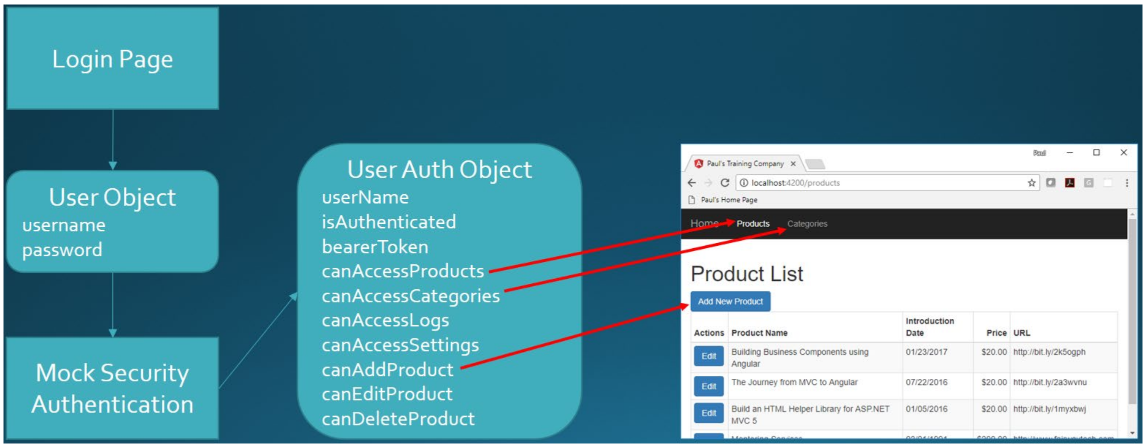
Security in Angular - Part 1

In most business applications, you are going to want to disable, or make invisible, different features such as menu items, buttons and other UI items, based on who is logged in and what roles or permissions they have. Angular does not have anything built-in to help you with this, so you must create it yourself. There are two different pieces to security you must worry about with Angular applications. First, you must develop the client-side security, which is the subject of this article. Second, you must secure your Web API calls, which will be the subject of another article.

Approaches to Security

There are many different approaches you can take to securing HTML items in Angular. You can create a simple security object that has one property for each item in your application you wish to secure as illustrated in Figure 1. This approach is great for smaller Angular applications as you won't have that many items to secure. For larger Angular applications, you will want to employ a claims-based and/or a role-based solution. This first article is going to focus on the simple security object with one property for each item to secure. This approach helps you focus on how to accomplish security before you tackle claims and roles. This article is using mock security objects, so you don't need to use any Web API calls. You are going to learn to retrieve security objects from a Web API in the next article.



Actions	Product Name	Introduction Date	Price	URL
Edit	Building Business Components using Angular	01/23/2017	$20.00	http://bit.ly/2k5ogph
Edit	The Journey from MVC to Angular	07/22/2016	$20.00	http://bit.ly/2a3wvnu
Edit	Build an HTML Helper Library for ASP.NET MVC 5	01/05/2016	$20.00	http://bit.ly/1myxbwj

Figure 1: Security authentication and authorization using single properties.

