RESTful Web Service with Spring Boot Actuator

Spring Boot Actuator 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'll build an application and then see how to add these services.

What you'll do

This guide will take you through creating a "hello world" RESTful web service with Spring Boot Actuator. You'll build a service that accepts an HTTP GET request:

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

It responds with the following JSON:

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

There are also has many features added to your application out-of-the-box 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. You don't need to use that guide to take advantage of this one, although it might be interesting to compare the results.

What you'll need

About 15 minutes

A favorite text editor or IDE

JDK 1.8 or later

Gradle 2.3+ or Maven 3.0+

You can also import the code from this guide as well as view the web page directly into Spring Tool Suite (STS) and work your way through it from there.

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 Build with Gradle.

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-actuator-service.git`

cd into `gs-actuator-service/initial`

Jump ahead to Create a representation class.

When you're finished, you can check your results against the code in `gs-actuator-service/complete`.

Gradle

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with Gradle and Maven is included here. If you're not familiar with either, refer to Building Java Projects with Gradle or Building Java Projects with Maven.

Create the directory structure

In a project directory of your choosing, create the following subdirectory structure; for example, with `mkdir -p src/main/java/hello` on *nix systems:

```
└── src
    └── main
        └── java
            └── hello
```

Create a Gradle build file

Below is the initial Gradle build file.

`build.gradle`

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:1.4.2.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'gs-actuator-service'
    version =  '0.1.0'
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    compile("org.springframework.boot:spring-boot-starter-actuator")
    testCompile("org.springframework.boot:spring-boot-starter-test")
    testCompile("junit:junit")
}
```

The Spring Boot gradle plugin provides many convenient features:

It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.

It searches for the `public static void main()` method to flag as a runnable class.

It provides a built-in dependency resolver that sets the version number to match Spring Boot dependencies. You can override any version you wish, but it will default to Boot's chosen set of versions.

Maven

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with Maven is included here. If you're not familiar with Maven, refer to Building Java Projects with Maven.

Create the directory structure

In a project directory of your choosing, create the following subdirectory structure; for example, with `mkdir -p src/main/java/hello` on *nix systems:

```
└── src
    └── main
        └── java
            └── hello
```

`pom.xml`

```
<?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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.springframework</groupId>
    <artifactId>gs-actuator-service</artifactId>
    <version>0.1.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.2.RELEASE</version>
    </parent>

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

```
    <properties>
        <java.version>1.8</java.version>
    </properties>

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

</project>
```

The Spring Boot Maven plugin provides many convenient features:
It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.
It searches for the `public static void main()` method to flag as a runnable class.
It provides a built-in dependency resolver that sets the version number to match Spring Boot dependencies. You can override any version you wish, but it will default to Boot's chosen set of versions.

IDE

- Read how to import this guide straight into Spring Tool Suite.
- Read how to work with this guide in IntelliJ IDEA.

Run the empty service

For starters, here's an empty Spring MVC application.

`src/main/java/hello/HelloWorldConfiguration.java`

```
package hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloWorldConfiguration {

    public static void main(String[] args) {
        SpringApplication.run(HelloWorldConfiguration.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 that activates web endpoints.

There aren't any endpoints defined in this application, but there's 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 this command.

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

You hardly written any code yet, so what's happening? Wait for the server to start and go to another terminal to try it out:

```
$ curl localhost:8080
{"timestamp":1384788106983,"error":"Not Found","status":404,"message":""}
```

So the server is running, but you haven't 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. Try a few out, for example

```
$ curl localhost:8080/health
{"status":"UP"}
```

You're "UP", so that's good.

Check out Spring Boot's Actuator Project for more details.

Create a representation class

First, 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 will send back JSON, representing a greeting, that looks something like this:

```
{
    "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 representation class:

`src/main/java/hello/Greeting.java`

```
package hello;

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'll create the endpoint controller that will serve the representation class.

Create a resource controller

In Spring, REST endpoints are just Spring MVC controllers. The following Spring MVC controller handles a GET request for /hello-world and returns the `Greeting` resource:

`src/main/java/hello/HelloWorldController.java`

```
package hello;

import java.util.concurrent.atomic.AtomicLong;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/hello-world")
public class HelloWorldController {

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

    @RequestMapping(method=RequestMethod.GET)
    public @ResponseBody Greeting sayHello(@RequestParam(value="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 simply returns the data to be written directly to the body of the response. The `@ResponseBody` 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 this by using one of Spring's message converters. Because Jackson 2 is in the classpath, this means that `MappingJackson2HttpMessageConverter` will handle the conversion of Greeting 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'll get a detailed tree of dependencies which shows

Jackson 2.x. You can also see that it comes from spring-boot-starter-web.

Create an executable main class

You can launch the application from a custom main class, or we can do that directly from one of the configuration classes. The easiest way is to use the `SpringApplication` helper class:

`src/main/java/hello/HelloWorldConfiguration.java`

```
package hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloWorldConfiguration {

	public static void main(String[] args) {
		SpringApplication.run(HelloWorldConfiguration.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` also brings in a `@ComponentScan`, which tells Spring to scan the `hello` 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. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This 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 are using Gradle, you can run the application using `./gradlew bootRun`. Or you can build the JAR file using `./gradlew build`. Then you can run the JAR file:

```
java -jar build/libs/gs-actuator-service-0.1.0.jar
```

If you are using Maven, you can run the application using `./mvnw spring-boot:run`. Or you can build the JAR file with `./mvnw clean package`. Then you can run the JAR file:

```
java -jar target/gs-actuator-service-0.1.0.jar
```

The procedure above will create a runnable JAR. You can also opt to build a classic WAR fileinstead.

```
... service comes up ...
```

Test it:

```
$ curl localhost:8080/hello-world
{"id":1,"content":"Hello, Stranger!"}
```

Switch to a different server port

Spring Boot Actuator defaults to run on port 8080. By adding an `application.properties` file, you can override that setting.

`src/main/resources/application.properties`

```
server.port: 9000
management.port: 9001
management.address: 127.0.0.1
```

Run the server again:

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

... service comes up on port 9000 ...
```

Test it:

```
$ curl localhost:8080/hello-world
curl: (52) Empty reply from server
$ curl localhost:9000/hello-world
{"id":1,"content":"Hello, Stranger!"}
$ curl localhost:9001/health
{"status":"UP"}
```

Test your application

In order to check if your application is functional you should write unit / integration tests of your application. Below you can find an example of such a test that checks:

if your controller is responsive

if your management endpoint is responsive

As you can see for tests we're starting the application on a random port.

`src/test/java/hello/HelloWorldConfigurationTests.java`

```
/*
 * Copyright 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 copy of the License at
 *
 *      http://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 hello;

import java.util.Map;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.test.context.TestPropertySource;
import org.springframework.test.context.junit4.SpringRunner;

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

/**
 * Basic integration tests for service demo application.
 *
 * @author Dave Syer
 */
@RunWith(SpringRunner.class)
@SpringBootTest(classes = HelloWorldConfiguration.class, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
@TestPropertySource(properties = {"management.port=0"})
public class HelloWorldConfigurationTests {

	@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 + "/info", Map.class);

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

```
}
```

Summary

Congratulations! You have just developed a simple RESTful service using Spring. You added some useful built-in services thanks to Spring Boot Actuator.

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