Security in Angular - Part 3

In my last two blogs, you created a set of Angular classes to support user authentication and authorization. You also built a .NET Core Web API project to authenticate a user by calling a Web API method. An authorization object was created with individual properties for each item you wished to secure in your application. In this blog, you are going to build an array of claims, and eliminate the use of single properties for each item you wish to secure. Using an array of claims is a much more flexible approach for large applications.

The Starting Application

To follow along with this article, download the accompanying ZIP. After extracting the sample from the ZIP file, there is a VS Code workspace file you can use to load the two projects in this application. If you double-click on this workspace file, the solution is loaded that looks like Figure 1. There are two projects; **PTC** is the Angular application. **PtcApi** is the ASP.NET Core Web API project.

In the last post, mock data was used for products, categories, users and authorization. The starting application in this post has removed all mock data and now connects to an SQL Server database to get this data. No changes were made to the Angular application.

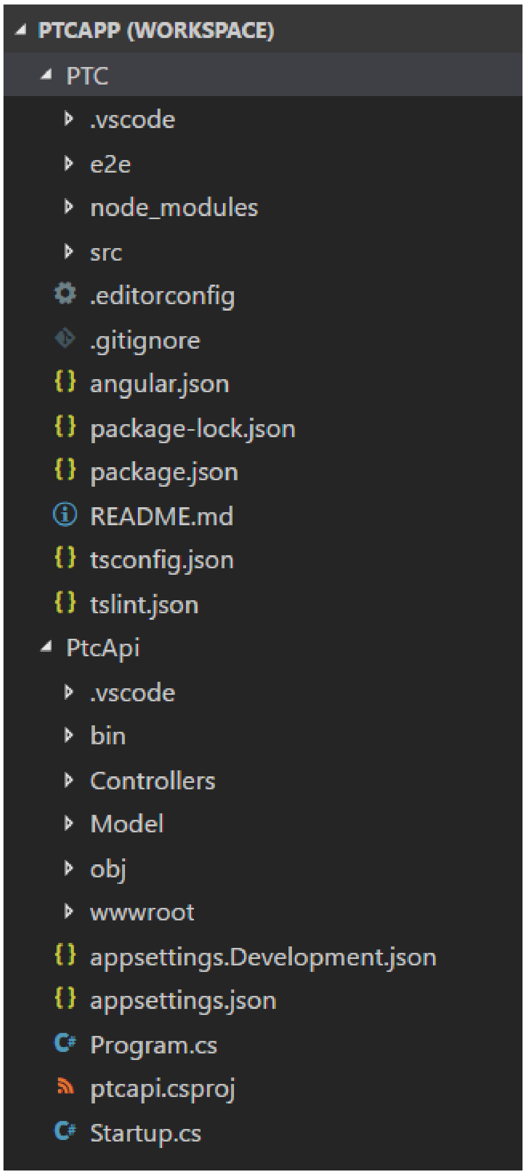

```
PTCAPP (WORKSPACE)
  PTC
    .vscode
    e2e
    node_modules
    src
    .editorconfig
    .gitignore
    angular.json
    package-lock.json
    package.json
    README.md
    tsconfig.json
    tslint.json
  PtcApi
    .vscode
    bin
    Controllers
    Model
    obj
    wwwroot
    appsettings.Development.json
    appsettings.json
    Program.cs
    ptcapi.csproj
    Startup.cs
```

Figure 1: The starting application has two projects; the Angular project (PTC) and the .NET Core Web API project (PtcApi).

The PTC Database

There is an SQL Server Express database named PTC included in the ZIP file. Open the **PtcDbContext.cs** file located in the **\PtcApi\Model** folder. Change the path in connection string constant to point to the folder in which you installed the files from this ZIP file. If you do not have SQL Server Express installed, you can use the **PTC.sql** file located in the **\SqlData** folder to create the appropriate tables in your own SQL Server instance.

Security Tables Overview

The PTC database has two tables besides the product and category tables; User and UserClaim (Figure 2). These tables are like the ones you find in the ASP.NET Identity System from Microsoft. I have simplified the structure just to keep the code small for this sample application.

