Spring Boot REST Services using JPA Part 2

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: This project expands on the program built using document titled "Spring Boot REST Services using JPA Part 1" which is based on Building REST services with Spring and just expands on the screenshots and step by step instructions. That program created a simple payroll service that manages the employees of a company. The website mentioned that at the point where the program was first tested, the program was not truly RESTful. The program is updated to add modification that will make it RESTful by definition.

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

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: payroll

This project builds on the project created using “Spring Boot 300 REST Services Part 1” and on the website Building REST services with Spring. The payroll project must exist to following the details in this document.

Right click on the project “payroll” ➔ select “Copy”
Right click in open area in “Package Explorer” ➔ select “Paste”
Enter “Project name:” **payroll-rest** ➔ Click “Copy”

We can skip refactoring any items in this project. The package name and main class can remain unchanged.

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.

This project continues building a simple payroll application using restful APIs, an H2 memory database, and JPA for database access.

At the end of the first project the following was discussed and the projected ended at the website section titled “What makes something RESTful?”.

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.

Update POM.xml

Add HATEOAS dependency to the pom.xml file.

> “Spring HATEOAS provides some APIs to ease creating REST representations that follow the HATEOAS principle when working with Spring and especially Spring MVC. The core problem it tries to address is link creation and representation assembly.” -- Spring HATEOAS.

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

```
                </dependency>
```

Update the Maven project

The project should automatically update the Maven library. In some cases it takes a manual update.
Right click project ➔ Highlight "Maven" ➔ Select "Update Project..."
The project should be selected ➔ click “OK”

Update Controller Class

“A critical ingredient to any RESTful service is adding links to relevant operations. To make your controller more RESTful, add links…” -- Building REST services with Spring

Update Controller Class: EmployeeController

Update Endpoint @GetMapping("/employees/{id}")

Existing code GetMapping by id method:

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

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

Code update for GetMapping by id method (copy and replace existing code):

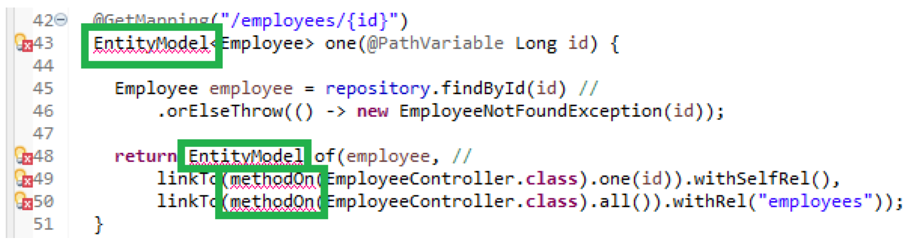
```
  @GetMapping("/employees/{id}")
  EntityModel<Employee> one(@PathVariable Long id) {

    Employee employee = repository.findById(id) //
        .orElseThrow(() -> new EmployeeNotFoundException(id));

    return EntityModel.of(employee, //
        linkTo(methodOn(EmployeeController.class).one(id)).withSelfRel(),
        linkTo(methodOn(EmployeeController.class).all()).withRel("employees"));
  }
```

Resolve the import errors, add the following imports:

```
 import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;
 import org.springframework.hateoas.EntityModel;
```

The EntityModel import error was resolved by the 2nd import above. The methodOn error was not straight forward the IDE Wizard did not find the correct import. The website gave a clue but did not specify what import was needed. Code examination from Git Hub showed the import.

This tutorial is based on Spring MVC and uses the static helper methods from `WebMvcLinkBuilder` to build these links.



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

  Employee employee = repository.findById(id) //
      .orElseThrow(() -> new EmployeeNotFoundException(id));

