Building REST services with Spring

REST has quickly become the de-facto standard for building web services on the web because they're easy to build and easy to consume.

There's a much larger discussion to be had about how REST fits in the world of microservices, but — for this tutorial — let's just look at building RESTful services.

Why REST? REST embraces the precepts of the web, including its architecture, benefits, and everything else. This is no surprise given its author, Roy Fielding, was involved in probably a dozen specs which govern how the web operates.

What benefits? The web and its core protocol, HTTP, provide a stack of features:

- Suitable actions (`GET`, `POST`, `PUT`, `DELETE`, ...)
- Caching
- Redirection and forwarding
- Security (encryption and authentication)

These are all critical factors on building resilient services. But that is not all. The web is built out of lots of tiny specs, hence it's been able to evolve easily, without getting bogged down in "standards wars".

Developers are able to draw upon 3rd party toolkits that implement these diverse specs and instantly have both client and server technology at their fingertips.

By building on top of HTTP, REST APIs provide the means to build:

- Backwards compatible APIs
- Evolvable APIs
- Scaleable services
- Securable services
- A spectrum of stateless to stateful services

What's important to realize is that REST, however ubiquitous, is not a standard, *per se*, but an approach, a style, a set of *constraints* on your architecture that can help you build web-scale systems. In this tutorial we will use the Spring portfolio to build a RESTful service while leveraging the stackless features of REST.

Getting Started

As we work through this tutorial, we'll use Spring Boot. Go to Spring Initializr and add the following dependencies to a project:

- Web
- JPA
- H2

Change the Name to "Payroll" and then choose "Generate Project". A `.zip` will download. Unzip it. Inside you'll find a simple, Maven-based project including a `pom.xml` build file (NOTE: You *can* use Gradle. The examples in this tutorial will be Maven-based.)

Spring Boot can work with any IDE. You can use Eclipse, IntelliJ IDEA, Netbeans, etc. The Spring Tool Suite is an open-source, Eclipse-based IDE distribution that provides a superset of the Java EE distribution of Eclipse. It includes features that make working with Spring applications even easier. It is, by no means, required.

But consider it if you want that extra **oomph** for your keystrokes. Here's a video demonstrating how to get started with STS and Spring Boot. This is a general introduction to familiarize you with the tools.

The Story so Far...

Let's start off with the simplest thing we can construct. In fact, to make it as simple as possible, we can even leave out the concepts of REST. (Later on, we'll add REST to understand the difference.)

Big picture: We're going to 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).

The following code defines an Employee in our system.

nonrest/src/main/java/payroll/Employee.java

COPY

```
package 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 + '\'' + '}';
  }
}
```

Despite being small, this Java class contains much:

- `@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.

With this domain object definition, we can now turn to Spring Data JPA to handle the tedious database interactions.

Spring Data JPA repositories are interfaces with methods supporting creating, reading, updating, and deleting records against a back end data store. Some repositories also support data paging, and sorting, where appropriate. Spring Data synthesizes implementations based on conventions found in the naming of the methods in the interface.

> There are multiple repository implementations besides JPA. You can use Spring Data MongoDB, Spring Data GemFire, Spring Data Cassandra, etc. For this tutorial, we'll stick with JPA.

Spring makes accessing data easy. By simply declaring the following `EmployeeRepository` interface we automatically will be able to

- Create new Employees
- Update existing ones
- Delete Employees
- Find Employees (one, all, or search by simple or complex properties)

nonrest/src/main/java/payroll/EmployeeRepository.java

```
package payroll;

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

interface EmployeeRepository extends JpaRepository<Employee, Long> {
```

```
}
```

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.

Believe it or not, this is enough to launch an application! A Spring Boot application is, at a minimum, a `public static void main` entry-point and the `@SpringBootApplication` annotation. This tells Spring Boot to help out, wherever possible.

nonrest/src/main/java/payroll/PayrollApplication.java

```
package 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);
  }
}
```

`@SpringBootApplication` is a meta-annotation that pulls in **component scanning**, **autoconfiguration**, and **property support**. We won't dive into the details of Spring Boot in this tutorial, but in essence, it will fire up a servlet container and serve up our service.