Preparing for this Article

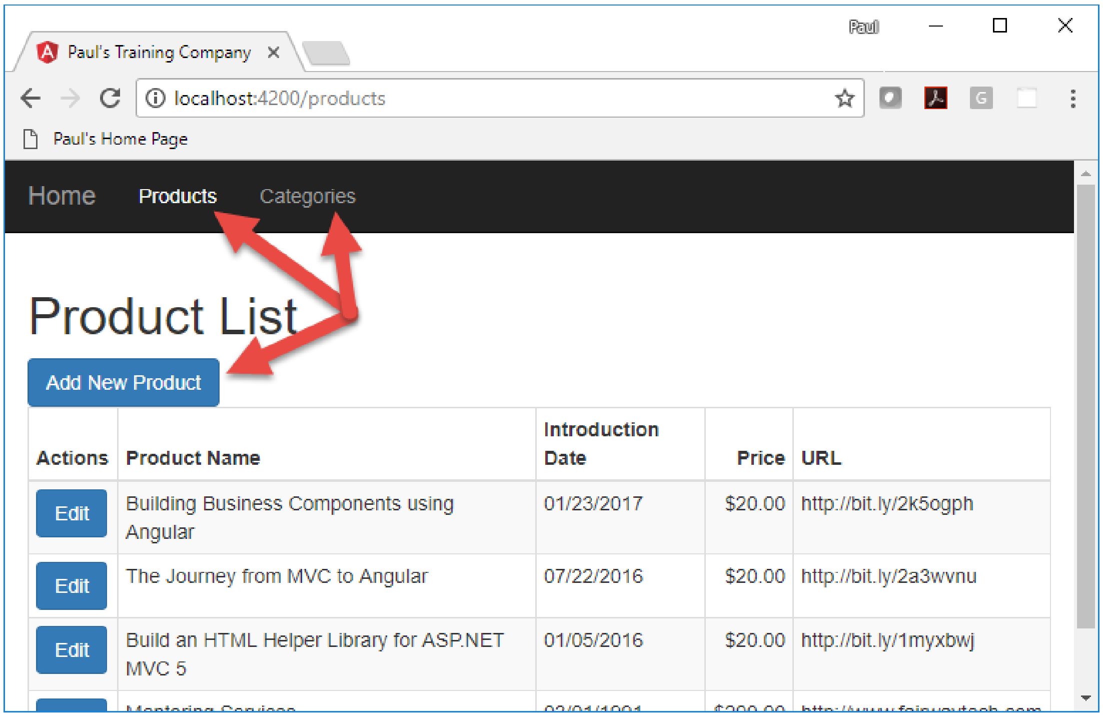
To demonstrate how to apply security to an Angular application, I created a sample application with a few pages to display products, display a single product, and display a list of product categories. You can download this sample from **http://pdsa.com/downloads**. Select "PDSA/Fairway Blog" from the Category drop-down, then choose "Security in Angular - Part 1".

This article assumes you have the following tools installed.

- Visual Studio Code
- Node
- Node Package Manager (npm)
- Angular CLI

A Look at the Sample Application

In the sample you downloaded, there are two menus, Products and Categories (Figure 2), that you may wish to turn off based on permissions assigned to a user. On the product and category list page (Figure 2), you may want to turn off the Add button based on permissions.

