Blog

ALL BLOG POSTS

Integrating Angular 2 with Spring Boot, JWT, and CORS, Part 1

Posted on Feb 28, 2017 by **Rich Freedman**

A current trend among developers writing web applications using "traditional" server-side languages, including Java, is to move the user interface entirely to the browser, and to limit the server-side code to just providing business logic via an API.

One of the most popular ways to implement the front end at the moment is as a Single Page Application (SPA) using the Angular 2 framework (soon to be renamed simply, "Angular", and released as version 4).

Here at Chariot, we've been using the Spring Framework to write web applications for quite a long time. While we're also heavily involved with other technologies, such as Scala, Clojure, and on occasion, Ruby, we're not about to give up on Spring any time soon. We're particularly fond of the latest incarnation of Spring - "Spring Boot", as it makes it easier than ever to get a Spring application up and running.

In the past, we would typically have used server-side templating (JSP, FreeMarker, etc.), with some Javascript mixed in, to present the user interface. More recently, we might have used Angular 1 to write a Javascript-based user interface, and served that from the static resources directory of our web app.

However, preferences in application architecture tend to change over time, and we now generally like to have our Javascript UI code stand alone. The npm-based tools that we use for Angular development make it convenient to run our Angular front-end app on one port (typically 3000), while our Spring Boot backend runs on another (typically 8080).

Having the two parts of the application (the UI and the API) served from different ports presents a problem, though - by default, the web browser prevents the UI application from accessing the API on a port different from the one on which the UI was served (this is known as the Single-Origin Policy, or SOP).

At first glance, you might think that this is a problem that we're unnecessarily inflicting on ourselves, just for the sake of ease of Angular development. However, we're likely to run into the SOP problem in production as well, for instance, if we want to serve our application's UI from http://example.com and it's API from http://api.example.com, or if we want to provide our API for other people's applications to consume in addition to our own.

A popular solution to this problem is the use of Cross-Origin Resource Sharing (CORS). CORS is a W3C "Recommendation", supported by all modern browsers, that involves a set of procedures and HTTP headers that together allow a browser to access data (notably Ajax requests) from a site other than the one from which the current page was served.

The rest of this post (Part 1) will show in detail how to implement a Spring Boot-based REST API for an Angular 2 app, and how to

implement CORS so as to allow the Angular UI application to work with the REST API served from a different port and/or a different domain.

Future posts will show how to add authentication and authorization via Spring Security and JSON Web Tokens (JWT).

The Application

Front-End - Tour of Heroes

Let's start with the "Tour of Heroes" example from the Angular web site.
You may already be familiar with it, but if not, now is a good time to go check it out at https://angular.io/docs/ts/latest/tutorial and work through the tutorial.

Go ahead, I'll wait here for you until you get back.

(If you're the impatient type, and want to just download the finished code for the Tour of Heroes demo, you can clone my git repo at https://github.com/rfreedman/angular-tour-of-heroes-complete)

Ok, welcome back.

Now that we have a working stand-alone demo front-end (with "faked" API access), let's create a real API for it.

Back-End - Custom Spring Boot App

The Tour of Heroes front-end expects an API for listing, searching and doing CRUD operations on Hero objects. If you look at the Angular app's hero.ts source file, you'll see that a Hero has two properties - a numeric id and a name:

```
export class Hero {
    id: number;
    name: string;
}
```

Let's use Spring Boot to create this API.

The easiest way to start is to use the SPRING INITIALIZR app at http://start.spring.io
Go there, select "Gradle Project", name the group "heroes" and name the artifact "back-end", add dependencies for "JPA", "H2" and "Rest Repositories", and then hit the "Generate Project" button. You'll get a zip file containing a starter project. Unzip it, and open it in your favorite IDE / editor. You should now have a "build.gradle" file in the root folder of the project that looks like this:

```
buildscript {
    ext {
        springBootVersion = '1.4.3.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradl
    }
}

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

jar {
    baseName = 'back-end'
    version = '0.0.1-SNAPSHOT'
}

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}


dependencies {
    compile('org.springframework.boot:spring-boot-starter-dat
    compile('org.springframework.boot:spring-boot-starter-dat
    compile("com.h2database:h2")
    testCompile('org.springframework.boot:spring-boot-starter
}
```

build.gradle

This gives us a build that includes the required libraries for an H2 embedded database, JPA (Java Persistence API), and Spring Rest in addition to Spring Boot's default embedded Tomcat server.