Nevertheless, an application with no data isn't very interesting, so let's preload it. The following class will get loaded automatically by Spring:

nonrest/src/main/java/payroll/LoadDatabase.java

```
package 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")));
    };
  }
}
```

What happens when it gets loaded?

- Spring Boot will run ALL `CommandLineRunner` beans once the application context is loaded.
- This runner will request a copy of the `EmployeeRepository` you just created.
- Using it, it will create two entities and store them.

Right-click and **Run** `PayRollApplication`, and this is what you get:

Fragment of console output showing preloading of data

```
...
2018-08-09 11:36:26.169  INFO 74611 --- [main] payroll.LoadDatabase : Preloading Employee(
2018-08-09 11:36:26.174  INFO 74611 --- [main] payroll.LoadDatabase : Preloading Employee(
...
```

This isn't the **whole** log, but just the key bits of preloading data. (Indeed, check out the whole console. It's glorious.)

HTTP is the Platform

To wrap your repository with a web layer, you must turn to Spring MVC. Thanks to Spring Boot, there is little in infrastructure to code. Instead, we can focus on actions:

nonrest/src/main/java/payroll/EmployeeController.java

COPY

```
package 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);
  }
}
```

- `@RestController` indicates that the data returned by each method will be written straight into the response body instead of rendering a template.
- An `EmployeeRepository` is injected by constructor into the controller.
- We have routes for each operation (`@GetMapping`, `@PostMapping`, `@PutMapping` and `@DeleteMapping`, corresponding to HTTP `GET`, `POST`, `PUT`, and `DELETE` calls). (NOTE: It's useful to read each method and understand what they do.)
- `EmployeeNotFoundException` is an exception used to indicate when an employee is looked up but not found.

nonrest/src/main/java/payroll/EmployeeNotFoundException.java

COPY

```
package payroll;

class EmployeeNotFoundException extends RuntimeException {

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

When an `EmployeeNotFoundException` is thrown, this extra tidbit of Spring MVC configuration is used to render an **HTTP 404**:

nonrest/src/main/java/payroll/EmployeeNotFoundAdvice.java

COPY

```
package 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();
  }
}
```

- `@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.

To launch the application, either right-click the `public static void main` in `PayRollApplication` and select **Run** from your IDE, or:

Spring Initializr uses maven wrapper so type this:

```
$ ./mvnw clean spring-boot:run
```

Alternatively using your installed maven version type this:

```
$ mvn clean spring-boot:run
```

When the app starts, we can immediately interrogate it.

```
$ curl -v localhost:8080/employees
```

This will yield:

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
```

```
> GET /employees HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Thu, 09 Aug 2018 17:58:00 GMT
<
* Connection #0 to host localhost left intact
[{"id":1,"name":"Bilbo Baggins","role":"burglar"},{"id":2,"name":"Frodo Baggins","role":"t
```

Here you can see the pre-loaded data, in a compacted format.

If you try and query a user that doesn't exist...

```
$ curl -v localhost:8080/employees/99
```

You get...

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /employees/99 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 404
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 26
< Date: Thu, 09 Aug 2018 18:00:56 GMT
<
* Connection #0 to host localhost left intact
Could not find employee 99
```

This message nicely shows an **HTTP 404** error with the custom message **Could not find employee 99**.

It's not hard to show the currently coded interactions...

> If you are using Windows Command Prompt to issue cURL commands, chances are the below command won't work properly. You must either pick

> a terminal that support single quoted arguments, or use double quotes and then escape the ones inside the JSON.

To create a new `Employee` record we use the following command in a terminal—the `$` at the beginning signifies that what follows it is a terminal command:

```
$ curl -X POST localhost:8080/employees -H 'Content-type:application/json' -d '{"name": "S
```

Then it stores newly created employee and sends it back to us:

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

You can update the user. Let's change his role.