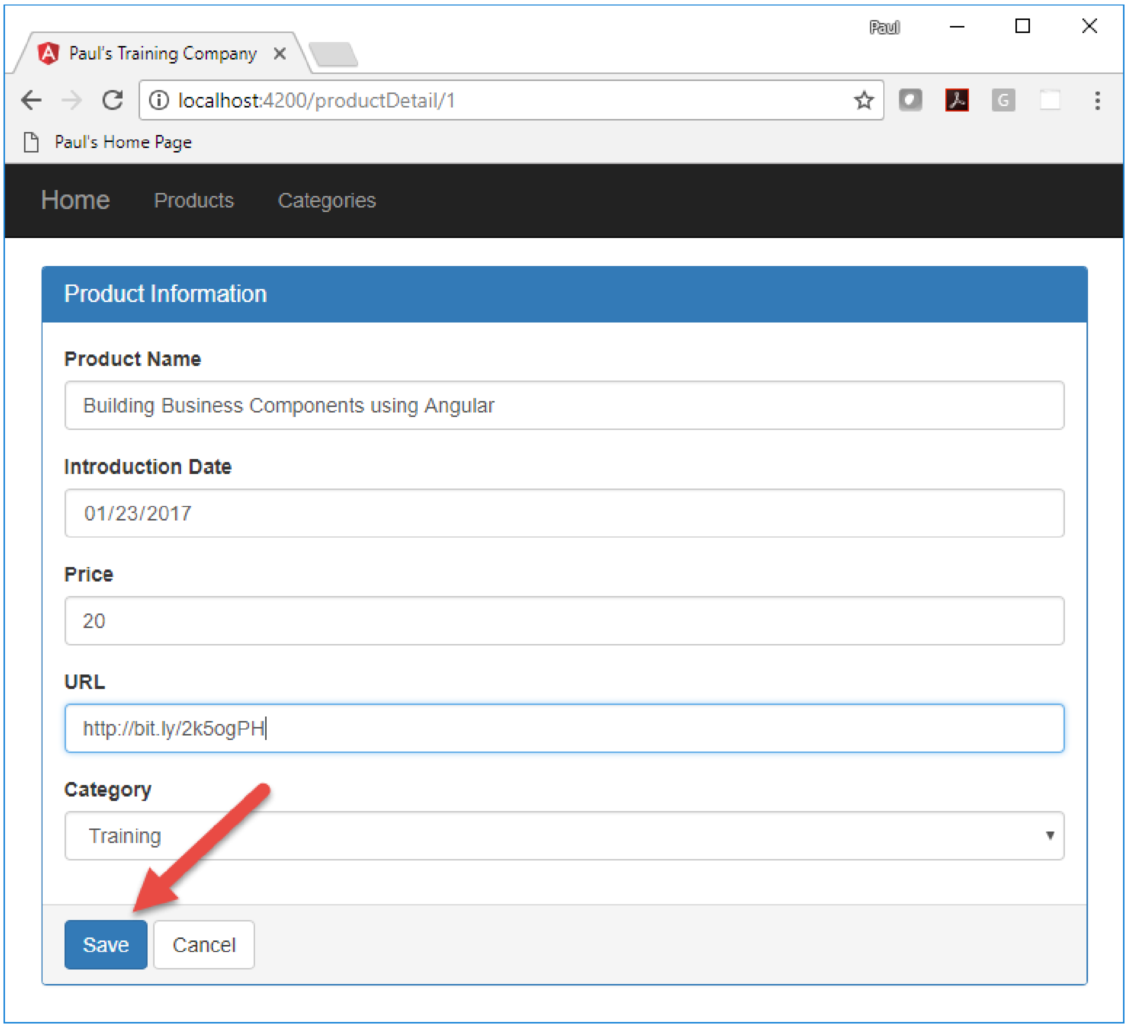


Product List

Actions	Product Name	Introduction Date	Price	URL
Edit	Building Business Components using Angular	01/23/2017	$20.00	http://bit.ly/2k5ogph
Edit	The Journey from MVC to Angular	07/22/2016	$20.00	http://bit.ly/2a3wvnu
Edit	Build an HTML Helper Library for ASP.NET MVC 5	01/05/2016	$20.00	http://bit.ly/1myxbwj

Figure 2: Product list page

On the product detail page (Figure 3), the Save button may be something you wish to turn off. Perhaps someone can view product detail, but not modify the data.



Paul's Training Company

localhost:4200/productDetail/1

Paul's Home Page

Product Information

Product Name: Building Business Components using Angular

Introduction Date: 01/23/2017

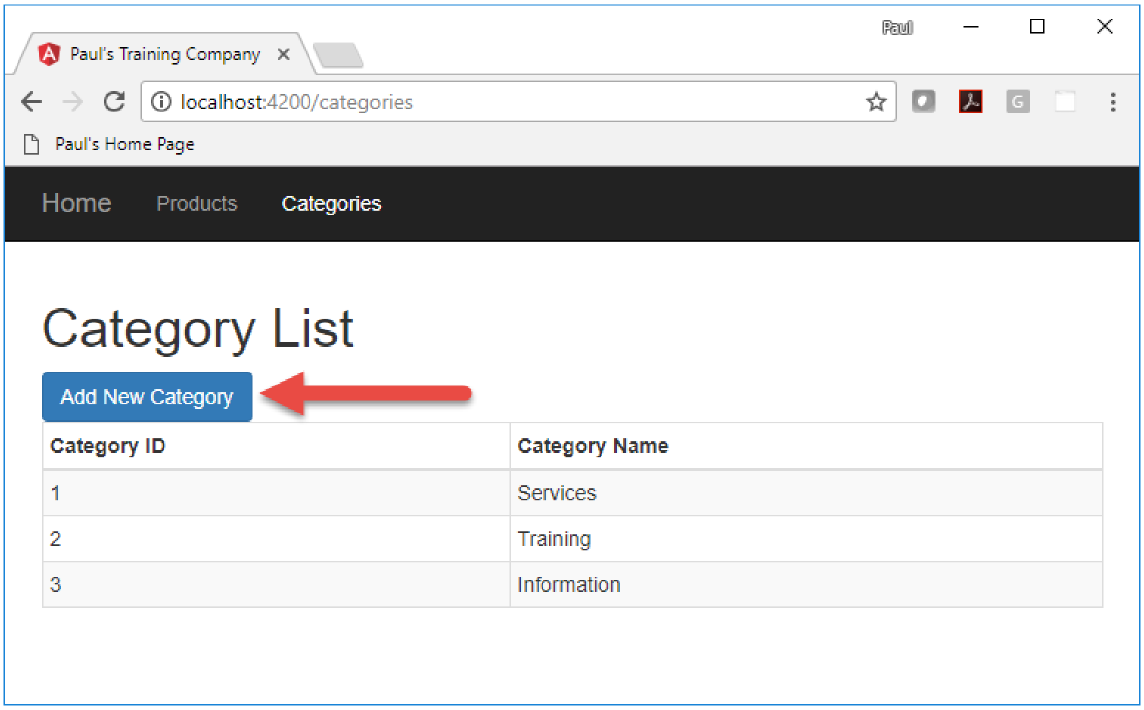
Price: 20

URL: http://bit.ly/2k5ogPH

Category: Training

Figure 3: Turn off the Save button based on permissions

Finally, on the Categories page (Figure 4), you may wish to make the Add New Category button invisible.