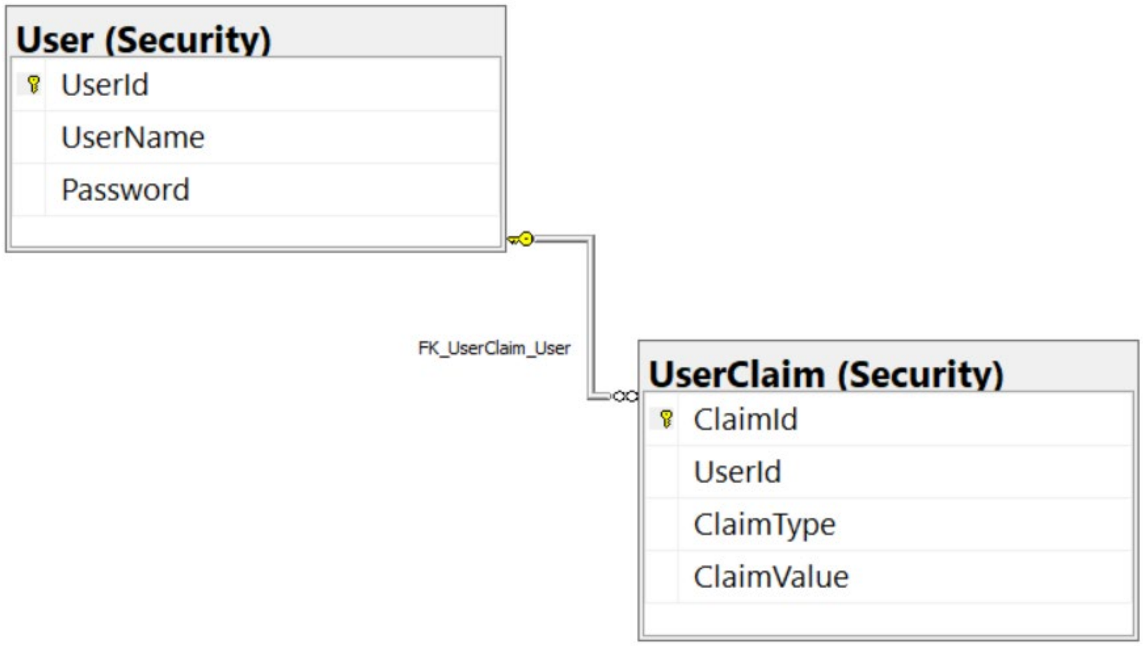
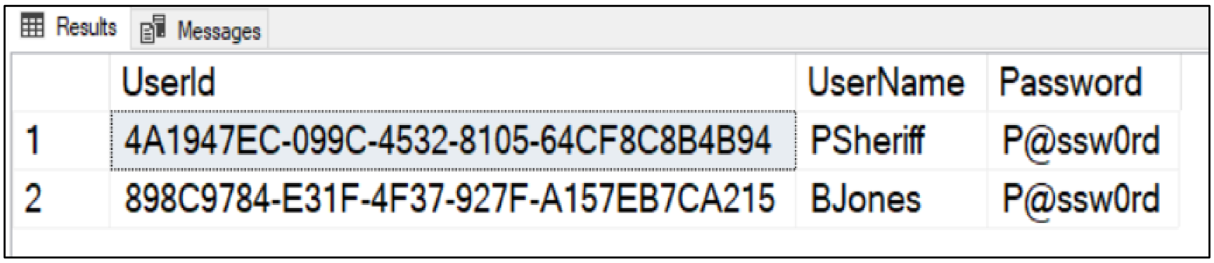


Figure 2: Two security tables are needed to authenticate and authorize a user.

User Table

The user table contains information about a specific user such as their user name, password, first name, last name, etc. For purposes of this blog, I have simplified this table to just a unique id (UserId), the name for the login (UserName) and the Password for the user.

Please note that I am using a plain-text password in the sample for this application. In a production application, this password should always be either encrypted or hashed.