```
$ curl -X PUT localhost:8080/employees/3 -H 'Content-type:application/json' -d '{"name": "
```

And we can see the change reflected in the output.

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

> The way you construct your service can have significant impacts. In this situation, we said **update**, but **replace** is a better description. For example, if the name was NOT provided, it would instead get nulled out.

Finally, you can delete users like this:

```
$ curl -X DELETE localhost:8080/employees/3

# Now if we look again, it's gone
$ curl localhost:8080/employees/3
Could not find employee 3
```

This is all well and good, but do we have a RESTful service yet? (If you didn't catch the hint, the answer is no.)

What's missing?

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.

Introducing Spring HATEOAS, a Spring project aimed at helping you write hypermedia-driven outputs. To upgrade your service to being RESTful, add this to your build:

Adding Spring HATEOAS to `dependencies` section of `pom.xml`

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

COPY

This tiny library will give us the constructs to define a RESTful service and then render it in an acceptable format for client consumption.

A critical ingredient to any RESTful service is adding links to relevant operations. To make your controller more RESTful, add links like this:

Getting a single item resource

```
@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"));
}
```

COPY

> This tutorial is based on Spring MVC and uses the static helper methods from `WebMvcLinkBuilder` to build these links. If you are using Spring WebFlux in your project, you must instead use `WebFluxLinkBuilder`.

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.

Roy Fielding encourages building APIs with the same techniques that made the web successful, and links are one of them.

If you restart the application and query the employee record of *Bilbo*, you'll get a slightly different response than earlier:

> Curling prettier
>
> When your curl output gets more complex it can become hard to read. Use thi
> json returned by curl:
>
> ```
> # The indicated part pipes the output to json_pp and asks it to make your JSON pretty.
> #                                  v------------------v
> curl -v localhost:8080/employees/1 | json_pp
> ```

RESTful representation of a single employee

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

COPY

This decompressed output shows not only the data elements you saw earlier (`id`, `name` and `role`), but also a `_links` entry containing two URIs. This entire document is formatted using HAL.

HAL is a lightweight mediatype that allows encoding not just data but also hypermedia controls, alerting consumers to other parts of the API they can navigate toward. In this case, there is a "self" link (kind of like a `this` statement in code) along with a link back to the **aggregate root**.

To make the aggregate root ALSO more RESTful, you want to include top level links while ALSO including any RESTful components within.

So we turn this

Getting an aggregate root

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

COPY

into this

Getting an aggregate root **resource**

```
@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")))
```

COPY

```
      .collect(Collectors.toList());

  return CollectionModel.of(employees,
linkTo(methodOn(EmployeeController.class).all()).withSelfRel());
}
```

Wow! That method, which used to just be `repository.findAll()`, is all grown up! Not to worry. Let's unpack it.

`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!)

If you restart the application and fetch the aggregate root, you can see what it looks like now.

RESTful representation of a collection of employee resources

COPY

```
{
  "_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.

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.

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.

Simply put, you need to define a function that converts `Employee` objects to `EntityModel<Employee>` objects. While you could easily code this method yourself,

there are benefits down the road of implementing Spring HATEOAS's `RepresentationModelAssembler` interface—which will do the work for you.

evolution/src/main/java/payroll/EmployeeModelAssembler.java

```
package 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 EmployeeModelAssembler implements RepresentationModelAssembler<Employee,
EntityModel<Employee>> {

  @Override
  public EntityModel<Employee> toModel(Employee employee) {

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

COPY

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

All the code you saw earlier in the controller can be moved into this class. And by applying Spring Framework's `@Component` annotation, the assembler will be automatically created when the app starts.

> Spring HATEOAS's abstract base class for all models is `RepresentationModel`. But for simplicity, I recommend using `EntityModel<T>` as your mechanism to easily wrap all POJOs as models.

To leverage this assembler, you only have to alter the `EmployeeController` by injecting the assembler in the constructor.

Injecting EmployeeModelAssembler into the controller

```
@RestController
class EmployeeController {
```

COPY

```
  private final EmployeeRepository repository;

  private final EmployeeModelAssembler assembler;

  EmployeeController(EmployeeRepository repository, EmployeeModelAssembler assembler) {

    this.repository = repository;
    this.assembler = assembler;
  }

  ...