  return EntityModel.of(employee, //
      linkTo(methodOn(EmployeeController.class).one(id)).withSelfRel(),
      linkTo(methodOn(EmployeeController.class).all()).withRel("employees"));
}
```

Discussion Update to Endpoint @GetMapping("/employees/{id}")

This is very similar to what we had before, but a few things have changed:

- The return type of the method has changed from `Employee` to `EntityModel<Employee>`. `EntityModel<T>` is a generic container from Spring HATEOAS that includes not only the data but a collection of links.
- `linkTo(methodOn(EmployeeController.class).one(id)).withSelfRel()` asks that Spring HATEOAS build a link to the `EmployeeController`'s `one()` method, and flag it as a self link.
- `linkTo(methodOn(EmployeeController.class).all()).withRel("employees")` asks Spring HATEOAS to build a link to the aggregate root, `all()`, and call it "employees".

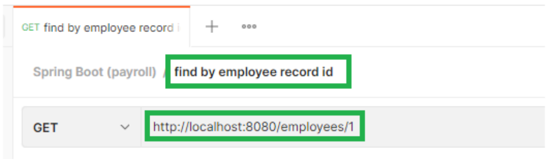
What do we mean by "build a link"? One of Spring HATEOAS's core types is `Link`. It includes a **URI** and a **rel** (relation). Links are what empower the web. Before the World Wide Web, other document systems would render information or links, but it was the linking of documents WITH this kind of relationship metadata that stitched the web together.

-- Building REST services with Spring

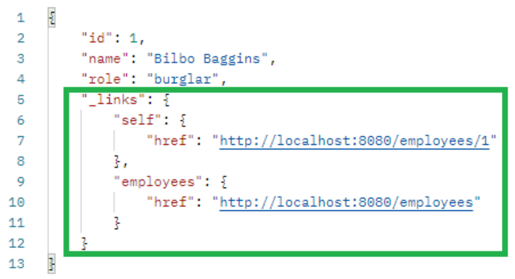
Test Changes to Endpoint @GetMapping("/employees/{id}")

Start the application inside the IDE as Spring Boot App. Send Postman test created in previous document GET "find by employee record id" for record id 1. Stop the application server at the end of this test.

Due to the code modifications, the response now shows two links (self, employees). The self link directly access record id of 1. The employees link brings back all employee records in the database.



GET find by employee record
Spring Boot (payroll)
find by employee record id
GET
http://localhost:8080/employees/1



```
{
    "id": 1,
    "name": "Bilbo Baggins",
    "role": "burglar",
    "_links": {
        "self": {
            "href": "http://localhost:8080/employees/1"
        },
        "employees": {
            "href": "http://localhost:8080/employees"
        }
    }
}
```

Update Endpoint @GetMapping("/employees")

Existing code GetMapping method getting all employees:

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

Code update for GetMapping method getting all employees (copy and replace existing code):

```
	// Aggregate root
	// tag::get-aggregate-root[]
	@GetMapping("/employees")
	CollectionModel<EntityModel<Employee>> all() {

		List<EntityModel<Employee>> employees = repository.findAll().stream()
						.map(employee -> EntityModel.of(employee,

	linkTo(methodOn(EmployeeController.class)).one(employee.getId())).withSelfRel(),

	linkTo(methodOn(EmployeeController.class)).all()).withRel("employees")))
						.collect(Collectors.toList());

		return CollectionModel.of(employees, linkTo(methodOn(EmployeeController.class).all()).withSelfRel());
	} // end::get-aggregate-root[]

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

Resolve the import errors, add the following imports:

```
import java.util.stream.Collectors;
import org.springframework.hateoas.CollectionModel;
```

Discussion Update to Endpoint @GetMapping("/employees")

`CollectionModel<>` is another Spring HATEOAS container; it's aimed at encapsulating collections of resources—instead of a single resource entity, like `EntityModel<>` from earlier. `CollectionModel<>`, too, lets you include links.

Don't let that first statement slip by. What does "encapsulating collections" mean? Collections of employees?

Not quite.

Since we're talking REST, it should encapsulate collections of **employee resources**.

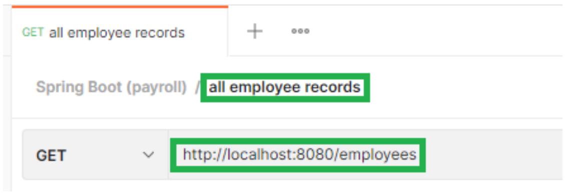
That's why you fetch all the employees, but then transform them into a list of `EntityModel<Employee>` objects. (Thanks Java 8 Streams!)