Now, let's add our Hero domain object. Create a Hero.java file in the src/main/java/heroes folder, and add the following code:

```
package heroes;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Hero {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String name;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Hero.java

This is a standard POJO (plain old java object), a.k.a. a "Bean", with JPA annotations to mark this as a JPA Entity and to tell JPA how to generate ids.

> The one thing that is slightly different here from a normal Spring Data Entity is the inclusion of the **getId** and **setId** methods. These would typically be omitted because JPA doesn't need them, and a true REST API would employ HATEOAS, and not expose the id as part of an object's JSON representation. However, like many "REST-ish" applications, the Tour

> of Heroes front-end does not not make use of API-provided hypertext links, and instead uses object ids and well-known endpoint URI patterns to form the URLs used to interact with the API. Since HATEOAS is not the subject of this article, we'll just have the API expose object ids.

This isn't quite enough to start exposing Entity ids, though. By default, Spring will still omit the id when serializing objects to Json. To change this, we need to add a configuration, and in keeping with Spring Boot convention, we'll do so with a configuration class.

Add the following RepositoryConfig.java source file to the src/main/java/com/example folder:

```
1  package heroes;
2
3  import org.springframework.context.annotation.Configuration;
4  import org.springframework.data.rest.core.config.RepositoryRe
5  import org.springframework.data.rest.webmvc.config.Repository
6
7  @Configuration
8  public class RepositoryConfig extends RepositoryRestConfigure
9      @Override
10     public void configureRepositoryRestConfiguration(Reposito
11         config.exposeIdsFor(Hero.class);
12     }
13 }
```

RepositoryConfig.java

This will configure the Repository (which we haven't created yet) to expose the ids of Hero objects when serializing to Json. Note that if you add more entities, you'll have to add them here as well.

Next, we'll add the Repository, which will be responsible for database operations involving our Heros. With Spring Boot, we can simply provide an interface for our Repository, and Spring Data will implement it via a proxy at runtime. Add the following HeroRepository.java file to src/main/java/com/example:

```
1  package heroes;
2
3  import java.util.List;
4
```

```
5  import org.springframework.data.repository.CrudRepository;
6  import org.springframework.data.repository.query.Param;
7  import org.springframework.data.rest.core.annotation.Reposit
8
9  @RepositoryRestResource(collectionResourceRel = "heroes", pat
10 public interface HeroRepository extends CrudRepository<Hero,
11
12     @Query("from Hero h where lower(h.name) like CONCAT('%',
13     public Iterable<Hero> findByName(@Param("name") String na
14
15 }
```

HeroRepository.java

This simple interface, by extending Spring Data's CrudRepository, gives us Create (POST), Update (PUT), Delete (DELETE), and List functionality. The @RepositoryRestResource causes Spring Data to wrap our Repository at runtime with a Controller that handles the HTTP REST requests.

We have added only one method to the interface, beyond what CrudRepository provides by default, to support the Angular app's search function.

This is all of the code that we need to write to get our basic Heroes REST api up and running.

There's one more thing to do. As the application stands right now, when it starts up, there will be no Heroes in the database. We could just add some, but since we're using an in-memory database (H2), we'd lose our Heroes every time we shut down the application.

Fortunately, Spring Data provides an easy facility for loading data into the database at start-up.
If it is creating a new database (which it will do every time we start the app), and it finds a file named "data.sql" in the root of the classpath, it will execute the contents as sql statements.

Add a "data.sql" file to the src/main/resources directory with the following contents (or make up your own!):

```
1 insert into hero(name) values('Black Widow');
2 insert into hero(name) values('Superman');
3 insert into hero(name) values('Rogue');
4 insert into hero(name) values('Batman');
```

data.sql

Then, to cause the data.sql file to be copied to the classpath at build-time, add the following to the build.gradle file, just after the "apply plugin" lines:

```
task copySqlImport(type: Copy) {
    from 'src/main/resources'
    into 'build/classes'
}

build.dependsOn(copySqlImport)
```

Ok, go ahead and start up the Spring Boot applicaiton using the Gradle task **bootRun** (either select it from your IDE if it supports Gradle builds, or run "./gradlew bootRun" from the command line (or ".\gradlew.bat bootRun" if you're on Windows).

You can try out the API at http://localhost:8080/heroes. If you point your web browser at that endpoint, you should get a list of the heroes that you specified in your data.sql file. Fire up a REST client (I recommend Postman), and you should also be able to add, update, and delete Heroes.

> Note that Spring-Data is still including HATEOAS links (in addition to Hero ids), and there's no easy way to turn them off. We're just going to ignore them.

Integrating the front-end and back-end

To integrate the Angular app with our new Spring Boot - based API, we'll need to make some changes to both the Angular front-end application and the Spring Boot API.

First, we need to disable the Angular app's "in-memory-database".

To do this, edit app.module.ts, and comment out or delete the imports for InMemoryWebApiModule and InMemoryDataService at the top of the file, and inside the @NgModule declaration.

Then, in hero.service.ts, change the URLs to point to http://localhost:8080/heroes instead of /app/heroes

These two changes will cause the app to use the api provided by the Spring Boot application instead of the internally mocked api.

Next, we need to deal with the fact that the Spring Boot api responds with data in the HAL format (see https://en.wikipedia.org/wiki/Hypertext_Application_Language). This format is somewhat different from what the original Tour of Heroes app expected, so we need to modify the Angular app's "Hero Service" slightly to look for data in the right places.

(You could modify the Spring Boot app to provide the format expected by the Angular app instead, but that would be quite a bit more work).

- Modify hero.service.ts, changing the getHeros() function from

