References

https://spring.io/guides/tutorials/spring-security-and-angular-js/#_the_login_page_angular_js_and_spring_security_part_ii

The Login Page

In this section we continue our discussion of how to use Spring Security with Angular JS in a "single page application". Here we show how to use Angular JS to authenticate a user via a form and fetch a secure resource to render in the UI. This is the second in a series of sections, and you can catch up on the basic building blocks of the application or build it from scratch by reading the first section, or you can just go straight to the source code in Github. In the first section we built a simple application that used HTTP Basic authentication to protect the backend resources. In this one we add a login form, give the user some control over whether to authenticate or not, and fix the issues with the first iteration (principally lack of CSRF protection). Reminder: if you are working through this section with the sample application, be sure to clear your browser cache of cookies and HTTP Basic credentials. In Chrome the best way to do that for a single server is to open a new incognito window.

Add Navigation to the Home Page

The core of a single page application is a static "index.html". We already had a really basic one, but for this application we need to offer some navigation features (login, logout, home), so let's modify it (in "src/main/resources/static"):
index.html

```
<!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" ng-cloak class="ng-cloak">
	<div ng-controller="navigation as nav" class="container">
		<ul class="nav nav-pills" role="tablist">
			<li class="active"><a href="#/">home</a></li>
			<li><a href="#/login">login</a></li>
			<li ng-show="authenticated"><a href=""
ng-click="nav.logout()">logout</a></li>
		</ul>
	</div>
```

```
        <div ng-view class="container"></div>
        <script src="js/angular-bootstrap.js" type="text/javascript"></script>
        <script src="js/hello.js"></script>
</body>
</html>
```

It’s not much different than the original in fact. Salient features:

- There is a <ul> for the navigation bar. All the links come straight back to the home page, but in a way that Angular will recognize once we get it set up with "routes".
- All the content is going to be added as "partials" in the <div> labelled "ng-view".
- The "ng-cloak" has been moved up to the body because we want to hide the whole page until Angular can work out which bits to render. Otherwise the menus and content can "flicker" as they are moved around when the page loads.
- As in the first section, the front end assets "angular-bootstrap.css" and "angular-bootstrap.js" are generated from JAR libraries at build time.

Add Navigation to the Angular Application

Let’s modify the "hello" application (in "src/main/resources/public/js/hello.js") to add some navigation features. We can start by adding some configuration for routes, so that the links in the home page actually do something. E.g.

hello.js

```
angular.module('hello', [ 'ngRoute' ])
  .config(function($routeProvider, $httpProvider) {

    $routeProvider.when('/', {
      templateUrl : 'home.html',
      controller : 'home',
      controllerAs: 'controller'
    }).when('/login', {
      templateUrl : 'login.html',
      controller : 'navigation',
      controllerAs: 'controller'
    }).otherwise('/');

    $httpProvider.defaults.headers.common["X-Requested-With"] = 'XMLHttpRequest';

  })
  .controller('home', function($http) {
    var self = this;
    $http.get('/resource/').success(function(data) {
      self.greeting = data;
    })
  })
  .controller('navigation', function() {});
```

We added a dependency on an Angular module called "ngRoute" and this allowed us to inject a magic $routeProvider into the config function (Angular does dependency injection by naming convention, and recognizes the names of your function parameters). The $routeProvider is then used inside the function to set up links to "/" (the "home" controller) and "/login" (the "login"

controller). The "templateUrls" are relative paths from the root of the routes (i.e. "/") to "partial" views that will be used to render the model created by each controller.
The custom "X-Requested-With" is a conventional header sent by browser clients, and it used to be the default in Angular but they took it out in 1.3.0. Spring Security responds to it by not sending a "WWW-Authenticate" header in a 401 response, and thus the browser will not pop up an authentication dialog (which is desirable in our app since we want to control the authentication).
In order to use the "ngRoute" module, we need to add a line to the "wro.xml" configuration that builds the static assets (in "src/main/wro"):
wro.xml
```
<groups xmlns="http://www.isdc.ro/wro">
  <group name="angular-bootstrap">
    ...
    <js>webjar:angularjs/1.3.8/angular-route.min.js</js>
   </group>
</groups>
```

