Spring Boot REST Services using JPA Part 4

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 documents "Spring Boot REST Services using JPA" Part 1 thru 3 which is based on Building REST services with Spring. The Part 1 program created a simple payroll service that manages the employees of a company. Part 2 updated the controller class adding links to the response to make it more RESTful. Part 3 simplified adding links to the response messages. This document Part 4 works on evolving the REST APIs. Part 4 will update the employee entity model and add customer order processing.

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-rest-2

This project builds on the project created using “Spring Boot 300 REST Services Part 3” and on the website Building REST services with Spring. The payroll-rest-2 project must exist to following the details in this document. The project built in Part 3 is used as the starting point for this project. The overall project is enhanced adding elements to the employee model and adding customer ordering processing.

Right click on the project “payroll-rest-2” ➔ select “Copy”
Right click in open area in “Package Explorer” ➔ select “Paste”
Enter “Project name:” **payroll-rest-3** ➔ 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.

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.

At the end of the second project links were added to two of the application’s endpoints. This modification made those endpoints RESTful.

At the end of the third project modification continued using the website section titled “Simplifying Link Creation”. It created a common way to create links for response messages. It also modifies the entity class.

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

This project continues modification at the website section titled “Evolving REST APIs”. It creates a common way to create links for response messages. It also modifies the entity class.

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

Update Domain Object Model / Entity Class

Supporting changes to the API

"Imagine this design problem: You've rolled out a system with this `Employee`-based record. The system is a major hit. You've sold your system to countless enterprises. Suddenly, the need for an employee's name to be split into `firstName` and `lastName` arises.

Uh oh. Didn't think of that.

Before you open up the `Employee` class and replace the single field `name` with `firstName` and `lastName`, stop and think for a second. Will that break any clients? How long will it take to upgrade them. Do you even control all the clients accessing your services?

Downtime = lost money. Is management ready for that?

There is an old strategy that precedes REST by years.

Never delete a column in a database.

— Unknown" -- Building REST services with Spring

The above strategy / principle is applied to database entities and RESTful APIs. In this project the model class Employee is updated to accept first and last name.

Update Model Class: Employee

Updates to class Employee is intended to be backward compatible to handle just one name field and to support a new API using first and last name fields.

Split the name field into first and last name fields.

```
  private String firstName;
  private String lastName;
```

Replace existing parameterized constructor:

```
Existing:
        Employee(String name, String role) {

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

Replace with:
        Employee(String firstName, String lastName, String role) {

                this.firstName = firstName;
                this.lastName = lastName;
                this.role = role;
        }
```

```
 9 @Entity
10 class Employee {
11
12     private @Id @GeneratedValue Long id;
13     private String firstName;
14     private String lastName;
15     private String role;
16
17     Employee() {
18     }
19
20     Employee(String firstName, String lastName, String role) {
21
22         this.firstName = firstName;
23         this.lastName = lastName;
24         this.role = role;
25     }
26
```

Update existing get and set name to the following:

```
        public String getName() {
                return this.firstName + " " + this.lastName;
        }

        public void setName(String name) {
                String[] parts = name.split(" ");
                this.firstName = parts[0];
                this.lastName = parts[1];
        }
```

Add getters and setters for new class variables:

```
        public String getFirstName() {
                return this.firstName;
        }

        public String getLastName() {
                return this.lastName;
        }

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

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

Update the override methods:

```
        @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.firstName, employee.firstName)
                                && Objects.equals(this.lastName, employee.lastName) && Objects.equals(this.role,
employee.role);
        }

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

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

Update Helper Class Related to Employee Changes

The changes to the Model class Employee creates errors in the LoadDatabase class. Update this class to clear the errors and continue to load initial employee data.

Update Helper Class: LoadDatabase to Correct Errors

The IDE will show errors in the LoadDatabase class related to the changes made in the Model class Employee. There is no a constructor to handle a single “name” parameter.

Replace the return statement with:

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

Update Controller Class

Update three controller class methods making them backward compatible plus other enhancements.