}
```

From here, you can use that assembler in the single-item employee method:

Getting single item resource using the assembler

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

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

  return assembler.toModel(employee);
}
```

COPY

This code is almost the same, except instead of creating the `EntityModel<Employee>` instance here, you delegate it to the assembler. Maybe that doesn't look like much.

Applying the same thing in the aggregate root controller method is more impressive:

Getting aggregate root resource using the assembler

```
@GetMapping("/employees")
CollectionModel<EntityModel<Employee>> all() {

  List<EntityModel<Employee>> employees = repository.findAll().stream() //
      .map(assembler::toModel) //
      .collect(Collectors.toList());

  return CollectionModel.of(employees,
linkTo(methodOn(EmployeeController.class).all()).withSelfRel());
}
```

COPY

The code is, again, almost the same, however you get to replace all that `EntityModel<Employee>` creation logic with `map(assembler::toModel)`. Thanks to Java 8

method references, it's super easy to plug it in and simplify your controller.

> A key design goal of Spring HATEOAS is to make it easier to do The Right Thing™. In this scenario: adding hypermedia to your service without hard coding a thing.

At this stage, you've created a Spring MVC REST controller that actually produces hypermedia-powered content! Clients that don't speak HAL can ignore the extra bits while consuming the pure data. Clients that DO speak HAL can navigate your empowered API.

But that is not the only thing needed to build a truly RESTful service with Spring.

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.

In the "olden" days, upgrades were notorious for breaking clients. In other words, an upgrade to the server required an update to the client. In this day and age, hours or even minutes of downtime spent doing an upgrade can cost millions in lost revenue.

Some companies require that you present management with a plan to minimize downtime. In the past, you could get away with upgrading at 2:00 a.m. on a Sunday when load was at a minimum. But in today's Internet-based e-commerce

with international customers in other time zones, such strategies are not as effective.

SOAP-based services and CORBA-based services were incredibly brittle. It was hard to roll out a server that could support both old and new clients. With REST-based practices, it's much easier. Especially using the Spring stack.

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

You can always add columns (fields) to a database table. But don't take one away. The principle in RESTful services is the same.

Add new fields to your JSON representations, but don't take any away. Like this:

JSON that supports multiple clients

```
{
  "id": 1,
  "firstName": "Bilbo",
  "lastName": "Baggins",
```

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

Notice how this format shows `firstName`, `lastName`, AND `name`? While it sports duplication of information, the purpose is to support both old and new clients. That means you can upgrade the server without requiring clients upgrade at the same time. A good move that should reduce downtime.

And not only should you show this information in both the "old way" and the "new way", you should also process incoming data both ways.

How? Simple. Like this:

Employee record that handles both "old" and "new" clients

COPY

```
package 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 firstName;
  private String lastName;
  private String role;

  Employee() {}

  Employee(String firstName, String lastName, String role) {

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

  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];
  }

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

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

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

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

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

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

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

  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.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 + '\'' + '}';
```

```
  }
}
```

This class is very similar to the previous version of `Employee`. Let's go over the changes:

- Field `name` has been replaced by `firstName` and `lastName`.
- A "virtual" getter for the old `name` property, `getName()` is defined. It uses the `firstName` and `lastName` fields to produce a value.
- A "virtual" setter for the old `name` property is also defined, `setName()`. It parses an incoming string and stores it into the proper fields.

Of course not EVERY change to your API is as simple as splitting a string or merging two strings. But it's surely not impossible to come up with a set of transforms for most scenarios, right?

