Security in Angular - Part 2

In Part 1 of this article series, you created a set Angular classes for users and user authentication/authorization. You used these classes to login a user, create a set of properties in a class to turn menus and buttons on and off. In this article you learn to authenticate users against a Web API method. That method returns an authorization object with the same properties as the classes you created in Angular. You are also going to learn to secure your Web API methods using JSON Web Tokens (JWT). You use the [Authorize] attribute to secure your methods, and you learn to add security policies too.

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 article, you did everything client-side. For the starting application in this article, I pre-built an ASP.NET Core Web API project and hooked up the Product and Category pages to the appropriate controllers in this Web API project. In this article you build the security classes necessary to return user authentication and authorization information to your Angular application.

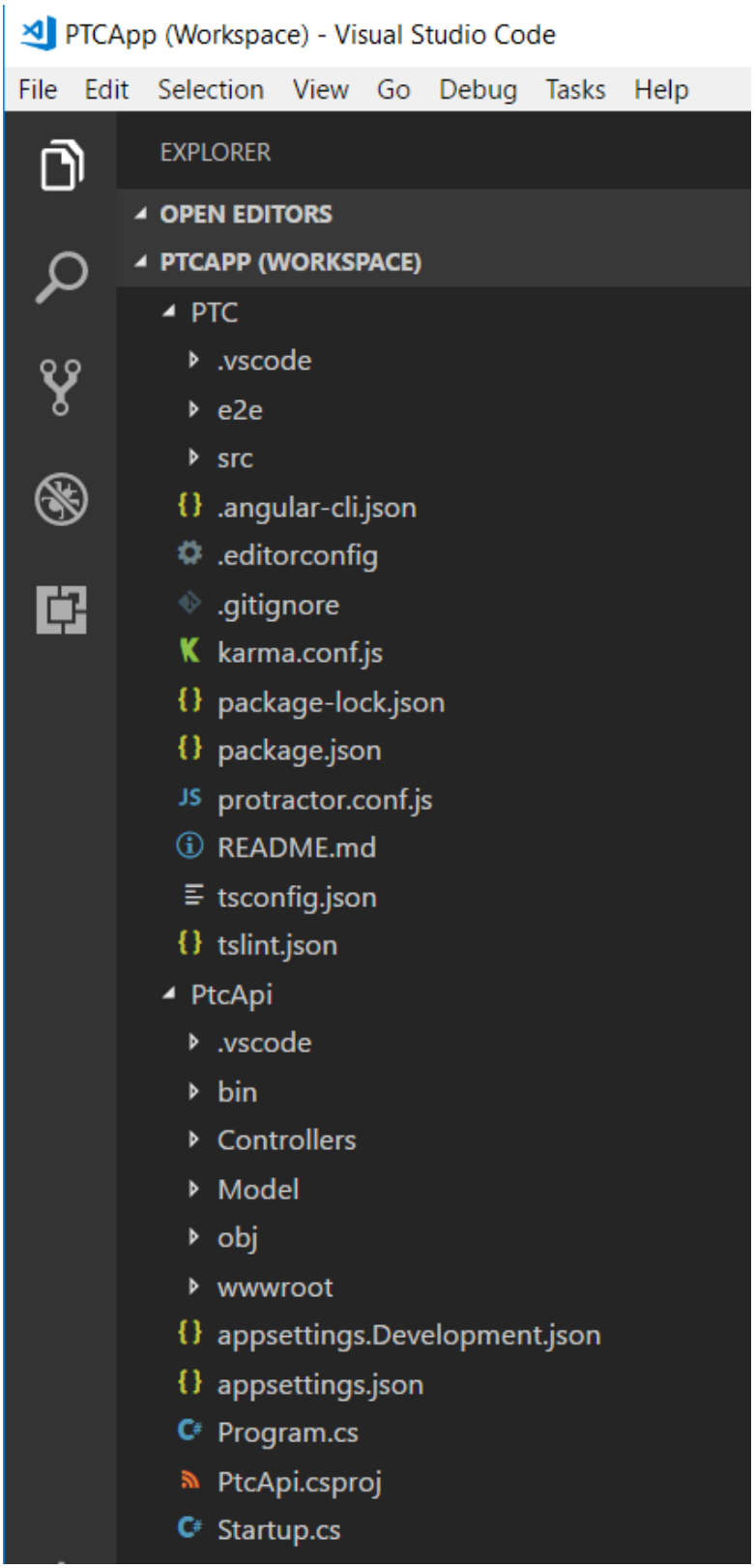

Figure 1: The starting application has two projects; the Angular project (PTC) and the .NET Core Web API project (PtcApi).

Create Application Security Classes

In Part 1 of this article series, you created Angular classes to represent a user and a user security object with properties you can bind to. You need to create these same classes as C# classes in the PtcApi project. You are going to build the C# classes with the same names and almost the same properties as those in TypeScript.

AppUser Class

The AppUser class is a simplified version of a user class with just *UserName* and *Password* properties. In this article, I am keeping the classes simple and will not be using a database, I am just going to use mock data, just like what I did in Part 1 of this article series. This helps you focus on how things work, and not have to worry about a lot of configuration. Don't worry, you are going to use real SQL tables in the article. Right mouse-click on the **Model** folder and add a new file named **AppUser.cs**. Add the following code into this file.

```
namespace PtcApi.Model
{
  public class AppUser
  {
    public int UserId { get; set; }
    public string UserName { get; set; }
    public string Password { get; set; }
  }
}
```

Create AppUserAuth Class