Update Controller Class: EmployeeController

Replace the POST, PUT, and DELETE endpoint with the following code:

```
    @PostMapping("/employees")
    ResponseEntity<?> newEmployee(@RequestBody Employee newEmployee) {

      EntityModel<Employee> entityModel = assembler.toModel(repository.save(newEmployee));

      return ResponseEntity //
          .created(entityModel.getRequiredLink(IanaLinkRelations.SELF).toUri()) //
          .body(entityModel);
    }

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

      Employee updatedEmployee = 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);
          });

      EntityModel<Employee> entityModel = assembler.toModel(updatedEmployee);

      return ResponseEntity //
          .created(entityModel.getRequiredLink(IanaLinkRelations.SELF).toUri()) //
          .body(entityModel);
    }

    @DeleteMapping("/employees/{id}")
    ResponseEntity<?> deleteEmployee(@PathVariable Long id) {

      repository.deleteById(id);

      return ResponseEntity.noContent().build();
    }
```

Add imports to resolve the errors:

```
import org.springframework.hateoas.IanaLinkRelations;
import org.springframework.http.ResponseEntity;
```

Discuss Updates to Controller Class: EmployeeController

POST method:

"The new `Employee` object is saved as before. But the resulting object is wrapped using the `EmployeeModelAssembler`.

Spring MVC's `ResponseEntity` is used to create an **HTTP 201 Created** status message. This type of response typically includes a **Location** response header, and we use the URI derived from the model's self-related link. Additionally, return the model-based version of the saved object.

With these tweaks in place, you can use the same endpoint to create a new employee resource, and use the legacy `name` field" -- Building REST services with Spring

PUT method:

"The `Employee` object built from the `save()` operation is then wrapped using the `EmployeeModelAssembler` into an `EntityModel<Employee>` object. Using the `getRequiredLink()` method, you can retrieve the `Link` created by the `EmployeeModelAssembler` with a `SELF` rel. This method returns a `Link` which must be turned into a `URI` with the `toUri` method.
Since we want a more detailed HTTP response code than **200 OK**, we will use Spring MVC's `ResponseEntity` wrapper. It has a handy static method `created()` where we can plug in the resource's URI. It's debatable if **HTTP 201 Created** carries the right semantics since we aren't necessarily "creating" a new resource. But it comes pre-loaded with a **Location** response header, so run with it." -- Building REST services with Spring

Test Initial Project Updates

The Postman request created in the previous project documents are used to perform regression testing on the now legacy endpoints related to the one field for name in the Employee class. Execution of all request in sequence should produce the results seen in the last project document.

New Postman request are created to test the splitting of the name in the Employee class. These tests are to validate the new endpoints created for new consumers of the application.

Test Initial Project Updates: Regression Testing

Start the application inside the IDE as Spring Boot App.
Run the sequence of Postman request and validate the response.

GET: all employee records
GET: find by employee record id
GET: find by employee record id not found
POST: create an employee
GET: all employee records
PUT: update an existing employee record
GET: all employee records
DELETE: delete an employee record
GET: all employee records

The main thing to notice in all responses that contain employee information the split of the first and last name plus the combined name is included in the response.

```
6           "firstName": "Bilbo",
7           "lastName": "Baggins",
8           "role": "burglar",
9           "name": "Bilbo Baggins"
```

Test Initial Project Updates: Create New Requests

The GET and DELETE requests are not impacted by the changes made to the Employee class. New requests are required for the POST and PUT endpoints to accept first and last names.

POST Request: create an employee by first and last name

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

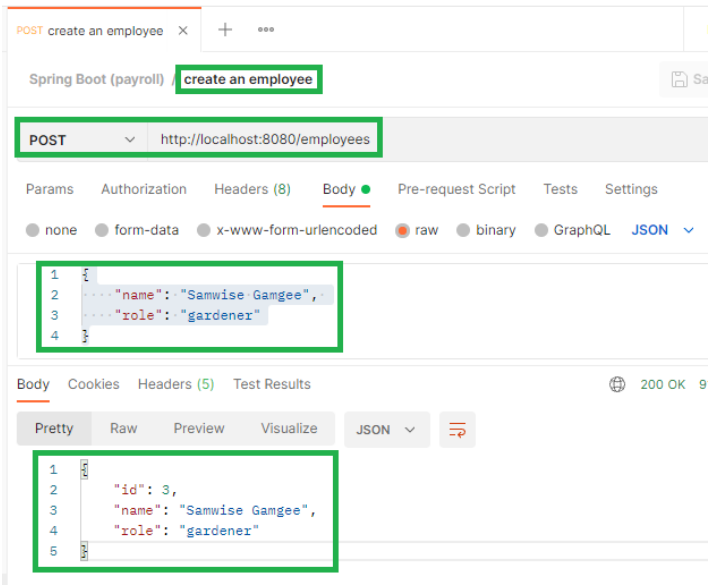
```
{
    "name": "Samwise Gamgee",
    "role": "gardener"
}
```

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.



