Enabling Cross Origin Requests for a RESTful Web Service

This guide walks you through the process of creating a "hello world" RESTful web service with Spring that includes headers for Cross-Origin Resource Sharing (CORS) in the response. You will find more information about Spring CORS support in this blog post.

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!"}
```

This service differs slightly from the one described in Building a RESTful Web Service in that it will use Spring Framework CORS support to add the relevant CORS response headers.

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 straight into your IDE:

    - Spring Tool Suite (STS)
    - IntelliJ IDEA

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-rest-service-cors.git`
- cd into `gs-rest-service-cors/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-cors/complete`.

Build with Gradle

Build with 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-rest-service-cors</artifactId>

    <version>0.1.0</version>



    <parent>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-parent</artifactId>

        <version>1.5.2.RELEASE</version>

    </parent>



    <dependencies>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-web</artifactId>

        </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.

Build with your IDE

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`

```java
package hello;



public class Greeting {



    private final long id;

    private final String content;

    public Greeting() {

        this.id = -1;

        this.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

In Spring's approach to building RESTful web services, HTTP requests are handled by a controller. These components are easily identified by the `@Controller` 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.GetMapping;

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

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

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(required=false,
defaultValue="World") String name) {

        System.out.println("==== in greeting ====");
        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 uses the `@GetMapping` annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.GET)`.

`@RequestParam` binds the value of the query string parameter `name` into the `name` parameter of the `greeting()` method. This query string parameter is not `required`; 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.

To accomplish this, the `@ResponseBody` annotation on the `greeting()` method tells Spring MVC that it does not need to render the greeting object through a server-side view layer, but that instead the greeting object returned *is* the response body, and should be written out directly.

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 is on the classpath, Spring's `MappingJackson2HttpMessageConverter` is automatically chosen to convert the `Greeting` instance to JSON.

Enabling CORS

Controller method CORS configuration

So that the RESTful web service will include CORS access control headers in its response, you just have to add a `@CrossOrigin` annotation to the handler method:

`src/main/java/hello/GreetingController.java`

```
    @CrossOrigin(origins = "http://localhost:9000")

    @GetMapping("/greeting")

    public Greeting greeting(@RequestParam(required=false,
defaultValue="World") String name) {

        System.out.println("==== in greeting ====");

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

    }
```

This `@CrossOrigin` annotation enables cross-origin requests only for this specific method. By default, its allows all origins, all headers, the HTTP methods specified in the `@RequestMapping`annotation and a maxAge of 30 minutes is used. You can customize this behavior by specifying the value of one of the annotation attributes: `origins`, `methods`, `allowedHeaders`, `exposedHeaders`, `allowCredentials` or `maxAge`. In this example, we only allow `http://localhost:9000` to send cross-origin requests.

it is also possible to add this annotation at controller class level as well, in order to enable CORS on all handler methods of this class.

Global CORS configuration

As an alternative to fine-grained annotation-based configuration, you can also define some global CORS configuration as well. This is similar to using a `Filter` based solution, but can be declared within Spring MVC and combined with fine-grained `@CrossOrigin` configuration. By default all origins and `GET`, `HEAD` and `POST` methods are allowed.

`src/main/java/hello/GreetingController.java`

```
    @GetMapping("/greeting-javaconfig")

    public Greeting greetingWithJavaconfig(@RequestParam(required=false,
defaultValue="World") String name) {

        System.out.println("==== in greeting ====");

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

    }
```

`src/main/java/hello/Application.java`

```
    @Bean

    public WebMvcConfigurer corsConfigurer() {

        return new WebMvcConfigurerAdapter() {

            @Override

            public void addCorsMappings(CorsRegistry registry) {

                registry.addMapping("/greeting-
javaconfig").allowedOrigins("http://localhost:9000");

            }

        };

    }
```

You can easily change any properties (like the `allowedOrigins` one in the example), as well as only apply this CORS configuration to a specific path pattern. Global and controller level CORS configurations can also be combined.