The Greeting

The greeting content from the old home page can go in "home.html" (right next to the "index.html" in "src/main/resources/static"):
home.html
```
<h1>Greeting</h1>
<div ng-show="authenticated">
    <p>The ID is {{controller.greeting.id}}</p>
    <p>The content is {{controller.greeting.content}}</p>
</div>
<div  ng-show="!authenticated">
    <p>Login to see your greeting</p>
</div>
```
Note that we are binding to the controller as "controller" because that is how it was declared in the route provider configuration.
Since the user now has the choice whether to login or not (before it was all controlled by the browser), we need to distinguish in the UI between content that is secure and that which is not. We have anticipated this by adding references to an (as yet non-existent) authenticated variable.

The Login Form

The login form goes in "login.html":
login.html
```
<div class="alert alert-danger" ng-show="controller.error">
    There was a problem logging in. Please try again.
</div>
<form role="form" ng-submit="controller.login()">
    <div class="form-group">
        <label for="username">Username:</label> <input type="text"
            class="form-control" id="username" name="username"
```

```
ng-model="controller.credentials.username"/>
    </div>
    <div class="form-group">
        <label for="password">Password:</label> <input type="password"
            class="form-control" id="password" name="password"
ng-model="controller.credentials.password"/>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

This is a very standard login form, with 2 inputs for username and password and a button for submitting the form via ng-submit. You don't need an action on the form tag, so it's probably better not to put one in at all. There is also an error message, shown only if the angular model contains an error. The form controls use ng-model to pass data between the HTML and the Angular controller, and in this case we are using a credentials object to hold the username and pasword. According to the routes we defined the login form is linked with the "navigation" controller, which is so far empty, so let's head over to that to fill in some gaps.

The Authentication Process

To support the login form we just added we need to add some more features. On the client side these will be implemented in the "navigation" controller, and on the server it will be Spring Security configuration.

Submitting the Login Form

To submit the form we need to define the login() function that we referenced already in the form via ng-submit, and the credentials object that we referenced via ng-model. Let's flesh out the "navigation" controller in "hello.js" (omitting the routes config and the "home" controller):

hello.js

```
angular.module('hello', [ 'ngRoute' ]) // ... omitted code
.controller('navigation',

  function($rootScope, $http, $location) {

  var self = this

  var authenticate = function(credentials, callback) {

    var headers = credentials ? {authorization : "Basic "
        + btoa(credentials.username + ":" + credentials.password)
    } : {};

    $http.get('user', {headers : headers}).success(function(data) {
      if (data.name) {
        $rootScope.authenticated = true;
      } else {
        $rootScope.authenticated = false;
      }
      callback && callback();
    }).error(function() {
      $rootScope.authenticated = false;
      callback && callback();
    });

  }

  authenticate();
  self.credentials = {};
  self.login = function() {
      authenticate(self.credentials, function() {
        if ($rootScope.authenticated) {
          $location.path("/");
          self.error = false;
        } else {
          $location.path("/login");
          self.error = true;
        }
      });
  };
});
```
All of the code in the "navigation" controller will be executed when the page loads because the <div> containing the menu bar is visible and is decorated with ng-controller="navigation". In addition to initializing the credentials object, it defines 2 functions, the login() that we need in the form, and a local helper function authenticate() which tries to load a "user" resource from the backend. The authenticate() function is called when the controller is loaded to see if the user is actually already authenticated (e.g. if he had refreshed the browser in the middle of a session). We need the authenticate() function to make a remote call because the actual authentication is done by the server, and we don't want to trust the browser to keep track of it.
The authenticate() function sets an application-wide flag called authenticated which we have already used in our "home.html" to control which parts of the page are rendered. We do this using $rootScope because it's convenient and easy to follow, and we need to share the authenticated flag between the "navigation" and the "home" controllers. Angular experts might prefer to share data through a shared user-defined service (but it ends up being the same mechanism).
The authenticate() makes a GET to a relative resource (relative to the deployment root of your application) "/user". When called from the login() function it adds the Base64-encoded credentials in the headers so on the server it does an authentication and accepts a cookie in return. The login() function also sets a local $scope.error flag accordingly when we get the result of the authentication, which is used to control the display of the error message above the login form.

The Currently Authenticated User

To service the authenticate() function we need to add a new endpoint to the backend:
UiApplication.java

```
@SpringBootApplication
@RestController
public class UiApplication {

  @RequestMapping("/user")
  public Principal user(Principal user) {
    return user;
  }

  ...