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:

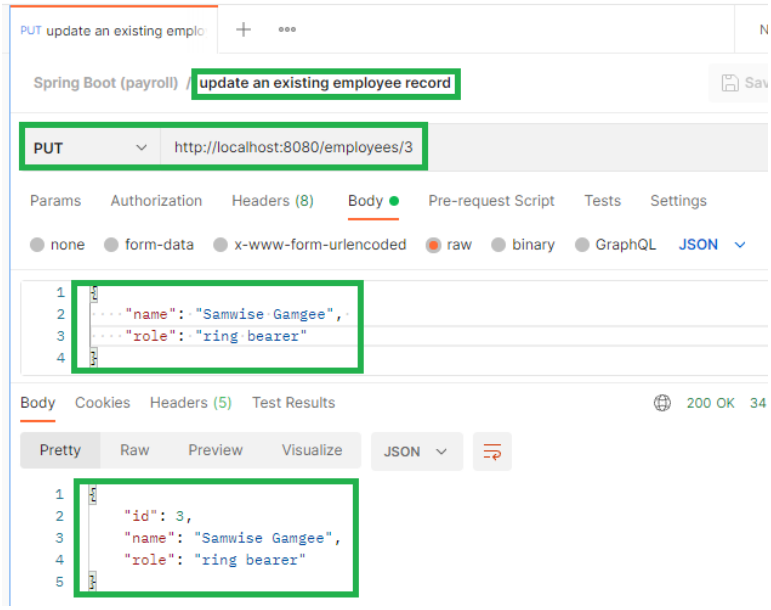
```
{
    "name": "Samwise Gamgee",
    "role": "ring bearer"
}
```

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.



Create Order Processing Classes

Building links into your REST API

"So far, you've built an evolvable API with bare bones links. To grow your API and better serve your clients, you need to embrace the concept of **Hypermedia as the Engine of Application State**.

What does that mean? In this section, you'll explore it in detail.

Business logic inevitably builds up rules that involve processes. The risk of such systems is we often carry such server-side logic into clients and build up strong coupling. REST is about breaking down such connections and minimizing such coupling.

To show how to cope with state changes without triggering breaking changes in clients, imagine adding a system that fulfills orders." -- Building REST services with Spring

Create Enum Class: Status

Class to capture the status of an order.

Copy and replace the code shell in its entirety.

```
package com.example.payroll;

public enum Status {

          IN_PROGRESS, //
          COMPLETED, //
          CANCELLED

}
```

Create Domain Model Entity Class: Order

Like the Employee class the Order class is defined as an @Entity class that defines records for database storage.

"The class requires a JPA @Table annotation changing the table's name to CUSTOMER_ORDER because ORDER is not a valid name for table.

It includes a description field as well as a status field.

Orders must go through a certain series of state transitions from the time a customer submits an order and it is either fulfilled or cancelled. This can be captured as a Java enum:" -- Building REST services with Spring

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;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMER_ORDER")
class Order {

  private @Id @GeneratedValue Long id;

  private String description;
  private Status status;

  Order() {}

  Order(String description, Status status) {

    this.description = description;
    this.status = status;
  }

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

  public String getDescription() {
    return this.description;
  }

  public Status getStatus() {
    return this.status;
  }

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

  public void setDescription(String description) {
    this.description = description;
  }

  public void setStatus(Status status) {
    this.status = status;
  }

  @Override
  public boolean equals(Object o) {

    if (this == o)
      return true;
    if (!(o instanceof Order))
      return false;
    Order order = (Order) o;
    return Objects.equals(this.id, order.id) && Objects.equals(this.description, order.description)
        && this.status == order.status;
  }

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

  @Override
  public String toString() {
    return "Order{" + "id=" + this.id + ", description='" + this.description + '\'' + ", status=" + this.status + '}';
  }
}
```

Create Repository Interface: OrderRepository

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 interface named “OrederRepository” in package “com.example.payroll” that extends JpaRepository<Order, Long>. This interface allows creating, reading, updating, and deleting order data in the memory database.

Copy and replace the code shell in its entirety.