The AppUserAuth class has properties to represent a user security profile. Again, to keep things simple, I have listed individual properties that can be directly bound to UI elements in Angular. In a future article, I will explore how to use Roles and Claims/Permissions with a security system like the one presented here. Right mouse-click on the **Model** folder and add a new file named **AppUserAuth.cs**. Add the following code into this file.

```
namespace PtcApi.Model
{
  public class AppUserAuth
  {
    public AppUserAuth() : base()
    {
      UserName = string.Empty;
      BearerToken = string.Empty;
      IsAuthenticated = false;
      CanAccessProducts = false;
      CanAddProduct = false;
      CanSaveProduct = false;
      CanAccessCategories = false;
      CanAddCategory = false;
    }

    public string UserName { get; set; }
    public string BearerToken { get; set; }
    public bool IsAuthenticated { get; set; }
    public bool CanAccessProducts { get; set; }
    public bool CanAddProduct { get; set; }
    public bool CanSaveProduct { get; set; }
    public bool CanAccessCategories { get; set; }
    public bool CanAddCategory { get; set; }
  }
}
```

Build Security Manager Class

Instead of building the mock data and creating the user security object in a Web API controller class, place that code in another class. Right mouse-click on the **PtcApi** project and create a new folder named **Security**. Right mouse-click on the new **Security** folder and add a new file named **SecurityManager.cs**. Add the following code into this file.

```
using System;
using System.Collections.Generic;
using System.Linq;
using PtcApi.Model;

namespace PtcApi.Security
{
  public class SecurityManager
  {
    private AppUser _user = null;
    private AppUserAuth _auth = null;
  }
}
```

Add CreateMockUsers() Method

Add a method to the SecurityManager class to create a set of mock users. It is against this collection of users, you are going to check the credentials of the user submitted from the Angular application.

```
private List<AppUser> CreateMockUsers()
{
  List<AppUser> list = new List<AppUser>();

  list.Add(new AppUser()
  {
    UserId = 1,
    UserName = "PSheriff",
    Password = "P@ssw0rd"
  });
  list.Add(new AppUser()
  {
    UserId = 2,
    UserName = "Bjones",
    Password = "P@ssw0rd"
  });

  return list;
}
```