Category List

Category ID	Category Name
1	Services
2	Training
3	Information

Figure 4: Turn off the Add New Category button based on permissions

Create User Security Classes

To secure an application, you need a couple of classes to hold user information. First, you need a user class to hold the user name and password that can be entered on a login page and verified against some data source. In the first part of this article, a mock set of logins is used for verification. Secondly, a user authentication/authorization class is used with properties for each item in your application you wish to secure.

Next, you need a security service class to authenticate a user and set properties in the user authentication/authorization object. The property values determine the permissions the logged in user has. You use the properties to turn on and off different menus, buttons or other UI elements on your pages.

User Class

Create the user class to hold the user name and password the user types into a login page. Right mouse-click on the **\src\app** folder and add a new folder named **security**. Right mouse-click on the new **security** folder and add a file named **app-user.ts**. Add two properties into this AppUser class as shown in the following code.

```
export class AppUser  {
  userName: string = "";
  password: string = "";
}
```

User Authentication/Authorization Class

It is now time to create that class used to turn menus and button off and on. Right mouse-click on the **security** folder and add a new file named **app-user-auth.ts**. This class contains the *username* property to hold the user name of the authenticated user, a *bearerToken* to be used when interacting with Web API calls, and a boolean property named *isAuthenticated* which is only set to true when a user has been authenticated. The rest of the boolean properties contained in this class are specific for each menu and button you wish to secure.

```
export class AppUserAuth {
  userName: string = "";
  bearerToken: string = "";
  isAuthenticated: boolean = false;
  canAccessProducts: boolean = false;
  canAddProduct: boolean = false;
  canSaveProduct: boolean = false;
  canAccessCategories: boolean = false;
  canAddCategory: boolean = false;
}
```

Login Mocks

In the first part of this article, you are going to keep all authentication and authorization local within this Angular application. To do this, create a file with mock logins. Right mouse-click on the **security** folder and add a new file named **login-mocks.ts**. Create a constant named LOGIN_MOCKS that is an array of AppUserAuth objects. Create a couple of literal objects to simulate two different user objects you might retrieve from a database on a backend server.

```
import { AppUserAuth } from "./app-user-auth";

export const LOGIN_MOCKS: AppUserAuth[] = [
  {
    userName: "PSheriff",
    bearerToken: "abi393kdkd9393ikd",
    isAuthenticated: true,
    canAccessProducts: true,
    canAddProduct: true,
    canSaveProduct: true,
    canAccessCategories: true,
    canAddCategory: false
  },
  {
    userName: "BJones",
    bearerToken: "sd9f923k3kdmcjkhd",
    isAuthenticated: true,
    canAccessProducts: false,
    canAddProduct: false,
    canSaveProduct: false,
    canAccessCategories: true,
    canAddCategory: true
  }
];
```

Security Service

Angular is all about services, so it makes sense that you should create a security service class to authenticate a user and return the user's authorization object with all the appropriate properties set. Open a VS Code terminal window and type in the following command to generate a service class named **SecurityService**. Add the -m option to register this service in the app.module file.

```
ng g s security/security --flat
```

Register this new service with the AppModule class. Open the **app.module.ts** file and add a new import.

```
import { SecurityService }
  from './security/security.service';
```

Into the providers property, add the SecurityService class.

```
providers: [ProductService, CategoryService, SecurityService],
```

Open the generated **security.service.ts** file and add the following import statements.

```
import { Observable, of } from 'rxjs';

import { AppUserAuth } from './app-user-auth';
import { AppUser } from './app-user';
import { LOGIN_MOCKS } from './login-mocks';
```

Add a property to the SecurityService class to hold the user authorization object. Initialize this object to a new instance of the AppUserAuth class so it creates the object in memory.

```
securityObject: AppUserAuth = new AppUserAuth();
```