```
package com.example.payroll;

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

interface OrderRepository extends JpaRepository<Order, Long> {
}
```

Create Exception Class: OrderNotFoundException

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

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

Copy and replace the code shell in its entirety.

```
package com.example.payroll;

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

	OrderNotFoundException(Long id) {
		super("Could not find order " + id);
	}
}
```

Create Exception Class: OrderNotFoundAdvice

Create class named “OrderNotFoundAdvice” 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.

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
public class OrderNotFoundAdvice {

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

Create Controller Class: OrderController

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 class named “OrderController” in package “com.example.payroll”.

Copy and replace the code shell in its entirety.

```
package com.example.payroll;

import java.util.List;
import java.util.stream.Collectors;

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

import org.springframework.hateoas.CollectionModel;
import org.springframework.hateoas.EntityModel;
import org.springframework.http.ResponseEntity;
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.RequestBody;
import org.springframework.web.bind.annotation.RestController;

public @RestController class OrderController {

        private final OrderRepository orderRepository;
        private final OrderModelAssembler assembler;

        OrderController(OrderRepository orderRepository, OrderModelAssembler assembler) {

                this.orderRepository = orderRepository;
                this.assembler = assembler;
        }

        @GetMapping("/orders")
        CollectionModel<EntityModel<Order>> all() {

                List<EntityModel<Order>> orders = orderRepository.findAll().stream() //
                                .map(assembler::toModel) //
                                .collect(Collectors.toList());
```

```
                return CollectionModel.of(orders, //
                                linkTo(methodOn(OrderController.class).all()).withSelfRel());
        }

        @GetMapping("/orders/{id}")
        EntityModel<Order> one(@PathVariable Long id) {

                Order order = orderRepository.findById(id) //
                                .orElseThrow(() -> new OrderNotFoundException(id));

                return assembler.toModel(order);
        }

        @PostMapping("/orders")
        ResponseEntity<EntityModel<Order>> newOrder(@RequestBody Order order) {

                order.setStatus(Status.IN_PROGRESS);
                Order newOrder = orderRepository.save(order);

                return ResponseEntity //

        .created(linkTo(methodOn(OrderController.class).one(newOrder.getId())).toUri()) //
                                .body(assembler.toModel(newOrder));
        }
}
```

The above is the initial creation of the OrderController class. It will contains errors since the OrderModelAssember class has not been created. Even then the OrderModelAssember will contain errors, the OrderController class is missing some methods that will be added later.

- It contains the same REST controller setup as the controllers you've built so far.
- It injects both an `OrderRepository` as well as a (not yet built) `OrderModelAssembler`.
- The first two Spring MVC routes handle the aggregate root as well as a single item `Order` resource request.
- The third Spring MVC route handles creating new orders, by starting them in the `IN_PROGRESS` state.
- All the controller methods return one of Spring HATEOAS's `RepresentationModel` subclasses to properly render hypermedia (or a wrapper around such a type).

-- Building REST services with Spring

Create Model Assembler Class: OrderModelAssembler

Create a Model Assembler class that uses HATEOAS to simplifies converting an Order object to an EntityModel object. The EntityModel object will generate the links in response messages.

Create class named "OrderModelAssembler" in package "com.example.payroll". This class contains a method to convert an Employee object into an EntityModel object.

"Before building the `OrderModelAssembler`, let's discuss what needs to happen. You are modeling the flow of states between `Status.IN_PROGRESS`, `Status.COMPLETED`, and `Status.CANCELLED`. A natural thing when serving up such data to clients is to let the clients make the decision on what it can do based on this payload.

But that would be wrong.

What happens when you introduce a new state in this flow? The placement of various buttons on the UI would probably be erroneous.

What if you changed the name of each state, perhaps while coding international support and showing locale-specific text for each state? That would most likely break all the clients.

Enter **HATEOAS** or **Hypermedia as the Engine of Application State**. Instead of clients parsing the payload, give them links to signal valid actions. Decouple state-based actions from the payload of data. In other words, when **CANCEL** and **COMPLETE** are valid actions, dynamically add them to the list of links. Clients only need show users the corresponding buttons when the links exist.

This decouples clients from having to know WHEN such actions are valid, reducing the risk of the server and its clients getting out of sync on the logic of state transitions.

Having already embraced the concept of Spring HATEOAS `RepresentationModelAssembler` components, putting such logic in the `OrderModelAssembler` would be the perfect place to capture this business rule:"

-- Building REST services with Spring

Copy and replace the code shell in its entirety.

