**Building a RESTful Web Service with Spring Boot Actuator**

[Spring Boot Actuator](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready) is a sub-project of Spring Boot. It adds several production grade services to your application with little effort on your part. In this guide, you will build an application and then see how to add these services.

What You Will build

This guide takes you through creating a “Hello, world” RESTful web service with Spring Boot Actuator. You will build a service that accepts the following HTTP GET request:

$ curl http://localhost:9000/hello-world

It responds with the following JSON:

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

There are also many features added to your application for managing the service in a production (or other) environment. The business functionality of the service you build is the same as in [Building a RESTful Web Service](https://spring.io/guides/gs/rest-service). You need need not use that guide to take advantage of this one, although it might be interesting to compare the results.

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

Fork the repo and follow the guide.

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=actuator-service&name=actuator-service&description=Demo%20project%20for%20Spring%20Boot&packageName=com.example.actuator-service&dependencies=web,actuator) 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** and **Spring Boot Actuator**.
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. |

Run the Empty Service

The Spring Initializr creates an empty application that you can use to get started. The following example (from src/main/java/com/example/actuatorservice/ActuatorServiceApplication in the initial directory) shows the class created by the Spring Initializr:

package com.example.actuatorservice;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ActuatorServiceApplication {

public static void main(String[] args) {

SpringApplication.run(ActuatorServiceApplication.class, args);

}

}

The @SpringBootApplication annotation provides a load of defaults (like the embedded servlet container), depending on the contents of your classpath and other things. It also turns on Spring MVC’s @EnableWebMvc annotation, which activates web endpoints.

There are no endpoints defined in this application, but there is enough to launch things and see some of Actuator’s features. The SpringApplication.run() command knows how to launch the web application. All you need to do is run the following command:

$ ./gradlew clean build && java -jar build/libs/gs-actuator-service-0.1.0.jar

You have yet to write any code, so what is happening? To see the answer, wait for the server to start, open another terminal, and try the following command (shown with its output):

$ curl localhost:8080

{"timestamp":1384788106983,"error":"Not Found","status":404,"message":""}

The output of the preceding command indicates that the server is running but that you have not defined any business endpoints yet. Instead of a default container-generated HTML error response, you see a generic JSON response from the Actuator /error endpoint. You can see in the console logs from the server startup which endpoints are provided out of the box. You can try a few of those endpoints, including the /health endpoint. The following example shows how to do so:

$ curl localhost:8080/actuator/health

{"status":"UP"}

The status is UP, so the actuator service is running.

See Spring Boot’s [Actuator Project](https://github.com/spring-projects/spring-boot/tree/main/spring-boot-project/spring-boot-actuator) for more details.

Create a Representation Class

First, you need to give some thought to what your API will look like.

You want to handle GET requests for /hello-world, optionally with a name query parameter. In response to such a request, you want to send back JSON, representing a greeting, that looks something like the following:

{

"id": 1,

"content": "Hello, World!"

}

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

To model the greeting representation, create a representation class. The following listing (from src/main/java/com/example/actuatorservice/Greeting.java) shows the Greeting class:

package com.example.actuatorservice;

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;

}

}

Now that you need to create the endpoint controller that will serve the representation class.

Create a Resource Controller

In Spring, REST endpoints are Spring MVC controllers. The following Spring MVC controller (from src/main/java/com/example/actuatorservice/HelloWorldController.java) handles a GET request for the /hello-world endpoint and returns the Greeting resource:

package com.example.actuatorservice;

import java.util.concurrent.atomic.AtomicLong;

import org.springframework.stereotype.Controller;

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

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

import org.springframework.web.bind.annotation.ResponseBody;

@Controller

public class HelloWorldController {

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

private final AtomicLong counter = new AtomicLong();

@GetMapping("/hello-world")

@ResponseBody

public Greeting sayHello(@RequestParam(name="name", required=false, defaultValue="Stranger") String name) {

return new Greeting(counter.incrementAndGet(), String.format(template, name));

}

}

The key difference between a human-facing controller and a REST endpoint controller is in how the response is created. Rather than rely on a view (such as JSP) to render model data in HTML, an endpoint controller returns the data to be written directly to the body of the response.

