# Blog

ALL BLOG POSTS

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

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

In the previous blog post, we created a Spring Boot - based API for the Angular Tour of Heroes demo front-end application, and integrated the two with CORS support.

We're going to continue developing the project from the previous post, so if you haven't followed along with that, you should go do it now before proceeding.

Let's add Authentication and Authorization to our Tour of Heroes app via Spring Security.

# Authentication and Authorization (a.k.a. Security)

Now that the Angular front-end and the Spring Boot back-end are working together, let's add authentication and authorization via Spring Security. Our goal is to add users and passwords to the Spring Boot application, and to require login to access any of the API's endpoints. In the Angular app, whenever we get an HTTP 401 (Unauthorized) response code from the API, we'll present the user with a login UI, and attempt to log them in.

We'll be using Json Web Tokens (JWT) for authenticating the user. When the user successfully logs in, they'll be issued a token (the API will return it in a response header). The Angular app will then include this token in a header with each subsequent request. When the API sees a valid token in a request, it will respond as it does now. If the token is missing or invalid, it will again return an HTTP response with a 401 status code.

First, let's modify the Angular app to add a 'home page'. When we're finished, this will be the one part of the application that an unauthenticated user will be able to access. The rest of the application will require the user to log in.

- Add a new file named home.component.ts to the app/ directory, with the following code:

```
import { Component } from '@angular/core'

@Component({
  moduleId: module.id,
  selector: 'home',
  template: `<h1>Welcome to the Tour of Heroes</h1>`
})
export class HomeComponent {
}
```

**home.component.ts**

This is a very simple component, that just displays the static text, "Welcome to the Tour of Heroes"

- Modify app.module.ts to import and declare the new home component

    - add `import { HomeComponent } from './home.component';` to the imports at the top
    - add `HomeComponent` to the declarations section of the NgModule metadata

- Modify app-routing.module.ts to add the HomeComponent to the app's router, making the new component the default view. Your app-routing.module.ts should now look like this:

```
import { NgModule }              from '@angula
import { RouterModule, Routes } from '@angular
import { HomeComponent}          from './home.c
import { DashboardComponent }    from './dashbo
import { HeroesComponent }       from './heroes
import { HeroDetailComponent }   from './hero-d
const routes: Routes = [
{ path: '', redirectTo: '/home', pathMatch: 'f
{ path: 'home',       component: HomeComponent
{ path: 'dashboard',  component: DashboardComp
{ path: 'detail/:id', component: HeroDetailCom
{ path: 'heroes',     component: HeroesCompone
];
@NgModule({
imports: [ RouterModule.forRoot(routes) ],
exports: [ RouterModule ]
})
export class AppRoutingModule {}
```

**app-routing.module.ts**

- Finally, modify app.component.ts to add a menu item for the 'Home' view, so that we can get back to it after navigating to one of the other views. Your app.component.ts should now look like this:

```
import { Component } from '@angular/core';

@Component({
moduleId: module.id,
selector: 'my-app',
template: `
<h1>{{title}}</h1>
<nav>
```

```
    <a routerLink="/home" routerLinkActive="acti
    <a routerLink="/dashboard" routerLinkActive=
    <a routerLink="/heroes" routerLinkActive="ac
  </nav>
  <router-outlet></router-outlet>
  `,
  styleUrls: ['./app.component.css']
  })
  export class AppComponent {
  title = 'Tour of Heroes';
  }
```

**app.component.ts**

Reload the app, and you should wind up on the new home view, and you should see a new navigation menu item for "Home". Make sure that everything still works, including the new menu item.