Reset Security Object Method

Once you have created this security object, you do not ever want to reset it to a new object, instead, just change the properties of this object based on a new user that logs in. Add a method to reset this security object to a default value.

```
resetSecurityObject(): void {
  this.securityObject.userName = "";
  this.securityObject.bearerToken = "";
  this.securityObject.isAuthenticated = false;

  this.securityObject.canAccessProducts = false;
  this.securityObject.canAddProduct = false;
  this.securityObject.canSaveProduct = false;
  this.securityObject.canAccessCategories = false;
  this.securityObject.canAddCategory = false;

  localStorage.removeItem("bearerToken");
}
```

Login Method

Soon, you are going to create a login page. That login component creates an instance of the AppUser class and binds the properties to input fields on that page. Once the user has typed in their user name and password, this instance of the AppUser class is going to be passed to a login() method in the SecurityService class to determine if the user exists. If the user exists, the appropriate properties are filled into a AppUserAuth object and returned from the login() method.

```
login(entity: AppUser): Observable<AppUserAuth> {
  // Initialize security object
  this.resetSecurityObject();

  // Use object assign to update the current object
  // NOTE: Don't create a new AppUserAuth object
  //       because that destroys all references to object
  Object.assign(this.securityObject,
    LOGIN_MOCKS.find(user => user.userName.toLowerCase() ===
                             entity.userName.toLowerCase()));
  if (this.securityObject.userName !== "") {
    // Store into local storage
    localStorage.setItem("bearerToken",
       this.securityObject.bearerToken);
  }

  return of<AppUserAuth>(this.securityObject);
}
```

The first thing to do is to reset the security object, so the resetSecurityObject() is called. Next, you use the Object.assign() method to replace all the properties in the securityObject property with the properties from the AppUserAuth object returned from the find() method on the LOGIN_MOCKS array. If the user is found, the bearer token is stored into local storage. This is done for when you need to pass this value to the Web API. This article is not going to cover that, but a future article will.

Logout Method

If you have a login method, you should always have a logout() method. The logout() method resets the properties in the *securityObject* property to empty fields, or false values. By resetting the properties, any bound properties such as menus, reread those properties and may change their state from visible to invisible.

```
logout(): void {
  this.resetSecurityObject();
}
```

Login Page

Now that you have a security service to perform a login, you need to retrieve a user name and password from the user. Create a Login page by opening a terminal window and type in the following command to generate a login page.

```
ng g c security/login --flat
```

Open the **login.component.html** file and delete the HTML that was generated. Create three distinct rows on the new login page.

1. Invalid User Name/Password message.
2. Row to display the instance of the securityObject property.
3. Panel for entering user name and password.

Use Bootstrap styles to create each of these rows on this login page. The first div contains a *ngIf directive to only display the message if the securityObject exists, and the *isAuthenticated* property is false. The second div element contains a binding to the *securityObject* property. This object is sent to the json pipe to display the object as a string within a label element. The last row is a Bootstrap panel into which you place the appropriate user name and password input fields.

```
<div class="row">
  <div class="col-xs-12">
    <div class="alert alert-danger"
    *ngIf="securityObject &&
           !securityObject.isAuthenticated">
      <p>Invalid User Name/Password.</p>
    </div>
  </div>
</div>

<!-- TEMPORARY CODE TO VIEW SECURITY OBJECT -->
<div class="row">
  <div class="col-xs-12">
    <label>{{securityObject | json}}</label>
  </div>
</div>

<form>
  <div class="row">
    <div class="col-xs-12 col-sm-6">
      <div class="panel panel-primary">
        <div class="panel-heading">
          <h3 class="panel-title">Log in</h3>
        </div>
        <div class="panel-body">
          <div class="form-group">
            <label for="userName">User Name</label>
            <div class="input-group">
              <input id="userName" name="userName"
                     class="form-control" required
                     [(ngModel)]="user.userName"
                     autofocus="autofocus" />
              <span class="input-group-addon">
                <i class="glyphicon glyphicon-envelope"></i>
              </span>
            </div>
          </div>
          <div class="form-group">
            <label for="password">Password</label>
            <div class="input-group">
              <input id="password" name="password"
                     class="form-control" required
                     [(ngModel)]="user.password"
                     type="password" />
              <span class="input-group-addon">
                <i class="glyphicon glyphicon-lock"></i>
              </span>
            </div>
          </div>
        </div>
        <div class="panel-footer">
          <button class="btn btn-primary" (click)="login()">
            Login
          </button>
        </div>
      </div>
    </div>
  </div>
</form>
```

