< ALL GUIDES

# Building a RESTful Web Service

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

## What You Will Build

You will build a service that will accept HTTP GET requests at `http://localhost:8080/greeting`.

It will respond with a JSON representation of a greeting, as the following listing shows:

```
{"id":1,"content":"Hello, World!"}
```
COPY

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:

```
{"id":1,"content":"Hello, 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.

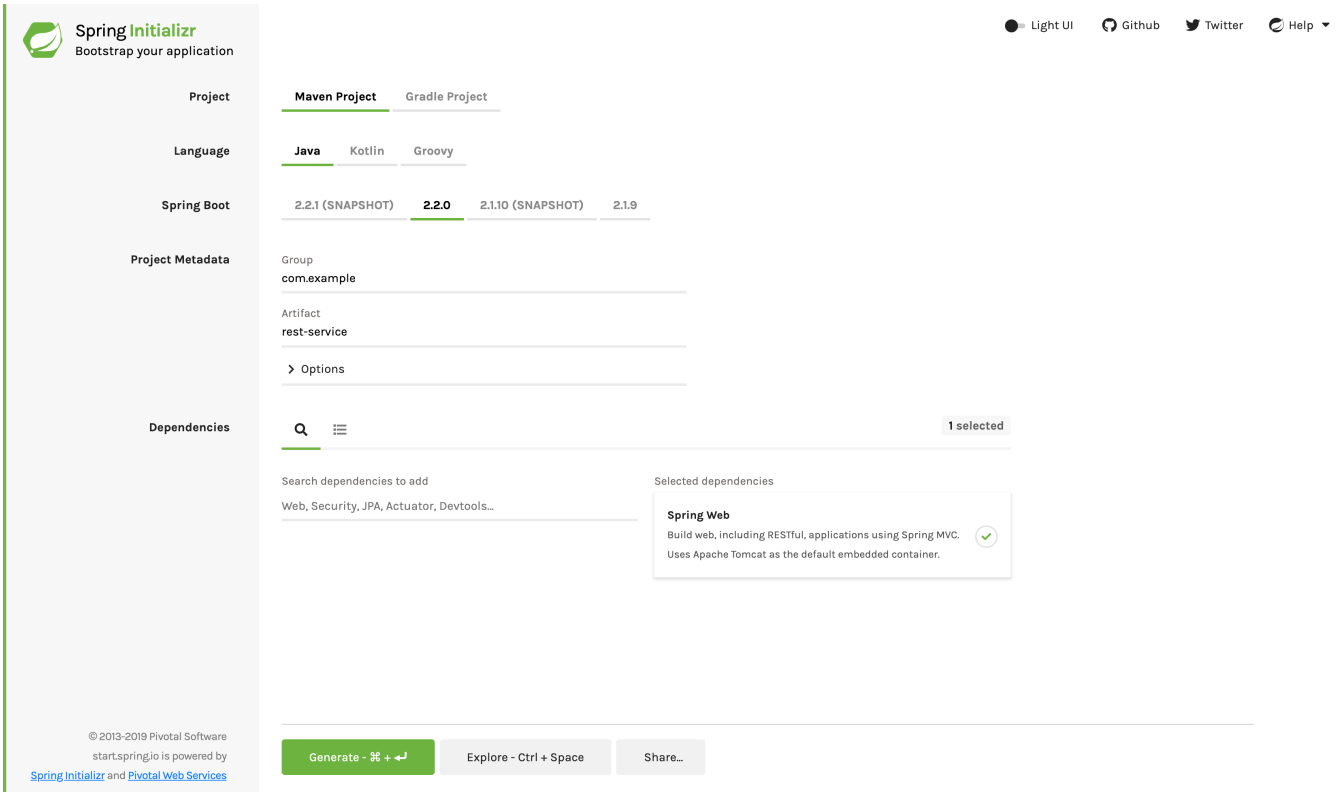
To **skip the basics**, do the following:

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

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

## Starting with Spring Initializr

For all Spring applications, you should start with the Spring Initializr. The Initializr offers a fast way to pull in all the dependencies you need for an application and does a lot of the setup for you. This example needs only the Spring Web dependency. The following image shows the Initializr set up for this sample project:




The preceding image shows the Initializr with Maven chosen as the build tool. You can also use Gradle. It also shows values of `com.example` and `rest-service` as the Group and Artifact, respectively. You will use those values throughout the rest of this sample.

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>2.2.2.RELEASE</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>rest-service</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>rest-service</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-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
			<exclusions>
				<exclusion>
					<groupId>org.junit.vintage</groupId>
					<artifactId>junit-vintage-
engine</artifactId>
				</exclusion>
			</exclusions>
		</dependency>
	</dependencies>

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

</project>
```