## Adding Spring Security to the API

To add Spring Security to the Spring Boot app, edit the build.gradle file and add the spring-boot-starter-security and io.jsonwebtoken.jjwt libraries to the dependencies. Your build.gradle file's dependency section should now look like this:

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-data
    compile("org.springframework.boot:spring-boot-starter-data
    compile("com.h2database:h2")
    compile("org.springframework.boot:spring-boot-starter-secu
    compile("io.jsonwebtoken:jjwt:0.7.0")
    testCompile("org.springframework.boot:spring-boot-starter-
}
```

If you try the application now, you'll see that all requests to the back-end result in an HTTP 401 Unauthorized response. By default, Spring Security secures the entire web application with 'basic' authentication, and a single default user named 'user' with a random password that is printed to the console on startup. If you use Postman, curl, wget, or something similar, and can set a Basic authentication header, with that user and password, you could, in theory still access the API.

But we want real users and passwords, backed by the database, so let's proceed with setting that up.

Most of the spring-security-specific Java code that we're going to add is from a very informative demo project written by Stephan Zerhusen, which is available on GitHub at https://github.com/szerhusenBC/jwt-spring-security-demo.

I'm using Stephan's code here with his permission, and under the code's MIT license (Thanks, Stephan!). Stephan also has a very informative companion blog post at https://www.toptal.com/java/rest-security-with-jwt-spring-security-and-java.

The code's original package was `org.zerhusen` and I've moved it to `heroes.org.zerhusen`. I moved it there to allow Spring to scan for annotations like @Configuration, @Repository, @Entity, etc. (there are other ways to achieve that, but putting everything under one root package is the simplest).

We have a fair amount of (Stephan's) security code to add, so instead of showing it here, I've made it available to download from my GitHub repo. Download the code and add it to your Spring Boot project.

> Note that I made one change to Stephan's code in the JwtAuthenticationTokenFilter class - the original code used the entire value of the HTTP authorization header. However, while not required, it is common practice for the client to form the header by prepending the word "Bearer" to the actual token, to explicitly indicate that it is a Bearer Token. So, I added the code to strip off the prefix before attempting to parse the token.

if you look at the WebSecurityConfig class, you'll see that we're allowing authenticated users (regardless of role) to access all API endpoints. In a real application, you would probably be a bit more selective.

We're also allowing unauthenticated users to send OPTIONS requests to all endpoints. This is because the the browser (not our code) is responsible for sending the OPTIONS request during the pre-flight phase of an API call, and there is no way to have the browser include the authentication token. If we don't do this, the pre-flight OPTIONS request will get a 401 response, and our call will fail.

There's some SpEL (Spring Expression Language) variables in a few places in the JwtAuthenticationFilter, and we need to define those variables now. So add an "application.yml" file to /src/main/resources, with the following content:

```
jwt:
  header: Authorization
  secret: mySecret
  expiration: 604800
  route:
    authentication:
      path: auth
      refresh: refresh
```

And finally, let's add some users to the database on app startup, by adding the following to the bottom of /src/main/resources/data.sql:

```
INSERT INTO USER (ID, USERNAME, PASSWORD, FIRSTNAME, LASTNA
INSERT INTO USER (ID, USERNAME, PASSWORD, FIRSTNAME, LASTNAME
INSERT INTO USER (ID, USERNAME, PASSWORD, FIRSTNAME, LASTNAME

INSERT INTO AUTHORITY (ID, NAME) VALUES (1, 'ROLE_USER');
INSERT INTO AUTHORITY (ID, NAME) VALUES (2, 'ROLE_ADMIN');

INSERT INTO USER_AUTHORITY (USER_ID, AUTHORITY_ID) VALUES (1,
INSERT INTO USER_AUTHORITY (USER_ID, AUTHORITY_ID) VALUES (1,
INSERT INTO USER_AUTHORITY (USER_ID, AUTHORITY_ID) VALUES (2,
INSERT INTO USER_AUTHORITY (USER_ID, AUTHORITY_ID) VALUES (3,
```

We haven't modified the Angular app to log in and get and present tokens yet, but we can test out our back-end changes by using Postman (or curl, or wget, or whatever your favorite tool is).

Fire up the back-end app, and then try to log in by POSTing to http://localhost:8080/auth, with a body containing

```
{
  "username": "admin",
  "password": "admin"
}
```

and a 'content-type' header of 'application/json'

For curl, that would be

```
curl --request POST \
> --url http://localhost:8080/auth \
> --header 'content-type: application/json' \
> --data '{"username": "admin", "password": "admin"}'
```

You should get a response with a token, that looks something like this:

```
{
  "token" : "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbiIsImF1ZGl
}
```

This means that you have successfully authenticated with the back-end, and you now have your Json Web Token (JWT) to prove your identity.

Let's use this token to make a call to the API. Again, using Postman, curl, or your favorite tool, send a GET request to http://localhost:8080/heroes with a header with the name `Authorization` and a value of `Bearer ey....` but with the second part replaced by the value of the token that you got in the previous step.

> If you are using Postman, you can't use the "Authorization" tab to set the authorization header, because Postman (as of version 4.9.2) doesn't support Bearer Tokens there. You'll have to use the Headers tab and create your own Authorization header.

For curl, the request is

```
curl -X GET -H "Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbiIs
```

but again, with the token value replaced with your token.

You should get a JSON response containing the list of Heroes. If you don't, go back and check your code, and make sure that it works before you proceed any farther.

# Adding Log-in and Authentication by JWT to the Angular front-end

On the front-end, we're going to do our best to keep the user from trying to access back-end services when they are not logged in, but will also fall back to presenting a login UI if we do happen to get a 401 Unauthorized response from the API.

First, let's create the login component.

Here is the template for the component. Add it to the Angular app as app/login.component.html:

```
<div class="col-md-6 col-md-offset-3">
    <h2>Login</h2>

    <div class="alert alert-info">
        Username: admin<br/>
        Password: admin
    </div>

    <form name="form" (ngSubmit)="f.form.valid && login()" #f
        <div class="form-group" [ngClass]="{ 'has-error': f.s
            <label for="username">Username</label>
            <input type="text" class="form-control" name="use
            <span *ngIf="f.submitted && !username.valid" clas
        </div>
        <div class="form-group" [ngClass]="{ 'has-error': f.s
            <label for="password">Password </label>
            <input type="password" class="form-control" name=
            <span *ngIf="f.submitted && !password.valid" clas
        </div>
        <div class="form-group">
            <button [disabled]="loading" class="btn btn-prima
            <img *ngIf="loading" src="
        </div>
        <div *ngIf="error" class="alert alert-error">{{error
    </form>
</div>
```

Here is the css for the login component. Add it as app/login.component.css:

```
.alert {
  width:200px;
  margin-top:20px;
  margin-bottom:20px;
}

.alert.alert-info {
  color:#607D8B;
```

```
}

.alert.alert-error {
  color:red;
}

.help-block {
  width:200px;
  color:white;
  background-color:gray;
}

.form-control {
  width: 200px;
  margin-bottom:10px;
}

.btn {
  margin-top:20px;
}
```

Here is the code for the login component. Add it as app/login.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

import { AuthenticationService } from './authentication.servi

@Component({
    moduleId: module.id,
    templateUrl: 'login.component.html',
    styleUrls: ['login.component.css']
})

export class LoginComponent implements OnInit {
    model: any = {};
    loading = false;
    error = '';

    constructor(
        private router: Router,
        private authenticationService: AuthenticationService)

    ngOnInit() {
        // reset login status
        this.authenticationService.logout();
    }

    login() {
        this.loading = true;
        this.authenticationService.login(this.model.username,
            .subscribe(result => {
                if (result === true) {
                    // login successful
                    this.router.navigate(['home']);
```

```
                } else {
                    // login failed
                    this.error = 'Username or password is inc
                    this.loading = false;
                }
            }, error => {
              this.loading = false;
              this.error = error;
            });
    }
}
```

The login component presents inputs for username and password, and when the 'Login' button is clicked, calls the `login` function of the AuthenticationService (which we haven't written yet). If the login succeeds, the login component navigates to the home page. If the login fails, it displays an error message.

Now, we need to add the AuthenticationService, in app/authentication.service.ts:

```
import { Injectable } from '@angular/core';
import { Http, Headers, Response } from '@angular/http';
import { Observable } from 'rxjs/Rx';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';
import 'rxjs/add/observable/throw';

@Injectable()
export class AuthenticationService {
    private authUrl = 'http://localhost:8080/auth';
    private headers = new Headers({'Content-Type': 'applicati

    constructor(private http: Http) {
    }

    login(username: string, password: string): Observable<boo
        return this.http.post(this.authUrl, JSON.stringify({u
            .map((response: Response) => {
                // login successful if there's a jwt token in
                let token = response.json() && response.json(
                if (token) {
                    // store username and jwt token in local
                    localStorage.setItem('currentUser', JSON.

                    // return true to indicate successful log
                    return true;
                } else {
                    // return false to indicate failed login
                    return false;
                }
            }).catch((error:any) => Observable.throw(error.js
    }
```

```
33
34      getToken(): String {
35        var currentUser = JSON.parse(localStorage.getItem('curr
36        var token = currentUser && currentUser.token;
37        return token ? token : "";
38      }
39
40      logout(): void {
41          // clear token remove user from local storage to log
42          localStorage.removeItem('currentUser');
43      }
44  }
```

