Spring and Angular JS: A Secure Single Page Application

ENGINEERING | DAVE SYER | JANUARY 12, 2015 | 50 COMMENTS

> Note: the source code and test for this blog continue to evolve, but the changes to the text are not being maintained here. Please see the tutorial version for the most up to date content.

In this article we show some nice features of Spring Security, Spring Boot and Angular JS working together to provide a pleasant and secure user experience. It should be accessible to beginners with Spring and Angular JS, but there also is plenty of detail that will be of use to experts in either. This is actually the first in a series of articles on Spring Security and Angular JS, with new features exposed in each one successively. We'll improve on the application in the second and subsequent installments, but the main changes after this are architectural rather than functional.

Spring and the Single Page Application

HTML5, rich browser-based features, and the "single page application" are extremely valuable tools for modern developers, but any meaningful interactions will involve a backend server, so as well as static content (HTML, CSS and JavaScript) we are going to need a backend server. The backend server can play any or all of a number of roles: serving static content, sometimes (but not so often these days) rendering dynamic HTML, authenticating users, securing access to protected resources, and (last but not least) interacting with JavaScript in the browser through HTTP and JSON (sometimes referred to as a REST API).

Spring has always been a popular technology for building the backend features (especially in the enterprise), and with the advent of Spring Boot things have never been easier. Let's have a look at how to build a new single page application from nothing using Spring Boot, Angular JS and Twitter Bootstrap. There's no particular reason to choose that specific stack, but it is quite popular, especially with the core Spring constituency in enterprise Java shops, so it's a worthwhile starting point.

Create a New Project

We are going to step through creating this application in some detail, so that anyone who isn't completely au fait with Spring and Angular can follow what is happening. If you prefer to cut to this chase, you can skip to the end where the application is working, and see how it all fits together. There are various options for creating a new project:

- Using curl on the command line
- Using Spring Boot CLI
- Using the Spring Initializr website
- Using Spring Tool Suite

The source code for the complete project we are going to build is in Github here, so you can just clone the project and work directly from there if you want. Then jump to the next section.

Using Curl

The easiest way to create a new project to get started is via the Spring Boot Initializr. E.g. using curl on a UN*X like system:

```
$ mkdir ui && cd ui
$ curl https://start.spring.io/starter.tgz -d style=web \
-d style=security -d name=ui | tar -xzvf -
```

COPY

You can then import that project (it's a normal Maven Java project by default) into your favourite IDE, or just work with the files and "mvn" on the command line. Then jump to the next section.

Using Spring Boot CLI

You can create the same project using the Spring Boot CLI, like this:

```
$ spring init --dependencies web,security ui/ && cd ui
```

COPY

Then jump to the next section.

Using the Initializr Website

If you prefer you can also get the same code directly as a .zip file from the Spring Boot Initializr. Just open it up in your browser and select dependencies "Web" and "Security", then click on "Generate Project". The .zip file contains a standard Maven or Gradle project in the root directory, so you might want to create an empty directory before you unpack it. Then jump to the next section.

Using Spring Tool Suite

In Spring Tool Suite (a set of Eclipse plugins) you can also create and import a project using a wizard at `File->New->Spring Starter Project`. Then jump to the next section.

Add a Home Page

The core of a single page application is a static "index.html", so let's go ahead and create one (in "src/main/resources/static" or "src/main/resources/public"):

COPY

```
<!doctype html>
<html>
<head>
<title>Hello AngularJS</title>
<link href="css/angular-bootstrap.css" rel="stylesheet">
<style type="text/css">
[ng\:cloak], [ng-cloak], .ng-cloak {
  display: none !important;
}
</style>
</head>

<body ng-app="hello">
  <div class="container">
    <h1>Greeting</h1>
    <div ng-controller="home" ng-cloak class="ng-cloak">
      <p>The ID is {{greeting.id}}</p>
      <p>The content is {{greeting.content}}</p>
    </div>
  </div>
  <script src="js/angular-bootstrap.js" type="text/javascript"></script>
  <script src="js/hello.js"></script>
```

```
</body>
</html>
```

It's pretty short and sweet because it is just going to say "Hello World".

