Spring Boot REST Services using JPA Part 1

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: Create a simple payroll service that manages the employees of a company. We'll store employee objects in a (H2 in-memory) database, and access them (via something called JPA). Then we'll wrap that with something that will allow access over the internet (called the Spring MVC layer). This project is based on Building REST services with Spring 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 Technology:
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: "payroll" 3
Project Payroll Discussion 4
Main Class: PayrollApplication 4
Create Domain Object Model / Entity Class 4
Create Model Class: Employee 4
Copy and paste source code for class: Employee 5
Create Repository Class 6
Create Repository Interface: EmployeeRepository 6
Copy and paste source code for class: EmployeeRepository 6
Create Helper Class 7
Create Helper Class: LoadDatabase 7
Copy and paste source code for class: LoadDatabase 7
Create Exception Classes 8

Create Exception Class: EmployeeNotFoundException 8
Copy and paste source code for class: EmployeeNotFoundException 8
Create Exception Class: EmployeeNotFoundAdvice 8
Copy and paste source code for class: EmployeeNotFoundAdvice 9
Create Controller Class 10
Create Controller Class: EmployeeController 10
Copy and paste source code for class: EmployeeController 10
Test the Project Using Spring Boot Tomcat Server 11
Test the Project Using Postman 11
Create Postman HTTP Requests for Project (payroll) 11
GET Request: all employee records 11
GET Request: find by employee record id 12
POST Request: create an employee 14
PUT Request: update an existing employee record 14
DELETE Request: delete an employee record 15
Project Conclusion 17

Glossary of Terminology

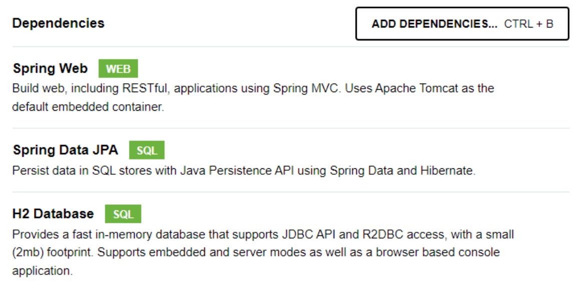
For a list of key terms and definitions used throughout this and various Spring Boot demo documents see the document titled "Appendix 01 Glossary".

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:

Spring Web
Spring Data JPA
H2 Database



Spring Web
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

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

"**Group**" use "**com.example**"
"**Artifact**" use "**payroll**"
"**Name**" use "**payroll**"
"**Package name**" use "**com.example.payroll**"

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: "payroll"

Follow the instructions in the document title "Appendix 03 Import Spring Tool Suite Project" and import the "**payroll**" project that was created with Spring Initializr.

Project Payroll Discussion

This project builds on the knowledge from other projects using these documents. A basic understanding of using the Spring Tool Suite 4 for creating program element, resolving import issue, and other items is assumed.

The project will be a simple payroll application using restful APIs, an H2 memory database, and JPA for database access.

Main Class: PayrollApplication

This project uses the generated application without modification to start the applications.

The PayrollApplication 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.

```
package com.example.payroll;

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

@SpringBootApplication
public class PayrollApplication {

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

}
```

Create Domain Object Model / Entity Class

Create a Domain Object also known as Model / Entity 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 Model Class: Employee

Create class named "Employee" in package "com.example.payroll". This class is an entity class mapping to the H2 memory database. It generates unique record id, contains an employee name and their role. The class has getters and setters along with several override methods.

- `"@Entity` is a JPA annotation to make this object ready for storage in a JPA-based data store.

- `id`, `name`, and `role` are attributes of our Employee domain object. `id` is marked with more JPA annotations to indicate it's the primary key and automatically populated by the JPA provider.
- a custom constructor is created when we need to create a new instance, but don't yet have an id."

-- Building REST services with Spring.

