**Building a RESTful Web Microservice**

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!"}

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

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!"}

What You Need

* About 15 minutes
* A favorite text editor or IDE
* [Java 17](https://www.oracle.com/java/technologies/downloads/) or later
* [Gradle 7.5+](https://gradle.org/install/) or [Maven 3.5+](https://maven.apache.org/download.cgi)
* You can also import the code straight into your IDE:
  + [Spring Tool Suite (STS)](https://spring.io/guides/gs/sts)
  + [IntelliJ IDEA](https://spring.io/guides/gs/intellij-idea/)
  + [VSCode](https://spring.io/guides/gs/guides-with-vscode/)

How to complete this guide

Clone this repository and follow the instructions listed here.

Starting with Spring Initializr (SKIP THIS STEP)

You can use this [pre-initialized project](https://start.spring.io/#!type=maven-project&groupId=com.example&artifactId=rest-service&name=rest-service&description=Demo%20project%20for%20Spring%20Boot&packageName=com.example.rest-service&dependencies=web) and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

To manually initialize the project:

1. Navigate to [https://start.spring.io](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 Web**.
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. |
|  | You can also fork the project from Github and open it in your IDE or other editor. |

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:

{

"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 Java record class for the id and content data, as the following listing (from src/main/java/com/example/restservice/Greeting.java) shows:

package com.example.restservice;

public record Greeting(long id, String content) { }

|  |  |
| --- | --- |
|  | This application uses the [Jackson JSON](https://github.com/FasterXML/jackson) 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](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) 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](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) 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](https://github.com/FasterXML/jackson) is on the classpath, Spring’s [MappingJackson2HttpMessageConverter](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/converter/json/MappingJackson2HttpMessageConverter.html) 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](https://spring.io/guides/gs/convert-jar-to-war/). |

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:

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

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:

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

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.