```
package com.example.payroll;


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

import org.springframework.hateoas.EntityModel;
import org.springframework.hateoas.server.RepresentationModelAssembler;
import org.springframework.stereotype.Component;



@Component
class OrderModelAssembler implements RepresentationModelAssembler<Order, EntityModel<Order>> {

  @Override
  public EntityModel<Order> toModel(Order order) {
```

```
    // Unconditional links to single-item resource and aggregate root

    EntityModel<Order> orderModel = EntityModel.of(order,
        linkTo(methodOn(OrderController.class).one(order.getId())).withSelfRel(),
        linkTo(methodOn(OrderController.class).all()).withRel("orders"));

    // Conditional links based on state of the order

    if (order.getStatus() == Status.IN_PROGRESS) {
      orderModel.add(linkTo(methodOn(OrderController.class).cancel(order.getId())).withRel("cancel"));
      orderModel.add(linkTo(methodOn(OrderController.class).complete(order.getId())).withRel("complete"));
    }

    return orderModel;
  }
}
```

Like mentioned before, this class will contain errors until the OrderController class adds two methods. A cancel and a complete method.

> "This simple interface has one method: `toModel()`. It is based on converting a non-model object (`Employee`) into a model-based object (`EntityModel<Employee>`)." -- Building REST services with Spring

Update Controller Class: OrderController

Add the two missing methods to the class cancel and complete.

Add the following imports to the class.

```
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.mediatype.problem.Problem;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.DeleteMapping;
```

Add cancel and complete methods.

```
    @DeleteMapping("/orders/{id}/cancel")
    ResponseEntity<?> cancel(@PathVariable Long id) {

      Order order = orderRepository.findById(id) //
          .orElseThrow(() -> new OrderNotFoundException(id));

      if (order.getStatus() == Status.IN_PROGRESS) {
        order.setStatus(Status.CANCELLED);
        return ResponseEntity.ok(assembler.toModel(orderRepository.save(order)));
      }

      return ResponseEntity //
          .status(HttpStatus.METHOD_NOT_ALLOWED) //
          .header(HttpHeaders.CONTENT_TYPE, MediaTypes.HTTP_PROBLEM_DETAILS_JSON_VALUE) //
          .body(Problem.create() //
              .withTitle("Method not allowed") //
              .withDetail("You can't cancel an order that is in the " + order.getStatus() + " status"));
    }

    @PutMapping("/orders/{id}/complete")
    ResponseEntity<?> complete(@PathVariable Long id) {

      Order order = orderRepository.findById(id) //
          .orElseThrow(() -> new OrderNotFoundException(id));

      if (order.getStatus() == Status.IN_PROGRESS) {
        order.setStatus(Status.COMPLETED);
        return ResponseEntity.ok(assembler.toModel(orderRepository.save(order)));
      }

      return ResponseEntity //
          .status(HttpStatus.METHOD_NOT_ALLOWED) //
          .header(HttpHeaders.CONTENT_TYPE, MediaTypes.HTTP_PROBLEM_DETAILS_JSON_VALUE) //
```

```
                .body(Problem.create() //
                    .withTitle("Method not allowed") //
                    .withDetail("You can't complete an order that is in the " + order.getStatus() + " status"));
        }
```

Creating the methods above will clear the errors in the OrderModelAssembler class.

Update Helper Class: LoadDatabase

Update the helper class to initially load the memory database.

Update 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 replace the @Bean code in its entirety.

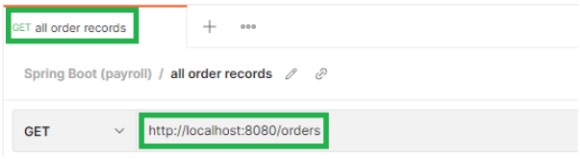
```
        @Bean
        CommandLineRunner initDatabase(EmployeeRepository employeeRepository, OrderRepository orderRepository) {

                return args -> {
                        employeeRepository.save(new Employee("Bilbo", "Baggins", "burglar"));
                        employeeRepository.save(new Employee("Frodo", "Baggins", "thief"));

