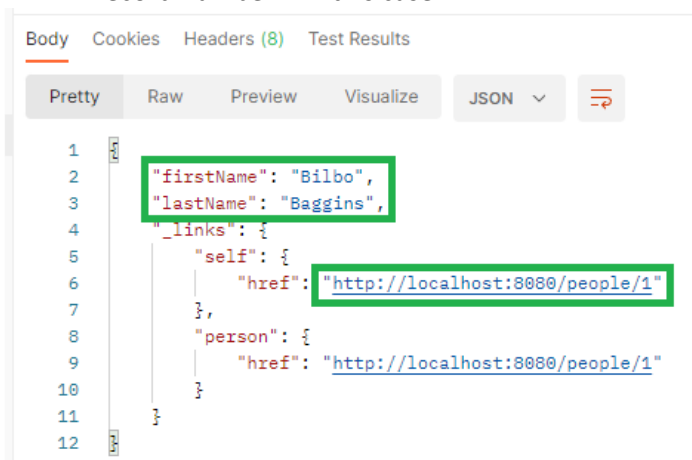
```
{
  "firstName": "Bilbo",
  "lastName": "Baggins"
}
```

Save then 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 people 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.

Add new request name “update a people record field”

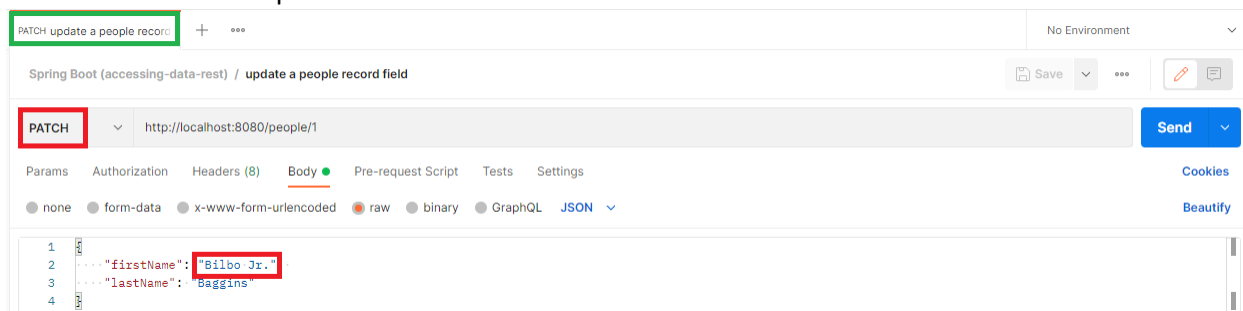
Change request type to “PATCH”

Enter URL: “http://localhost:8080/people/1”

Enter request “Body”

```
{
  "firstName": "Bilbo Jr.",
  "lastName": "Baggins"
}
```

Save then Send the request.



Examine the response:

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



DELETE Request: delete a people record

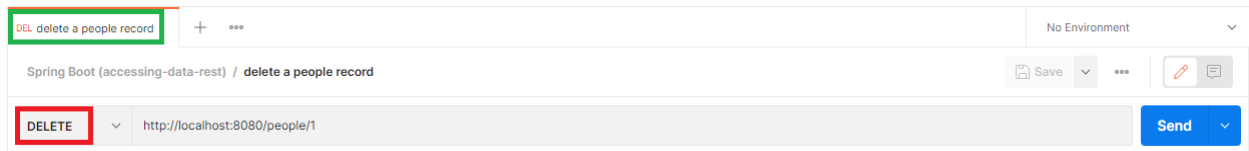
Request to delete an existing record from the database. If the record does not exist, nothing happened, and no error message is seen.

Add new request name “delete a people record”

Change request type to “DELETE”

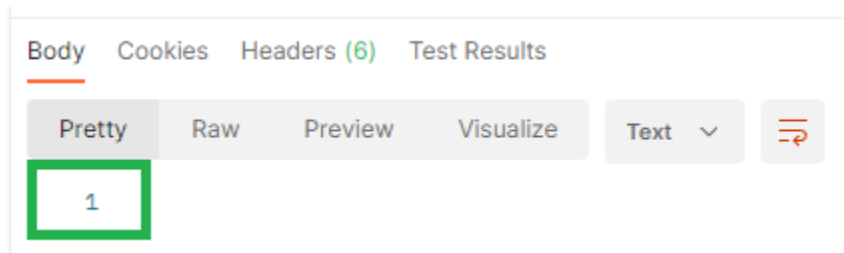
Enter URL: “http://localhost:8080/people/1”

Save then Send the request.