> Don't forget to go and change how you preload your database (in `LoadDatabase`) to use this new constructor.
>
> ```
> log.info("Preloading " + repository.save(new Employee("Bilbo", "Baggins",
> "burglar")));
> log.info("Preloading " + repository.save(new Employee("Frodo", "Baggins",
> "thief")));
> ```
> COPY

Proper Responses

Another step in the right direction involves ensuring that each of your REST methods returns a proper response. Update the POST method like this:

POST that handles "old" and "new" client requests

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

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

  return ResponseEntity //
```
COPY

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

- 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:

```
$ curl -v -X POST localhost:8080/employees -H 'Content-Type:application/json' -d '{"name":
```

The output is shown below:

```
> POST /employees HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
> Content-Type:application/json
> Content-Length: 46
>
< Location: http://localhost:8080/employees/3
< Content-Type: application/hal+json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Fri, 10 Aug 2018 19:44:43 GMT
<
{
  "id": 3,
  "firstName": "Samwise",
  "lastName": "Gamgee",
  "role": "gardener",
  "name": "Samwise Gamgee",
  "_links": {
    "self": {
      "href": "http://localhost:8080/employees/3"
    },
    "employees": {
      "href": "http://localhost:8080/employees"
    }
  }
}
```

This not only has the resulting object rendered in HAL (both `name` as well as `firstName` / `lastName`), but also the **Location** header populated with `http://localhost:8080/employees/3`. A hypermedia powered client could opt to "surf" to this new resource and proceed to interact with it.

The PUT controller method needs similar tweaks:

Handling a PUT for different clients

```
@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);
}
```

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.

```
$ curl -v -X PUT localhost:8080/employees/3 -H 'Content-Type:application/json' -d '{"name"
```

```
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> PUT /employees/3 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
> Content-Type:application/json
> Content-Length: 49
>
< HTTP/1.1 201
< Location: http://localhost:8080/employees/3
< Content-Type: application/hal+json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Fri, 10 Aug 2018 19:52:56 GMT
{
        "id": 3,
        "firstName": "Samwise",
        "lastName": "Gamgee",
        "role": "ring bearer",
        "name": "Samwise Gamgee",
        "_links": {
                "self": {
                        "href": "http://localhost:8080/employees/3"
                },
                "employees": {
                        "href": "http://localhost:8080/employees"
                }
        }
}
```

That employee resource has now been updated and the location URI sent back.

Finally, update the DELETE operation suitably:

Handling DELETE requests

COPY

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

  repository.deleteById(id);

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

This returns an **HTTP 204 No Content** response.

```
$ curl -v -X DELETE localhost:8080/employees/1

* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> DELETE /employees/1 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
```

```
< HTTP/1.1 204
< Date: Fri, 10 Aug 2018 21:30:26 GMT
```

> Making changes to the fields in the `Employee` class will require coordination with your database team, so that they can properly migrate existing content into the new columns.

You are now ready for an upgrade that will NOT disturb existing clients while newer clients can take advantage of the enhancements!

By the way, are you worried about sending too much information over the wire? In some systems where every byte counts, evolution of APIs may need to take a backseat. But don't pursue such premature optimization until you measure.

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.

As a first step, define an `Order` record:

links/src/main/java/payroll/Order.java

```
package 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 + '}';
  }
}
```

- 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`:

links/src/main/java/payroll/Status.java

```
package payroll;

enum Status {

  IN_PROGRESS, //
  COMPLETED, //
  CANCELLED
}
```

COPY

This `enum` captures the various states an `Order` can occupy. For this tutorial, let's keep it simple.

To support interacting with orders in the database, you must define a corresponding Spring Data repository:

Spring Data JPA's `JpaRepository` base interface

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

COPY

With this in place, you can now define a basic `OrderController`:

links/src/main/java/payroll/OrderController.java

```
@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));
  }
}
```

- 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).

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:

links/src/main/java/payroll/OrderModelAssembler.java

```
package 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("ca
ncel"));

orderModel.add(linkTo(methodOn(OrderController.class).complete(order.getId())).withRel("
complete"));
    }

    return orderModel;
  }
}
```

This resource assembler always includes the **self** link to the single-item resource as well as a link back to the aggregate root. But it also includes two conditional links to `OrderController.cancel(id)` as well as `OrderController.complete(id)` (not yet defined). These links are ONLY shown when the order's status is `Status.IN_PROGRESS`.

If clients can adopt HAL and the ability to read links instead of simply reading the data of plain old JSON, they can trade in the need for domain knowledge about the order system. This naturally reduces coupling between client and server. And it opens the door to tuning the flow of order fulfillment without breaking clients in the process.

To round out order fulfillment, add the following to the `OrderController` for the `cancel` operation:

Creating a "cancel" operation in the OrderController