Copy and paste source code for class: Employee

Copy and replace the code shell in its entirety.

```java
package com.example.payroll;

import java.util.Objects;

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

@Entity
class Employee {

  private @Id @GeneratedValue Long id;
  private String name;
  private String role;

  Employee() {}

  Employee(String name, String role) {

    this.name = name;
    this.role = role;
  }

  public Long getId() {
    return this.id;
  }

  public String getName() {
    return this.name;
  }

  public String getRole() {
    return this.role;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public void setName(String name) {
    this.name = name;
  }

  public void setRole(String role) {
    this.role = role;
  }

  @Override
  public boolean equals(Object o) {

    if (this == o)
      return true;
    if (!(o instanceof Employee))
      return false;
    Employee employee = (Employee) o;
    return Objects.equals(this.id, employee.id) && Objects.equals(this.name, employee.name)
        && Objects.equals(this.role, employee.role);
  }

  @Override
  public int hashCode() {
    return Objects.hash(this.id, this.name, this.role);
  }

  @Override
  public String toString() {
    return "Employee{" + "id=" + this.id + ", name='" + this.name + '\'' + ", role='" + this.role + '\'' + '}';
  }
}
```

Create Repository Class

Create a repository interface that extends JpaRepository<>. There are no custom employee methods to access the memory database. All access is through CRUD methods defined in the extended JpaRepository<> class.

Create Repository Interface: EmployeeRepository

Create interface named "EmployeeRepository" in package "com.example.payroll" that extends JpaRepository<Employee, Long>. This interface allows creating, reading, updating, and deleting employee data in the memory database.

"To get all this free functionality, all we had to do was declare an interface which extends Spring Data JPA's `JpaRepository`, specifying the domain type as `Employee` and the id type as `Long`.

Spring Data's repository solution makes it possible to sidestep data store specifics and instead solve a majority of problems using domain-specific terminology."

-- Building REST services with Spring.

Copy and paste source code for class: EmployeeRepository

Copy and replace the code shell in its entirety.

```
package com.example.payroll;

import org.springframework.data.jpa.repository.JpaRepository;

interface EmployeeRepository extends JpaRepository<Employee, Long> {

}
```

Create Helper Class

Create a helper class to initially load the memory database.

Create Helper Class: LoadDatabase

Create class named “LoadDatabase” in package “com.example.payroll”. This class is a helper class performing an initial load in the H2 memory database. This class has a logger, @Configuration annotation and contains a bean.

@Configuration annotation indicates that the class has @Bean definition methods. The Spring container can process the class and generate Spring Beans to be used in the application.

Copy and paste source code for class: LoadDatabase

Copy and replace the code shell in its entirety.

```java
package com.example.payroll;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
class LoadDatabase {

  private static final Logger log = LoggerFactory.getLogger(LoadDatabase.class);

  @Bean
  CommandLineRunner initDatabase(EmployeeRepository repository) {

    return args -> {
      log.info("Preloading " + repository.save(new Employee("Bilbo Baggins", "burglar")));
      log.info("Preloading " + repository.save(new Employee("Frodo Baggins", "thief")));
    };
  }
}
```

Create Exception Classes

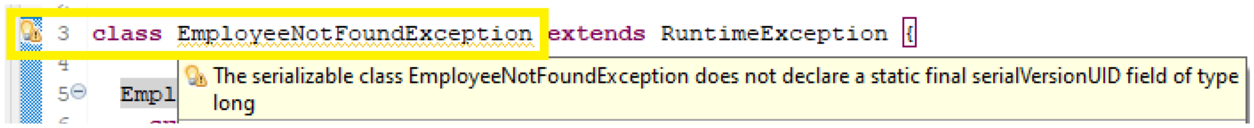
Create customized application exception classes to gracefully handle certain runtime exceptions.

Create Exception Class: EmployeeNotFoundException

Create class named “EmployeeNotFoundException” in package “com.example.payroll”. This class extends RuntimeException and provides a custom exception for when an employee record is not found.