Features of the Home Page

Salient features include:

- Some CSS imported in the `<head>`, one placeholder for a file that doesn't yet exist, but is named suggestively ("angular-bootstrap.css") and one inline stylesheet defining the "ng-cloak" class.
- The "ng-cloak" class is applied to the content `<div>` so that dynamic content is hidden until Angular JS has had a chance to process it (this prevents "flickering" during the initial page load).
- The `<body>` is marked as `ng-app="hello"` which means we need to define a JavaScript module that Angular will recognise as an application called "hello".
- All the CSS classes (apart from "ng-cloak") are from Twitter Bootstrap. They will make things look pretty once we get the right stylesheets set up.
- The content in the greeting is marked up using handlebars, e.g. `{{greeting.content}}` and this will be filled in later by Angular (using a "controller" called "home" according to the `ng-controller` directive on the surrounding `<div>`).
- Angular JS (and Twitter Bootstrap) are included at the bottom of the `<body>` so that the browser can process all the HTML before it gets processed.
- We also include a separate "hello.js" which is where we are going to define the application behaviour.

We are going to create the script and stylesheet assets in a minute, but for now we can ignore the fact that they don't exist.

Running the Application

Once the home page file is added, your application will be loadable in a browser (even though it doesn't do much yet). On the command line you can do this

```
$ mvn spring-boot:run
```
COPY

and go to a browser at http://localhost:8080. When you load the home page you should get a browser dialog asking for username and password (the username is "user" and the password is printed in the console logs on startup). There's actually no content yet, so you should get a blank page with a "Greeting" header once you successfully authenticate.

> Tip: if you don't like scraping the console log for the password just add this to the "application.properties" (in "src/main/resources"): `security.user.password=password` (and choose your own password). We did this in the sample code using "application.yml".

In an IDE, just run the `main()` method in the application class (there is only one class, and it is called `UiApplication` if you used the "curl" command above).

To package and run as a standalone JAR, you can do this:

```
$ mvn package
$ java -jar target/*.jar
```
COPY

Front End Assets

Entry-level tutorials on Angular and other front end technologies often just include the library assets directly from the internet (e.g. the Angular JS website itself recommends downloading from Google CDN). Instead of doing that we are going to generate the "angular-bootstrap.js" asset by concatenating several files from such libraries. This is not strictly necessary to get the application working, but it *is* best practice for a production

application to consolidate scripts to avoid chatter between the browser and the server (or content delivery network). Since we aren't modifying or customizing the CSS stylesheets it is also unecessary to generate the "angular-bootstrap.css", and we could just use static assets from Google CDN for that as well. However, in a real application we almost certainly would want to modify the stylesheets and we wouldn't want to edit the CSS sources by hand, so we would use a higher level tool (e.g. Less or Sass), so we are going to use one too.

There are many different ways of doing this but for the purposes of this article we are going to use wro4j, which is a Java-based toolchain for preprocessing and packaging front end assets. It can be used as a JIT (Just in Time) `Filter` in any Servlet application, but it also has good support for build tools like Maven and Eclipse, and that is how we are going to use it. So we are going to build static resource files and bundle them in our application JAR.

> Aside: Wro4j is probably not the tool of choice for hard-core front end developers - they would probably be using a node-based toolchain, with bower and/or grunt. These are definitely excellent tools, and covered in great detail all over the internet, so please feel free to use them if you prefer. If you just put the outputs from those toolchains in "src/main/resources/static" then it will all work. I find wro4j comfortable because I am not a hard-core front end developer and I know how to use Java-based tooling.

To create static resources at build time we add some magic to the Maven `pom.xml` (it's quite verbose, but boilerplate, so it could be extracted into a parent pom in Maven, or a shared task or plugin for Gradle):

COPY

```
<build>
  <resources>
    <resource>
      <directory>${project.basedir}/src/main/resources</directory>
    </resource>
    <resource>
      <directory>${project.build.directory}/generated-resources</directory>
    </resource>
  </resources>
  <plugins>
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <executions>
    <execution>
      <!-- Serves *only* to filter the wro.xml so it can get an absolute
        path for the project -->
      <id>copy-resources</id>
      <phase>validate</phase>
      <goals>
        <goal>copy-resources</goal>
      </goals>
      <configuration>
        <outputDirectory>${basedir}/target/wro</outputDirectory>
        <resources>
          <resource>
            <directory>src/main/wro</directory>
            <filtering>true</filtering>
          </resource>
        </resources>
```

```
      </configuration>
    </execution>
  </executions>
</plugin>
<plugin>
  <groupId>ro.isdc.wro4j</groupId>
  <artifactId>wro4j-maven-plugin</artifactId>
  <version>1.7.6</version>
  <executions>
    <execution>
      <phase>generate-resources</phase>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <wroManagerFactory>ro.isdc.wro.maven.plugin.manager.factory.ConfigurableWroManagerFactory</wroManagerFactory>
    <cssDestinationFolder>${project.build.directory}/generated-resources/static/css</cssDestinationFolder>
    <jsDestinationFolder>${project.build.directory}/generated-resources/static/js</jsDestinationFolder>
    <wroFile>${project.build.directory}/wro/wro.xml</wroFile>
    <extraConfigFile>${basedir}/src/main/wro/wro.properties</extraConfigFile>
    <contextFolder>${basedir}/src/main/wro</contextFolder>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.webjars</groupId>
      <artifactId>jquery</artifactId>
      <version>2.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.webjars</groupId>
      <artifactId>angularjs</artifactId>
      <version>1.3.8</version>
    </dependency>
    <dependency>
      <groupId>org.webjars</groupId>
      <artifactId>bootstrap</artifactId>
      <version>3.2.0</version>
    </dependency>
  </dependencies>
</plugin>
</plugins>
</build>
```