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

COPY

```
plugins {
	id 'org.springframework.boot' version '2.2.0.RELEASE'
	id 'io.spring.dependency-management' version '1.0.8.RELEASE'
	id 'java'
}

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

repositories {
	mavenCentral()
}

dependencies {
	implementation 'org.springframework.boot:spring-boot-starter-web'
	testImplementation('org.springframework.boot:spring-boot-starter-test')
{
		exclude group: 'org.junit.vintage', module: 'junit-vintage-
engine'
	}
```

```
}

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 handle `GET` requests for `/greeting`, optionally with a `name` parameter in the query string. The `GET` request should return a `200 OK` response with JSON in the body that represents a greeting. It should resemble the following output:

COPY

```
{
    "id": 1,
    "content": "Hello, World!"
}
```

The `id` field is a unique identifier for the greeting, and `content` is the textual representation of the greeting.

To model the greeting representation, create a resource representation class. To do so, provide a plain old Java object with fields, constructors, and accessors for the `id` and `content` data, as the following listing (from `src/main/java/com/example/restservice/Greeting.java`) shows:

COPY

```
package com.example.restservice;

public class Greeting {

        private final long id;
        private final String content;

        public Greeting(long id, String content) {
                this.id = id;
                this.content = content;
        }

        public long getId() {
                return id;
        }

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

```
        }
}
```

This application uses the Jackson JSON library to automatically marshal instances of type `Greeting` into JSON. Jackson is included by default by the web starter.

## Create a Resource Controller

In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. These components are identified by the `@RestController` annotation, and the `GreetingController` shown in the following listing (from `src/main/java/com/example/restservice/GreetingController.java`) handles `GET` requests for `/greeting` by returning a new instance of the `Greeting` class:

```
package com.example.restservice;

import java.util.concurrent.atomic.AtomicLong;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {

        private static final String template = "Hello, %s!";
        private final AtomicLong counter = new AtomicLong();

        @GetMapping("/greeting")
        public Greeting greeting(@RequestParam(value = "name", defaultValue =
"World") String name) {
                return new Greeting(counter.incrementAndGet(),
String.format(template, name));
        }
}
```

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

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

There are companion annotations for other HTTP verbs (e.g. `@PostMapping` for

POST). There is also a `@RequestMapping` annotation that they all derive from, and can serve as a synonym (e.g. `@RequestMapping(method=GET)`).

`@RequestParam` binds the value of the query string parameter `name` into the `name` parameter of the `greeting()` method. If the `name` parameter is absent in the request, the `defaultValue` of `World` is used.

The implementation of the method body creates and returns a new `Greeting` object with `id` and `content` attributes based on the next value from the `counter` and formats the given `name` by using the greeting `template`.

A key difference between a traditional MVC controller and the RESTful web service controller shown earlier is the way that the HTTP response body is created. Rather than relying on a view technology to perform server-side rendering of the greeting data to HTML, this RESTful web service controller populates and returns a `Greeting` object. The object data will be written directly to the HTTP response as JSON.

This code uses Spring `@RestController` annotation, which marks the class as a controller where every method returns a domain object instead of a view. It is shorthand for including both `@Controller` and `@ResponseBody`.

The `Greeting` object must be converted to JSON. Thanks to Spring's HTTP message converter support, you need not do this conversion manually. Because Jackson 2 is on the classpath, Spring's `MappingJackson2HttpMessageConverter` is automatically chosen to convert the `Greeting` instance to JSON.

`@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-service-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-service-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:

```
{"id":1,"content":"Hello, World!"}
```
COPY

Provide a `name` query string parameter by visiting `http://localhost:8080/greeting?name=User`. Notice how the value of the `content`

attribute changes from `Hello, World!` to `Hello, User!`, as the following listing shows:

```
{"id":2,"content":"Hello, User!"}
```

COPY

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

Notice also how the `id` attribute has changed from `1` to `2`. This proves that you are working against the same `GreetingController` instance across multiple requests and that its `counter` field is being incremented on each call as expected.

## Summary

Congratulations! You have just developed a RESTful web service with Spring.

## See Also

The following guides may also be helpful:

- 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
- Building a Hypermedia-Driven RESTful Web Service
- Circuit Breaker

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