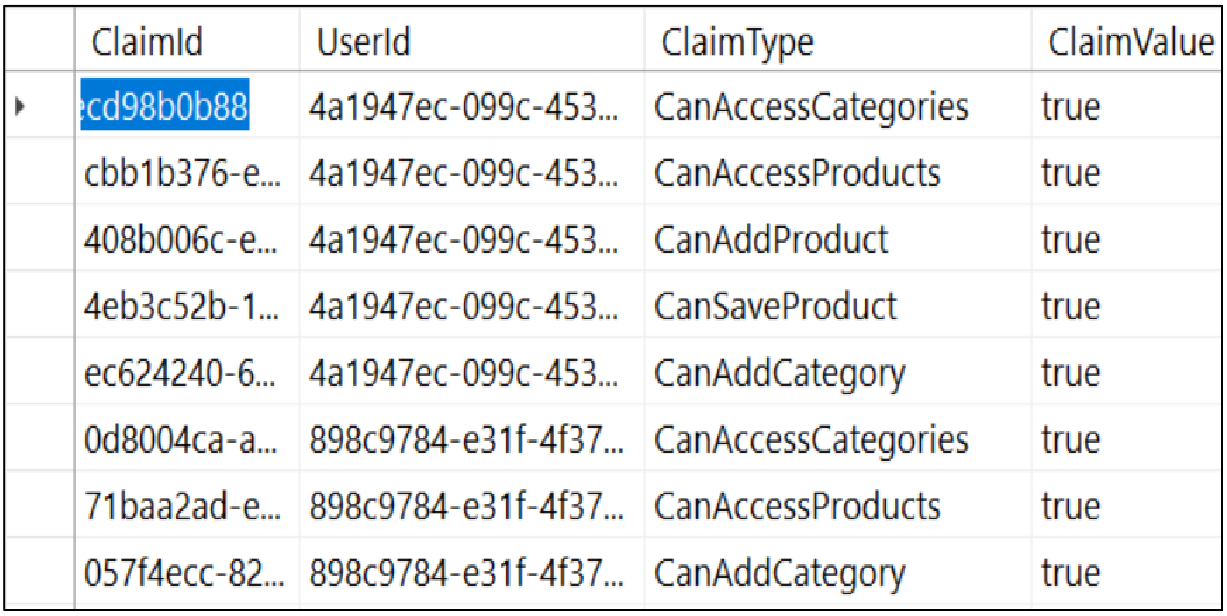
	UserId	UserName	Password
1	4A1947EC-099C-4532-8105-64CF8C8B4B94	PSheriff	P@ssw0rd
2	898C9784-E31F-4F37-927F-A157EB7CA215	BJones	P@ssw0rd

Figure 3: Example data in the User table.

User Claim Table

In the UserClaim table there are four fields; ClaimId, UserId, ClaimType and ClaimValue. The ClaimId is a unique identifier for the claim record. The UserId is a foreign key relation to the User table. The value in the ClaimType field is the one that is used in your Angular application to determine if the user has the appropriate authorization to perform some action. The value in the ClaimValue can be any value you want, I am using a "true" or "false" value for this blog.

You do not need to enter a record for a specific user and claim type if you do not wish to give the user that claim. For example, the CanAddProduct property in the authorization object may either be eliminated for the user "bjones", or you can enter a "false" value for the ClaimValue field. Later in this blog, you learn how this process works.



	ClaimId	UserId	ClaimType	ClaimValue
▸	cd98b0b88	4a1947ec-099c-453...	CanAccessCategories	true
	cbb1b376-e...	4a1947ec-099c-453...	CanAccessProducts	true
	408b006c-e...	4a1947ec-099c-453...	CanAddProduct	true
	4eb3c52b-1...	4a1947ec-099c-453...	CanSaveProduct	true
	ec624240-6...	4a1947ec-099c-453...	CanAddCategory	true
	0d8004ca-a...	898c9784-e31f-4f37...	CanAccessCategories	true
	71baa2ad-e...	898c9784-e31f-4f37...	CanAccessProducts	true
	057f4ecc-82...	898c9784-e31f-4f37...	CanAddCategory	true

Figure 4: Example data in the UserClaim table.

Modify Web API Project

In the previous blog, the **AppUserAuth** class contained a Boolean property for each claim. You tested this Boolean using an Angular *ngIf directive to remove HTML elements from the DOM, thus eliminating the ability for a user to perform some action. Now that you have database tables for your users and claims, you need to modify a few things in the Web API application to support an array of claims.

Using individual properties for each claim makes your AppUserAuth class become quite large and unmanageable when you have more than just a few claims. It also means that when you wish to add another claim, you must add a new record to your

SQL Server, add a property to the AppUserAuth class in your Web API, add a property to the AppUserAuth class in your Angular application, and add a directive to any DOM element you wish to secure.

Using an array-based approach, you only need to add a record to your SQL Server and add a directive to a DOM element you wish to secure. This means you have less code to modify, less testing to perform, and thus, your time deploy a new security change decreases.

Modify AppUserAuth Class

Open the **AppUserAuth.cs** file in the **\PtcApi\Model** folder and remove each of the individual claim properties you created in the last blog. Add a generic list of AppUserClaim objects with the property name of *Claims*. You need to add a using statement to import the **System.Collections.Generic** namespace. You should also initialize the *Claims* property to an empty list in the constructor of this class. After making these changes, the AppUserAuth class should look like Listing 1.

```
using System.Collections.Generic;

namespace PtcApi.Model
{
  public class AppUserAuth
  {
    public AppUserAuth()
    {
      UserName = "Not authorized";
      BearerToken = string.Empty;
      Claims = new List<AppUserClaim>();
    }

    public string UserName { get; set; }
    public string BearerToken { get; set; }
    public bool IsAuthenticated { get; set; }
    public List<AppUserClaim> Claims { get; set; }
  }
}
```

Listing 1: Modify the AppUserAuth class to use an array of user claims.