You can copy that verbatim into your POM, or just scan it if you are following along from the source in Github. The main points are:

- We are including some webjars libraries as dependencies (jquery and bootstrap for CSS and styling, and Angular JS for business logic). Some of the static resources in those jar files will be included in our generated "angular-bootstrap.*" files, but the jars themselves don't need to be packaged with the application.
- Twitter Bootstrap has a dependency on jQuery, so we include that as well. An Angular JS application that didn't use Bootstrap wouldn't need that since Angular has its own version of the features it needs from jQuery.
- The generated resources will go in "target/generated-resources", and because that is declared in the `<resources/>` section, they will be packaged in the output JAR from the project, and available on the classpath in the IDE (as long as we are using Maven tooling, e.g. m2e in Eclipse).
- The wro4j-maven-plugin has some Eclipse integration features and you can install it from the Eclipse Marketplace (try it later if this is your first time - it's not needed to complete the application). If you do that then Eclipse will watch the source files and re-generate the outputs if they change. If you run in debug mode then changes are immediately re-loadable in a browser.
- Wro4j is controlled from an XML configuration file that doesn't know about your build classpath, and only understand absolute file paths, so we have to create an absolute file location and insert it in `wro.xml`. For that purpose we use Maven resource filtering and that is why there is an explicit "maven-resources-plugin" declaration.

That's all of the changes we are going to need to the POM. It remains to add the wro4j build files, which we have specified are going to live in "src/main/wro".

Wro4j Source Files

If you look in the source code in Github you will see there are only 3 files (and one of those is empty, ready for later customization):

- `wro.properties` is a configuration file for the preprocessing and rendering engine in wro4j. You can use it to switch on and off various parts of the toolchain. In this case we use it to compile CSS from Less and to minify JavaScript, ultimately combining the sources from all the libraries we need in two files.

```
preProcessors=lessCssImport
postProcessors=less4j,jsMin
```

- `wro.xml` declares a single "group" of resources called "angular-bootstrap", and this ends up being the base name of the static resources that are generated. It includes references to `<css>` and `<js>` elements in the webjars we added, and also to a local source file `main.less`.

```
<groups xmlns="http://www.isdc.ro/wro">
  <group name="angular-bootstrap">
  <css>webjar:bootstrap/3.2.0/less/bootstrap.less</css>
    <css>file:${project.basedir}/src/main/wro/main.less</css>
    <js>webjar:jquery/2.1.1/jquery.min.js</js>
    <js>webjar:bootstrap/3.2.0/bootstrap.js</js>
    <js>webjar:angularjs/1.3.8/angular.min.js</js>
  </group>
</groups>
```

- `main.less` is empty, but could be used to customise the look and feel, changing the default settings in Twitter Bootstrap. E.g. to change the colours from default blue to light pink you could add a single line: `@brand-primary: #de8579;`.

Copy those files to your project and run "mvn package" and you should see the "bootstrap-angular.*" resources show up in your JAR file. If you run the app now, you should see the CSS take effect, but the business logic and navigation is still missing.

Create the Angular Application

Let's create the "hello" application (in "src/main/resources/static/js/hello.js" so that the `<script/>` at the bottom of our "index.html" finds it in the right place).

A minimal Angular JS application looks like this:

```
angular.module('hello', [])
  .controller('home', function($scope) {
    $scope.greeting = {id: 'xxx', content: 'Hello World!'}
})
```

The name of the application is "hello" and it has an empty (and redundant) "config" and an empty "controller" called "home". The "home" controller will be called when we load the "index.html" because we have decorated the content `<div>` with `ng-controller="home"` .