Write CreateMockSecurityObjects() Method

If you read Part 1 of this article series, the following method should look like the LOGIN_MOCKS constant created in the PTC Angular application. Add a method named CreateMockSecurityObjects() to build a list of AppUserAuth objects with different properties filled in for each user.

```
private List<AppUserAuth> CreateMockSecurityObjects()
{
  List<AppUserAuth> list = new List<AppUserAuth>();

  list.Add(new AppUserAuth()
  {
    UserName = "PSheriff",
    BearerToken = "abi393kdkd9393ikd",
    IsAuthenticated = true,
    CanAccessProducts = true,
    CanAddProduct = true,
    CanSaveProduct = true,
    CanAccessCategories = true,
    CanAddCategory = false
  });
  list.Add(new AppUserAuth()
  {
    UserName = "Bjones",
    BearerToken = "sd9f923k3kdmcjkhd",
    IsAuthenticated = true,
    CanAccessProducts = true,
    CanAddProduct = false,
    CanSaveProduct = false,
    CanAccessCategories = true,
    CanAddCategory = true
  });

  return list;
}
```

In the above code I have hard-coded a value into the *BearerToken* property. Later in this article you generate a real bearer token using JWT.

Write BuildUserAuthObject Method

When the user logs in to the Angular application, the user name and password are sent to a controller in a AppUser object. That user object is going to be passed into a ValidateUser() method you are going to write in just a minute. Write a method named BuildUserAuthObject() that is going to be called from the ValidateUser() method to build the user's security object.

This method checks to ensure that the private field variable named *_user* is not null. If it isn't, it creates the list of mock security objects and looks through the list searching for where the user name passed in matches one of the objects in the list. If it finds that object, it assigns it to the *_auth* field. If the user security object is not found, a new instance of the AppUserAuth class is created and assigned to the *_auth* field. It is the value in this *_auth* field that is returned to the Angular application.

```
protected void BuildUserAuthObject()
{
  if (_user != null)
  {
    _auth = CreateMockSecurityObjects()
      .Where(u => u.UserName ==
             _user.UserName).FirstOrDefault();
  }

  if (_auth == null)
  {
    _auth = new AppUserAuth();
  }
}
```

Add ValidateUser Method

The ValidateUser() method is the only method exposed from the SecurityManager class. The security controller you are going to create to accept the AppUser object from the Angular application calls this method and passes in that user object to validate the user.

The ValidateUser() object assigns the user passed in, into the **_user** field. The user is validated against the list of mock users to ensure that the user name and password match one of the users in the list. The BuildUserAuthObject() method is called to build a AppUserAuth object and passed back to the controller.

```
public AppUserAuth ValidateUser(AppUser user)
{
  // Assign current user
  _user = user;

  // Attempt to validate user
  _user = CreateMockUsers().Where(
    u => u.UserName.ToLower() == _user.UserName.ToLower()
    && u.Password == _user.Password).FirstOrDefault();

  // Build User Security Object
  BuildUserAuthObject();

  return _auth;
}
```

Add Security Controller

It is now time to build the security controller the Angular application calls. Angular passes the user credentials in an AppUser object. The security controller returns an AppUserAuth object with the appropriate properties set. Right mouse-click on the

Controllers folder and add a new file named **SecurityController.cs**. Add the following code into this file.

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using PtcApi.Security;
using PtcApi.Model;

namespace PtcApi.Controllers
{
  [Route("api/[controller]")]
  public class SecurityController : Controller
  {

  }
}
```

Within this new controller class, add a Login() method that accepts an AppUser object. Create an instance of the SecurityManager and pass that AppUser object to the ValidateUser() method. If the user comes back authenticated, the IsAuthenticated property is a true value. Set the *ret* variable to an IActionResult generated by the StatusCode() method. A status code of 200 is passed back to the Angular application with the payload of the user security object. If the user is not authenticated, pass back a status code of 404.

```
[HttpPost("login")]
public IActionResult Login([FromBody]AppUser user)
{
  IActionResult ret = null;
  AppUserAuth auth = new AppUserAuth();
  SecurityManager mgr = new SecurityManager();

  auth = (AppUserAuth)mgr.ValidateUser(user);
  if (auth.IsAuthenticated)
  {
    ret = StatusCode(StatusCodes.Status200OK, auth);
  }
  else
  {
    ret = StatusCode(StatusCodes.Status404NotFound,
                     "Invalid User Name/Password.");
  }

  return ret;
}
```

Call Web API

To call the SecurityController's Login() method, use the HttpClient service in Angular. Go back to the **PTC** project and open the **security.service.ts** file located in the **\src\app\security** folder and add the following import statements.

