Using Claims to Secure Angular - Part 1

I previously published a couple of articles on how to create a security system in Angular. In those articles, a set Angular classes for users' authentication/authorization were created. You used these classes to login a user and create a set of properties in a class to turn menus and buttons on and off. For each menu or button you want to turn on or off, you have a corresponding property in a AppUserAuth class. This works for smaller applications, but for larger applications, you would be best to use a claims-based approach.

If you have used the Microsoft Identity system, you know it creates a set of "AspNet" tables. One of these tables is AspNetUserClaims into which you may assign a claim name and the value for the claim. You can think of claims like permissions. A claim may be something like 'I can add a Product', 'I can save a Product'. You can assign one or more claims to a user, or to a role.

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 "Using Claims to Secure Angular - Part 1". Within the zip file you download, there are two folders. One has a suffix of "-Start"; this is the sample without security. The other has a suffix of "-End" and is the final sample once you have completed all the steps in this article.

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 1), that you may wish to turn off based on claims assigned to a user. On the product and category list pages (Figure 1), you may want to turn off the Add button based on claims.

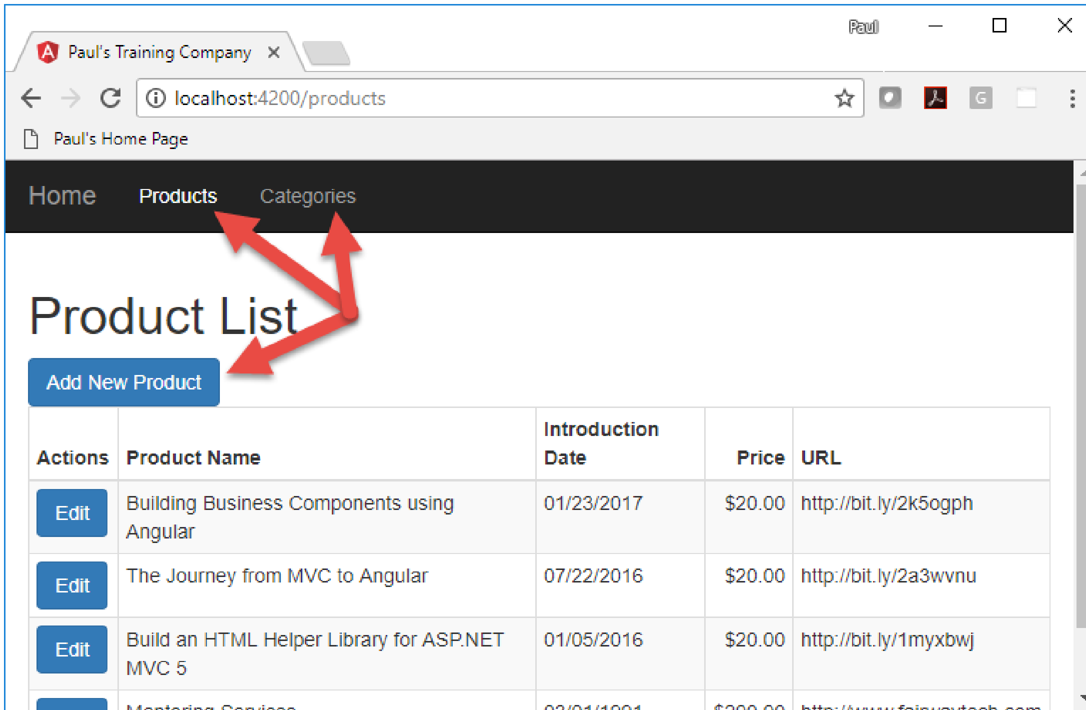

Figure 1: Product list page

On the product detail page (Figure 2), the Save button may be something you wish to turn off. Perhaps someone can view product detail, but not modify the data.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