Copy and paste source code for class: EmployeeNotFoundException

Copy and replace the code shell in its entirety. The code copied from the website produced a warning. To resolve the warning the serialVersionUID class variable was created.



```
class EmployeeNotFoundException extends RuntimeException {
```

The serializable class EmployeeNotFoundException does not declare a static final serialVersionUID field of type long

```
package com.example.payroll;

class EmployeeNotFoundException extends RuntimeException {
        private static final long serialVersionUID = 1L;

EmployeeNotFoundException(Long id) {
    super("Could not find employee " + id);
  }
}
```

Create Exception Class: EmployeeNotFoundAdvice

Create class named “EmployeeNotFoundAdvice” in package “com.example.payroll”. This class is used to render an HTTP 404 error.

@ControllerAdvice which allows to handle exceptions across the whole application in one global handling component.

- “`@ResponseBody` signals that this advice is rendered straight into the response body.
- `@ExceptionHandler` configures the advice to only respond if an `EmployeeNotFoundException` is thrown.
- `@ResponseStatus` says to issue an `HttpStatus.NOT_FOUND`, i.e. an **HTTP 404**.
- The body of the advice generates the content. In this case, it gives the message of the exception.”

-- Building REST services with Spring.

Copy and paste source code for class: EmployeeNotFoundAdvice

Copy and replace the code shell in its entirety.

```
package com.example.payroll;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
class EmployeeNotFoundAdvice {

  @ResponseBody
  @ExceptionHandler(EmployeeNotFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  String employeeNotFoundHandler(EmployeeNotFoundException ex) {
    return ex.getMessage();
  }
}
```

Create Controller Class

Create a controller class as part of the Model View Controller (MVC) architecture. The Model class Employee and associated repository class was created in a previous section. The View would be a web page accessing the application. In this case Postman will be used as the View component. The Controller contains the entry point, the URL mapping into the application.

Create Controller Class: EmployeeController

Create class named “EmployeeController” in package “com.example.payroll”.

Copy and paste source code for class: EmployeeController

Copy and replace the code shell in its entirety.

```java
package com.example.payroll;

import java.util.List;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
class EmployeeController {

  private final EmployeeRepository repository;

  EmployeeController(EmployeeRepository repository) {
    this.repository = repository;
  }


  // Aggregate root
  // tag::get-aggregate-root[]
  @GetMapping("/employees")
  List<Employee> all() {
    return repository.findAll();
  }
  // end::get-aggregate-root[]

  @PostMapping("/employees")
  Employee newEmployee(@RequestBody Employee newEmployee) {
    return repository.save(newEmployee);
  }

  // Single item

  @GetMapping("/employees/{id}")
  Employee one(@PathVariable Long id) {

    return repository.findById(id)
      .orElseThrow(() -> new EmployeeNotFoundException(id));
  }

  @PutMapping("/employees/{id}")
  Employee replaceEmployee(@RequestBody Employee newEmployee, @PathVariable Long id) {

    return repository.findById(id)
      .map(employee -> {
        employee.setName(newEmployee.getName());
        employee.setRole(newEmployee.getRole());
        return repository.save(employee);
      })
      .orElseGet(() -> {
        newEmployee.setId(id);
        return repository.save(newEmployee);
      });
  }

  @DeleteMapping("/employees/{id}")
  void deleteEmployee(@PathVariable Long id) {
    repository.deleteById(id);
    }
  }
```

Test the Project Using 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.
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 other project documents using Postman. 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-based version for "localhost" testing.

