

Building a RESTful Web Service

指导你完成使用Spring创建一个"hello world" RESTful web服务的过程

This guide walks you through the process of creating a "hello world" RESTful web service with Spring.

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 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!"}
```

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 如何完成本指南

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.

从头开始并完成每个步骤，或者绕过你熟悉的基本设置步骤

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-rest-service.git
```
- cd into `gs-rest-service/initial`
- Jump ahead to Create a resource representation class.

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

Create a resource representation class 1、创建资源表示类

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 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 2、创建资源控制器

In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. These components are easily identified by the @RestController 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. This query string parameter is explicitly marked as optional (required=true by default): if it 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 above 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 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 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` is on the classpath, Spring's `MappingJackson2HttpMessageConverter` is automatically chosen to convert the `Greeting` instance to JSON.

Make the application executable 3、使应用程序可执行

Although it is possible to package this service as a traditional `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 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 the `hello` package, allowing it to find the controllers.

启动应用程序

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. Web应用程序是100%的纯Java

Build an executable JAR 构建可执行的JAR文件

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-rest-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-rest-service-0.1.0.jar
```

The procedure above will create a runnable JAR. You can also opt to [build a classic WAR file](#) instead.

Logging output is displayed. The service should be up and running within a few seconds.

Test the service 4、测试服务

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.

Summary

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

Want to write a new guide or contribute to an existing one? Check out our [contribution guidelines](#).

All guides are released with an ASLv2 license for the code, and an [Attribution, NoDerivatives creative commons license](#) for the writing.