Modify Security Manager

The **SecurityManager.cs** file, located in the **\PtcApi\Model** folder is responsible for interacting with the Entity Framework to retrieve security information from your SQL Server tables. Open the SecurityManager.cs file and remove the for loop in the BuildUserAuthObject() method that uses reflection to set property names. The following code snippet is what the BuildUserAuthObject() method should look like after you have made these changes.

```
protected AppUserAuth BuildUserAuthObject(AppUser authUser)
{
  AppUserAuth ret = new AppUserAuth();

  // Set User Properties
  ret.UserName = authUser.UserName;
  ret.IsAuthenticated = true;
  ret.BearerToken = BuildJwtToken(ret);

  // Get all claims for this user
  ret.Claims = GetUserClaims(authUser);

  return ret;
}
```

You also need to locate the BuildJWTToken() method and remove the individual properties being set. Each line where these properties are being set should be presenting a syntax error in Visual Studio code because those properties no longer exist.

Modify the Angular Application

As is frequently the case with Angular applications, if you make changes in the Web API project, you need to make changes in the Angular application as well. Let's make those changes now.

Add a AppUserClaim Class

Since you are now going to be returning an array of AppUserClaim objects from the Web API, you need a class named **AppUserClaim** in your Angular application. 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: string = "";
  userId: string = "";
  claimType: string = "";
  claimValue: string = "";
}
```

Modify the AppUserAuth Class

Open the **app-user-auth.ts** file and remove all the individual Boolean claim properties. Just like you removed them from the Web API class, you need to remove them from your Angular application as well. Next, add an array of AppUserClaim objects to this class as shown in the following code snippet.

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

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

Modify Security Service

Open the **security.service.ts** file located in the **\src\app\security** folder. Locate the resetSecurityObject() method and remove the individual Boolean properties. Add a line of code to reset the claims array to an empty array of claims as shown in the following code snippet.

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

  this.securityObject.claims = [];

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

Claim Validation

Now that you have made code changes on both the server and client-side, the Web API call returns the authorization class with an array of user claims. You now need to be able to check if a user has a valid claim (authorization) to perform an action, or to remove an HTML element from the DOM. You are eventually going to create a custom structural directive that you can use on a menu as shown here.

```
<a routerLink="/products"
   *hasClaim="'canAccessProducts'">Products</a>
```

To be able to do this, you need a method that takes the string passed to the *hasClaim directive and verifies that this claim exists in the array downloaded from the Web API. This method should also able to check for a claim value set with this claim type. Remember the ClaimValue field in the SQL Server UserClaim table is of the type string. You can place any value you want into this field. This means you also want to be able to pass in a value to check as shown here.

```
<a routerLink="/products"
   *hasClaim="'canAccessProducts:false'">Products</a>
```

Notice the use of a colon, then the value you want to check for this claim. This string containing the claim type, a colon, and the claim value is passed to the hasClaim directive. The new method you are going to create should be able to parse this string and determine the claim type and the value (if any). Add this new method to the **SecurityService** class, and give it the name isClaimValid() as shown in Listing 2.

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

  // 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].toLowerCase();
      claimValue = words[1];
    }
    else {
      claimType = claimType.toLowerCase();
      // Either get the claim value, or assume 'true'
      claimValue = claimValue ? claimValue : "true";
    }
    // Attempt to find the claim
    ret = auth.claims.find(
      c => c.claimType.toLowerCase() == claimType
           && c.claimValue == claimValue) != null;
  }

  return ret;
}
```

Listing 2: Check for a claim type and optionally a claim value in the isClaimValid() method.

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

```
hasClaim(claimType: any) : boolean {
  return this.isClaimValid(claimType);
}
```

Create Structural Directive to Check Claim

To add your own structural directive, hasClaim, open a terminal window in VS Code and type in the following Angular CLI command. This command adds a new directive into the **\security** folder.

```
ng g d security/hasClaim --flat
```

Open the newly created **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 in the @Directive function to read **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) { }
```

Just like when you bind properties from one element to another, you need to use the Input class to tell Angular to pass the data on the right-hand side of the equal sign in the directive to the hasClaim property in your directive class. Add the following code below the constructor.