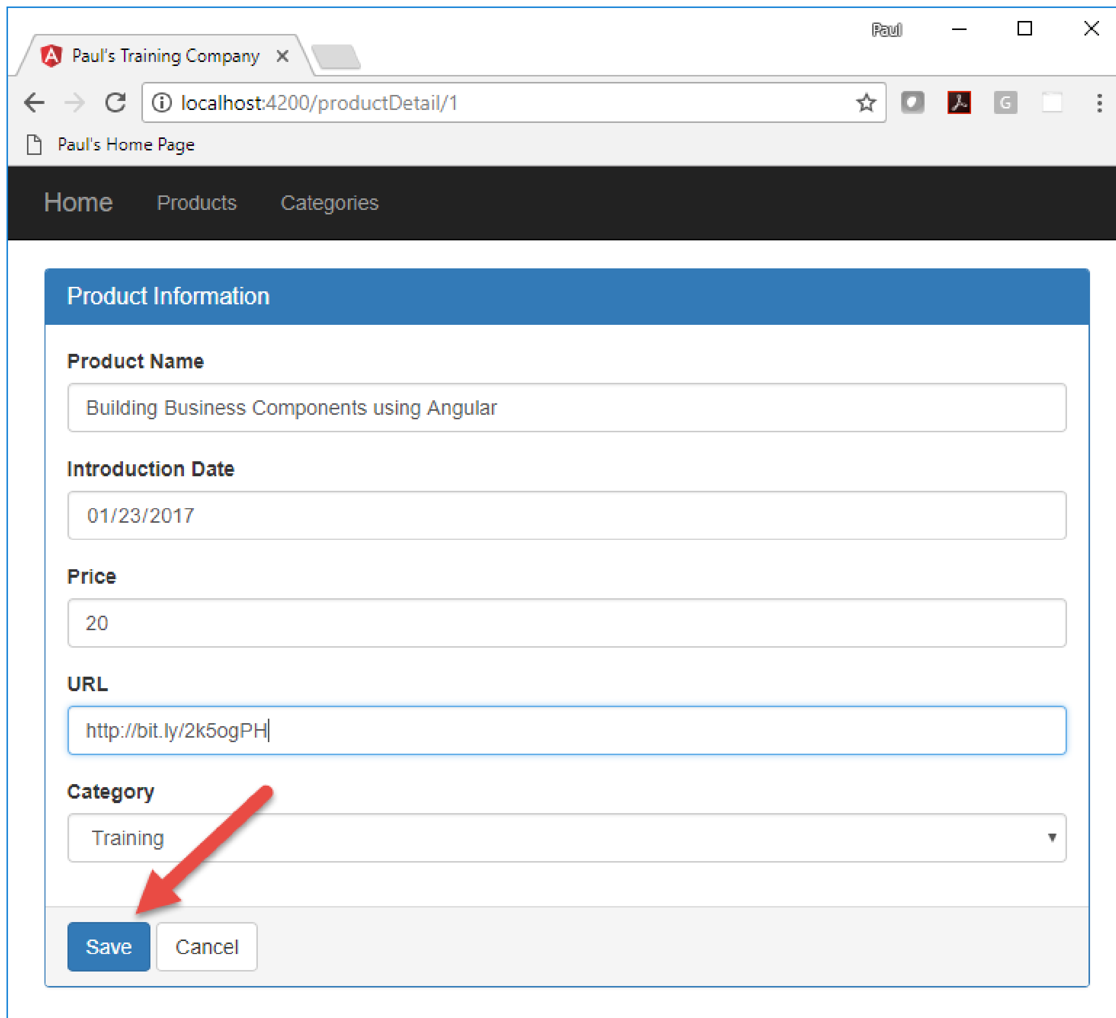

Figure 2: Turn off the Save button based on claims

Finally, on the Categories page (Figure 3), you may wish to make the Add New Category button invisible if someone does not have the appropriate claims.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