```
import {HttpClient, HttpHeaders} from '@angular/common/http';
import { tap } from 'rxjs/operators';
```

Add two constants just under the import statements. The first constant is the location of the Web API controller. The second constant holds header options required by the HttpClient when POSTing data to a Web API call.

```
const API_URL = "http://localhost:5000/api/security/";

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json'
  })
};
```

To use the HttpClient service in your security service class, modify the constructor to tell Angular to inject the HttpClient. Dependency injection is used extensively throughout Angular to provide services such as HttpClient to different components. The HttpClientModule has already been imported in the AppModule class. This module must be imported to allow components to use the HttpClient service.

```
constructor(private http: HttpClient) { }
```

Remove the hard-coded logic in the login() method, and make a Web API call. Modify the login() method to call the SecurityController login method. If valid data is returned from this call, the pipe() method is used to tap into the pipeline. You can retrieve the data returned and assign that data to the *securityObject* property.

It is very important you don't use the equal sign to assign to the *securityObject* property. If you do, bound references to the *securityObject* are wiped out and your menus and other UI elements bound to properties on the *securityObject* no longer work. Instead, use the Object.assign() method to copy the data from one object to another.

This method also takes the bearer token and stores it into local storage. The reason for this will become apparent later in this article when you learn about building HTTP Interceptor class.

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

  return this.http.post<AppUserAuth>(API_URL + "login",
    entity, httpOptions).pipe(
    tap(resp => {
      // 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, resp);
      // Store into local storage
      localStorage.setItem("bearerToken",
          this.securityObject.bearerToken);
    }));
}
```

Modify Login Component

Since it is possible the Web API may return a **404: Not Found** error, be sure to handle this status code so you can display the *Invalid User Name/Password* error message. Open the **login.component.ts** file and add the error block in the subscribe() method. The login HTML contains a Bootstrap alert which has the words "Invalid User Name/Password." in a <p> tag. This alert only shows up if the *securityObject* is not null and the *isAuthenticated* property is set to false.

```
login() {
  this.securityService.login(this.user)
    .subscribe(resp => {
      this.securityObject = resp;
      if (this.returnUrl) {
        this.router.navigateByUrl(this.returnUrl);
      }
    },
    () => {
      // Initialize security object to display error message
      this.securityObject = new AppUserAuth();
    });
}
```

Try it Out

Save all your changes and start the debugging of the Web API project. Start the Angular application by typing in **npm start** in a terminal window. Go to the browser and attempt to login using either "psheriff" or "bjones" with a password of "**P@ssw0rd**". If you have done everything correctly, the Web API is now being called for authenticating the user name and password. Try logging in with a bogus login id and password to make sure the error handling is working.

Authorizing Access to Web API Call

Now that you have created these Web API calls, you need to ensure that only those people authorized to call your APIs can do so. In .NET you can use the **[Authorize]** attribute to secure a controller class, or individual methods within the controller class. However, there must be some authentication / authorization component in the .NET runtime to provide data to this attribute. The [Authorize] attribute must be able to read that data to decide if the user has permissions to call the method.

There are many different authentication/authorization components you can use such as Microsoft Identity, OAuth, and JSON Web Tokens (JWT) just to mention a few. In this article, you are going to use JWT.

Secure Product Get() Method

To show you what happens when you apply the Authorize attribute to a method, open the **ProductController.cs** file and add the [Authorize] attribute to the Get() method.

```
[HttpGet]
[Authorize]
public IActionResult Get()
{
   // REST OF THE CODE
}
```

Try it Out

Save your change and run the PtcApi project. Login as "psheriff" and click on the Products menu. No product data is displayed because you attempted to call a method that was secured with the [Authorize] attribute. Press the F12 key to bring up the developer tools and you should see something that looks like Figure 2.