Now, in app.module.ts, import the LoginComponent and the AuthenticationService, add the LoginComponent to the @NgModule's imports section, and add the AuthenticationService to the @NgModule's providers section.

Your app.module.ts should look like this:

```
1   import './rxjs-extensions';
2
3   import { NgModule }      from '@angular/core';
4   import { BrowserModule } from '@angular/platform-browser';
5   import { FormsModule } from '@angular/forms';
6   import { HttpModule } from '@angular/http';
7
8   import {AppRoutingModule} from "./app-routing.module";
9
10  import { AppComponent }  from './app.component';
11  import { HomeComponent } from './home.component';
12  import { LoginComponent } from './login.component';
13  import { DashboardComponent } from './dashboard.component';
14  import { HeroesComponent } from './heroes.component';
15  import { HeroDetailComponent } from './hero-detail.component
16  import { HeroService } from './hero.service';
17  import { HeroSearchComponent } from './hero-search.component
18  import { AuthenticationService } from './authentication.servi
19
20
21  @NgModule({
22    imports:      [
23      BrowserModule,
24        FormsModule,
25      HttpModule,
26      AppRoutingModule
27    ],
28    declarations: [
29      AppComponent,
30      HomeComponent,
31      LoginComponent,
32      DashboardComponent,
33        HeroDetailComponent,
34      HeroesComponent,
```

```
    HeroSearchComponent
  ],
  providers: [ HeroService, AuthenticationService ],
  bootstrap: [ AppComponent ]
})
export class AppModule {
}
```

We need to add the path to the login component to the router - modify app-routing.module.ts, and add a route for /login, pointing to the LoginComponent.

We'll also add a link to the menu to get to the new login component - modify app.component.ts and add a RouterLink to /login

You should now see the new login menu item when you start up the app, and clicking the login menu item should take you to the login page. If you login with admin / admin, and watch the network tab in the browser's developer's tools, you should see that the authentication request is posted, and that a token is returned. The app should then display the home component.

After you have successfully logged in and returned to the home component, you can verify that the JWT token is stored in the browser's localStorage. Go to the browser's console, and type in `localStorage.getItem("currentUser")`. You should get back a JSON string containing the user's name and the JWT token.

## Using the JWT Token

Now that we have a valid token, let's make use of it. We need to give the hero service access to the token, so that it can include it in an authorization header when it makes api requests. We could access it directly from localStorage, but it's a much better idea to encapsulate access to the token in the Authentication service, and then inject the Authentication service into the hero service.

Modify hero.service.ts:

- import the AuthenticationService
- add the AuthenticationService to the constructor parameters
- modify the "headers" map to include the Authorization header with the Bearer token

- modify the getHeroes() function to use the headers

Your hero.service.ts should now look like this:

```
import { Injectable } from '@angular/core';
import { Headers, Http } from "@angular/http";
import { AuthenticationService } from './authentication.servi
import 'rxjs/add/operator/toPromise';

import { Hero } from './hero'

@Injectable()
export class HeroService {

  private heroesUrl = 'http://localhost:8080/heroes';

  private headers = new Headers({
     'Content-Type': 'application/json',
     'Authorization': 'Bearer ' + this.authenticationService.
     });

  constructor(
    private http: Http,
    private authenticationService: AuthenticationService) {
  }

  getHeroes(): Promise<Hero[]> {
     return this.http
      .get(this.heroesUrl, {headers: this.headers})
      .toPromise()
      .then(response => response.json()._embedded.heroes as H
      .catch(this.handleError);

  }

  private handleError(error: any): Promise<any> {
    console.error('An error occurred: ', error); // for demo
    return Promise.reject(error.message || error);
  }


  getHero(id: number): Promise<Hero> {
    return this.getHeroes()
      .then(heroes => heroes.find(hero => hero.id === id))
  }

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

  update(hero: Hero): Promise<Hero> {
    const url = `${this.heroesUrl}/${hero.id}`;
    return this.http
```

```
      .put(url, JSON.stringify(hero), {headers: this.headers})
      .toPromise()
      .then(() => hero)
      .catch(this.handleError);
  }

  delete(id: number): Promise<void> {
    console.log(`hero.service - deleting ${id}`);
    const url = `${this.heroesUrl}/${id}`;
    return this.http
      .delete(url, {headers: this.headers})
      .toPromise()
      .then(() => null)
      .catch(this.handleError);
  }
}
```