}
```

This is a useful trick in a Spring Security application. If the "/user" resource is reachable then it will return the currently authenticated user (an Authentication), and otherwise Spring Security will intercept the request and send a 401 response through an AuthenticationEntryPoint.

Handling the Login Request on the Server

Spring Security makes it easy to handle the login request. We just need to add some configuration to our main application class (e.g. as an inner class):

UiApplication.java

```
@SpringBootApplication
@RestController
public class UiApplication {

  ...

  @Configuration
  @Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
  protected static class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
      http
        .httpBasic()
      .and()
        .authorizeRequests()
          .antMatchers("/index.html", "/home.html", "/login.html", "/").permitAll()
          .anyRequest().authenticated();
    }
  }

}
```

This is a standard Spring Boot application with Spring Security customization, just allowing anonymous access to the static (HTML) resources (the CSS and JS resources are already accessible by default). The HTML resources need to be available to anonymous users, not just ignored by Spring Security, for reasons that will become clear.

Logout

The application is almost finished functionally. The last thing we need to do is implement the logout feature that we sketched in the home page. Here’s a reminder what the navigation bar looks like:

index.html

```
<div ng-controller="navigation as nav" class="container">
  <ul class="nav nav-pills" role="tablist">
    <li class="active"><a href="#/">home</a></li>
    <li><a href="#/login">login</a></li>
    <li ng-show="authenticated"><a href="" ng-click="nav.logout()">logout</a></li>
  </ul>
</div>
```

If the user is authenticated then we show a "logout" link and hook it to a logout() function in the "navigation" controller. The implementation of the function is relatively simple:

hello.js

```
angular.module('hello', [ 'ngRoute' ]).
// ...
.controller('navigation', function(...) {

...

self.logout = function() {
  $http.post('logout', {}).finally(function() {
    $rootScope.authenticated = false;
    $location.path("/");
  });
}

...

});
```

It sends an HTTP POST to "/logout" which we now need to implement on the server. This is straightforward because it is added for us already by Spring Security (i.e. we don’t need to do anything for this simple use case). For more control over the behaviour of logout you could use the HttpSecurity callbacks in your WebSecurityAdapter to, for instance execute some business logic after logout.

CSRF Protection

The application is almost ready to use, and in fact if you run it you will find that everything we built so far actually works except the logout link. Try using it annd look at the responses in the browser and you will see why:

```
POST /logout HTTP/1.1
...
Content-Type: application/x-www-form-urlencoded

username=user&password=password
```

```
HTTP/1.1 403 Forbidden
Set-Cookie: JSESSIONID=3941352C51ABB941781E1DF312DA474E; Path=/; HttpOnly
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
...

{"timestamp":1420467113764,"status":403,"error":"Forbidden","message":"Expected CSRF token not found. Has your session expired?","path":"/login"}
```

That's good because it means that Spring Security's built-in CSRF protection has kicked in to prevent us from shooting ourselves in the foot. All it wants is a token sent to it in a header called "X-CSRF". The value of the CSRF token was available server side in the HttpRequest attributes from the initial request that loaded the home page. To get it to the client we could render it using a dynamic HTML page on the server, or expose it via a custom endpoint, or else we could send it as a cookie. The last choice is the best because Angular has built in support for CSRF (which it calls "XSRF") based on cookies.

So on the server we need a custom filter that will send the cookie. Angular wants the cookie name to be "XSRF-TOKEN" and Spring Security provides it as a request attribute, so we just need to transfer the value from a request attribute to a cookie:

CsrfHeaderFilter.java