                        employeeRepository.findAll().forEach(employee -> log.info("Preloaded " + employee));

                        orderRepository.save(new Order("MacBook Pro", Status.COMPLETED));
                        orderRepository.save(new Order("iPhone", Status.IN_PROGRESS));

                        orderRepository.findAll().forEach(order -> {
                                log.info("Preloaded " + order);
                        });

                };
        }
```

Test New OrderController Endpoints

Start the application inside the IDE as Spring Boot App.

Create Test Endpoint @GetMapping("/orders")

Create Postman test GET “all order records”.



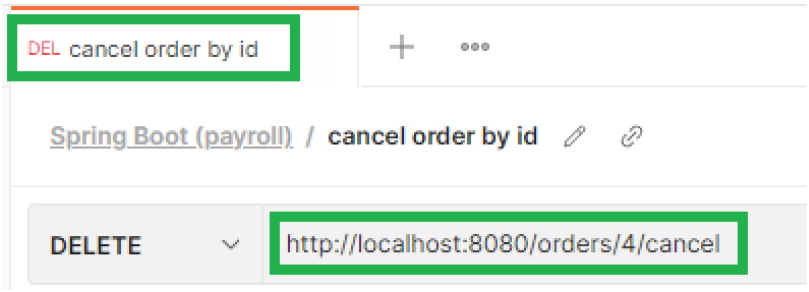
The response contains the completed and in progress orders preloaded in the memory database. The response contains the links making the endpoint RESTful.


```
{
    "_embedded": {
        "orderList": [
            {
                "id": 3,
                "description": "MacBook Pro",
                "status": "COMPLETED",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/orders/3"
                    },
                    "orders": {
                        "href": "http://localhost:8080/orders"
                    }
                }
            },
            {
                "id": 4,
                "description": "iPhone",
                "status": "IN_PROGRESS",
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/orders/4"
                    },
                    "orders": {
                        "href": "http://localhost:8080/orders"
                    },
                    "cancel": {
                        "href": "http://localhost:8080/orders/4/cancel"
                    },
                    "complete": {
```

```
                        "href": "http://localhost:8080/orders/4/complete"
                    }
                }
            }
        ]
    },
    "_links": {
        "self": {
            "href": "http://localhost:8080/orders"
        }
    }
}
```


Create Test Endpoint for Cancel @DeleteMapping("/orders")

Create Postman test DELETE “cancel order by id”.



Examine the response.

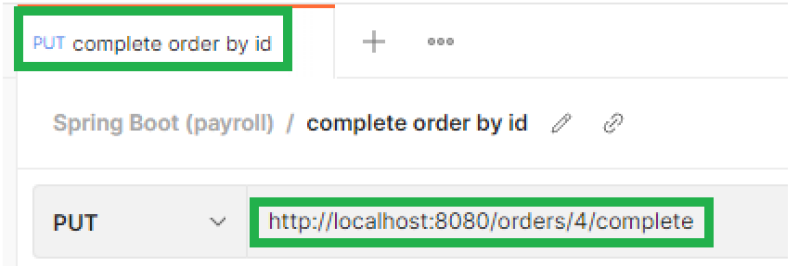

```
{
    "id": 4,
    "description": "iPhone",
    "status": "CANCELLED",
    "_links": {
        "self": {
            "href": "http://localhost:8080/orders/4"
        },
        "orders": {
            "href": "http://localhost:8080/orders"
        }
    }
}
```


Run the request again and receive the following response. The message is defined in the OrderController class.