Notice that you are getting status code of 500 instead of a 401 (Unauthorized) or 403 (Forbidden). The reason is you have not registered an authentication service with .NET core. Microsoft Identity, OAuth, or JWT must be registered with .NET core to return a 401 instead of a 500.

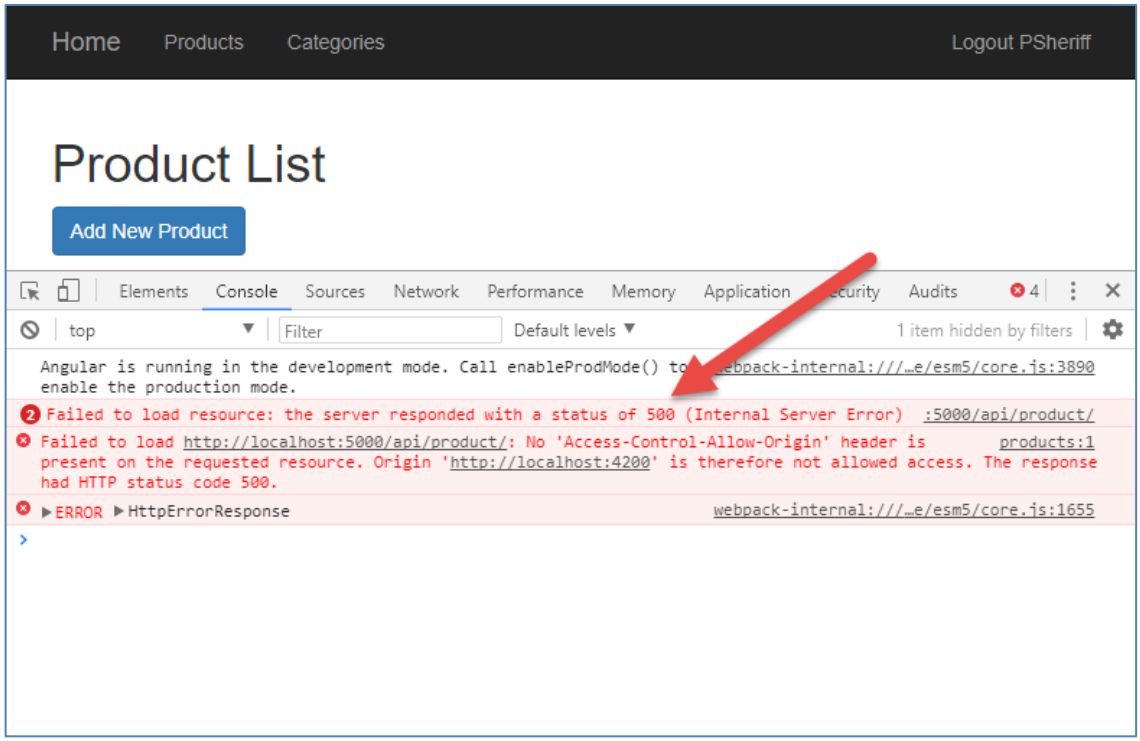

Figure 2: You receive a status code of 500 when you use the Authorize attribute without registering an authentication service.

Add JWT Configuration

To use the JSON Web Token system, there are a few steps you must perform.

- Add JWT package.
- Add JWT bearer token checking package.
- Store default JWT settings in configuration file.
- Register JWT as the authentication service.
- Add bearer token options to validate incoming token.
- Build JWT token and add to user security object.

Add JWT Package

The first you must do in your Web API project is to add some packages to use the JSON Web Token system. Open a terminal window in your **PtcApi** project and enter the following command.

```
dotnet add package System.IdentityModel.Tokens.Jwt
```

Add Bearer Token Check

In addition to the Jwt package, add a package to ensure the bearer token is passed in from the client. Add this package to your PtcApi project using the following command in your terminal window.

