Building a Hypermedia-Driven RESTful Web Service

This guide walks you through the process of creating a “Hello, World” Hypermedia-driven REST web service with Spring.

Hypermedia is an important aspect of REST. It lets you build services that decouple client and server to a large extent and let them evolve independently. The representations returned for REST resources contain not only data but also links to related resources. Thus, the design of the representations is crucial to the design of the overall service.

What You Will Build

You will build a hypermedia-driven REST service with Spring HATEOAS: a library of APIs that you can use to create links that point to Spring MVC controllers, build up resource representations, and control how they are rendered into supported hypermedia formats (such as HAL).

The service will accept HTTP GET requests at `http://localhost:8080/greeting`.

It will respond with a JSON representation of a greeting that is enriched with the simplest possible hypermedia element, a link that points to the resource itself. The following listing shows the output:

```
{
  "content":"Hello, World!",
  "_links":{
    "self":{
      "href":"http://localhost:8080/greeting?name=World"
    }
  }
}
```

The response already indicates that you can customize the greeting with an optional `name` parameter in the query string, as the following listing shows:

```
http://localhost:8080/greeting?name=User
```

COPY

The `name` parameter value overrides the default value of `World` and is reflected in the response, as the following listing shows:

```
{
  "content":"Hello, User!",
  "_links":{
    "self":{
      "href":"http://localhost:8080/greeting?name=User"
    }
  }
}
```

COPY

What You Need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 4+ or Maven 3.2+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Starting with Spring Initializr.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-rest-hateoas.git`
- cd into `gs-rest-hateoas/initial`
- Jump ahead to Create a Resource Representation Class.

When you finish, you can check your results against the code in `gs-rest-hateoas/complete`.

Starting with Spring Initializr

If you use Maven, visit the Spring Initializr to generate a new project with the required dependency (Spring HATEOAS).

The following listing shows the `pom.xml` file that is created when you choose Maven:

COPY

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.4.3</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>rest-hateoas</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>rest-hateoas</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>1.8</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-hateoas</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
```

```
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>

</project>
```

If you use Gradle, visit the Spring Initializr to generate a new project with the required dependency (Spring HATEOAS).

The following listing shows the `build.gradle` file that is created when you choose Gradle:

COPY

```
plugins {
        id 'org.springframework.boot' version '2.4.3'
        id 'io.spring.dependency-management' version '1.0.11.RELEASE'
        id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
        mavenCentral()
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-hateoas'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
        useJUnitPlatform()
}
```

Manual Initialization (optional)

If you want to initialize the project manually rather than use the links shown earlier, follow the steps given below:

1. Navigate to https://start.spring.io. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring HATEOAS**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

> If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

Adding a JSON Library

Because you will use JSON to send and receive information, you need a JSON library. In this guide, you will use the Jayway JsonPath library.

To include the library in a Maven build, add the following dependency to your `pom.xml` file:

```
<dependency>
  <groupId>com.jayway.jsonpath</groupId>
  <artifactId>json-path</artifactId>
  <scope>test</scope>
</dependency>
```

The following listing shows the finished `pom.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
```

```
		<version>2.4.3</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.example</groupId>
	<artifactId>rest-hateoas</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>rest-hateoas</name>
	<description>Demo project for Spring Boot</description>
	<properties>
		<java.version>1.8</java.version>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-hateoas</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

To include the libary in a Gradle build, add the following dependency to your `build.gradle` file:

```
testCompile 'com.jayway.jsonpath:json-path'
```

COPY

The following listing shows the finished `build.gradle` file:

COPY

```
plugins {
	id 'org.springframework.boot' version '2.4.3'
	id 'io.spring.dependency-management' version '1.0.11.RELEASE'
	id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
	mavenCentral()
}
```

```
dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-hateoas'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
        useJUnitPlatform()
}
```

Create a Resource Representation Class

Now that you have set up the project and build system, you can create your web service.

Begin the process by thinking about service interactions.

The service will expose a resource at `/greeting` to handle `GET` requests, optionally with a `name` parameter in the query string. The `GET` request should return a `200 OK` response with JSON in the body to represent a greeting.

Beyond that, the JSON representation of the resource will be enriched with a list of hypermedia elements in a `_links` property. The most rudimentary form of this is a link that points to the resource itself. The representation should resemble the following listing:

COPY

```
{
  "content":"Hello, World!",
  "_links":{
    "self":{
      "href":"http://localhost:8080/greeting?name=World"
    }
  }
}
```

The `content` is the textual representation of the greeting. The `_links` element contains a list of links (in this case, exactly one with the relation type of `rel` and the `href` attribute pointing to the resource that was accessed).

To model the greeting representation, create a resource representation class. As the `_links` property is a fundamental property of the representation model,

Spring HATEOAS ships with a base class (called `RepresentationModel`) that lets you add instances of `Link` and ensures that they are rendered as shown earlier.

Create a plain old java object that extends `RepresentationModel` and adds the field and accessor for the content as well as a constructor, as the following listing (from `src/main/java/com/example/resthateoas/Greeting.java`) shows:

```
package com.example.resthateoas;

import org.springframework.hateoas.RepresentationModel;

import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;

public class Greeting extends RepresentationModel<Greeting> {

        private final String content;