```java
public class CsrfHeaderFilter extends OncePerRequestFilter {
  @Override
  protected void doFilterInternal(HttpServletRequest request,
      HttpServletResponse response, FilterChain filterChain)
      throws ServletException, IOException {
    CsrfToken csrf = (CsrfToken) request.getAttribute(CsrfToken.class
        .getName());
    if (csrf != null) {
      Cookie cookie = WebUtils.getCookie(request, "XSRF-TOKEN");
      String token = csrf.getToken();
      if (cookie==null || token!=null && !token.equals(cookie.getValue())) {
        cookie = new Cookie("XSRF-TOKEN", token);
        cookie.setPath("/");
        response.addCookie(cookie);
      }
    }
    filterChain.doFilter(request, response);
  }
}
```

To finish the job and make it completely generic we should be careful to set the cookie path to the context path of the application (instead of hard-coded to "/"), but this is good enough for the application we are working on.

We need to install this filter in the application somewhere, and it needs to go after the Spring Security CsrfFilter so that the request attribute is available. Since we have Spring Security protecting these resources there's no better place than in the Spring Security filter chain, e.g. extending the SecurityConfiguration above:

SecurityConfiguration.java

```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
protected static class SecurityConfiguration extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .httpBasic().and()
      .authorizeRequests()
        .antMatchers("/index.html", "/home.html", "/login.html", "/").permitAll().anyRequest()
        .authenticated().and()
      .addFilterAfter(new CsrfHeaderFilter(), CsrfFilter.class);
  }
}
```
The other thing we have to do on the server is tell Spring Security to expect the CSRF token in the format that Angular wants to send it back (a header called "X-XRSF-TOKEN" instead of the default "X-CSRF-TOKEN"). We do this by customizing the CSRF filter:
SecurityConfiguration.java
```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    .httpBasic().and()
    ...
    .csrf().csrfTokenRepository(csrfTokenRepository());
}