Reload the app in your browser, and you should now see everything working - almost. If you log in, everything works (yay!).

But we have two problems that we need to fix:

- If you hit the Dashboard or Heroes links without logging in first, you just get an error in the console, and no Heroes. It would be better to redirect to the login component.
- It would actually be preferrable to not let the user access the Dashboard or Heroes links at all if they are not logged in.

Let's fix the redirect problem first. Edit dashboard.component.ts, and change it so that the application's Router is injected, and add error handling to navigate to the login page. The code should now look like this:

```
import { Component, OnInit } from '@angular/core'
import { Router } from '@angular/router';

import { Hero } from './hero'
import { HeroService } from './hero.service'

@Component({
  moduleId: module.id,
  selector: 'my-dashboard',
  templateUrl: 'dashboard.component.html',
  styleUrls: ['dashboard.component.css']
})
export class DashboardComponent implements OnInit {
```

```
  heroes: Hero[] = [];

  constructor(private router: Router, private heroService: He
  }

  ngOnInit(): void {
    this.heroService.getHeroes()
      .then(
        heroes => this.heroes = heroes.slice(0, 4),
        error => {
          this.router.navigate(['login']);
          console.error('An error occurred in dashboard compo
        }
      );
  }
}
```

Now, make the same changes to heroes.component.ts:

```
import { Component } from '@angular/core';
import { OnInit } from '@angular/core';
import { Router } from '@angular/router';

import { Hero } from './hero';
import { HeroService } from './hero.service'

@Component({
  moduleId: module.id,
  selector: 'my-heroes',
  templateUrl: 'heroes.component.html',
  styleUrls: ['heroes.component.css'],
  providers: [HeroService]

})
export class HeroesComponent implements OnInit {
    heroes: Hero[];
    selectedHero: Hero;

    constructor(private router: Router, private heroService:

    ngOnInit(): void {
      this.getHeroes();
  }

    getHeroes(): void {
      this.heroService.getHeroes()
      .then(
        heroes => this.heroes = heroes,
        error => {
        this.router.navigate(['login']);
        console.error('An error occurred in heroes component,
        }
      )
  }


```

```
38      onSelect(hero: Hero): void {
39          this.selectedHero = hero;
40      }
41
42      gotoDetail(): void {
43        this.router.navigate(['/detail', this.selectedHero.id])
44    }
45
46    add(name: string): void {
47      name = name.trim();
48      if(!name) { return; }
49      this.heroService.create(name)
50        .then(hero => {
51          this.heroes.push(hero);
52          this.selectedHero = null;
53        });
54    }
55
56    delete(hero: Hero): void {
57        this.heroService
58        .delete(hero.id)
59        .then(() => {
60            this.heroes = this.heroes.filter(h => h !== hero);
61            if(this.selectedHero === hero) {
62              this.selectedHero = null;
63          }
64        });
65    }
66  }
```

Reload the application, and hit the Dashboard and Heroes links. Each time, you should see error messages in the browser's console indicating an error, and the app should navigate to the login page.

That's a nicer UI, and a good fall-back in case, say, the token expires, or something else goes wrong during the API call, but we should also prevent the user from attempting to access protected content when they're not logged in.

The answer for this is Angular 2 Auth Guards. An Auth Guard is a relatively simple class that makes one or more yes/no decisions about routing or module loading. We're interested in just one of those - the CanActivate function.