Create a Postman Collection: Spring Boot (payroll)

Create Postman HTTP Requests for Project (payroll)

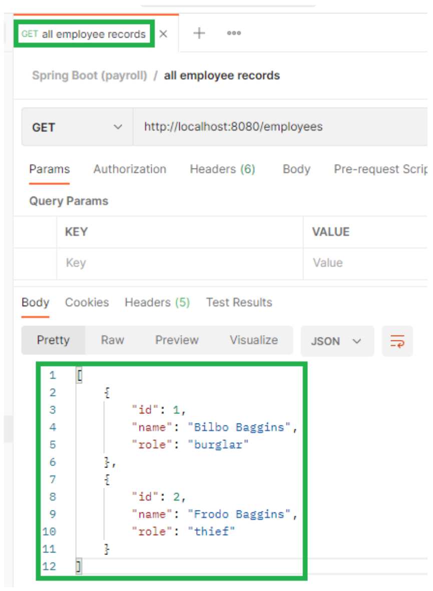
This section describes a collection of various HTTP request for testing this project. While creating the requests in this section keep in mind that no custom queries were created. All endpoints will use the default CRUD methods from the extended class JpaRepository.

GET Request: all employee records

Request to get the list of all employee records in the database.
URL: "http://localhost:8080/employees"
This request maps to the Controller class EmployeeController endpoint:

```
  @GetMapping("/employees")
  List<Employee> all() {
    return repository.findAll();
  }
```

Examine the response: two records were preloaded in the memory database by class LoadDatabase. The response shows the records preloaded.



URL: http://localhost:8080/employees

```
[
    {
        "id": 1,
        "name": "Bilbo Baggins",
        "role": "burglar"
    },
    {
        "id": 2,
        "name": "Frodo Baggins",
        "role": "thief"
    }
]
```

GET Request: find by employee record id

Define a query request to find an employee record by the unique record id. In this request there is "/id" notation. This notation is called a path parameter.
"http://localhost:8080/employees/#"
This request maps to the Controller class EmployeeController endpoint:

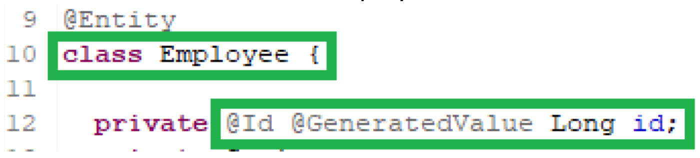
```
  @GetMapping("/employees/{id}")
  Employee one(@PathVariable Long id) {

    return repository.findById(id)
      .orElseThrow(() -> new EmployeeNotFoundException(id));
  }
```

GET Request: find by employee record id success

URL: "http://localhost:8080/employees/1"

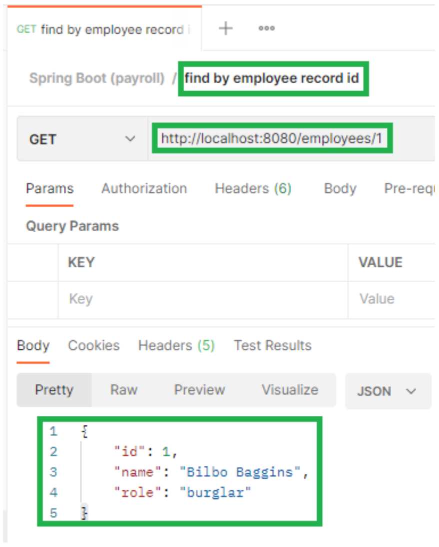
Examine the response:

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



```
@Entity
class Employee {

  private @Id @GeneratedValue Long id;
```

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


GET find by employee record
Spring Boot (payroll) / find by employee record id
GET
Params Authorization Headers (6) Body Pre-req
Query Params
KEY VALUE
Key Value
Body Cookies Headers (5) Test Results
Pretty Raw Preview Visualize JSON

```
http://localhost:8080/employees/1
```

