# **Building a RESTful Web Service**

This guide walks you through the process of creating a "hello world" [RESTful web service](https://spring.io/understanding/REST) with Spring and then containerizing it for use as a microservice.

## **What you’ll build**

You’ll build a service that will accept HTTP GET requests at:

http://localhost:8080/greeting

and respond with a [JSON](https://spring.io/understanding/JSON) representation of a greeting:

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

You can customize the greeting with an optional name parameter in the query string:

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

The name parameter value overrides the default value of "World" and is reflected in the response:

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

## **Copy over the pom.xml file**

We need to start with a clean pom.xml file.

## **Create a resource representation class**

Now that you’ve 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 look 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, you create a resource representation class. Provide a plain old java object with fields, constructors, and accessors for the id and content data:

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;

}

}

|  | As you see in steps below, Spring uses the [Jackson JSON](https://github.com/FasterXML/jackson) library to automatically marshal instances of type Greeting into JSON. |
| --- | --- |

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

## **Create a resource controller**

In Spring’s approach to building RESTful web services, HTTP requests are handled by a controller. These components are easily identified by the [@RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) annotation, and the GreetingController below handles GET requests for /greeting by returning a new instance of the Greeting class:

src/main/java/hello/GreetingController.java

package hello;

import java.util.concurrent.atomic.AtomicLong;

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

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();

@RequestMapping("/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’s plenty going on under the hood. Let’s 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 @RequestMapping(method=GET) to narrow this mapping. |
| --- | --- |

@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 idand 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 above is the way that the HTTP response body is created. Rather than relying on a [view technology](https://spring.io/understanding/view-templates) to perform server-side rendering of the greeting data to HTML, this RESTful web service controller simply populates and returns a Greeting object. The object data will be written directly to the HTTP response as JSON.

This code uses Spring 4’s new [@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’s shorthand for @Controller and @ResponseBody rolled together.

The Greeting object must be converted to JSON. Thanks to Spring’s HTTP message converter support, you don’t need to do this conversion manually. Because [Jackson 2](https://wiki.fasterxml.com/JacksonHome) 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.

## **Make the application executable**

Although it is possible to package this service as a traditional [WAR](https://spring.io/understanding/WAR) file for deployment to an external application server, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a good old Java main() method. Along the way, you use Spring’s support for embedding the [Tomcat](https://spring.io/understanding/Tomcat) servlet container as the HTTP runtime, instead of deploying to an external instance.

src/main/java/hello/Application.java

package hello;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

@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.
* Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This 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 hello package, allowing it to find the controllers.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application. Did you notice that there wasn’t a single line of XML? No **web.xml** file either. This web application is 100% pure Java and you didn’t 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. 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 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/rest-0.1.0.jar

## **Test the service**

Now that the service is up, visit <http://localhost:8080/greeting>, where you see:

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

Provide a name query string parameter with <http://localhost:8080/greeting?name=User>. Notice how the value of the content attribute changes from "Hello, World!" to "Hello, User!":

{"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 always 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.

Let’s containerize the application!

In the pom file directory, create a file

nano Dockerfile --- and add these contents:

FROM openjdk:8-jdk-alpine

VOLUME /tmp

ARG DEPENDENCY=target/dependency

COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib

COPY ${DEPENDENCY}/META-INF /app/META-INF

COPY ${DEPENDENCY}/BOOT-INF/classes /app

ENTRYPOINT ["java","-cp","app:app/lib/\*","hello.Application"]

We added a VOLUME pointing to "/tmp" because that is where a Spring Boot application creates working directories for Tomcat by default. The effect is to create a temporary file on your host under "/var/lib/docker" and link it to the container under "/tmp". This step is optional for the simple app that we wrote here, but can be necessary for other Spring Boot applications if they need to actually write in the filesystem.

To reduce Tomcat startup time we added a system property pointing to "/dev/urandom" as a source of entropy. This is not necessary with more recent versions of Spring Boot, if you use the "standard" version of Tomcat (or any other web server).

This Dockerfile has a DEPENDENCY parameter pointing to a directory where we have unpacked the fat jar. If we get that right, it already contains a BOOT-INF/lib directory with the dependency jars in it, and a BOOT-INF/classes directory with the application classes in it. Notice that we are using the application’s own main class hello.Application (this is faster than using the indirection provided by the fat jar launcher).

Update the pom file with these docker add-ons:

<properties>

<docker.image.prefix>learningvoyage</docker.image.prefix>

</properties>

<build>

<plugins>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>dockerfile-maven-plugin</artifactId>

<version>1.4.9</version>

<configuration>

<repository>${docker.image.prefix}/${project.artifactId}</repository>

</configuration>

</plugin>

</plugins>

</build>

The configuration specifies 1 mandatory thing: the repository with the image name, which will end up here as fenago/rest.

Some other properties are optional:

The name of the directory where the fat jar is going to be unpacked, exposing the Maven configuration as a build argument for docker. It can be specified using the <buildArgs/> of the plugin configuration.

The image tag, which ends up as "latest" if not specified. It can be set with the <tag/> element.

Lastly, the make sure that the jar is unpacked before the docker image is created we need to add more pom file configurations:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<executions>

<execution>

<id>unpack</id>

<phase>package</phase>

<goals>

<goal>unpack</goal>

</goals>

<configuration>

<artifactItems>

<artifactItem>

<groupId>${project.groupId}</groupId>

<artifactId>${project.artifactId}</artifactId>

<version>${project.version}</version>

</artifactItem>

</artifactItems>

</configuration>

</execution>

</executions>

</plugin>

Now – let’s build our tagged docker image from our Spring Boot Application with this command:

$ ./mvnw install dockerfile:build

docker images

$ docker run -p 8080:8080 -t fenago/rest

....

2015-03-31 13:25:48.035 INFO 1 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)

2015-03-31 13:25:48.037 INFO 1 --- [ main] hello.RestApplication : Started Application in 5.613 seconds (JVM running for 7.293)

The application is then available on http://localhost:8080 (visit that and it says "Hello Docker World"). To make sure the whole process is really working, change the prefix from "springio" to something else (e.g. ${env.USER}) and go through it again from the build through to the docker run.

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

81c723d22865 fenago/rest:latest "java -Djava.secur..." 34 seconds ago Up 33 seconds 0.0.0.0:8080->8080/tcp goofy\_brown

Stop it with YOUR id:

$ docker stop 81c723d22865

81c723d22865

Delete it with YOUR id (fyi – it is persisted in /var/lib/docker :

$ docker rm 81c723d22865

Launch it with a Spring Profile:

$ docker run -e "SPRING\_PROFILES\_ACTIVE=prod" -p 8080:8080 -t fenago/rest

Debugging the application in a Docker container

To debug the application JPDA Transport can be used. So we’ll treat the container like a remote server. To enable this feature pass a java agent settings in JAVA\_OPTS variable and map agent’s port to localhost during a container run.

$ docker run -e "JAVA\_OPTS=-agentlib:jdwp=transport=dt\_socket,address=5005,server=y,suspend=n" -p 8080:8080 -p 5005:5005 -t fenago/rest

## **Summary**

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