```
{
    "title": "Method not allowed",
    "detail": "You can't cancel an order that is in the CANCELLED status"
}
```

Create Test Endpoint for Complete @PutMapping("/orders")

Create Postman test PUT “complete order by id”.

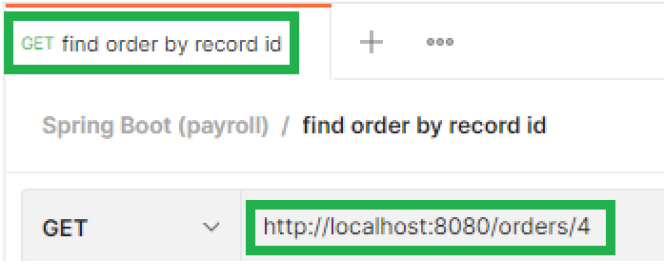


Examine the response. This PUT request is against the order that was cancelled. The message is defined in the OrderController class.

```
{
    "title": "Method not allowed",
    "detail": "You can't complete an order that is in the CANCELLED status"
}
```

Create Test Endpoint @GetMapping("/orders/{id}")

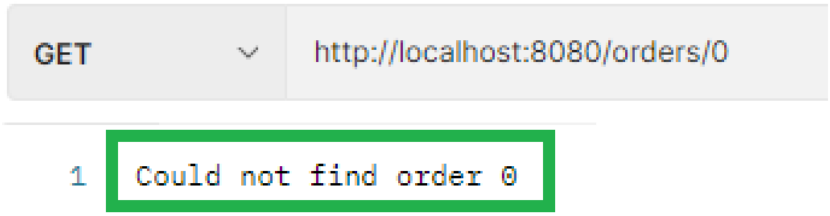
Create Postman test GET “find order by record id”.



Examine the response.

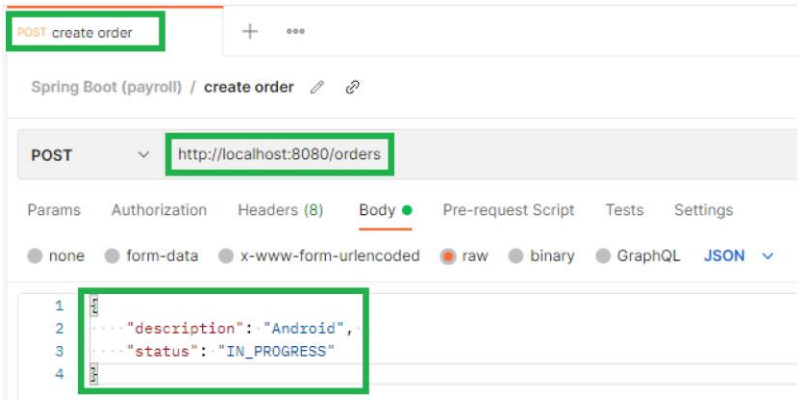
```
{
    "id": 4,
    "description": "iPhone",
    "status": "CANCELLED",
    "_links": {
        "self": {
            "href": "http://localhost:8080/orders/4"
        },
        "orders": {
            "href": "http://localhost:8080/orders"
        }
    }
}
```

Run the GET request again with an id that is not in the database. This will trigger the order not found exception.



Create Test Endpoint @PostMapping("/orders/{id}")

Create Postman test POST “create order”.



Examine the response.

Project Conclusion

Summary
"Throughout this tutorial, you have engaged in various tactics to build REST APIs. As it turns out, REST isn't just about pretty URIs and returning JSON instead of XML.

Instead, the following tactics help make your services less likely to break existing clients you may or may not control:
Don't remove old fields. Instead, support them.
Use rel-based links so clients don't have to hard code URIs.

Retain old links as long as possible. Even if you have to change the URI, keep the rels so older clients have a path onto the newer features.

Use links, not payload data, to instruct clients when various state-driving operations are available.

It may appear to be a bit of effort to build up RepresentationModelAssembler implementations for each resource type and to use these components in all of your controllers. But this extra bit of server-side setup (made easy thanks to Spring HATEOAS) can ensure the clients you control (and more importantly, those you don't) can upgrade with ease as you evolve your API."

-- Building REST services with Spring

Project Refactoring

As mentioned above from the website, the tutorial is finished. The project evolved over four evolutions into a project with two entity classes and two controller classes with multiple endpoints. All of the classes were developed under one package, com.example.payroll, not the best software engineering practice. The next evolution of the project will focus solely on refactoring using better software engineering practices.

- These changes will be discussed in document titled "Spring Boot 300 REST Services Part 5".