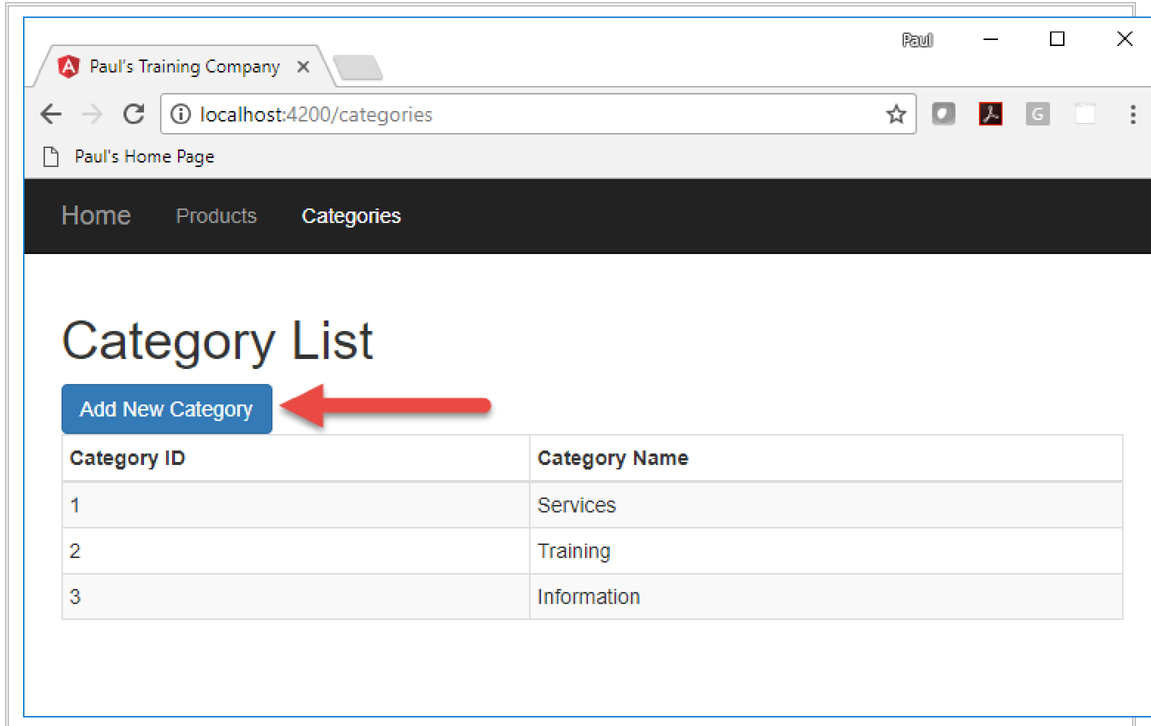

Figure 3: Turn off the Add New Category button based on permissions

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 for the logged in user. 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 entered intothe 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.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

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

AppUserClaim Class

To represent claims, create a class named AppUserClaim. This class mimics the Microsoft Identity table "AspNetUserClaims". Right mouse-click on the security folder and add a new file named **app-user-claim.ts**. Add the following code in this file.

```
export class AppUserClaim  {
  claimId: number = 0;
  userId: number = 0;
  claimType: string = "";
  claimValue: string = "";
}
```

User Authentication/Authorization Class

It is now time to create a class with an array of claims that will be 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 *userId* and *userName* properties to hold the user id and 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 final property, *claims*, holds an array of claims. These claims are going to be hard-coded in this application, but in a future article, you will retrieve these via a Web API call.

```
import { AppUserClaim } from "./app-user-claim";

export class AppUserAuth {
  userId: number = 0;
  userName: string = "";
  bearerToken: string = "";
  isAuthenticated: boolean = false;
  claims: AppUserClaim[] = [];
}
```

Login Mocks

In this article, you are going to keep all authentication and authorization local within the Angular application. 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

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

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.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

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

export const LOGIN_MOCKS: AppUserAuth[] = [
  {
    userId: 1,
    userName: "PSheriff",
    bearerToken: "abi393kdkd9393ikd",
    isAuthenticated: true,
    claims: [
      {
        userId: 1,
        claimId: 1,
        claimType: "isAuthenticated",
        claimValue: "true"
      },
      {
        userId: 1,
        claimId: 2,
        claimType: "canAccessProducts",
        claimValue: "true"
      },
      {
        userId: 1,
        claimId: 3,
        claimType: "canAddProduct",
        claimValue: "true"
      },
      {
        userId: 1,
        claimId: 4,
        claimType: "canSaveProduct",
        claimValue: "true"
      },
      {
        userId: 1,
        claimId: 5,
        claimType: "canAccessCategories",
        claimValue: "true"
      },
      {
        userId: 1,
        claimId: 6,
        claimType: "canAddCategory",
        claimValue: "false"
      }
    ]
  },
  {
    userId: 2,
    userName: "BJones",
    bearerToken: "sd9f923k3kdmcjkhd",
    isAuthenticated: true,
    claims: [
      {
        userId: 2,
```


Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