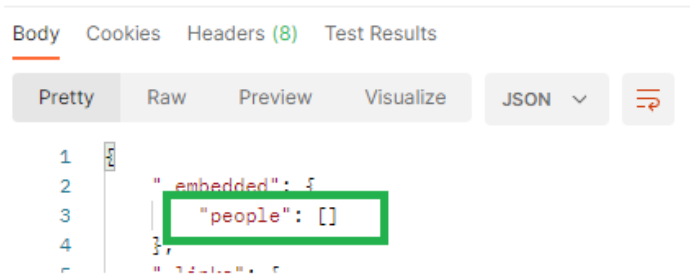


Examine the response:

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-rest) Requests

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

Create People Records

Use the “POST create a person” request with the following bodies to create multiple records for testing.

```
{
  "firstName": "Bilbo",
  "lastName": "Baggins"
}
```

```
{
  "firstName": "Ma",
  "lastName": "Baggins"
}
```

```
{
  "firstName": "Pa",
  "lastName": "Baggins"
}
```

```
{
  "firstName": "Ma & Pa",
  "lastName": "Kettle"
}
```

Retrieve All Records

Use the “GET all people records” request to retrieve record.

Examine the response:

Four records were created. All four 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 “Bilbo Baggins” record was first created and deleted. Once a record is deleted from the database, the record id is not reused.

```
1  {
2    "_embedded": {
3      "people": [
4        {
5          "firstName": "Bilbo",
6          "lastName": "Baggins",
7          "_links": {
8            "self": {
9              "href": "http://localhost:8080/people/2"
10           },
11           "person": {
12             "href": "http://localhost:8080/people/2"
13           }
14         }
15       },
16       {
17         "firstName": "Ma",
18         "lastName": "Baggins",
19         "_links": {
20           "self": {
21             "href": "http://localhost:8080/people/3"
22           },
23           "person": {
24             "href": "http://localhost:8080/people/3"
25           }
26         }
27       },
28       {
29         "firstName": "Pa",
30         "lastName": "Baggins",
31         "_links": {
32           "self": {
33             "href": "http://localhost:8080/people/4"
34           },
35           "person": {
36             "href": "http://localhost:8080/people/4"
37           }
38         }
39       },
40       {
41         "firstName": "Ma & Pa",
42         "lastName": "Kettle",
43         "_links": {
44           "self": {
45             "href": "http://localhost:8080/people/5"
46           },
47           "person": {
48             "href": "http://localhost:8080/people/5"
49           }
50         }
51       }
52     ]
53   },
54 }
```


Retrieve Records by Last Name

Use the “GET find by last name” request to retrieve record.

Examine the response:

Four records were created. Three records are retrieved. Records with last name of “Baggins” are the records retrieved. The record with last name “Kettle” is not retrieved.

```
1  {
2    "_embedded": {
3      "people": [
4        {
5          "firstName": "Bilbo",
6          "lastName": "Baggins",
7          "_links": {
8            "self": {
9              "href": "http://localhost:8080/people/2"
10             },
11            "person": {
12              "href": "http://localhost:8080/people/2"
13            }
14          }
15        },
16        {
17          "firstName": "Ma",
18          "lastName": "Baggins",
19          "_links": {
20            "self": {
21              "href": "http://localhost:8080/people/3"
22            },
23            "person": {
24              "href": "http://localhost:8080/people/3"
25            }
26          }
27        },
28        {
29          "firstName": "Pa",
30          "lastName": "Baggins",
31          "_links": {
32            "self": {
33              "href": "http://localhost:8080/people/4"
34            },
35            "person": {
36              "href": "http://localhost:8080/people/4"
37            }
38          }
39        },
40        {
41          "firstName": "Kettle",
42          "lastName": "Kettle",
43          "_links": {
44            "self": {
45              "href": "http://localhost:8080/people/5"
46            },
47            "person": {
48              "href": "http://localhost:8080/people/5"
49            }
50          }
51        }
52      ]
53    }
54  }
```