

3

Creating a RESTful Web Service with Spring Boot

In this chapter, we will first create a RESTful web service using the controller class. After that, we will demonstrate how to use Spring Data REST to create a RESTful web service that also covers all CRUD functionalities automatically. We are using the database application that we created in the previous chapter as a starting point.

In this chapter, we will look into the following:

- What the RESTful web service is
- How to create a RESTful web service with Spring Boot
- How to test the RESTful web service

Technical requirements

The Spring Boot application that has been created in previous chapters is necessary.

Postman, cURL, or another suitable tool for transferring data using various HTTP methods is necessary.

Creating a RESTful web service with Spring Boot

Web services are applications that communicate over the internet using the HTTP protocol. There are many different types of web service architectures, but the main idea across all designs is the same. In this book, we are creating a RESTful web service from what is a really popular design nowadays.

Basics of REST

REST (Representational State Transfer) is an architectural style for creating web services. REST is not standard, but it defines a set of constraints defined by Roy Fielding. The six constraints are the following:

- **Stateless:** The server doesn't hold any information about the client state.
- **Client server:** The client and server act independently. The server does not send any information without a request from the client.
- **Cacheable:** Many clients often request the same resources, therefore it is useful to cache responses in order to improve performance.
- **Uniform interface:** Requests from different clients look the same. Clients may be, for example, a browser, a Java application, and a mobile application.
- **Layered system:** REST allows us to use a layered system architecture.
- **Code on demand:** This is an optional constraint.

The uniform interface is an important constraint and it defines that every REST architecture should have following elements:

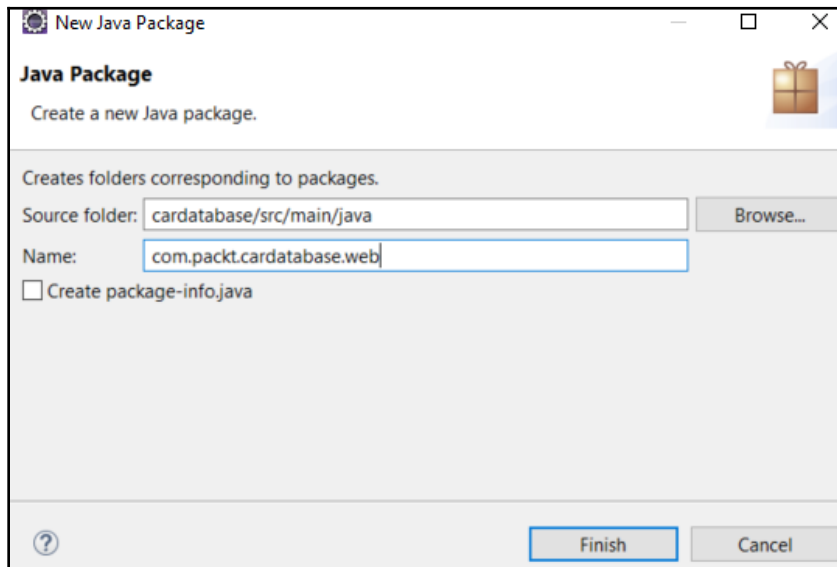
- **Identification of resources:** There are resources with their unique identifiers, for example, URIs in web-based REST services. REST resources should expose easily understood directory structure URIs. Therefore, a good resource naming strategy is very important.
- **Resource manipulation through representation:** When making a request to a resource, the server responds with a representation of the resource. Typically, the format of the representation is JSON or XML.
- **Self descriptive messages:** Messages should have enough information that the server knows how to process them.
- **Hypermedia and the Engine of Application State (HATEOAS):** Responses can contain links to other areas of service.

The RESTful web service that we are going to develop in the following topics follows the REST architectural principles.