```
@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 on the right-hand side of the equals sign to the 'set' property named hasClaim(). The parameter to the hasClaim property is named *claimType*. Pass this parameter to the new hasClaim() method you created in the SecurityService class. If this method returns a true, which means the claim exists, the UI element to which this directive is applied is displayed on the screen using the createEmbeddedView() method. If the claim does not exist, the UI element is removed by calling the clear() method on the viewContainer.

Modify Authorization Guard

Just because you remove a menu item does not mean that the user cannot directly navigate to the path pointed to by the menu. In the first blog, you created an Angular guard to stop a user from directly navigating to a route if they did not have the appropriate claim. As you now verify claims using an array instead of Boolean properties, you need to modify the authorization guard you created. Open the **auth.guard.ts** file, locate the canActivate() method, and change the if statement to look like the code shown below.

```
if (this.securityService.securityObject.isAuthenticated
    && this.securityService.hasClaim(claimName)) {
  return true;
}
```

Secure Menus

You are just about ready to try out all the changes you made. If you look at the Products and Categories menu items in the **app.component.html** file you see that you are using an *ngIf directive to only display menu items if the securityObject property is not null, and that the Boolean property is set to a true value.

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

Since the *ngIf directive is bound to the securityObject using two-way data-binding if this property changes, the menus are redrawn. The structural directive you just created, however, passes in a string to a 'set' property that executes code, so there is no binding to an actual property. This means that the menus are not be redrawn if you add the *hasClaim structural as shown previously. Another problem is you can't have two directives on a single HTML element. Not to worry, you may wrap the two anchor tags within an **ng-container** and use the *ngIf directive on those to bind to the isAuthenticated property of the securityObject. This property changes once a user logs in, so this allows you to control the visibility of the menus. Then you may use the *hasClaim on the anchor tags to control the visibility based on if the user's claim is valid. Open the **app.component.html** file and modify the two menu items as shown below.

```
<li>
  <ng-container *ngIf="securityObject.isAuthenticated">
    <a routerLink="/products"
       *hasClaim="'canAccessProducts'">Products</a>
  </ng-container>
</li>
<li>
  <ng-container *ngIf="securityObject.isAuthenticated">
    <a routerLink="/categories"
       *hasClaim="'canAccessCategories'">Categories</a>
  </ng-container>
</li>
```

Try it Out

You are finally ready to try out all your changes and verify your menu items are turned off and on based on the user being authenticated, and they have the appropriate claims in the UserClaim table. Save all the changes you have made in VS Code. Start the Web API and Angular projects and view the browser. The Products and Categories menus should not be visible. Click on the Login menu and login using a user name of "psheriff" and a password of 'P@ssw0rd'. You should now see both menus appear.

Open the User table, locate the "bjones" user and remember the UserId for this user. Open the UserClaim table, locate the CanAccessCategories record for "bjones", and change the value from a *true* to a *false* value. Back in the browser, logout as "psheriff", and log back in as "bjones". You should see the Products menu, but the Categories menu does not appear. Go back to the UserClaim table and set the CanAccessCategories claim value field back to a true for "bjones".

Secure Add New Product Button

Add the *hasClaim directive to the "Add New Product" button located on the **product-list.component.html** file. Remove the *ngIf directive that was bound to the old canAddProduct property and use your new structural directive 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 as "psheriff" and login as "bjones". The "Add New Product" button should now be gone.

Remember you added the capability to specify the claim value after the name of the claim. Add a colon after the claim type, then add 'false' to the Add New Product button as shown below.

```
<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.

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 users that have one claim type and other users that have another claim type. 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 a single string value, or an array is passed in. Open the **security.service.ts** file and modify the hasClaim() method to look like Listing 3.

```
hasClaim(claimType: any) : boolean {
  let ret: boolean = false;

  // See if an array of values was passed in.
  if (typeof claimType === "string") {
    ret = this.isClaimValid(claimType);
  }
  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;
}
```

Listing 3: Add the ability to pass multiple claim types to the hasClaim directive.

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.

Secure Other Buttons

Open the **product-list.component.html** file and modify the Add New Product button to use an array as shown in the following code snippet.

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

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 may view the "Add New Product" button.

Summary

In this final blog on Angular security you learned to build a security system that is more appropriate for enterprise type applications. Instead of individual properties for each item you wish to secure, you return an array of claims from your Web API call. You built a custom structural directive to which you may pass one or more claims. This directive takes care of including or removing an HTML element based on the users set of claims. This approach makes your code more flexible and requires less coding changes should you wish to add or delete claims.

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 3" from the drop-down.