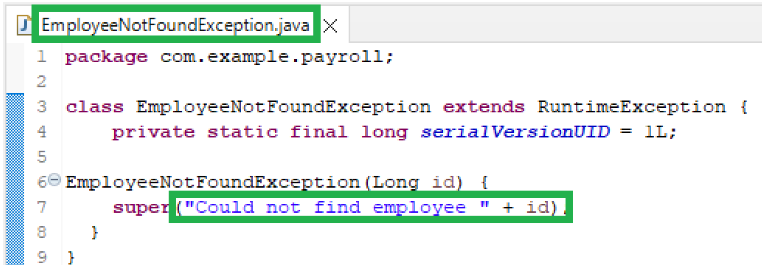
```
{
    "id": 1,
    "name": "Bilbo Baggins",
    "role": "burglar"
}
```

GET Request: find by employee record id not found

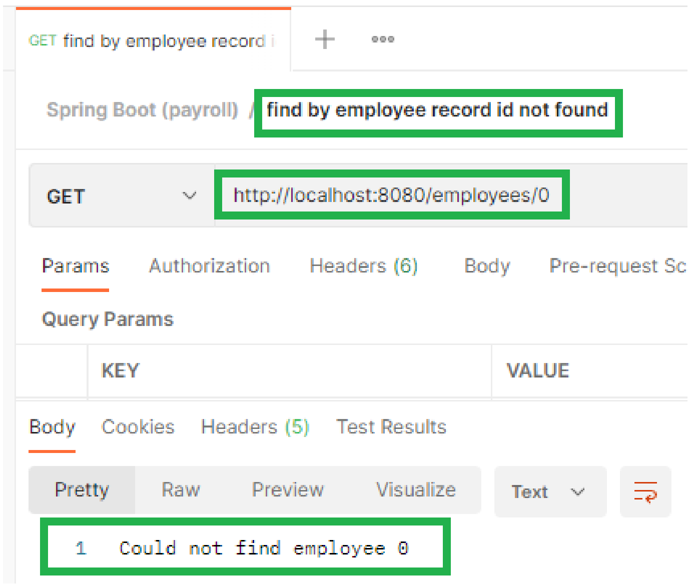
URL: "http://localhost:8080/employees/0"

Examine the response:

The message received in the response was generated by the exception class "EmployeeNotFoundException".



```
package com.example.payroll;

class EmployeeNotFoundException extends RuntimeException {
    private static final long serialVersionUID = 1L;

EmployeeNotFoundException(Long id) {
    super("Could not find employee " + id);
  }
}
```


GET find by employee record i
Spring Boot (payroll) / find by employee record id not found
GET
Params Authorization Headers (6) Body Pre-request Sc
Query Params
KEY VALUE
Body Cookies Headers (5) Test Results
Pretty Raw Preview Visualize Text

```
http://localhost:8080/employees/0
```

```
Could not find employee 0
```

POST Request: create an employee

Request to create an employee record in the database.
URL: “http://localhost:8080/employees”
Request body:

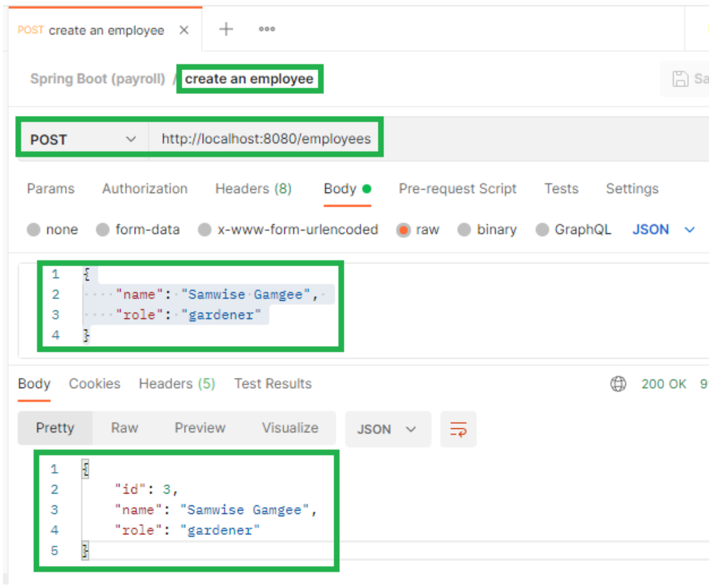
```
{
    "name": "Samwise Gamgee",
    "role": "gardener"
}
```

This request maps to the Controller class EmployeeController endpoint:

```
  @PostMapping("/employees")
  Employee newEmployee(@RequestBody Employee newEmployee) {
    return repository.save(newEmployee);
  }
```

Examine the response:

The response echoes back the record created including its record number, 3 in this case.