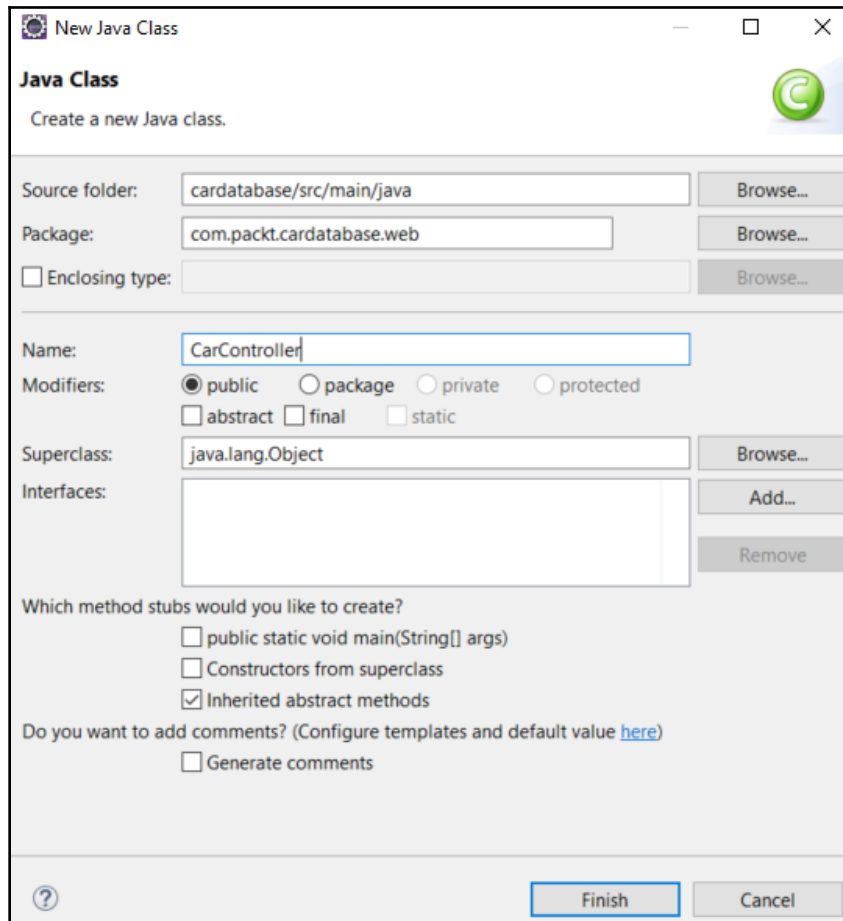
Creating a RESTful web service

In Spring Boot, all the HTTP requests are handled by controller classes. To be able to create a RESTful web service, first, we have to create a controller class. We will create our own Java package for our controller:

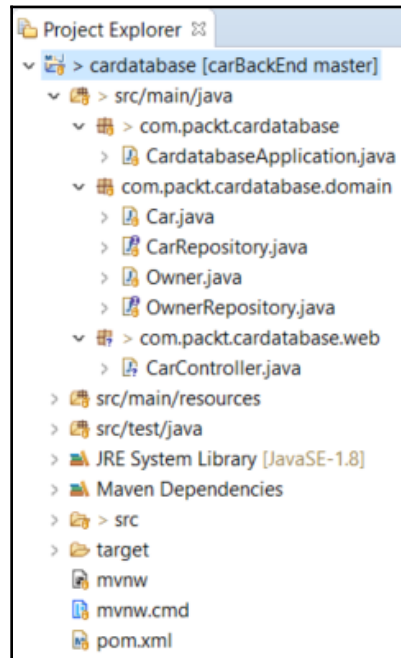
1. Activate the root package in the Eclipse **Project Explorer** and right-click. Select **New | Package** from the menu. We will name our new package `com.packt.cardatabase.web`:



2. Next, we will create a new controller class in a new web package. Activate the `com.packt.cardatabase.web` package in the Eclipse project explorer and right-click. Select **New** | **Class** from the menu. We will name our class `CarController`:



3. Now, your project structure should look like the following screenshot:



If you create classes in a wrong package accidentally, you can drag and drop the files between packages in the Eclipse **Project Explorer**. Sometimes, the Eclipse **Project Explorer** view might not be rendered correctly when you make some changes. Refreshing the project explorer helps (Activate **Project Explorer** and press *F5*).

4. Open your controller class in the editor window and add the `@RestController` annotation before the class definition. See the following source code. The `@RestController` annotation identifies that this class will be the controller for the RESTful web service:

```
package com.packt.cardatabase.web;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class CarController {
}
```

5. Next, we add a new method inside our controller class. The method is annotated with the `@RequestMapping` annotation, which defines the endpoint that the method is mapped to. Following, you can see the sample source code. In this example, when a user navigates to the `/cars` endpoint, the `getCars()` method is executed:

```
package com.packt.cardatabase.web;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class CarController {
    @RequestMapping("/cars")
    public Iterable<Car> getCars() {
    }
}
```

The `getCars()` method returns all the car objects, which are then marshalled to JSON objects by Jackson library.