An AuthGuard can be applied to one or more of the routes in the application's routing module.

Our Authguard will simply check to see if the user is logged in (has an auth token in local storage). If the token is present, the route will proceed as usual. If the token is not present, the route will be blocked

(an error is thrown), and the AuthGuard will navigate the UI to the /login route.

Add the following code to app/can-activate.authguard.ts:

```
import { Injectable } from '@angular/core';
import { Router, CanActivate, ActivatedRouteSnapshot, RouterS
import { AuthenticationService } from './authentication.servi

@Injectable()
export class CanActivateAuthGuard implements CanActivate {

  constructor(private router: Router, private authService: Au

  canActivate(route: ActivatedRouteSnapshot, state: RouterSta
        if (this.authService.isLoggedIn()) {
            // logged in so return true
            return true;
        }

        // not logged in so redirect to login page with the r
        this.router.navigate(['/login']);
        return false;
    }
}
```

Now, modify authentication.service.ts to add the isLoggedIn method:

```
isLoggedIn(): boolean {
  var token: String = this.getToken();
  return token && token.length > 0;
}
```

Modify app.module.ts to register the Auth Guard as a provider:

```
import './rxjs-extensions';

import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import {AppRoutingModule} from "./app-routing.module";

import { AppComponent }  from './app.component';
import { HomeComponent } from './home.component';
import { LoginComponent } from './login.component';
import { DashboardComponent } from './dashboard.component';
import { HeroesComponent } from './heroes.component';
import { HeroDetailComponent } from './hero-detail.component
import { HeroService } from './hero.service';
import { HeroSearchComponent } from './hero-search.component
import { AuthenticationService } from './authentication.servi
import { CanActivateAuthGuard } from './can-activate.authguar
```

```
20
21  @NgModule({
22    imports:      [
23      BrowserModule,
24        FormsModule,
25      HttpModule,
26      AppRoutingModule
27    ],
28    declarations: [
29      AppComponent,
30      HomeComponent,
31      LoginComponent,
32      DashboardComponent,
33        HeroDetailComponent,
34      HeroesComponent,
35      HeroSearchComponent
36    ],
37    providers: [ HeroService, AuthenticationService, CanActivat
38    bootstrap: [ AppComponent ]
39  })
40  export class AppModule {
41  }
```

And finally, protect the routes that use the API by modifying app-routing.module.ts, importing CanActivateAuthGuard and applying to each of the routes that uses the API.

```
1   import { NgModule }             from '@angular/core';
2   import { RouterModule, Routes } from '@angular/router';
3   import { HomeComponent }        from './home.component';
4   import { LoginComponent }       from './login.component';
5   import { DashboardComponent }   from './dashboard.component';
6   import { HeroesComponent }      from './heroes.component';
7   import { HeroDetailComponent }  from './hero-detail.component
8   import { CanActivateAuthGuard } from './can-activate.authguar
9
10  const routes: Routes = [
11    { path: '', redirectTo: '/home', pathMatch: 'full' },
12    { path: 'home',       component: HomeComponent },
13    { path: 'login',      component: LoginComponent},
14    { path: 'dashboard',  component: DashboardComponent, canAct
15    { path: 'detail/:id', component: HeroDetailComponent, canAc
16    { path: 'heroes',     component: HeroesComponent, canActiva
17  ];
18  @NgModule({
19    imports: [ RouterModule.forRoot(routes) ],
20    exports: [ RouterModule ]
21  })
22  export class AppRoutingModule {}
```

By monitoring the browser's admin network window, you should see that when not logged in, you now get navigated to the the login view without making an API call first.

There's lots more that we could do to further polish this app, but this should give you a good playground for investigating Angular 2, Spring Security, and CORS, and provide a good basis for you to build a real, production app based on those technologies.

TAGS: ANGULAR | ANGULAR 2 | JAVA | SPRING BOOT


<PREVIOUS POST

NEXT POST>

<< ALL BLOG POSTS


Share this entry

**Amitava Biswas** • 10 months ago

Did not find on how to get corresponding user roles in front end.
Pls help to give some direction . I am able to get the token in my angular front end . So the authentication is fine. How about the roles to read ? If the token also has the role information then which api or helper can help to read the roles from token in the angular app.