```
        claimId: 1,
        claimType: "isAuthenticated",
        claimValue: "true"
      },
      {
        userId: 2,
        claimId: 2,
        claimType: "canAccessProducts",
        claimValue: "false"
      },
      {
        userId: 2,
        claimId: 3,
        claimType: "canAddProduct",
        claimValue: "false"
      },
      {
        userId: 2,
        claimId: 4,
        claimType: "canSaveProduct",
        claimValue: "false"
      },
      {
        userId: 2,
        claimId: 5,
        claimType: "canAccessCategories",
        claimValue: "true"
      },
      {
        userId: 2,
        claimId: 6,
        claimType: "canAddCategory",
        claimValue: "true"
      }
    ]
  }
];
```

Security Service

Angular is all about services, so create a security service class to authenticate a user and return a user's authorization object with the appropriate properties. 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.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

```
ng g s security/security --flat -m app.module
```

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

```
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';
import { AppUserAuth } from './app-user-auth';
import { AppUser } from './app-user';
import { LOGIN_MOCKS } from './login-mocks';
import { AppUserClaim } from './app-user-claim';
```

Add a property named *securityObject* 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 when a new user authenticates. 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.claims = [];

  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 in that class to input fields on the 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.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

```
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 so that when you need to pass this value to the Web API, it is available and ready to use. 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. Resetting the properties (as opposed to creating a new instance of the class), keeps any bound properties on the *securityObject* from being thrown away. For instance, if you are turning off a menu such as the Products menu based on the value in the *isAuthenticated* property, if you create a new instance of the security Object, that bound property is released, and your menu visibility no longer works.

```
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.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

```
ng g c security/login --flat -m app.module
```

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.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

```
<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>
```


Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

```
    </div>
  </div>
</form>
```

Modify Login Component TypeScript

As you can see from the HTML you entered in 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* property, inject the SecurityService into the constructor of this class.

```
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. The login() method on the SecurityService class is subscribed, and the response returned is assigned into the *securityObject* property defined in this login component.

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

Add Login Menu

Now that you have a login page and a valid security object, you need to add a Login menu. The menu system is created in the **app.component.html** file, so you need to

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

open that file and add a new menu item to call the login page. Add the following HTML below the closing 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>
```

Modify the AppComponent Class

As you saw from the HTML you entered, you need to add a *securityObject* property to the component associated with the app. Open the **app.component.ts** file and add the *securityObject* property. Assign this property to a null value so the Invalid User Name/Password message does not display on the page.

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

Add a constructor to the AppComponent class to inject the SecurityService and assign the *securityObject* property from the SecurityService class 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();
}
```


Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

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.