By default, `@RequestMapping` handles all the HTTP method (GET, PUT, POST, and more) requests. You can define which method is accepted with the following `@RequestMapping("/cars", method=GET)` parameter. Now, this method handles only GET requests from the `/cars` endpoint.

6. To be able to return cars from the database, we have to inject our `CarRepository` into the controller. Then, we can use the `findAll()` method that the repository provides to fetch all cars. The following source code shows the controller code:

```
package com.packt.cardatabase.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.packt.cardatabase.domain.Car;
import com.packt.cardatabase.domain.CarRepository;


@RestController
public class CarController {
    @Autowired
    private CarRepository repository;
    @RequestMapping("/cars")
    public Iterable<Car> getCars() {
        return repository.findAll();
    }
}
```

7. Now, we are ready to run our application and navigate to `localhost:8080/cars`. We can see that there is something wrong, and the application seems to be in an infinite loop. That happens due to our one-to-many relationship between the car and owner tables. So, what happens in practice—first, the car is serialized, and it contains an owner that is then serialized, and that, in turn, contains cars that are then serialized... and so on. To avoid this, we have to add the `@JsonIgnore` annotation to the `cars` field in the `Owner` class:

```
// Owner.java

@OneToMany(cascade = CascadeType.ALL, mappedBy="owner")
@JsonIgnore
private List<Car> cars;
```

8. Now, when you run the application and navigate to `localhost:8080/cars`, everything should go as expected and you will get all the cars from the database in JSON format, as shown in the following screenshot:



```
[{"brand": "Ford", "model": "Mustang", "color": "Red", "registerNumber": "ADF-1121", "year": 2017, "price": 59000, "owner": {"ownerid": 1, "firstname": "John", "lastname": "Johnson"}}, {"brand": "Nissan", "model": "Leaf", "color": "White", "registerNumber": "SSJ-3002", "year": 2014, "price": 29000, "owner": {"ownerid": 2, "firstname": "Mary", "lastname": "Robinson"}}, {"brand": "Toyota", "model": "Prius", "color": "Silver", "registerNumber": "KKO-0212", "year": 2018, "price": 39000, "owner": {"ownerid": 2, "firstname": "Mary", "lastname": "Robinson"}}]
```

We have done our first RESTful web service, which return all the cars. Spring Boot provides a much more powerful way of creating RESTful Web Services and this is investigated in the next topic.

Using Spring Data REST

Spring Data REST is part of the Spring Data project. It offers an easy and fast way to implement RESTful Web Services with Spring. To start, using Spring Data REST you have to add the following dependency to the `pom.xml` file:

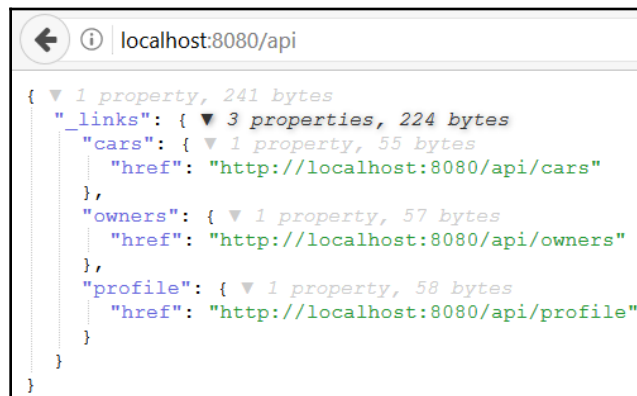
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

By default, Spring Data REST finds all public repositories from the application and creates automatically RESTful Web Services for your entities.