1 ^ | v • Reply • Share ›

**Juan Ignacio Liska** • a year ago

In the part
"After you have successfully logged in and returned to the home component, you can verify that the JWT token is stored in the browser's localStorage. Go to the browser's console, and type in localStorage.getItem("currentUser"). You should get back a JSON string containing the user's name and the JWT token."

I have the problem again of CORS

In other method TS, this trouble is not active, but in the login yes
The service of spring authorization is Ok with postman and angular.....but the CORS crash the app

entro al constructor del login
usuario-service.service.ts:23 entro a loguear con: http://localhost:8080/login {"username":"jliska","password":"ss123"} Object {headers: Headers}
:4200/login:1 XMLHttpRequest cannot load http://localhost:8080/login. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:4200' is therefore not allowed access.

**see more**

1 ^ | v • Reply • Share ›

**Rich Freedman** ➚ Juan Ignacio Liska • a year ago

Juan,
Sorry, I can't tell what's going from the code that you posted. I verified that your configuration is exactly the same as mine. Since it works ok for other requests, you'll have to debug to find what's different for the login request. Check to make sure that the back-end app is sending the Access-Control-Allow-Origin header. If not, you'll need to find out why. I expect that it is not sending it, as the inspection of the header on the client side is done by the browser, not by the client code.

^ | v • Reply • Share ›

**Juan Ignacio Liska** ➚ Rich Freedman • a year ago

Thanks Rich!
Now test other method. This Method (no login) in the part 1 is fine.....and the error of cross in this moment

Run the back-end app with postman:
1-
URL: http://localhost:8080/login
method: POST
header: Content-Type:application/json
body: {"username": "jliska", "password": "ss123"}

2- the result of this POST whit postman is:
body :
header:
Authorization
→eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJqbGlza2EiLCJleHAiOjE0O
Cache-Control →no-cache, no-store, max-age=0, must-revalidate
Content-Length →0

see more

• Reply • Share ›

**Rich Freedman** ➚ Juan Ignacio Liska • a year ago

I believe that you would need to send an Origin header from Postman to receive the Access-Control-Allow-Origin header from the API

• Reply • Share ›

**Peter Boomsma** • 7 months ago

I'm using this code for my spring application. And I have a many to many relation, for this I need to have the current user object available in a different controller. Is this possible without passing a user id through from the front-end?

• Reply • Share ›

**Bhargav Prasad** • 8 months ago

Lovely post, I was able to follow all of it & successfully decode how security is implemented from Angular app to Spring. Thanks again !!!

• Reply • Share ›

**vince carter** • 9 months ago

Do you have Github repo for this exercise?

• Reply • Share ›

**Amitava Biswas** • 10 months ago

Hi Rich,
how user authority/roles can be retrieved in angular front end ? Is it to be read from token ?

• Reply • Share ›

**tekkyru** • a year ago

Hi Rich
I move through this sample step by step
Sadly there's no final version on github e.g. app-routing.module.ts does not have /login route defined as advised in text
I'll keep pushing by guesses. but final version would probably increase overall use of this great post. Anyway I already learned a lot, many thanks.

Reply • Share ›

**Jyoti** • a year ago

You can see the token which is being fetched from API in the browser ,In every client request, this token can be added in header by tampering the input request using burp tool and send to the server. Then any attacker can get the response. So how this application is secure. Can you pls explain

Reply • Share ›

**Rich Freedman** ➚ Jyoti • a year ago

As with any authentication scheme, you would have to make the production API available only via ssl/tls (https) to avoid a "man in the middle" attack.

Old-style cookie-based "jsessionid" auth has exactly the same issue. I'm not aware of any scheme that is secure without a secure communication channel.

1 • Reply • Share ›

**Jyoti** ➚ Rich Freedman • a year ago

As per my understanding,SSL/TLS protects the data in a network which is sent from client to server.. However data can't be protected at the sender/receiver end and it needs additional security mechanism. Before data passed to the network it can be tampered by burp tool.