Modify Login Component TypeScript

As you can see from the HTML you entered into the login.component.html file, there are two properties required for binding to the HTML elements; *user* and *securityObject*. Open the **login.component.ts** file and add the following import statements, or if you wish, use VS Code to insert them for you as you add each class.

```
import { AppUser } from './app-user';
import { AppUserAuth } from './app-user-auth';
import { SecurityService } from './security.service';
```

Add two properties to hold the user and the user authorization object.

```
user: AppUser = new AppUser();
securityObject: AppUserAuth = null;
```

To set the *securityObject*, you need to inject the SecurityService into this class. Modify the constructor to inject the SecurityService.

```
constructor(private securityService: SecurityService) { }
```

The button in the footer area of the Bootstrap panel binds the click event to a method named login(). Add this login() method as shown below. This method first removes any previous bearer token that may have been stored in local storage. The login() method on the SecurityService class is subscribed to, and the response that is returned is assigned into the *securityObject* property defined in this login component.

```
login() {
  this.securityService.login(this.user)
    .subscribe(resp => {
      this.securityObject = resp;
    });
}
```

Secure Menus

Now that you have the login working and a valid security object, you need to bind this security object to the main menu. The menu system is created in the **app.component.html** file, so you need to open that file and add a new menu item to call the login page. Add the following HTML below the closing </ul> tag used to create the other menus. This HTML creates a right-justified menu that displays the word "Login" when the user is not yet authenticated. Once authenticated, the menu changes to Logout <User Name>.

```
<ul class="nav navbar-nav navbar-right">
  <li>
    <a routerLink="login"
       *ngIf="!securityObject.isAuthenticated">
      Login
    </a>
    <a href="#" (onclick)="logout()"
       *ngIf="securityObject.isAuthenticated">
      Logout {{securityObject.userName}}
    </a>
  </li>
</ul>
```

Just above the code you just added, modify the other two menu items to also check the security object to determine if they need to be displayed or not. Use the *ngIf directive to check the *securityObject* property you are going to add to the AppComponent class. You check the appropriate boolean properties that correspond to each menu.

```
<li>
  <a routerLink="/products"
     *ngIf="securityObject.canAccessProducts">Products</a>
</li>
<li>
  <a routerLink="/categories"
     *ngIf="securityObject.canAccessCategories">Categories</a>
</li>
```

Modify the AppComponent Class

As you saw from the HTML you entered, you need to add the securityObject to the component associated with the app. Open the **app.component.ts** file and add the securityObject property. You do need to set it equal to a null value to start with so the Invalid User Name/Password message does not show.

```
securityObject: AppUserAuth = null;
```

Modify the constructor of the AppComponent class to inject the SecurityService and assign the *securityObject* property to the property you just created.

```
constructor(private securityService: SecurityService) {
  this.securityObject = securityService.securityObject;
}
```

Add a logout() method to this class that calls the logout() method on the security service class. This method is bound to the click event on the Logout menu item you added in the HTML.

```
logout(): void {
  this.securityService.logout();
}
```

Add Login Route

To get to the login page, you need to add a route. Open the **app-routing.module.ts** file and add a new route like the one shown below.

```
{
  path: 'login',
  component: LoginComponent
},
```

Try it Out

Save all the changes you have made so far. Start the application using **npm start**. Click the Login menu and login with "psheriff" and notice the properties that are set in the returned security object. Click the logout button, then login back in as "bjones" and notice that different properties are set, and the Product link goes away. This is because the *canAccessProducts* property in the LOGIN_MOCKS array for BJones is set to false.

Open up the **logins-mock.ts** file and set the *canAccessProducts* property to true for the BJones object.

```
{
  userName: "BJones",
  bearerToken: "sd9f923k3kdmcjkhd",
  isAuthenticated: true,
  canAccessProducts: true,
  canAddProduct: false,
  canSaveProduct: false,
  canAccessCategories: true,
  canAddCategory: true
}
```

You are going to try out some of the different authorization properties and this needs to be set to true to try them out.

Secure Buttons

Besides the permissions you added to the menus, you also might want to apply the same to buttons that perform actions. For example, adding a new product or category. Or, saving product data. For this article, you are only learning how to hide HTML elements. If there were Web API method calls behind these buttons, those are not being secured here. You need to secure the Web API using some sort of token system. Those techniques will be covered in a future article.