You can define the endpoint of service in your `application.properties` file:

```
spring.data.rest.basePath=/api
```

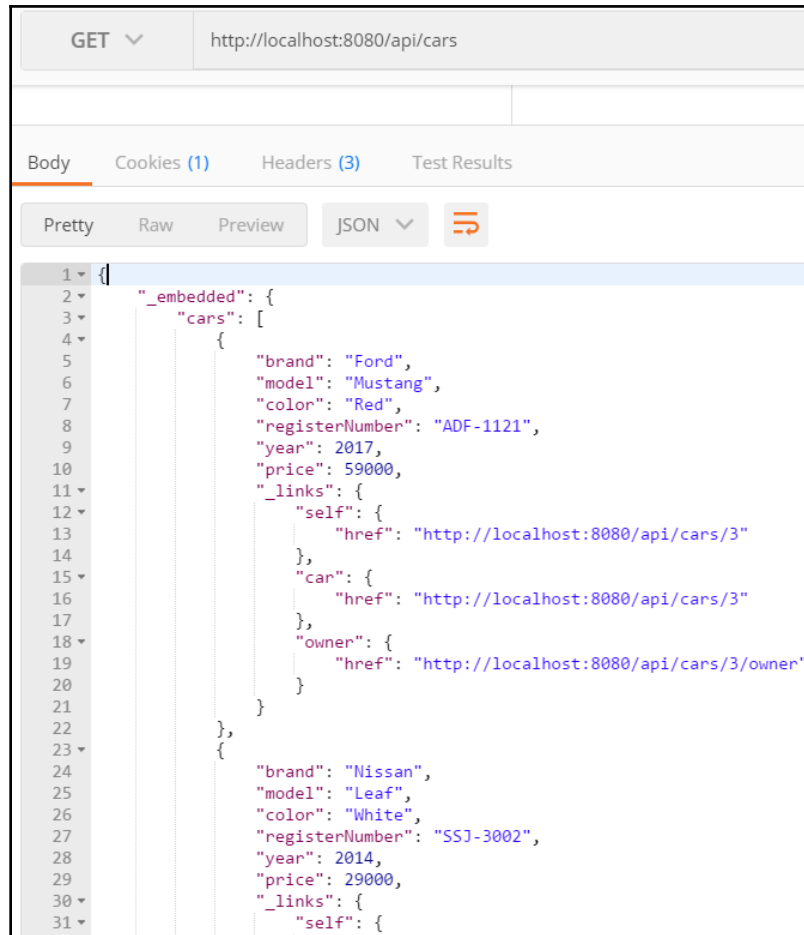
Now you can access the RESTful web service from the `localhost:8080/api` endpoint. By calling the root endpoint of the service it returns the resources that are available. Spring Data REST returns JSON data in the **HAL (Hypertext Application Language)** format. The HAL format provides a set of conventions for expressing hyperlinks in JSON and it makes your RESTful web service easier to use for frontend developers:



We can see that there are links to the car and owner entity services. The Spring Data Rest service pathname is derived from the entity name. The name will then be pluralized and uncapitalized. For example, the entity Car service pathname will be named `cars`. The profile link is generated by Spring Data Rest and it contains application-specific metadata.

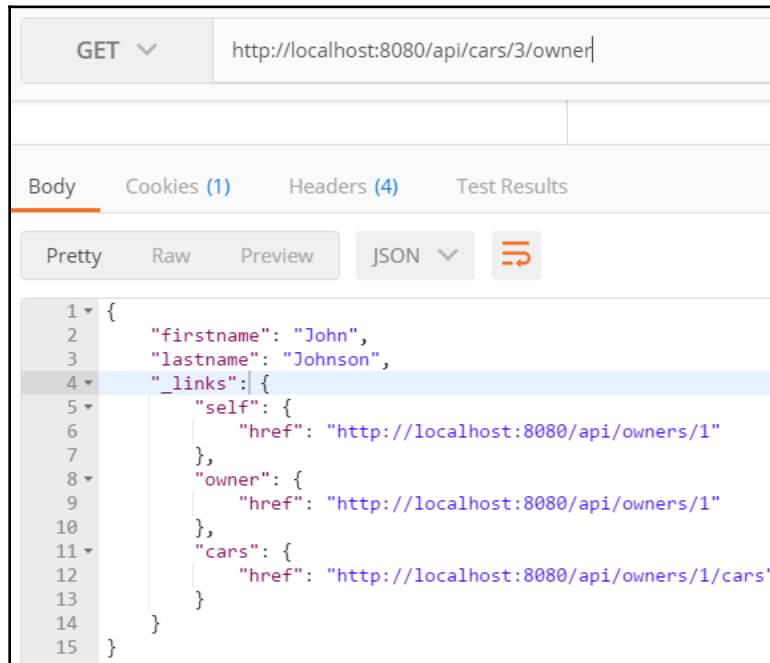
Now, we start to examine different services more carefully. There are multiple tools available for testing and consuming RESTful Web Services. In this book, we are using Postman, but you can use tools that you are familiar with, such as cURL. Postman can be acquired as a desktop application or as a browser plugin. cURL is also available for Windows 10 by using Windows Ubuntu Bash.