        @JsonCreator
        public Greeting(@JsonProperty("content") String content) {
                this.content = content;
        }

        public String getContent() {
                return content;
        }
}
```

COPY

- @JsonCreator: Signals how Jackson can create an instance of this POJO.
- @JsonProperty: Marks the field into which Jackson should put this constructor argument.

> As you will see in later in this guide, Spring will use the Jackson JSON library to automatically marshal instances of type `Greeting` into JSON.

Next, create the resource controller that will serve these greetings.

Create a REST Controller

In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. The components are identified by the `@RestController`

annotation, which combines the `@Controller` and `@ResponseBody` annotations. The following `GreetingController` (from `src/main/java/com/example/resthateoas/GreetingController.java`) handles `GET` requests for `/greeting` by returning a new instance of the `Greeting` class:

```
package com.example.resthateoas;

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

import org.springframework.http.HttpEntity;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@RestController
public class GreetingController {

        private static final String TEMPLATE = "Hello, %s!";

        @RequestMapping("/greeting")
        public HttpEntity<Greeting> greeting(
                @RequestParam(value = "name", defaultValue = "World") String name) {

                Greeting greeting = new Greeting(String.format(TEMPLATE, name));

greeting.add(linkTo(methodOn(GreetingController.class).greeting(name)).withSelfRel());

                return new ResponseEntity<>(greeting, HttpStatus.OK);
        }
}
```

COPY

This controller is concise and simple, but there is plenty going on. We break it down step by step.

The `@RequestMapping` annotation ensures that HTTP requests to `/greeting` are mapped to the `greeting()` method.

> The above example does not specify `GET` vs. `PUT`, `POST`, and so forth, because `@RequestMapping` maps *all* HTTP operations by default. Use `@GetMapping("/greeting")` to narrow this mapping. In that case you also want to `import org.springframework.web.bind.annotation.GetMapping;`.

`@RequestParam` binds the value of the query string parameter `name` into the `name` parameter of the `greeting()` method. This query string parameter is implicitly not `required` because of the use of the `defaultValue` attribute. If it is absent in the request, the `defaultValue` of `World` is used.

Because the `@RestController` annotation is present on the class, an implicit `@ResponseBody` annotation is added to the `greeting` method. This causes Spring MVC to render the returned `HttpEntity` and its payload (the `Greeting`) directly to the response.

The most interesting part of the method implementation is how you create the link that points to the controller method and how you add it to the representation model. Both `linkTo(…)` and `methodOn(…)` are static methods on `ControllerLinkBuilder` that let you fake a method invocation on the controller. The returned `LinkBuilder` will have inspected the controller method's mapping annotation to build up exactly the URI to which the method is mapped.

> Spring HATEOAS respects various `X-FORWARDED-` headers. If you put a Spring HATEOAS service behind a proxy and properly configure it with `X-FORWARDED-HOST` headers, the resulting links will be properly formatted.

The call to `withSelfRel()` creates a `Link` instance that you add to the `Greeting` representation model.

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration`: Tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if `spring-webmvc` is on the classpath, this annotation flags the application as a

web application and activates key behaviors, such as setting up a `DispatcherServlet`.

- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the `com/example` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there was not a single line of XML? There is no `web.xml` file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

```
java -jar build/libs/gs-rest-hateoas-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-rest-hateoas-0.1.0.jar
```

> The steps described here create a runnable JAR. You can also build a classic WAR file.

Logging output is displayed. The service should be up and running within a few seconds.

Test the Service

Now that the service is up, visit http://localhost:8080/greeting, where you should see the following content:

```
{
  "content":"Hello, World!",
  "_links":{
    "self":{
      "href":"http://localhost:8080/greeting?name=World"
    }
  }
}
```

Provide a `name` query string parameter by visiting the following URL: `http://localhost:8080/greeting?name=User`. Notice how the value of the `content` attribute changes from `Hello, World!` to `Hello, User!` and the `href` attribute of the `self` link reflects that change as well, as the following listing shows:

```
{
  "content":"Hello, User!",
  "_links":{
    "self":{
      "href":"http://localhost:8080/greeting?name=User"
    }
  }
}
```

This change demonstrates that the `@RequestParam` arrangement in `GreetingController` works as expected. The `name` parameter has been given a default value of `World` but can always be explicitly overridden through the query string.

Summary

Congratulations! You have just developed a hypermedia-driven RESTful web service with Spring HATEOAS.

See Also

The following guides may also be helpful:

- Building a RESTful Web Service
- Accessing GemFire Data with REST
- Accessing MongoDB Data with REST
- Accessing data with MySQL
- Accessing JPA Data with REST
- Accessing Neo4j Data with REST
- Consuming a RESTful Web Service
- Consuming a RESTful Web Service with AngularJS
- Consuming a RESTful Web Service with jQuery
- Consuming a RESTful Web Service with rest.js
- Securing a Web Application
- Building REST services with Spring
- React.js and Spring Data REST
- Building an Application with Spring Boot
- Creating API Documentation with Restdocs
- Enabling Cross Origin Requests for a RESTful Web Service

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