POST http://localhost:8080/employees

```
{
    "name": "Samwise Gamgee",
    "role": "gardener"
}
```

```
{
    "id": 3,
    "name": "Samwise Gamgee",
    "role": "gardener"
}
```

PUT Request: update an existing employee 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 “/#” is the unique record id to modify in the database. If the record does not exist, it will be created.

URL: “http://localhost:8080/employees/3”
Request body:

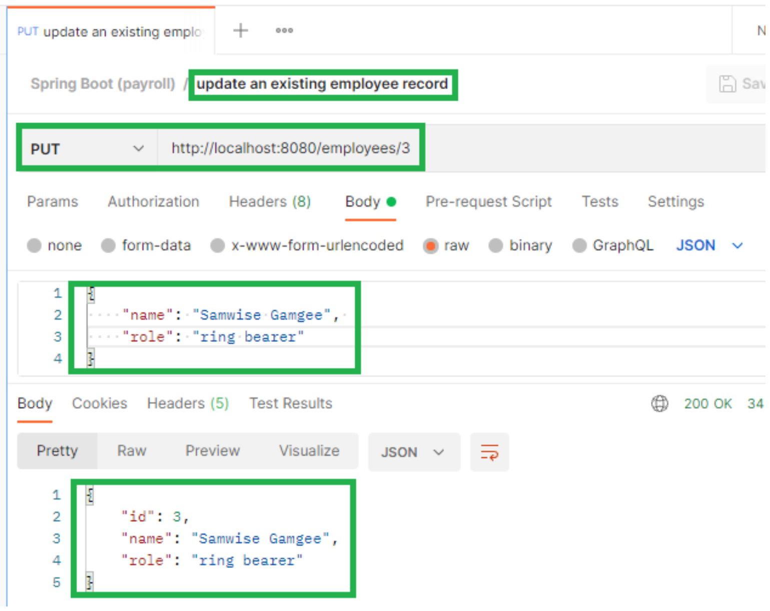
```
{
    "name": "Samwise Gamgee",
    "role": "ring bearer"
}
```

This request maps to the Controller class EmployeeController endpoint:

```
@PutMapping("/employees/{id}")
Employee replaceEmployee(@RequestBody Employee newEmployee, @PathVariable Long id) {

  return repository.findById(id)
    .map(employee -> {
      employee.setName(newEmployee.getName());
      employee.setRole(newEmployee.getRole());
      return repository.save(employee);
    })
    .orElseGet(() -> {
      newEmployee.setId(id);
      return repository.save(newEmployee);
    });
}
```

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.



PUT http://localhost:8080/employees/3

```
{
    "name": "Samwise Gamgee",
    "role": "ring bearer"
}
```

```
{
    "id": 3,
    "name": "Samwise Gamgee",
    "role": "ring bearer"
}
```

DELETE Request: delete an employee record

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

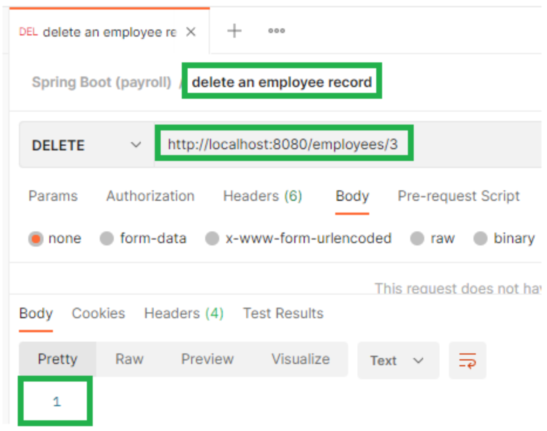
URL: "http://localhost:8080/employees/3"