If you make a request to the `cars` endpoint `http://localhost:8080/api/cars` using the `GET` method, you will get a list of all the `cars`, as shown in the following screenshot:



In the JSON response, you can see that there is an array of `cars` and each car contains car specific data. All the cars also have the `"_links"` attribute, which is a collection of links, and with these you can access the car itself or get the owner of the car. To access one specific car, the path will be `http://localhost:8080/api/cars/{id}`.

The request to `http://localhost:8080/api/cars/3/owner` returns the owner of the car. The response now contains owner data, a link to the owner, and links to other `cars` that the user owns:



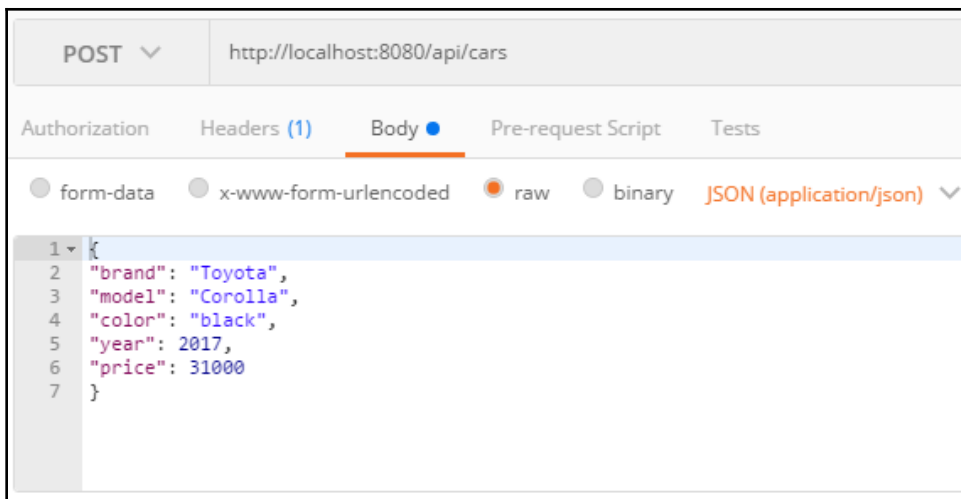
Spring Data Rest service provides all CRUD operations. The following table shows which HTTP methods you can use for different CRUD operations:

HTTP Method	CRUD
GET	Read
POST	Create
PUT/PATCH	Update
DELETE	Delete

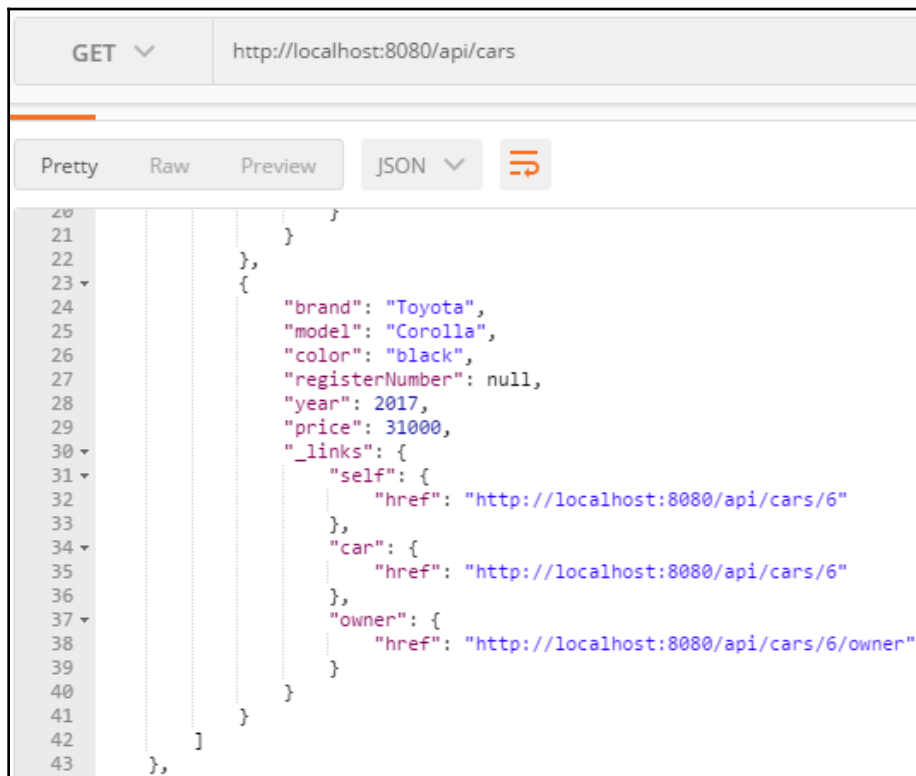
Next, we will look at how to delete a car from the database by using our RESTful web service. In a delete operation, you have to use the `DELETE` method and the link to the car that will be deleted (`http://localhost:8080/api/cars/{id}`). The following screenshot shows how you can delete one car with the ID 4 by using `cURL`. After the delete request, you can see that there are now two cars left in the database:

```
root@HMMX4717: ~  
root@HMMX4717:~# curl -X "DELETE" http://localhost:8080/api/cars/4
```

When we want to add a new car to the database, we have to use the `POST` method, and the link is `http://localhost:8080/api/cars`. The header must contain the **Content-Type** field with the value `Content-Type: application/json`, and the new car object will be embedded in the request body:



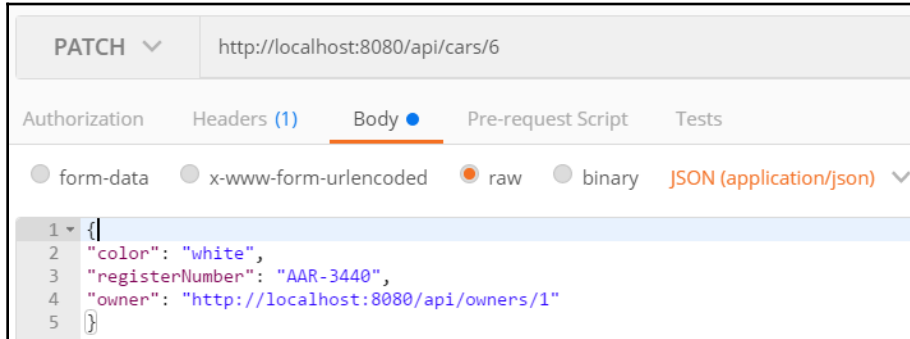
The response will send a newly created car object back. Now, if you again make a `GET` request to the `http://localhost:8080/api/cars` path, you can see that the new car exists in the database:



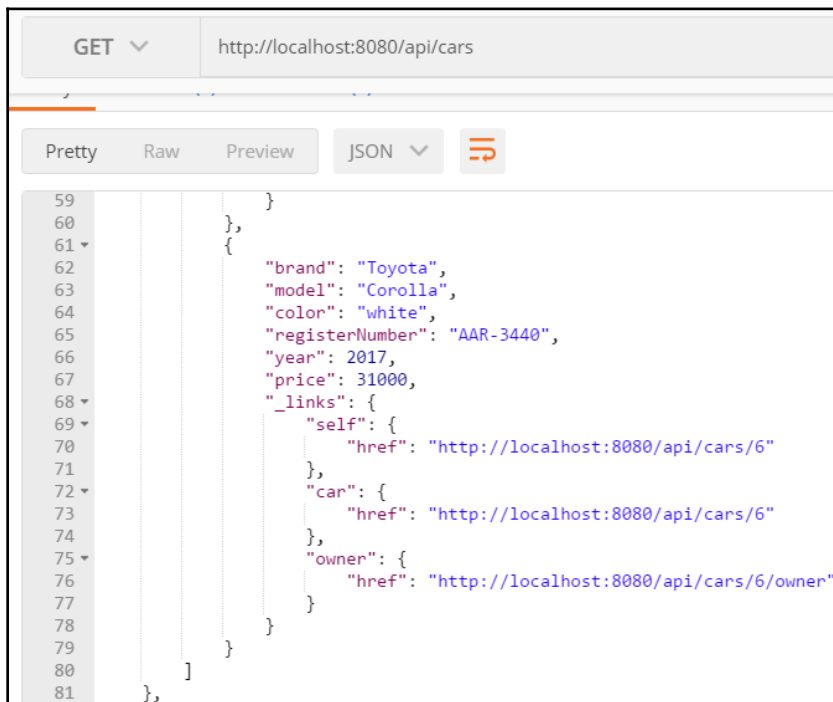
```
20 }
21 }
22 },
23 {
24   "brand": "Toyota",
25   "model": "Corolla",
26   "color": "black",
27   "registerNumber": null,
28   "year": 2017,
29   "price": 31000,
30   "_links": {
31     "self": {
32       "href": "http://localhost:8080/api/cars/6"
33     },
34     "car": {
35       "href": "http://localhost:8080/api/cars/6"
36     },
37     "owner": {
38       "href": "http://localhost:8080/api/cars/6/owner"
39     }
40   }
41 }
42 ],
43 },
```