```
getHeroes(): Promise<Hero[]> {
 return this.http.get(this.heroesUrl)
   .toPromise()
   .then(response => response.json().data as Her
   .catch(this.handleError);
}
```

to

```
getHeroes(): Promise<Hero[]> {
 return this.http.get(this.heroesUrl)
   .toPromise()
   .then(response => response.json()._embedded.h
   .catch(this.handleError);
}
```

> (change `response.json().data` to `response.json().embedded.heroes`)

and changing

```
create(name: string): Promise<Hero> {
return this.http
   .post(this.heroesUrl, JSON.stringify({name: n
   .toPromise()
   .then(res => res.json().data)
   .catch(this.handleError)
}
```

to

```
create(name: string): Promise<Hero> {
return this.http
  .post(this.heroesUrl, JSON.stringify({name: n
  .toPromise()
  .then(res => res.json())
  .catch(this.handleError)
}
```

> (change `.then(res => res.json().data)` to `.then(res => res.json())`)

- Modify heros-search.service.ts, changing the search function from

```
search(term: string): Observable<Hero[]> {
return this.http
  .get(`app/heroes/?name=${term}`)
  .map((r: Response) => r.json().data as Hero[]
}
```

to

```
search(term: string): Observable<Hero[]> {
return this.http
  .get(`http://localhost:8080/heroes/search/fin
  .map((r: Response) => r.json()._embedded.hero
}
```

> (change `app/heroes/?name=${term}` to `http://localhost:8080/heroes/search/fi name=${term}`)

Fire up the apps, and you can see (by using your browser's "developer tools" that the Angular app does indeed make a GET request to http://localhost:8080/heroes, and that the expected Json is returned with an HTTP 200 response code.

But no heroes show up in the dashboard! What happened?

Look at the web browser's console, and you'll see that there's an error that reads something like:

> XMLHttpRequest cannot load http://localhost:8080/heroes. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:3000' is therefore not allowed access.

The problem here is that the browser has a Same-Origin Policy for XMLHttpRequests (or at least most modern browsers do) [see https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy].

This means that without further intervention, XMLHttpRequests can only be made to the same domain that served the initial page. If the protocol, host, or port do not match the original page request, the response won't be received by the Angular code, and you'll see this error.

CORS

There is a way around this, and it's known as the **C**ross **O**rigin **R**esource **S**haring Protocol, or **CORS**.
For simple cases like this GET, when your Angular code makes an XMLHttpRequest that the browser determines is cross-origin, the browser looks for an HTTP header named "Access-Control-Allow-Origin" in the response. If the response header exists, and the value matches the origin domain, then the browser passes the response back to the calling javascript. If the response header does not exist, or it's value does not match the origin domain, then the browser does not pass the response back to the calling code, and you get the error that we just saw.

For more complex cases, like PUTs, DELETEs, or any request involving credentials (which will eventually be all of our requests), the process is slightly more involved. The browser will send an OPTION request to find out what methods are allowed. If the requested method is allowed,

then the browser will make the actual request, again passing or blocking the response depending on the Access-Control-Allow-Origin header in the response.

> Note that for security resasons, the browser is in complete control of the CORS protocol - it cannot be overridden by the calling code.

The problem right now, as the error message indicates, is that the API is not returning an Access-Control-Allow-Origin header. Let's correct this.

In the Spring Boot app, add the following configuration class to add a Filter that adds the required header.