Reply • Share ›

**Rich Freedman** ➚ Jyoti • a year ago

SSL will prevent an attacker from obtaining the JWT token. If the token is not available to the attacker, they can't inject it. JWT tokens are signed with the server's private key, so an attacker cannot forge one either. If an attacker has physical access to the user's browser to inspect a token, then there's no authentication scheme in the world that would protect them.

1 • Reply • Share ›

**Jyoti** ➚ Rich Freedman • a year ago

Thanks Rich.

Reply • Share ›

**Hareesh Veduraj** • a year ago

Im getting the below error, while starting the server,after integrating the security related code.
Caused by: org.h2.jdbc.JdbcSQLException: Table "USER" not found; SQL statement:
Am I missing any configuration here? Can somebody help me oiut

Reply • Share ›

**Mabrouk Achraf** • a year ago

Thanx can i add a new admin in the database who has the same abilities that has the first admin

Reply • Share ›

**Rich Freedman** ➚ Mabrouk Achraf • a year ago

Yes, of course!
Take a look at the data.sql file in src/main/resources. The 'password' value for the user is the bcrypt hash of the user's password. In addition to inserting the user record, make sure that you insert the appropriate rows into the user_authority table, so that the user gets the appropriate role(s).

If you are going to be creating users more than just once in a great while, it would probably be good to add this functionality to your application. When doing so, take note of the fact that Spring Security uses BCryptPasswordEncoder to create bcrypt hashes.

Reply • Share ›

**Mabrouk Achraf** ➚ Rich Freedman • a year ago

hello thanx for all this informations but i can't connect with user user only admin admin work

Reply • Share ›

**Sebastián Oliveros** • a year ago

Thanks for your post.
I have a problem using the Token in the header. I am debugging the Sprint Rest Service and i can see that the method request.getheader("Authorization") does not get the token.

I am sending the header in the http.get method:
getUsers(): Promise<user> {
console.log('Token:' + this.authenticationService.getToken());

let headers = new Headers({'Content-Type': 'application/json','Authorization': 'Bearer ' + this.authenticationService.getToken() });
let options = new RequestOptions({ headers: headers });

return this.http
.get(this.userUrl, options)
.toPromise()
.then(response => response.json()._embedded.user as User)
.catch(this.handleError);
}

JWT Filter
String authToken = request.getHeader(this.tokenHeader); ==> authToken always is null

Do you know what could be happening?

Regards.

^ | v • Reply • Share ›

**Rich Freedman** ➚ Sebastián Oliveros • a year ago

Sebastian,

Your TypeScript code looks correct to me.
There's a couple of things that you could try to figure out what's going on...

First, use the "network' tab of the Chrome Developer Tools to make sure that the header is being sent, and that it looks correct.

Assuming that the header is actually sent, then in your JWT Filter on the server side, enumerate all of the received headers with request.getHeaders(), and see what you get. Perhaps "this.tokenHeader" is something other than "Authorization", maybe mis-spelled, or with a space in it?

1 ^ | v • Reply • Share ›

**Juan Ignacio Liska** ➚ Rich Freedman • a year ago

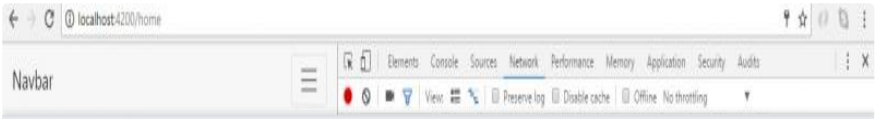
Same issue for me

In the network tab figure that result:

Request Headers
Provisional headers are shown
Access-Control-Request-Headers:access-control-allow-origin,authorization
Access-Control-Request-Method:GET
Origin:http://localhost:4200
Referer:http://localhost:4200/home
User-Agent:Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36

Chrome/56.0.3029.110 Safari/537.36

And if you see the image bellow i see the token:




**see more**

• Reply • Share ›

**Rich Freedman** ➚ Juan Ignacio Liska • a year ago

Juan,
Looks like your 'Authorization' header is actually 'authorization' (first letter is lower-case instead of upper-case). Header names are case sensitive, so I think that you need to change yours to start with an upper-case 'A'. That may or may not entirely fix your problem, but it is a necessary first step.

• Reply • Share ›