```
@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"));
}
```

It checks the `Order` status before allowing it to be cancelled. If it's not a valid state, it returns an RFC-7807 `Problem`, a hypermedia-supporting error container. If the transition is indeed valid, it transitions the `Order` to `CANCELLED`.

And add this to the `OrderController` as well for order completion:

Creating a "complete" operation in the OrderController

```
@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"));
}
```

This implements similar logic to prevent an `Order` status from being completed unless in the proper state.

Let's update `LoadDatabase` to pre-load some `Order`s along with the `Employee`s it was loading before.

Updating the database pre-loader

```
package 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 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);
      });

    };
  }
}
```

Now you can test things out!

To use the newly minted order service, just perform a few operations:

```
$ curl -v http://localhost:8080/orders

{
  "_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"
    }
  }
}
```


This HAL document immediately shows different links for each order, based upon its present state.

- The first order, being **COMPLETED** only has the navigational links. The state transition links are not shown.
- The second order, being **IN_PROGRESS** additionally has the **cancel** link as well as the **complete** link.

Try cancelling an order:

```
$ curl -v -X DELETE http://localhost:8080/orders/4/cancel

> DELETE /orders/4/cancel HTTP/1.1
```

```
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/hal+json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Mon, 27 Aug 2018 15:02:10 GMT
<
{
  "id": 4,
  "description": "iPhone",
  "status": "CANCELLED",
  "_links": {
    "self": {
      "href": "http://localhost:8080/orders/4"
    },
    "orders": {
      "href": "http://localhost:8080/orders"
    }
  }
}
```

This response shows an **HTTP 200** status code indicating it was successful. The response HAL document shows that order in its new state (`CANCELLED`). And the state-altering links gone.

If you try the same operation again...

```
$ curl -v -X DELETE http://localhost:8080/orders/4/cancel

* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> DELETE /orders/4/cancel HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 405
< Content-Type: application/problem+json
< Transfer-Encoding: chunked
< Date: Mon, 27 Aug 2018 15:03:24 GMT
<
{
  "title": "Method not allowed",
  "detail": "You can't cancel an order that is in the CANCELLED status"
}
```

...you see an **HTTP 405 Method Not Allowed** response. **DELETE** has become an invalid operation. The `Problem` response object clearly indicates that you aren't allowed to "cancel" an order already in the "CANCELLED" status.

Additionally, trying to complete the same order also fails:

```
$ curl -v -X PUT localhost:8080/orders/4/complete

* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> PUT /orders/4/complete HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 405
< Content-Type: application/problem+json
< Transfer-Encoding: chunked
< Date: Mon, 27 Aug 2018 15:05:40 GMT
<
{
  "title": "Method not allowed",
  "detail": "You can't complete an order that is in the CANCELLED status"
}
```

With all this in place, your order fulfillment service is capable of conditionally showing what operations are available. It also guards against invalid operations.

By leveraging the protocol of hypermedia and links, clients can be built sturdier and less likely to break simply because of a change in the data. And Spring HATEOAS eases building the hypermedia you need to serve to your clients.

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.

This concludes our tutorial on how to build RESTful services using Spring. Each section of this tutorial is managed as a separate subproject in a single github repo:

- **nonrest** — Simple Spring MVC app with no hypermedia
- **rest** — Spring MVC + Spring HATEOAS app with HAL representations of each resource
- **evolution** — REST app where a field is evolved but old data is retained for backward compatibility
- **links** — REST app where conditional links are used to signal valid state changes to clients

To view more examples of using Spring HATEOAS see https://github.com/spring-projects/spring-hateoas-examples.

To do some more exploring check out the following video by Spring teammate Oliver Gierke:

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.