Let's secure the Add New Product button by using the security object created after logging in. Open the **product-list.component.html** file and modify the Add New Product button to look like the following:

```
<button class="btn btn-primary"
        (click)="addProduct()"
        *ngIf="securityObject.canAddProduct">
  Add New Product
</button>
```

Open the **product-list.component.ts** file and add a property named *securityObject* that is of the type AppUserAuth. You are going to want to add this same property to any component you wish to use security upon.

```
securityObject: AppUserAuth = null;
```

Assign the *securityObject* property you just created to the *securityObject* property in the SecurityService class. Inject the service in the constructor and retrieve the security object. You are going to want to use this same design pattern in any component you wish to secure.

```
constructor(private productService: ProductService,
  private router: Router,
  private securityService: SecurityService) {
  this.securityObject = securityService.securityObject;
}
```

Open the **product-detail.component.html** file and modify the Save button to use the *canSaveProduct* property on the securityObject. The *ngIf directive will cause the button to disappear if the *canSaveProduct* property is false.

```
<button class="btn btn-primary"
       (click)="saveData()"
       *ngIf="securityObject.canSaveProduct">
  Save
</button>
```

Open the **product-detail.component.ts** file and add the *securityObject* property, just like you did in the product-list.component.ts file.

```
securityObject: AppUserAuth = null;
```

Modify the constructor to inject the SecurityService and to assign the *securityObject* property from the SecurityService to the *securityObject* property you just created in this class.

```
constructor(private categoryService: CategoryService,
  private productService: ProductService,
  private route: ActivatedRoute,
  private location: Location,
  private securityService: SecurityService) {
  this.securityObject = securityService.securityObject;
}
```

Open the **category-list.component.html** file and modify the Add New Category button to use the *canAddCategory* property on the *securityObject*.

```
<button class="btn btn-primary"
        (onclick)="addCategory()"
        *ngIf="securityObject.canAddCategory">
  Add New Category
</button>
```

Open the **category-list.component.ts** file and add the *securityObject* property.

```
securityObject: AppUserAuth = null;
```

Modify the constructor to inject the SecurityService and to assign the *securityObject* property from the SecurityService to the *securityObject* property you just created in this class.

```
constructor(private categoryService: CategoryService,
  private securityService: SecurityService) {
  this.securityObject = securityService.securityObject;
}
```

Try it Out

Save all the changes you have made and go to your browser. Click the Login menu and login with "psheriff" and notice the properties that are set in the returned security object.

Open the Products page and you can click on the Add New Product button. If you click on an Edit button next to one of the products, you can see the Save button on the product detail page. Open the Category page and notice that the Add New Category button is not visible to you.

Click the logout button and login as "bjones". Notice that different properties are set on the security object.

Open the Products page and notice you the Add New Product button is not visible. If you click on an Edit button next to one of the products, the Save button on the product detail page is not visible to you. Open the Category page and notice that the Add New Category button is visible.

Secure Routes Using a Guard

Even though you can control the visibility of menu items, just because you can't click on them doesn't mean you can't get to the route. You can type the route directly into the browser address bar and you can get to the products page even if you don't have the *canAccessProducts* property set to true.

To protect the route, you need to build a Route Guard. A Route Guard is a special class in Angular to determine if a page can be activated, or even deactivated. Let's learn how to build a CanActivate guard. Open a terminal and create a new guard named AuthGuard.

```
ng g g security/auth --flat
```

Register this new guard with the AppModule class. Open the **app.module.ts** file and add a new import.

```
import { AuthGuard } from './security/auth.guard';
```

Into the providers property, add the AuthGuard class.

```
providers: [ProductService, CategoryService,
            SecurityService, AuthGuard],
```

To protect a route, open the **app-routing.module.ts** file and add the *canActivate* property to those paths you wish to secure. You pass one or many guards to this property. In this case, add the AuthGuard class to the array of guards. For each route you also need to specify the name of the property to check on the security object that is associated with this route. Add a *data* property and pass in a property named *claimType* and set the value of that property to the name of the property associated with the route. This *data* property is passed to each Guard listed in the *canActivate* property.

```
{
  path: 'products',
  component: ProductListComponent,
  canActivate: [AuthGuard],
  data: {claimType: 'canAccessProducts'}
},
{
  path: 'productDetail/:id',
  component: ProductDetailComponent,
  canActivate: [AuthGuard],
  data: {claimType: 'canAccessProducts'}
},
{
  path: 'categories',
  component: CategoryListComponent,
  canActivate: [AuthGuard],
  data: {claimType: 'canAccessCategories'}
},
```