To update entities, we have to use the `PATCH` method and the link to the car that we want to update (`http://localhost:8080/api/cars/{id}`). The header must contain the **Content-Type** field with the value `Content-Type: application/json` and the car object, with edited data, will be given inside the request body. If you are using `PATCH`, you have to send only fields that are updates. If you are using `PUT`, you have to include all fields to request. Let's edit our car that we created in the previous example. We will change the color to white and fill in the register number that we left empty.

We will also link an owner to the car by using the owner field. The content of the owner field is the link to the owner (<http://localhost:8080/api/owners/{id}>). The following screenshot shows the PATCH request content:



You can see that the car is updated after you fetch all cars by using GET request:



In the previous chapter, we created queries to our repository. These queries can also be included in our service. To include queries, you have to add the `@RepositoryRestResource` annotation to the repository class. Query parameters are annotated with the `@Param` annotation. The following source code shows our `CarRepository` with these annotations:

```
package com.packt.cardatabase.domain;

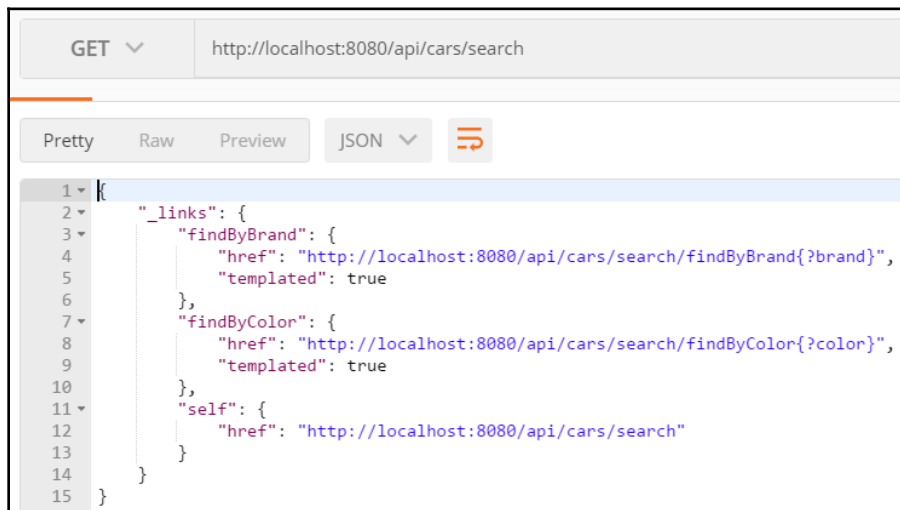
import java.util.List;

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

@RepositoryRestResource
public interface CarRepository extends CrudRepository <Car, Long> {
    // Fetch cars by brand
    List<Car> findByBrand(@Param("brand") String brand);

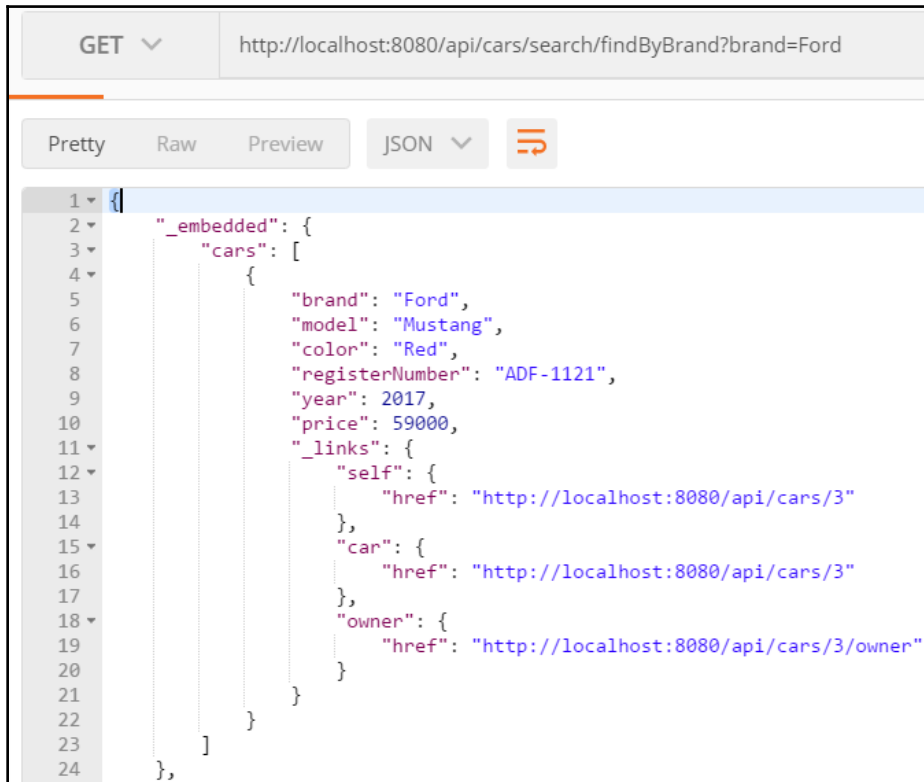
    // Fetch cars by color
    List<Car> findByColor(@Param("color") String color);
}
```