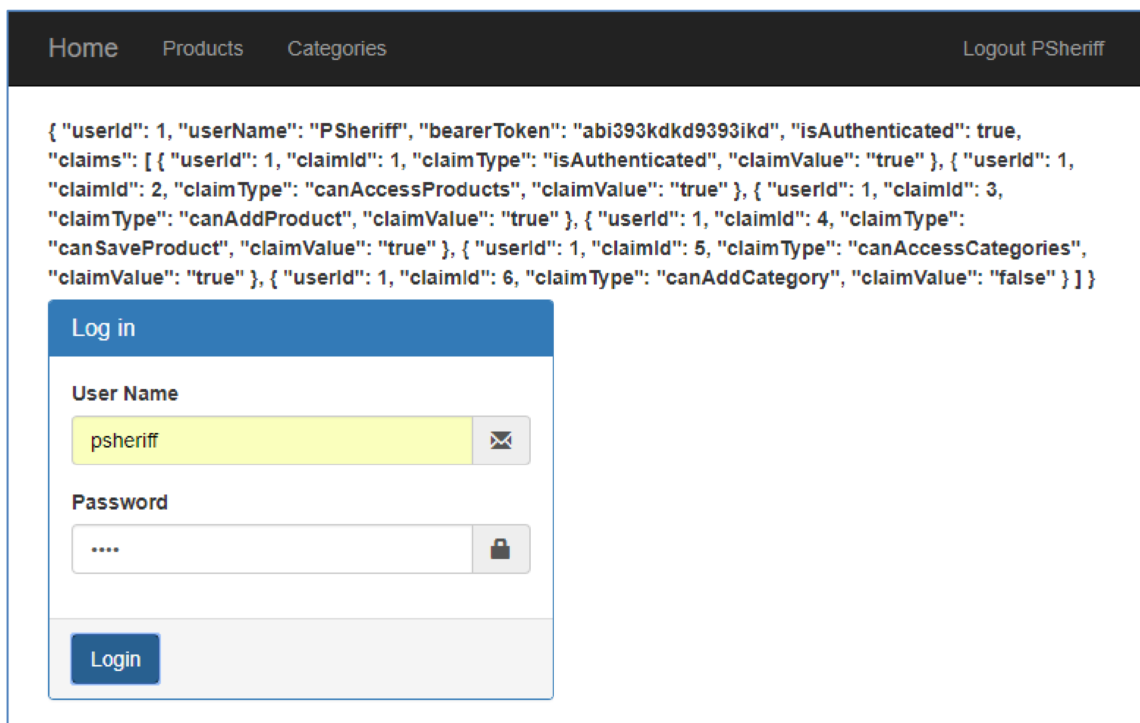

Figure 4: You can see all the various claims in the returned object for this logged in user.

Check for Claims

Open the **security.service.ts** file and add a new method named isClaimValid() to check if a user has a specified claim. This method accepts two parameters;

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

claimType and *claimValue*. The *claimValue* parameter is optional. Into the claimType parameter you may pass a claim like 'canAccessProduct' and if nothing is passed for the *claimValue* parameter, then the array of claims is searched for a claim that has *claimType* equals the value passed in, and the *claimValue* is equal to "true". If the claimValue is passed, then the *claims* array is searched for the *claimType* and the *claimValue* equal to what is passed in.

Another way to call this method is to pass the claim type and claim value in the *claimType* parameter separated by a colon. For example, you can pass "canAccessProducts:false" to the *claimType* parameter, and there is code in here to check if a colon exists and to separate out the claim type and the claim value before searching for these values in the *claims* array.

```
private isClaimValid(claimType: string, claimValue?: string) {
  let ret: boolean = false;
  let auth: AppUserAuth = null;

  // Retrieve security object
  auth = this.securityObject;
  if (auth) {
    // See if the claim type has a value
    // *hasClaim="'claimType:value'"
    if (claimType.indexOf(":") >= 0) {
      let words: string[] = claimType.split(":");
      claimType = words[0];
      claimValue = words[1];
    }
    else {
      // Either get the claim value, or assume 'true'
      claimValue = claimValue ? claimValue : "true";
    }
    // Attempt to find the claim
    ret = auth.claims.find(c => c.claimType == claimType
                       && c.claimValue == claimValue) != null;
  }

  return ret;
}
```

The isClaimValid is a private method in the security service class, so you need a public method to call this one. Create a hasClaim() method that looks like the following.

```
hasClaim(claimType: any, claimValue?: any) {
  return this.isClaimValid(claimType, claimValue);
}
```

The above method is very simple, but later, you are going to add code to check for an array of claims to be passed in, so just create this simple method for now.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

Has Claim Structural Directive

When you do not have properties to bind to on a class, but instead you have an array of objects, you can't use data-binding to secure buttons and menus. Instead you can use a structural directive like the *ngIf directive. You want to create a directive that can attach to a UI element like the following:

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

The directive is passed the value within the quotes as a string, checks the security object to see if that claim exists in the array for the current user, and if that claim value is true. If so, then the button is displayed, otherwise it is removed from the DOM.

Create this hasClaim Angular structural directive by opening a terminal window in Code and typing in the following command.

```
ng g d security/hasClaim --flat -m app.module
```

Open the **has-claim.directive.ts** file and modify the import statement to add a few more classes.

```
import { Directive, Input, TemplateRef, ViewContainerRef }
  from '@angular/core';
```

Modify the *selector* property from **ptcHasClaim** to **hasClaim**.

```
@Directive({ selector: '[hasClaim]' })
```

Modify the constructor to inject the TemplateRef, ViewContainerRef and the SecurityService.

```
constructor(
  private templateRef: TemplateRef<any>,
  private viewContainer: ViewContainerRef,
  private securityService: SecurityService) { }
```

Add the following @Input property to accept data from the right-hand side of the equal sign of the hasClaim directive as shown earlier.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

```
@Input() set hasClaim(claimType: any) {
  if (this.securityService.hasClaim(claimType)) {
    // Add template to DOM
    this.viewContainer.createEmbeddedView(this.templateRef);
  } else {
    // Remove template from DOM
    this.viewContainer.clear();
  }
}
```

The @Input() decorator tells Angular to pass the value to this property defined in the directive. The claimType is passed to the hasClaim() method you created earlier in the security service class. If the claim exists, the UI element to which this directive is applied is displayed on the screen.