private CsrfTokenRepository csrfTokenRepository() {
  HttpSessionCsrfTokenRepository repository = new HttpSessionCsrfTokenRepository();
  repository.setHeaderName("X-XSRF-TOKEN");
  return repository;
}
```
With those changes in place we don't need to do anything on the client side and the login form is now working.

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, may require a plugin in Firefox). Here's a summary:

Verb	Path	Status	Response
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	/user	401	Unauthorized
GET	/home.html	200	Home page
GET	/resource	401	Unauthorized
GET	/login.html	200	Angular login form partial
GET	/user	401	Unauthorized
GET	/user	200	Send credentials and get JSON
GET	/resource	200	JSON greeting

The responses that are marked "ignored" above are HTML responses received by Angular in an XHR call, and since we aren't processing that data the HTML is dropped on the floor. We do look for an authenticated user in the case of the "/user" resource, but since it isn't there in the first call, that response is dropped.
Look more closely at the requests and you will see that they all have cookies. If you start with a clean browser (e.g. incognito in Chrome), the very first request has no cookies going off to the server, but the server sends back "Set-Cookie" for "JSESSIONID" (the regular HttpSession) and "X-XSRF-TOKEN" (the CRSF cookie that we set up above). Subsequent requests all have those cookies, and they are important: the application doesn't work without them, and they are providing some really basic security features (authentication and CSRF protection). The values of the cookies change when the user authenticates (after the POST) and this is another important security feature (preventing session fixation attacks).

> It is not adequate for CSRF protection to rely on a cookie being sent back to the server because the browser will automatically send it even if you are not in a page loaded from your application (a Cross Site Scripting attack, otherwise known as XSS). The header is not automatically sent, so the origin is under control. You might see that in our application the CSRF token is sent to the client as a cookie, so we will see it being sent back automatically by the browser, but it is the header that provides the protection.

Help, How is My Application Going to Scale?

"But wait…" you are saying, "isn't it Really Bad to use session state in a single-page application?" The answer to that question is going to have to be "mostly", because it very definitely is a Good Thing to use the session for authentication and CSRF protection. That state has to be stored somewhere, and if you take it out of the session, you are going to have to put it somewhere else and manage it manually yourself, on both the server and the client. That's just more code and probably more maintenance, and generally re-inventing a perfectly good wheel.

"But, but…" you are going to respond, "how do I scale my application horizontally now?" This is the "real" question you were asking above, but it tends to get shortened to "session state is bad, I must be stateless". Don't panic. The main point to take on board here is that security *is* stateful. You can't have a secure, stateless application. So where are you going to store the state? That's all there is to it. Rob Winch gave a very useful and insightful talk at Spring Exchange 2014 explaining the need for state (and the ubiquity of it - TCP and SSL are stateful, so your system is stateful whether you knew it or not), which is probably worth a look if you want to look into this topic in more depth.