Now, when you make a GET request to the `http://localhost:8080/api/cars` path, you can see that there is a new endpoint called `/search`. Calling the `http://localhost:8080/api/cars/search` path returns the following response:



From the response, you can see that both queries are now available in our service. The following URL demonstrates how to fetch cars by brand:

`http://localhost:8080/api/cars/search/findByBrand?brand=Ford`



The screenshot shows a REST client interface with a GET request to `http://localhost:8080/api/cars/search/findByBrand?brand=Ford`. The response is displayed in JSON format, showing a list of cars with details like brand, model, color, register number, year, price, and links to self, car, and owner.

```
1 {
2   "_embedded": {
3     "cars": [
4       {
5         "brand": "Ford",
6         "model": "Mustang",
7         "color": "Red",
8         "registerNumber": "ADF-1121",
9         "year": 2017,
10        "price": 59000,
11        "_links": {
12          "self": {
13            "href": "http://localhost:8080/api/cars/3"
14          },
15          "car": {
16            "href": "http://localhost:8080/api/cars/3"
17          },
18          "owner": {
19            "href": "http://localhost:8080/api/cars/3/owner"
20          }
21        }
22      }
23    ]
24  },
25}
```

Summary

In this chapter, we created a RESTful web service with Spring Boot. First, we created a controller and one method that returns all cars in JSON format. Next, we used Spring Data REST to get a fully functional web service with all CRUD functionalities. We covered different types of requests that are needed to use CRUD functionalities of the service that we created. Finally, we also included our queries to service. In the next chapter we will secure our backend using Spring Security.

Questions

1. What is REST?
2. How can you create a RESTful web service with Spring Boot?
3. How can you fetch items using our RESTful web service?
4. How can you delete items using our RESTful web service?
5. How can you add items using our RESTful web service?
6. How can you update items using our RESTful web service?
7. How can you use queries with our RESTful web service?

Further reading

Pack has other great resources for learning about Spring Boot RESTful Web Services:

- <https://www.packtpub.com/application-development/learning-spring-boot-20-second-edition>
- <https://www.packtpub.com/web-development/spring-boot-getting-started-integrated-course>
- <https://www.packtpub.com/web-development/building-restful-web-service-spring>