Secure Add New Product Button

Try out your new directive by opening the **product-list.component.ts** file and adding this directive to the Add New Product button as shown in the code below.

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

Don't forget to add the single quotes inside the double quotes. If you forget them, Angular is going to try to bind to a property in your component named *canAddProduct* which does not exist.

Try it Out

Save all your changes and go back to the browser. Click on the Login menu and login as "psheriff". Click on the Products menu and you should see the Add New Product button appear. Logout psheriff and login as "bjones". The Add New Product button should now be gone. You may specify the value you want the claim to be by adding a colon, followed by the value after the claim name.

```
<button class="btn btn-primary" (click)="addProduct()"
        *hasClaim="'canAddProduct:false'">
  Add New Product
</button>
```

If you now login as "psheriff", the Add New Product button is gone. Login as "bjones" and it should appear. Remove the ":false" from the claim after you have tested this out.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

Add Multiple Claims

Sometimes your security requirements are such that you need to secure a UI element using multiple claims. For example, you want to display a button for people that have one claim, and for people that have another claim. To accomplish this, you need to pass an array of claims to the hasClaim directive as shown below.

```
*hasClaim="['canAddProduct', 'canAccessCategories']"
```

You need to modify the hasClaim method in the SecurityService class to check to see if just a single string value has been passed, or an array. Open the **security.service.ts** file and modify the hasClaim method to look like the following.

```
// This method can be called a couple of different ways
// *hasClaim="'claimType'"  // Assumes claimValue is true
// *hasClaim="'claimType:value'"  // Compares claimValue to value
// *hasClaim="['claimType1','claimType2:value',
               'claimType3']"
hasClaim(claimType: any, claimValue?: any) {
  let ret: boolean = false;

  // See if an array of values was passed in.
  if (typeof claimType === "string") {
    ret = this.isClaimValid(claimType, claimValue);
  }
  else {
    let claims: string[] = claimType;
    if (claims) {
      for (let index = 0; index < claims.length; index++) {
        ret = this.isClaimValid(claims[index]);
        // If one is successful, then let them in
        if (ret) {
          break;
        }
      }
    }
  }

  return ret;
}
```

As you now have two different data types that can be passed to the hasClaim() method, use the typeof operator to check if the *claimType* parameter is a string. If it is, call the isClaimValid() method passing in the two parameters. If it is not a string, assume it is an array. Cast the *claimType* parameter into a string array named claims. Verify it is an array, then loop through each element of the array and pass each element to the isClaimValid() method. If even one claim matches, then return a true from this method so the UI element is displayed.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

Secure Other Buttons

Open the **product-list.component.html** file and modify the Add New Product button to use an array.

```
*hasClaim="['canAddProduct', 'canAccessCategories']"
```

Open the **product-detail.component.html** file and modify the Save button.

```
<button class="btn btn-primary" (click)="saveData()"
        *hasClaim="'canSaveProduct'">
  Save
</button>
```

Open the **category-list.component.html** file and modify the Add New Category button.

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

Try it Out

Save all the changes in your application and go back to your browser. Login as "bjones" and because he has the canAccessCategories claim, he is allowed to view the Add New Product button. Change the hasClaim attribute in the **product-list.component.html** file so it is a single value again.

```
*hasClaim="'canAddProduct'"
```

Try it Out

Save all the changes and test the application to make sure that when you are logged in with the correct user, you see the correct buttons.

Create Observer Pattern

One consideration when using a structural directive that passes in a string value, instead of binding to a property on the component, is that if the securityObject gets a new set of claims due to a change in who is logged in, there is no automatic

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

refresh since there is no binding. This means that you need to come up with some other mechanism to inform the AppComponent page that claims have changed and any menus that are secured, need to potentially be re-displayed, or hidden. One method you can employ is an observer pattern where you inform any observers of the *securityObject* property in the security service that something has changed.