This request maps to the Controller class EmployeeController endpoint:

```
@DeleteMapping("/employees/{id}")
void deleteEmployee(@PathVariable Long id) {
  repository.deleteById(id);
}
```

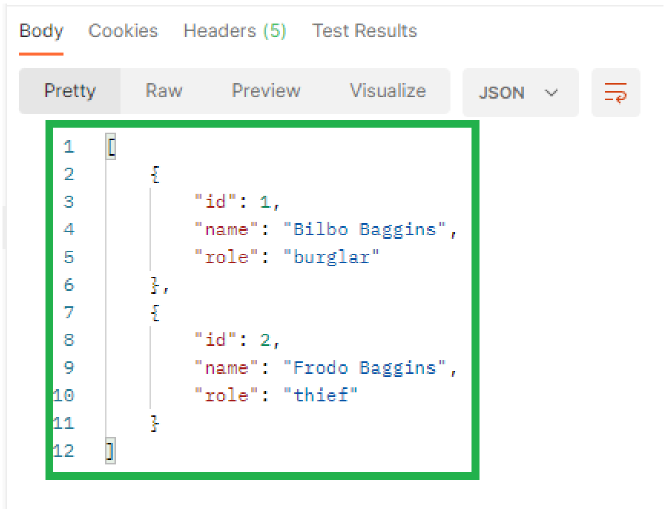
Examine the response:

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


DEL delete an employee re
Spring Boot (payroll) / delete an employee record
DELETE
http://localhost:8080/employees/3
Params Authorization Headers (6) Body Pre-request Script
none form-data x-www-form-urlencoded raw binary
This request does not ha
Body Cookies Headers (4) Test Results
Pretty Raw Preview Visualize Text

```
1
```

If you run the “GET all employee records” request, the results show record 3 is not in the list.


Body Cookies Headers (5) Test Results
Pretty Raw Preview Visualize JSON

```
[
    {
        "id": 1,
        "name": "Bilbo Baggins",
        "role": "burglar"
    },
    {
        "id": 2,
        "name": "Frodo Baggins",
        "role": "thief"
    }
]
```

Project Conclusion

The website describing this project "Building REST services with Spring" concludes that what was just built is in fact not a RESTful project but a Remote Procedure Call (RPC) project. Discussion below. In a project document named "Spring Boot 300 REST Services Part 2" this project is updated to become by definition RESTful.

"What makes something RESTful?

So far, you have a web-based service that handles the core operations involving employee data. But that's not enough to make things "RESTful".

- Pretty URLs like `/employees/3` aren't REST.
- Merely using `GET`, `POST`, etc. isn't REST.
- Having all the CRUD operations laid out isn't REST.

In fact, what we have built so far is better described as **RPC** (**Remote Procedure Call**). That's because there is no way to know how to interact with this service. If you published this today, you'd also have to write a document or host a developer's portal somewhere with all the details.

This statement of Roy Fielding's may further lend a clue to the difference between **REST** and **RPC**:

I am getting frustrated by the number of people calling any HTTP-based interface a REST API. Today's example is the SocialSite REST API. That is RPC. It screams RPC. There is so much coupling on display that it should be given an X rating.

What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. Period. Is there some broken manual somewhere that needs to be fixed?

— Roy Fielding
https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

The side effect of NOT including hypermedia in our representations is that clients MUST hard code URIs to navigate the API. This leads to the same brittle nature that predated the rise of e-commerce on the web. It's a signal that our JSON output needs a little help."

-- Building REST services with Spring