The [@ResponseBody](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html) annotation tells Spring MVC not to render a model into a view but, rather, to write the returned object into the response body. It does so by using one of Spring’s message converters. Because Jackson 2 is in the classpath, [MappingJackson2HttpMessageConverter](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/converter/json/MappingJackson2HttpMessageConverter.html) will handle the conversion of a Greeting object to JSON if the request’s Accept header specifies that JSON should be returned.

|  |  |
| --- | --- |
|  | How do you know Jackson 2 is on the classpath? Either run ` mvn dependency:tree` or ./gradlew dependencies, and you get a detailed tree of dependencies that includes Jackson 2.x. You can also see that it comes from [/spring-boot-starter-json](https://github.com/spring-projects/spring-boot/tree/main/spring-boot-project/spring-boot-starters/spring-boot-starter-json), itself imported by [spring-boot-starter-web](https://github.com/spring-projects/spring-boot/blob/main/spring-boot-starters/spring-boot-starter-web/pom.xml). |

Run the Application

You can run the application from a custom main class or directly from one of the configuration classes. For this simple example, you can use the SpringApplication helper class. Note that this is the application class that the Spring Initializr created for you, and you need not even modify it for it to work for this simple application. The following listing (from src/main/java/com/example/actuatorservice/HelloWorldApplication.java) shows the application class:

package com.example.actuatorservice;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class HelloWorldApplication {

public static void main(String[] args) {

SpringApplication.run(HelloWorldApplication.class, args);

}

}

In a conventional Spring MVC application, you would add @EnableWebMvc to turn on key behaviors, including configuration of a DispatcherServlet. But Spring Boot turns on this annotation automatically when it detects **spring-webmvc** on your classpath. This sets you up to build a controller in an upcoming step.

The @SpringBootApplication annotation also brings in a [@ComponentScan](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/ComponentScan.html) annotation, which tells Spring to scan the com.example.actuatorservice package for those controllers (along with any other annotated component classes).

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-actuator-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-actuator-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/). |

Once the service is running (because you ran spring-boot:run in a terminal), you can test it by running the following command in a separate terminal:

$ curl localhost:8080/hello-world

{"id":1,"content":"Hello, Stranger!"}

Switch to a Different Server Port

Spring Boot Actuator defaults to running on port 8080. By adding an application.properties file, you can override that setting. The following listing (from src/main/resources/application.properties)shows that file with the necessary changes:

server.port: 9000

management.server.port: 9001

management.server.address: 127.0.0.1

Run the server again by running the following command in a terminal:

$ ./gradlew clean build && java -jar build/libs/gs-actuator-service-0.1.0.jar

The service now starts on port 9000.

You can test that it is working on port 9000 by running the following commands in a terminal:

$ curl localhost:8080/hello-world

curl: (52) Empty reply from server

$ curl localhost:9000/hello-world

{"id":1,"content":"Hello, Stranger!"}

$ curl localhost:9001/actuator/health

{"status":"UP"}

Test Your Application

To check whether your application works, you should write unit and integration tests for your application. The test class in src/test/java/com/example/actuatorservice/HelloWorldApplicationTests.java ensures that

* Your controller is responsive.
* Your management endpoint is responsive.

Note that the tests start the application on a random port. The following listing shows the test class:

/\*

\* right 2012-2014 the original author or authors.

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a of the License at

\*

\* https://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

package com.example.actuatorservice;

import java.util.Map;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.boot.test.web.client.TestRestTemplate;

import org.springframework.boot.test.web.server.LocalServerPort;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.test.context.TestPropertySource;

import static org.assertj.core.api.BDDAssertions.then;

/\*\*

\* Basic integration tests for service demo application.

\*

\* @author Dave Syer

\*/

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

@TestPropertySource(properties = {"management.port=0"})

public class HelloWorldApplicationTests {

@LocalServerPort

private int port;

@Value("${local.management.port}")

private int mgt;

@Autowired

private TestRestTemplate testRestTemplate;

@Test

public void shouldReturn200WhenSendingRequestToController() throws Exception {

@SuppressWarnings("rawtypes")

ResponseEntity<Map> entity = this.testRestTemplate.getForEntity(

"http://localhost:" + this.port + "/hello-world", Map.class);

then(entity.getStatusCode()).isEqualTo(HttpStatus.OK);

}

@Test

public void shouldReturn200WhenSendingRequestToManagementEndpoint() throws Exception {

@SuppressWarnings("rawtypes")

ResponseEntity<Map> entity = this.testRestTemplate.getForEntity(

"http://localhost:" + this.mgt + "/actuator", Map.class);

then(entity.getStatusCode()).isEqualTo(HttpStatus.OK);

}

}

Summary

Congratulations! You have just developed a simple RESTful service by using Spring, and you added some useful built-in services with Spring Boot Actuator.