Authorization Guard

Let's write the appropriate code in the AuthGuard to secure the route. Since you are going to need to access the property passed in via the data property, open the **auth-guard.ts** file and add a constructor to inject the SecurityService.

```
constructor(private securityService: SecurityService) { }
```

Modify the canActivate() method to retrieve the *claimType* property in the *data* property. Remove the "return true" statement and add the following lines of code in its place.

```
canActivate(
  next: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Observable<boolean>
                             | Promise<boolean> | boolean {
  // Get property name on security object to check
  let claimType: string = next.data["claimType"];

  return this.securityService.securityObject.isAuthenticated
         && this.securityService.securityObject[claimType];
}
```

Retrieve the property name to check on the security object using the *next* parameter. This property is an ActivatedRouteSnapshot and contains the data object passed via the route you created earlier. A true value returned from this guard means that the user has the right to navigate to this route. Check to ensure that the *isAuthenticated* property on the securityObject is a true value and that the property name passed in the data object is also a true value.

Try it Out

Save all the changes you have made and go to the browser and type directly into the browser address bar http://localhost:4200/products. If you are not logged in, you are not able to get to the products page. Your guard is working; however, it ends up displaying a blank page. It would be better to redirect to the login page.

Redirect to Login Page

To redirect to the login page, modify the AuthGuard class to perform the redirection if the user is not authorized for the current route. Open the **auth-guard.ts** file and inject the Router service into the constructor.

```
constructor(private securityService: SecurityService,
  private router: Router) { }
```

Modify the canActivate() method. Remove the current **return** statement and replace it with the following lines of code.

```
if (this.securityService.securityObject.isAuthenticated
    && this.securityService.securityObject[claimType]) {
  return true;
}
else {
  this.router.navigate(['login'],
      { queryParams: { returnUrl: state.url } });
  return false;
}
```

If the user is authenticated and authorized, the Guard returns a true and Angular goes to the route. Otherwise, use the Router object to navigate to the login page. Pass the current route the user was attempting to get to as a query parameter. This places the route on the address bar for the login component to retrieve and use to go to the route requested after a valid login.

Try it Out

Save all your changes, go to the browser, and type directly into the browser address bar http://localhost:4200/products. The page will reset, and you will be directed to the login page. You should see a *returnUrl* parameter in the address bar. You can login, but you won't be redirected to the products page, you need to add some code to the login component.

Redirect Back to Requested Page

If the user logs in with the appropriate credentials that allows them to get to the requested page, then you want to direct them to that page after login. The LoginComponent class should return the *returnUrl* query parameter and attempt to navigate to that route after successful login. Open the **login.component.ts** file and inject the ActivatedRoute and the Router objects into the constructor.

```
constructor(private securityService: SecurityService,
            private route: ActivatedRoute,
            private router: Router) { }
```

Add a property to this class to hold the return url if any is retrieved from the address bar.

```
returnUrl: string;
```

Add a line to the ngOnInit() method to retrieve this returnUrl query parameter. If you click on the Login menu directly, the queryParamMap.get() method returns a null.

```
ngOnInit() {
  this.returnUrl =
     this.route.snapshot.queryParamMap.get('returnUrl');
}
```

Locate the login() method and add code after setting the *securityObject* to test for a valid url and to redirect to that route if there is one.

```
login() {
  localStorage.removeItem("bearerToken");

  this.securityService.login(this.user)
    .subscribe(resp => {
      this.securityObject = resp;
      if (this.returnUrl) {
        this.router.navigateByUrl(this.returnUrl);
      }
    });
}
```

Try it Out

Save all your changes, go to the browser, and type directly into the browser address bar http://localhost:4200/products and you will be directed to login page. Login as "psheriff" and you are redirected to the products list page.

Summary

In this article you learned to add client-side security to your Angular applications. Using a class with properties to represent each "permission" you want to grant to each user, makes securing menu links and buttons easy. Apply Route Guards to your routes to ensure no one can get to a page by typing directly into the address bar. One thing you can do instead of adding the securityObject property to each component, is create a custom directive to which you pass the permission you want to check.

Everything was done client-side in this article; however, you can authenticate users, and return a security authorization object using a Web API call. The techniques in this article do not address securing your Web API methods. We will look at how to do this in a future article.

Final Sample Code

You can download the complete sample code at my website. http://www.pdsa.com/downloads. Choose "PDSA/Fairway Blog", then "Security in Angular - Part 1 - Finished" from the drop-down.