The good news is you have a choice. The easiest choice is to store the session data in-memory, and rely on sticky sessions in your load balancer to route requests from the same session back to the same JVM (they all support that somehow). That's good enough to get you off the ground and will work for a *really* large number of use cases. The other choice is to share the session data between instances of your application. As long as you are strict and only store the security data, it is small and changes infrequently (only when users log in and out, or their session times out), so there shouldn't be any major infrastructure problems. It's also really easy to do with Spring Session. We'll be using Spring Session in the next section in this series, so there's no need to go into any detail about how to set it up here, but it is literally a few lines of code and a Redis server, which is super fast.

> Another easy way to set up shared session state is to deploy your application as a WAR file to Cloud Foundry Pivotal Web Services and bind it to a Redis service.

But, What about My Custom Token Implementation (it's Stateless, Look)?

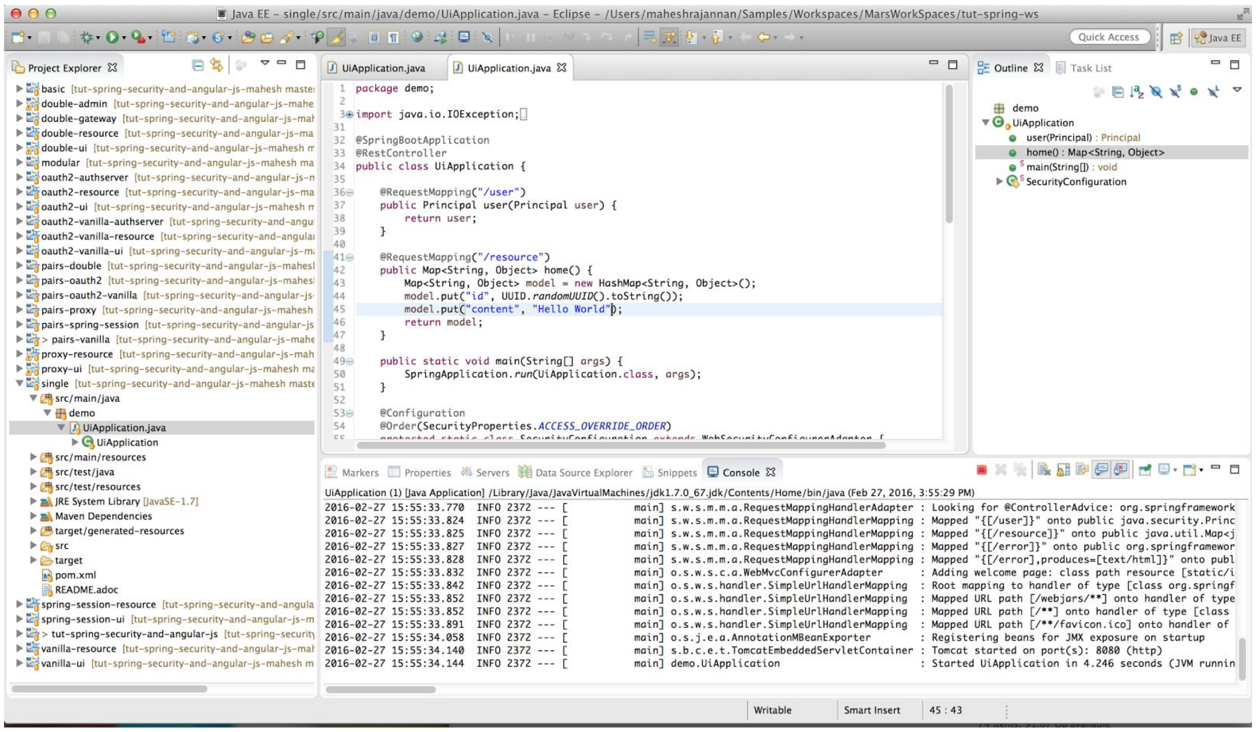
If that was your response to the last section, then read it again because maybe you didn't get it the first time. It's probably not stateless if you stored the token somewhere, but even if you didn't (e.g. you use JWT encoded tokens), how are you going to provide CSRF protection? It's important. Here's a rule of thumb (attributed to Rob Winch): if your application or API is going to be accessed by a browser, you need CSRF protection. It's not that you can't do it without sessions, it's just that you'd have to write all that code yourself, and what would be the point because it's already implemented and works perfectly well on top of HttpSession (which in turn is part of the container you are using and baked into specs since the very beginning)? Even if you decide you don't need CSRF, and have a perfectly "stateless" (non-session based) token implementation, you still had to write extra code in the client to consume and use it, where you could have just delegated to the browser and server's own built-in features: the browser always sends cookies, and the server always has a session (unless you switch it off). That code is not business logic, and it isn't making you any money, it's just an overhead, so even worse, it costs you money.

Conclusion

The application we have now is close to what a user might expect in a "real" application in a live environment, and it probably could be used as a template for building out into a more feature rich application with that architecture (single server with static content and JSON resources).

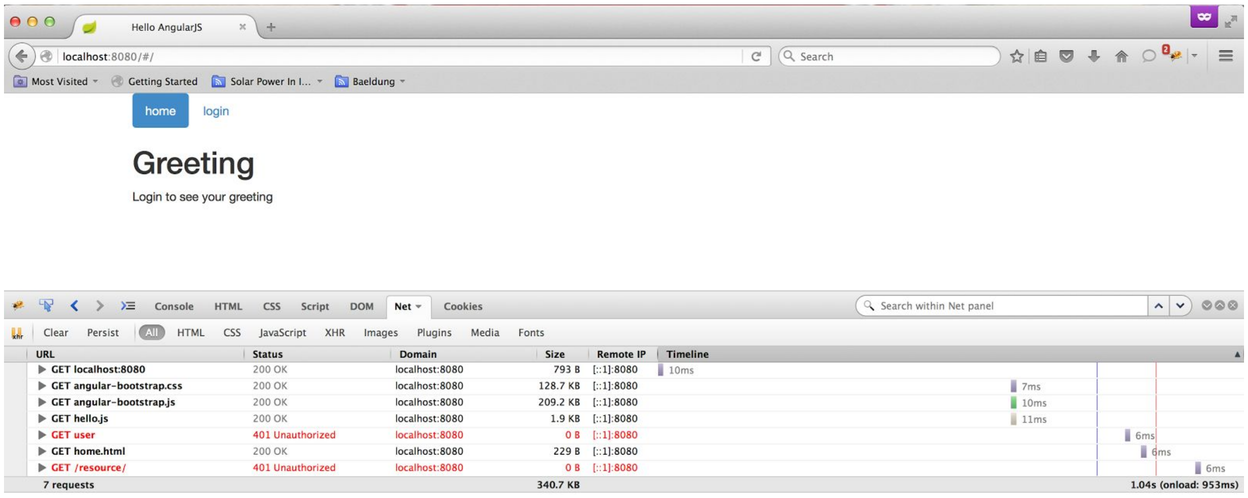
We are using the HttpSession for storing security data, relying on our clients to respect and use the cookies we send them, and we are comfortable with that because it lets us concentrate on our own business domain. In the [next section]() we expand the architecture to a separate authentication and UI server, plus a standalone resource server for the JSON. This is obviously easily generalised to multiple resource servers. We are also going to introduce Spring Session into the stack and show how that can be used to share authentication data.

Running Screens