Open the **security.service.ts** file and add the following import statement. This BehaviorSubject class from RxJS allows you to setup an observable and an observer. The reason to use a BehaviorSubject as opposed to a normal Observable is a BehaviorSubject allows you to send a message to any observers.

```
import { BehaviorSubject } from 'rxjs/BehaviorSubject';
```

Create a private property named *hasChanged* and assign it to a generic number of the type of BehaviorSubject. Assign that object an initial value of zero. It doesn't matter what the initial value is.

```
private hasChanged = new BehaviorSubject<number>(0);
```

Next, create a public observable called *securityReset*. It is this public property that any observer can subscribe to receive changes to the value.

```
securityReset = this.hasChanged.asObservable();
```

When a new user logs in, you want to inform any observer that the *securityObject* property has received a new set of claims, and a new user. To inform them, call the next() method on the private *hasChanged* property and pass in any value.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

```
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);

    // Inform everyone that the security object has changed.
    this.hasChanged.next(0);
  }

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

Modify the resetSecurityObject() method to also inform all observers that the *securityObject* property has changed.

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

  // Inform everyone that the security object has changed.
  this.hasChanged.next(0);

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

Secure Menus

Now that you have a method to communicate that the *securityObject* property in the security service class have changed, you may now secure the menus in the app.component.html file. For each menu you wish to secure, you are going to create a property for each one. When the *securityObject* changes you query the security service class to see if the corresponding claim for that property is true or false. If the value is true, then the menu property is updated and the menu is displayed, and vice versa. Open **app.component.html** file and add the *ngIf directive to the two menu items.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

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

Open the **app.component.ts** file and modify the first import statement to add a couple of interfaces. Also, import the Subscription class from RxJS. This helps you setup an observer to the *securityObject* property in the security service class.

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs/Subscription';
```

Modify the class definition to add the two additional methods you need to add to use the observer pattern.

```
export class AppComponent implements OnInit, OnDestroy
```

Add three new properties; *subscription, canAccessProducts, canAccessCategories*. The *subscription* property is used as the observer to the securityObject in the security service class. The other two properties are for binding to the menus in the app.component.html file.

```
subscription: Subscription;
canAccessProducts: boolean = false;
canAccessCategories: boolean = false;
```

Add an updateProperties() method to update each of the menu properties by calling the hasClaim() method in the security service class.

```
private updateProperties() {
  this.canAccessProducts =
    this.securityService.hasClaim("canAccessProducts", "true");
  this.canAccessCategories =
    this.securityService.hasClaim("canAccessCategories", "true");
}
```

Add an ngOnInit() method and setup your observer in this method. During the subscribe you call the updateProperties() method to modify the values in those menu properties each time the *securityObject* property changes in the security service class.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

```
ngOnInit() {
  this.subscription = this.securityService.securityReset
    .subscribe(() => this.updateProperties());
}
```

When you are explicitly creating your own subscription and not using something created by Angular, you need to unsubscribe from that subscription when you are done. Add an ngOnDestroy() method to this class and call the unsubscribe() method when this component is destroyed.

```
ngOnDestroy() {
  // prevent memory leak when component is destroyed
  this.subscription.unsubscribe();
}
```

Try it Out

Save all your changes, go to the browser and try logging in and out using the different user names. Every time you login and logout, the menus should change.

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* claim.

To protect the routes based on claims, 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 -m app.module
```

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, specify the name of the claim to check that is associated with this route. Add a data property and pass in a property named *claimType* and set the value to the name of the claim associated with the route. This *data* property is passed to each Guard listed in the *canActivate* property.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

```
{
  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 claim type on security object to check
  let claimType: string = next.data["claimType"];

  // Check security claim
  return this.securityService.hasClaim(claimType, "true");
}
```

Retrieve the *claimType* to validate using the *data* property on the ActivatedRouteSnaphot object passed into this method. A true value returned from this guard means the user has the right to navigate to this route.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

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.hasClaim(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 view 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.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

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, that 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.

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

Summary

In this article you learned to setup an array of claims and perform authorization using those claims, as well as how to turn menus and buttons on and off can be done by using claim types. You also learned to secure routes using a Route Guard that looks up claim information in the array of claims for the currently logged-in user.

Sample Code

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

Copyright © 2017-18 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.