Make the application executable

```
    }
```

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`

```java
package hello;



import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.Bean;

import org.springframework.web.servlet.config.annotation.CorsRegistry;

import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import
org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter
;



@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 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-cors-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-cors-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

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.

Now to test that the CORS headers are in place and allowing a Javascript client from another origin to access the service, you'll need to create a Javascript client to consume the service.

First, create a simple Javascript file named `hello.js` with the following content:

`public/hello.js`

```
$(document).ready(function() {

    $.ajax({

        url: "http://localhost:8080/greeting"

    }).then(function(data, status, jqxhr) {

       $('.greeting-id').append(data.id);
```

```
        $('.greeting-content').append(data.content);

        console.log(jqxhr);

    });

});
```

This script uses jQuery to consume the REST service at http://localhost:8080/greeting. It is loaded by `index.html` as shown here:

`public/index.html`

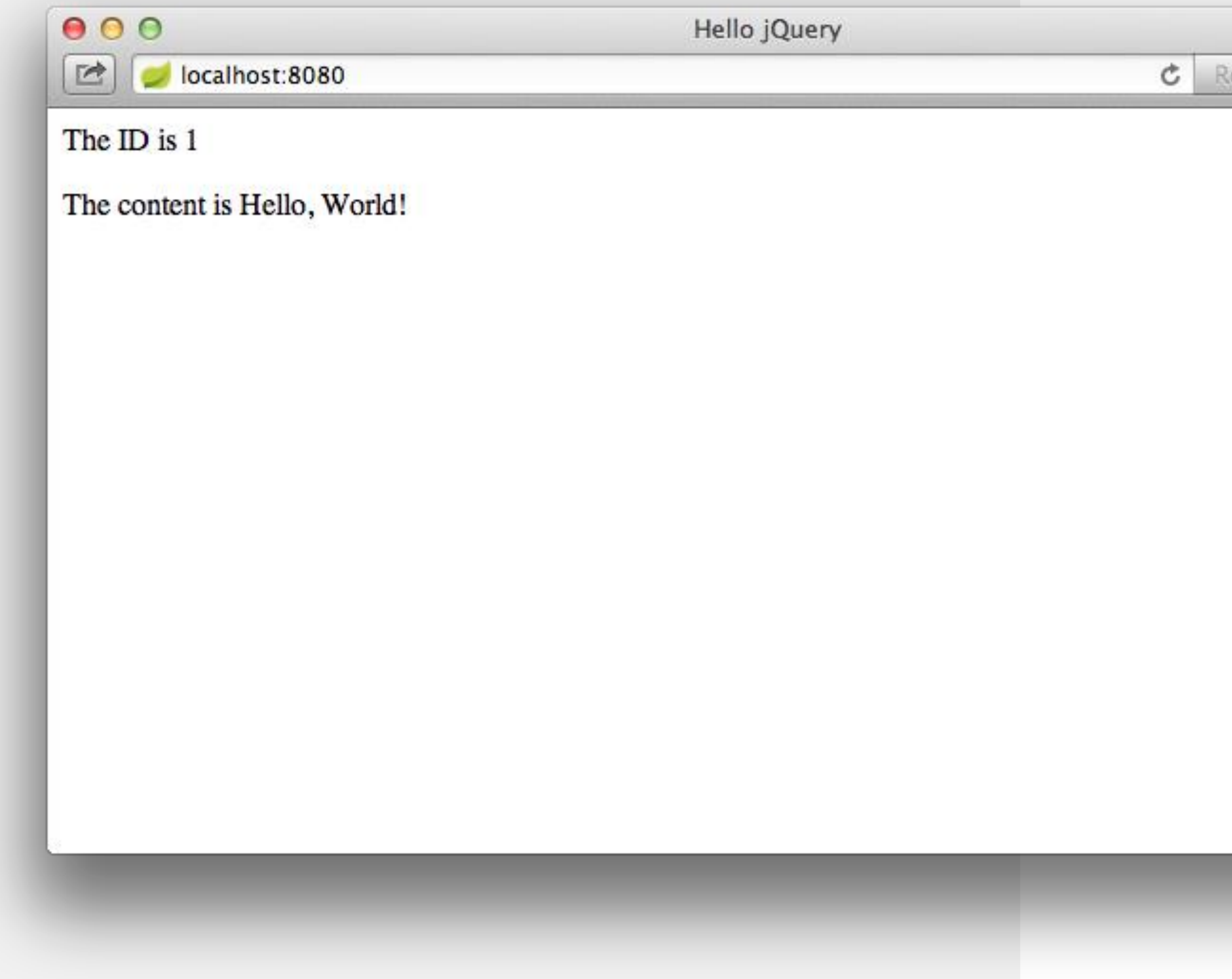
```
<!DOCTYPE html>

<html>

    <head>

        <title>Hello CORS</title>

        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"><
/script>

        <script src="hello.js"></script>

    </head>



    <body>

        <div>

            <p class="greeting-id">The ID is </p>

            <p class="greeting-content">The content is </p>

        </div>

    </body>

</html>
```

This is essentially the REST client created in Consuming a RESTful Web Service with jQuery, modified slightly to consume the service running on localhost, port 8080. See that guide for more details on how this client was developed.

Because the REST service is already running on localhost, port 8080, you'll need to be sure to start the client from another server and/or port. This will not only avoid a collision between the two applications, but will also ensure that the client code is served from a different origin than the service. To start the client running on localhost, port 9000:

```
mvn spring-boot:run -Dserver.port=9000
```

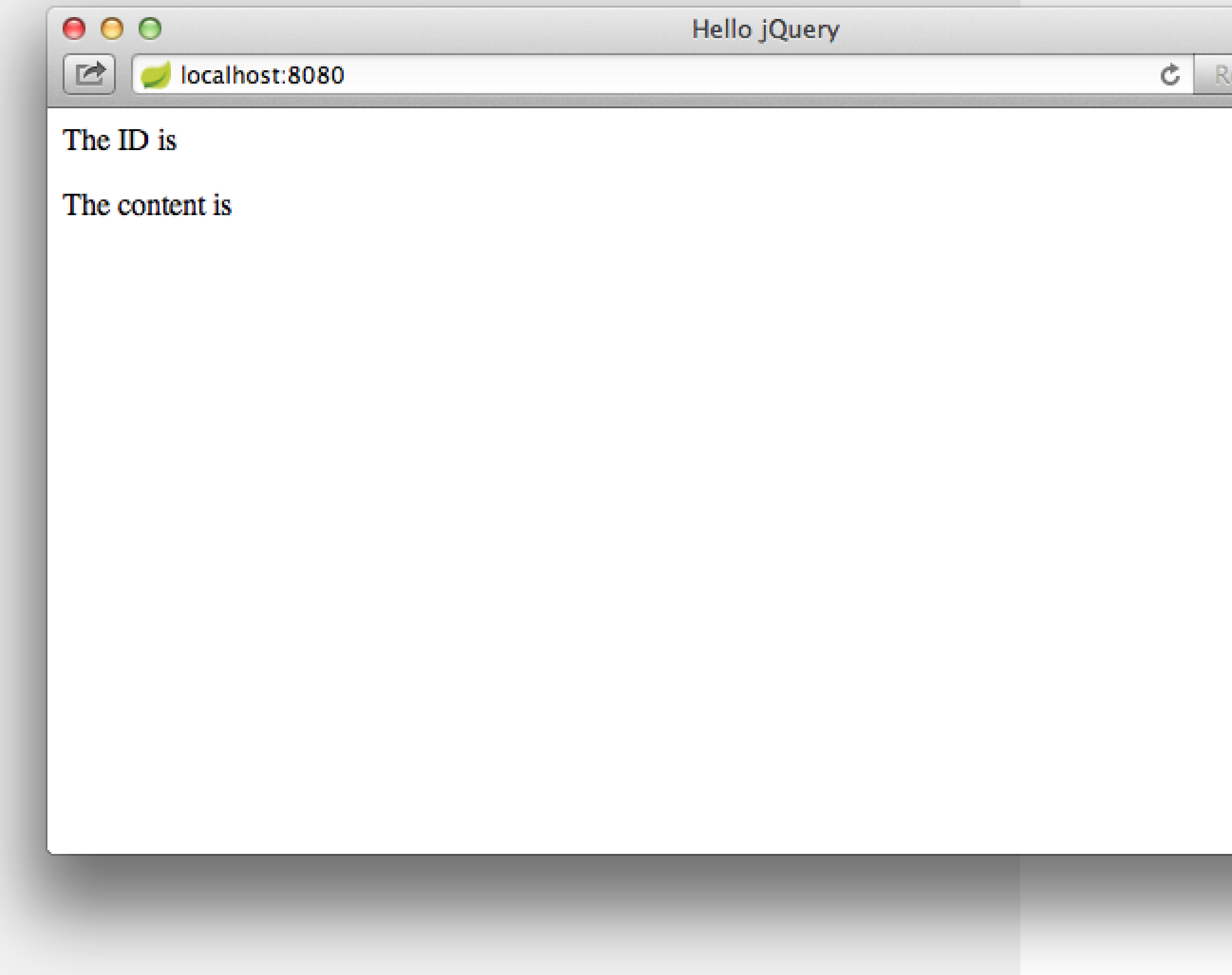
Once the client starts, open http://localhost:9000 in your browser, where you should see:



The ID is 1

The content is Hello, World!

If the service response includes the CORS headers, then the ID and content will be rendered into the page. But if the CORS headers are missing (or insufficiently defined for the client), then the browser will fail the request and the values will not be rendered into the DOM:



The ID is

The content is

Summary

Congratulations! You’ve just developed a RESTful web service including Cross-Origin Resource Sharing with Spring.

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