```java
package demo;

import java.io.IOException;

@SpringBootApplication
@RestController
public class UiApplication {

    @RequestMapping("/user")
    public Principal user(Principal user) {
        return user;
    }

    @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);
    }

    @Configuration
    @Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
```

```
UiApplication (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_67.jdk/Contents/Home/bin/java (Feb 27, 2016, 3:55:29 PM)
2016-02-27 15:55:33.770  INFO 2372 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework
2016-02-27 15:55:33.824  INFO 2372 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/user]}" onto public java.security.Princ
2016-02-27 15:55:33.825  INFO 2372 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/resource]}" onto public java.util.Map<j
2016-02-27 15:55:33.827  INFO 2372 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframewor
2016-02-27 15:55:33.828  INFO 2372 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto publ
2016-02-27 15:55:33.832  INFO 2372 --- [           main] o.s.w.s.c.a.WebMvcConfigurerAdapter      : Adding welcome page: class path resource [static/i
2016-02-27 15:55:33.842  INFO 2372 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Root mapping to handler of type [class org.springf
2016-02-27 15:55:33.852  INFO 2372 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type
2016-02-27 15:55:33.852  INFO 2372 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class
2016-02-27 15:55:33.891  INFO 2372 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of
2016-02-27 15:55:34.058  INFO 2372 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2016-02-27 15:55:34.140  INFO 2372 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2016-02-27 15:55:34.144  INFO 2372 --- [           main] demo.UiApplication                       : Started UiApplication in 4.246 seconds (JVM runnin
```



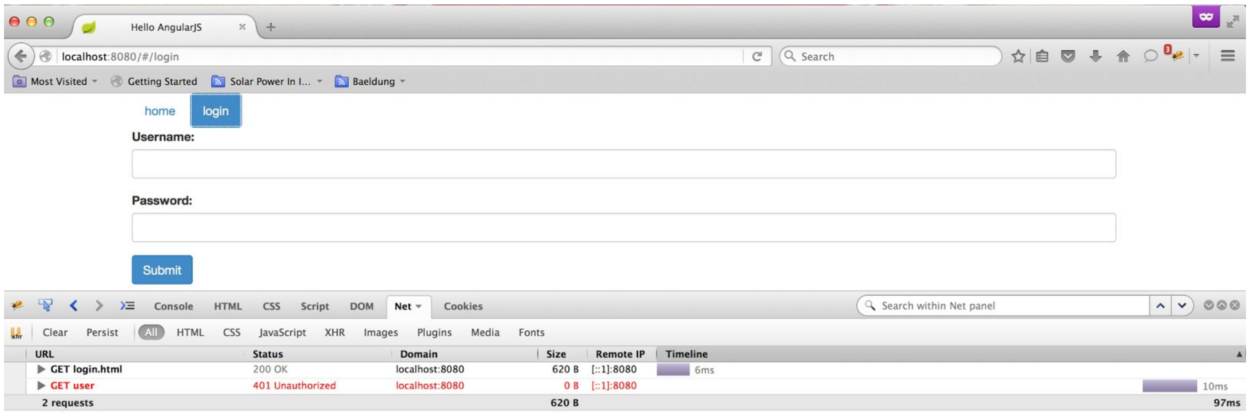
localhost:8080/#/

home login

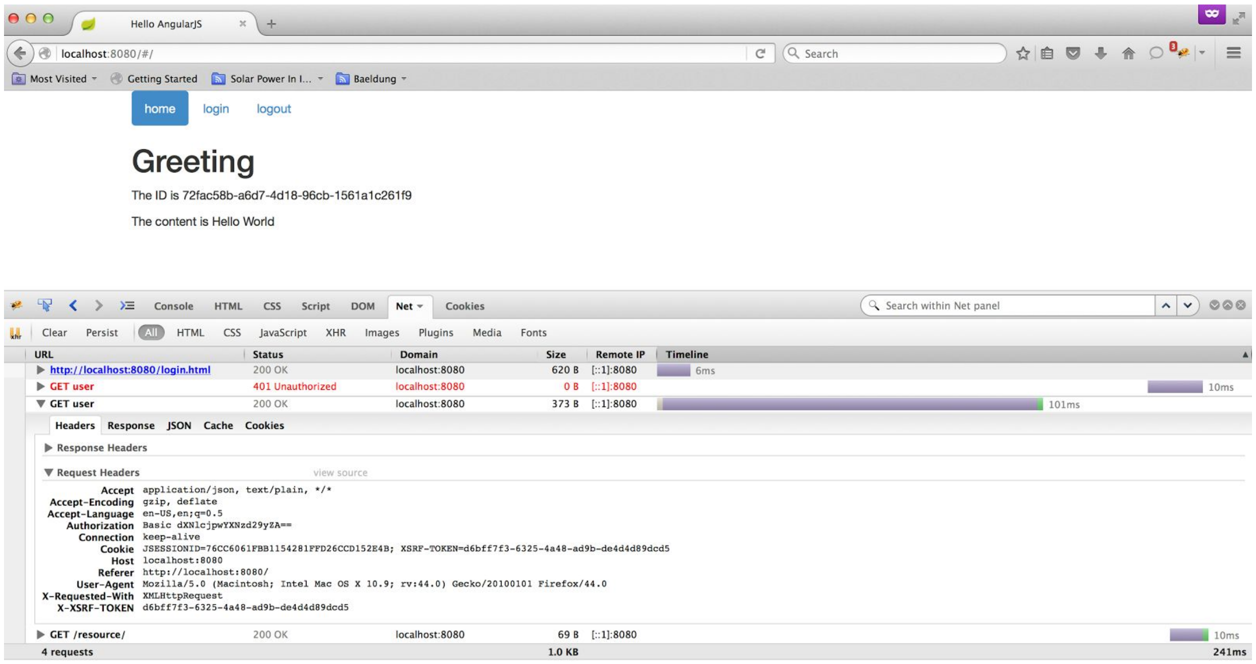
Greeting

Login to see your greeting

URL	Status	Domain	Size	Remote IP	Timeline
GET localhost:8080	200 OK	localhost:8080	793 B	[::1]:8080	10ms
GET angular-bootstrap.css	200 OK	localhost:8080	128.7 KB	[::1]:8080	7ms
GET angular-bootstrap.js	200 OK	localhost:8080	209.2 KB	[::1]:8080	10ms
GET hello.js	200 OK	localhost:8080	1.9 KB	[::1]:8080	11ms
GET user	401 Unauthorized	localhost:8080	0 B	[::1]:8080	6ms
GET home.html	200 OK	localhost:8080	229 B	[::1]:8080	6ms
GET /resource/	401 Unauthorized	localhost:8080	0 B	[::1]:8080	6ms
7 requests			340.7 KB		1.04s (onload: 953ms)