Notice that we injected a magic `$scope` into the controller function (Angular does dependency injection by naming convention, and recognises the names of your function parameters). The `$scope` is then used inside the function to set up content and behaviour for the UI elements that this controller is responsible for.

If you added that file under "src/main/resources/static/js" your app should now be secure and functional, and it will say "Hello World!". The `greeting` is rendered by Angular in the HTML using the handlebar placeholders, `{{greeting.id}}` and `{{greeting.content}}` .

Adding Dynamic Content

So far we have an application with a greeting that is hard coded. That's useful for learning how things fit together, but really we expect content to come from a backend server, so let's create an HTTP endpoint that we can use to grab a greeting. In your application class (in "src/main/java/demo"), add the `@RestController` annotation and define a new `@RequestMapping` :

```
@SpringBootApplication
@RestController
public class UiApplication {

  @RequestMapping("/resource")
  public Map<String,Object> home() {
    Map<String,Object> model = new HashMap<String,Object>();
    model.put("id", UUID.randomUUID().toString());
    model.put("content", "Hello World");
    return model;
  }

  public static void main(String[] args) {
    SpringApplication.run(UiApplication.class, args);
  }

}
```

> Note: Depending on the way you created your new project it might not be called `UiApplication` , and it might have `@EnableAutoConfiguration @ComponentScan @Configuration` instead of `@SpringBootApplication` .

Run that application and try to curl the "/resource" endpoint and you will find that it is secure by default:

```
$ curl localhost:8080/resource
{"timestamp":1420442772928,"status":401,"error":"Unauthorized","message":"Full authentication is required to access this resource","path":"/resource"}
```

Loading a Dynamic Resource from Angular

So let's grab that message in the browser. Modify the "home" controller to load the protected resource using XHR:

```
angular.module('hello', [])
  .controller('home', function($scope, $http) {
  $http.get('/resource/').success(function(data) {
    $scope.greeting = data;
  })
});
```

COPY

We injected an `$http` service, which is provided by Angular as a core feature, and used it to GET our resource. Angular passes us the JSON from the response body back to a callback function on success.

Run the application again (or just reload the home page in the browser), and you will see the dynamic message with its unique ID. So, even though the resource is protected and you can't curl it directly, the browser was able to access the content. We have a secure single page application in less than a hundred lines of code!

> Note: You might need to force your browser to reload the static resources after you change them. In Chrome (and Firefox with a plugin) you can use "developer tools" (F12), and that might be enough. Or you might have to use CTRL+F5.

How Does it Work?

The interactions between the browser and the backend can be seen in your browser if you use some developer tools (usually F12 opens this up, works in Chrome by default, requires a plugin in Firefox). Here's a summary:

Verb	Path	Status	Response
GET	/	401	Browser prompts for authentication
GET	/	200	index.html
GET	/css/angular-bootstrap.css	200	Twitter bootstrap CSS
GET	/js/angular-bootstrap.js	200	Bootstrap and Angular JS
GET	/js/hello.js	200	Application logic
GET	/resource	200	JSON greeting

You might not see the 401 because the browser treats the home page load as a single interaction, and you might see 2 requests for "/resource" because there is a CORS negotiation.

Look more closely at the requests and you will see that all of them have an "Authorization" header, something like this:

```
Authorization: Basic dXNlcjpwYXNzd29yZA==
```

COPY

The browser is sending the username and password with every request (so remember to use HTTPS exclusively in production). There's nothing "Angular" about that, so it works with your JavaScript framework or non-framework of choice.

What's Wrong with That?

On the face of it, it seems like we did a pretty good job, it's concise, easy to implement, all our data are secured by a secret password, and it would still work if we changed the front end or backend technologies. But there are some issues.

- Basic authentication is restricted to username and password authentication.

- The authentication UI is ubiquitous but ugly (browser dialog).
- There is no protection from Cross Site Request Forgery (CSRF).

CSRF isn't really an issue with our application as it stands since it only needs to GET the backend resources (i.e. no state is changed in the server). As soon as you have a POST, PUT or DELETE in your application it simply isn't secure any more by any reasonable modern measure.

In the next article in this series we will extend the application to use form-based authentication, which is a lot more flexible than HTTP Basic. Once we have a form we will need CSRF protection, and both Spring Security and Angular have some nice out-of-the box features to help with this. Spoiler: we are going to need to use the `HttpSession`.

> Thanks: I would like to thank everyone who helped me develop this series, and in particular Rob Winch and Thorsten Spaeth for their careful reviews of the articles and sources codes, and for teaching me a few tricks I didn't know even about the parts I thought I was most familar with.