```
1  package heroes;
2
3  import org.springframework.context.annotation.Bean;
4  import org.springframework.context.annotation.Configuration;
5  import org.springframework.web.cors.CorsConfiguration;
6  import org.springframework.web.cors.UrlBasedCorsConfiguratior
7  import org.springframework.web.filter.CorsFilter;
8
9  @Configuration
10 public class RestConfig {
11     @Bean
12     public CorsFilter corsFilter() {
13         UrlBasedCorsConfigurationSource source = new UrlBasec
14         CorsConfiguration config = new CorsConfiguration();
15         config.setAllowCredentials(true);
16         config.addAllowedOrigin("*");
17         config.addAllowedHeader("*");
18         config.addAllowedMethod("OPTIONS");
19         config.addAllowedMethod("GET");
20         config.addAllowedMethod("POST");
21         config.addAllowedMethod("PUT");
22         config.addAllowedMethod("DELETE");
23         source.registerCorsConfiguration("/**", config);
24         return new CorsFilter(source);
25     }
26 }
```

RestConfig.java

> In this configuration, we have `addAllowedOrigin("*")` - this is a wildcard that will simply copy the value of

> the request's Origin header into the value of the Response's Access-Control-Allow-Origin header, effectively allowing all origins. You can add specific origins instead if you wish to limit them.

Go back and reload the Angular app, and you should see that the Heros are now listed, and that the rest of the application works as expected.

In this Part 1 post, you have learned how to implement a simple Spring Boot REST API for use by an Angular 2 front-end, and how to allow them to be served from different ports and/or domains via CORS.

In a future post, I'll show you how to add Authentication and Authorization via Spring Security and JWT.

TAGS: ANGULAR ANGULAR2 JAVA SPRING BOOT SPRING-FRAMEWORK

<PREVIOUS POST NEXT POST>

<< ALL BLOG POSTS

Good tutorial!

A think you can add this new config to CORS. Its is more specifically to repositories.

See ya!

3 ^ v • Reply • Share ›

Sunil Roy • 5 months ago

I am connting Angular Project to Spring Boot Controller and used below annotion on calling method resolve the problems.
@CrossOrigin

^ v • Reply • Share ›

hari mada • 5 months ago

Hi,

I have downloaded the project and tried installing using npm install . it gives me warnings and when I do ng serve I get below error. Please help me.

Binary is fine
npm WARN rollback Rolling back ajv@4.11.8 failed (this is probably harmless): EPERM: operation not permitted, lstat 'C:\hari\vscode\workspace\angular-tour-of-heroes-complete\node_modules\fsevents\node_modules'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN @schematics/schematics@0.0.11 requires a peer of rxjs@^5.5.2 but none is installed. You must install peer dependencies yourself.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.1.3 (node_modules\fsevents):

see more

^ v • Reply • Share ›

Lim Ken • 7 months ago

Hi Please help
On the following step:

Ok, go ahead and start up the Spring Boot applicaiton using the Gradle task bootRun (either select it from your IDE if it supports Gradle builds, or run ./gradlew bootRun from the command line (or .\gradlew.bat bootRun if you're on Windows).

I get the following error

Caused by: org.h2.jdbc.JdbcSQLException: Table "HERO" not found; SQL statement:
insert into HERO("NAME") values("Black Widow") [42102-196]

Reply • Share ›

Carnal ➚ Lim Ken • 6 months ago

Me too. wtf

Reply • Share ›

Waqas Rana ➚ Carnal • 6 months ago

if you use this property in properties file it would generate table for u .
spring.jpa.hibernate.ddl-auto = update

1 Reply • Share ›

Lim Ken ➚ Waqas Rana • 6 months ago

Thank you so much, it worked for me :) I been stuck on this for at least a month

1 Reply • Share ›

Waqas Rana ➚ Lim Ken • 6 months ago

glad to hear it . :)

1 Reply • Share ›

dharlequin ➚ Waqas Rana • 5 months ago

Still does not work for me. The only thing that at least helped me to start the server is to add the following at the top of data.sql:

CREATE TABLE hero (
name TEXT
);

Although the database still seems empty, and going to the specified api address /heroes I see no heroes there.

Reply • Share ›

Anand Kumar • a year ago

Thanks for sharing the valuable info. Your suggested change works perfectly when there is no nested HATEOAS in the JSON object. Can you please also suggest how to handle them. following is example of JSON data, I am struggling with-

{
"_embedded" : {
"dIt" : [{
"vendor" : "XXXXXXXX-ZZ",
"model" : "XXXXXXXX-YY",
"description" : null,
"variant" : null,
"_links" : {
"self" : {
"href" : "http://localhost:8080/dIt/1"
},
"dIt" : {
"href" : "http://localhost:8080/dIt/1"

see more

• Reply • Share ›

reza • a year ago

Hi. Anyone know what's the equivalent of

task copySqlImport(type: Copy) {
from 'src/main/resources'
into 'build/classes'
}
build.dependsOn(copySqlImport
for a maven project?

Thanks

• Reply • Share ›

Rich Freedman ➚ reza • a year ago

Reza,

For maven, you shouldn't have to do anything at all - by default, it already copies files from src/main/resources into build/classes.

2 • Reply • Share ›

Jhonatan Batista • a year ago

Great tutorial, keep going! It helped me a lot

• Reply • Share ›

okon3 • a year ago

Really nice. Waiting for Part2!

• Reply • Share ›

Sammy • a year ago

Thank you! I'm really looking forward to Part 2, especially that now Spring OAuth 2.1.0 has been released with native JWT support!

Reply • Share ›

Sandeep sandy • 5 months ago

I want to use spring session too with angular 2, could you please suggest a way to do it.

Reply • Share ›