```
URL              Status            Domain          Size   Remote IP    Timeline
GET login.html   200 OK            localhost:8080  620 B  [::1]:8080   6ms
GET user         401 Unauthorized  localhost:8080    0 B  [::1]:8080   10ms
2 requests                                         620 B               97ms
```



Greeting

The ID is 72fac58b-a6d7-4d18-96cb-1561a1c261f9

The content is Hello World

```
URL                                 Status            Domain          Size   Remote IP    Timeline
http://localhost:8080/login.html    200 OK            localhost:8080  620 B  [::1]:8080   6ms
GET user                            401 Unauthorized  localhost:8080    0 B  [::1]:8080   10ms
GET user                            200 OK            localhost:8080  373 B  [::1]:8080   101ms
  Headers  Response  JSON  Cache  Cookies
  Response Headers
  Request Headers                              view source
             Accept  application/json, text/plain, */*
    Accept-Encoding  gzip, deflate
    Accept-Language  en-US,en;q=0.5
      Authorization  Basic dXNlcjpwYXNzd29yZA==
         Connection  keep-alive
             Cookie  JSESSIONID=76CC6061FBB1154281FFD26CCD152E4B; XSRF-TOKEN=d6bff7f3-6325-4a48-ad9b-de4d4d89dcd5
               Host  localhost:8080
            Referer  http://localhost:8080/
         User-Agent  Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:44.0) Gecko/20100101 Firefox/44.0
   X-Requested-With  XMLHttpRequest
       X-XSRF-TOKEN  d6bff7f3-6325-4a48-ad9b-de4d4d89dcd5
GET /resource/                      200 OK            localhost:8080   69 B  [::1]:8080   10ms
4 requests                                            1.0 KB                 241ms
```