-- Building REST services with Spring

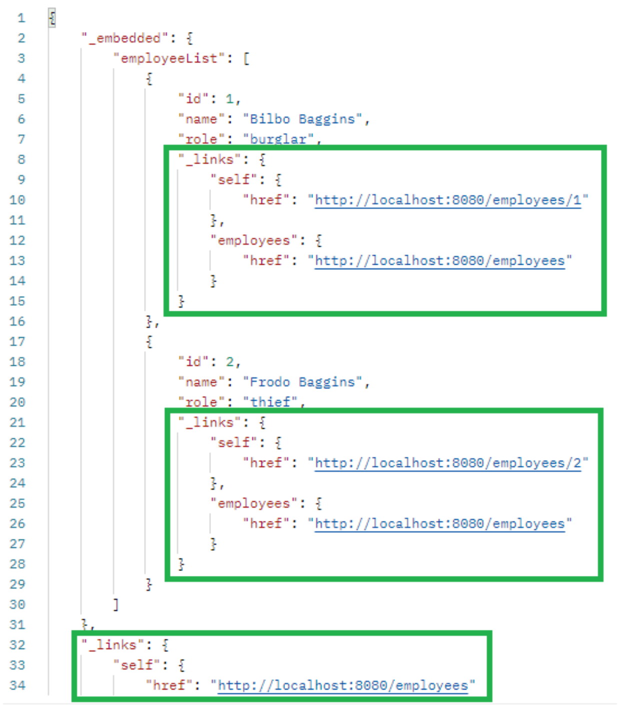
Test Changes to Endpoint @GetMapping("/employees")

Start the application inside the IDE as Spring Boot App. Send Postman test created in previous document GET “all employee records”.

The code modifications for this method also produces links in the response message.



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



```
{
    "_embedded": {
        "employeeList": [
            {
                "id": 1,
                "name": "Bilbo Baggins",
                "role": "burglar",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/employees/1"
                    },
                    "employees": {
                        "href": "http://localhost:8080/employees"
                    }
                }
            },
            {
                "id": 2,
                "name": "Frodo Baggins",
                "role": "thief",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/employees/2"
                    },
                    "employees": {
                        "href": "http://localhost:8080/employees"
                    }
                }
            }
        ]
    },
    "_links": {
        "self": {
            "href": "http://localhost:8080/employees"
```

“For this aggregate root, which serves up a collection of employee resources, there is a top-level **"self"** link.

The **"collection"** is listed underneath the **"_embedded"** section; this is how HAL represents collections.

And each individual member of the collection has their information as well as related links.”

-- Building REST services with Spring

The Reason for the Controller Class Link Updates

"What is the point of adding all these links? It makes it possible to evolve REST services over time. Existing links can be maintained while new links can be added in the future. Newer clients may take advantage of the new links, while legacy clients can sustain themselves on the old links. This is especially helpful if services get relocated and moved around. As long as the link structure is maintained, clients can STILL find and interact with things." -- Building REST services with Spring

Project Conclusion

The project updates in this document added links to two EmployeeController class endpoints:

- @GetMapping("/employees")
- @GetMapping("/employees/{id}")

There are additional updates to this project to make adding links to the endpoints simpler. These updates are discussed in the next section "Simplifying Link Creation". Updates are also talked about in the subsection "Evolving REST APIs". These two section updates will be handled in Parts 3 and 4 of this series of documents.

Simplifying Link Creation

Simplifying Link Creation
"In the code earlier, did you notice the repetition in single employee link creation? The code to provide a single link to an employee, as well as to create an "employees" link to the aggregate root, was shown twice. If that raised your concern, good! There's a solution." -- Building REST services with Spring

- These changes will be discussed in document titled "Spring Boot 300 REST Services Part 3".

Evolving REST APIs

Further updates to the project involve updating the employee entity and adding customer order processing.

Evolving REST APIs
"With one additional library and a few lines of extra code, you have added hypermedia to your application. But that is not the only thing needed to make your service RESTful. An important facet of REST is the fact that it's neither a technology stack nor a single standard.

REST is a collection of architectural constraints that when adopted make your application much more resilient. A key factor of resilience is that when you make upgrades to your services, your clients don't suffer from downtime."
-- Building REST services with Spring

- These changes will be discussed in document titled "Spring Boot 300 REST Services Part 4".