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Add JWT Settings to Configuration

There are a few things that are needed for a JwtToken to be issued.

- A secret key used for hashing data sent to the client.
- The name of the issuer of the token.
- The intended audience of the token.
- How many minutes to allow the token to be valid.

Store JWT Information in appsettings.json

You are going to need all the above items in two places in your code; once when you configure JWT in the .NET Core Startup class, and once when you generate a new token specifically for a user. You do not want to hard-code data into two places, so use the .NET Core configuration system to retrieve this data from the **appsettings.json** file located in the root folder of the PtcApi project.

Open the **appsettings.json** file and add a new entry named *JwtSettings*. Add the following properties and values into this JwtSettings object.

```
{
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  },
  "JwtSettings": {
    "key": "This*Is&A!Long)Key(For%Creating@A$SymmetricKey",
    "issuer": "http://localhost:5000",
    "audience": "PTCUsers",
    "minutesToExpiration": "10"
  }
}
```

Add JwtSettings Class

You can just use the configuration system in .NET core to retrieve this data, but I prefer using a class. Right mouse-click on the **Security** folder and add a new file named **JwtSettings.cs** and add the following code.

```
public class JwtSettings {
  public string Key { get; set; }
  public string Issuer { get; set; }
  public string Audience { get; set; }
  public int MinutesToExpiration { get; set; }
}
```

Read the Settings

Open the **Startup.cs** file and add a new method to read these settings using the .NET Core configuration object and create a new instance of the JwtSettings class.

```
public JwtSettings GetJwtSettings() {
  JwtSettings settings = new JwtSettings();

  settings.Key = Configuration["JwtSettings:key"];
  settings.Audience = Configuration["JwtSettings:audience"];
  settings.Issuer = Configuration["JwtSettings:issuer"];
  settings.MinutesToExpiration =
   Convert.ToInt32(
      Configuration["JwtSettings:minutesToExpiration"]);

  return settings;
}
```

Create a Singleton for the JWT Settings

Modify the ConfigureServices() method to create an instance of the JwtSettings class and call the GetJwtSettings() method. When this object is created, add it as a Singleton to the .NET core services so you can inject this into any controller.

```
public void ConfigureServices(IServiceCollection services)
{
  // Get JWT Token Settings from JwtSettings.json file
  JwtSettings settings;
  settings = GetJwtSettings();
  services.AddSingleton<JwtSettings>(settings);

  // REST OF THE CODE HERE
}
```

Register JWT as the Authentication Provider

Just below this code is where you are going to register JWT as an Authentication provider. Configure the settings of JWT with the security key, issuer, audience and minutes to expiration. This is the object that ensures the bearer token passed in from Angular is valid.

```
public void ConfigureServices(IServiceCollection services)
{
  // Get JWT Token Settings from JwtSettings.json file
  JwtSettings settings;
  settings = GetJwtSettings();
  services.AddSingleton<JwtSettings>(settings);

  // Register Jwt as the Authentication service
  services.AddAuthentication(options =>
  {
    options.DefaultAuthenticateScheme = "JwtBearer";
    options.DefaultChallengeScheme = "JwtBearer";
  })
  .AddJwtBearer("JwtBearer", jwtBearerOptions =>
  {
    jwtBearerOptions.TokenValidationParameters =
        new TokenValidationParameters
    {
      ValidateIssuerSigningKey = true,
      IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.UTF8.GetBytes(settings.Key)),
      ValidateIssuer = true,
      ValidIssuer = settings.Issuer,

      ValidateAudience = true,
      ValidAudience = settings.Audience,

      ValidateLifetime = true,
      ClockSkew = TimeSpan.FromMinutes(
                  settings.MinutesToExpiration)
    };
  });

  // REST OF THE CODE HERE
}
```

The last thing to do in the Startup class is modify the Configure() method and tell it to use authentication.

```
public void Configure(IApplicationBuilder app,
                      IHostingEnvironment env)
{
   // REST OF THE CODE HERE

   app.UseAuthentication();

   app.UseMvc();
}
```

Inject Settings into SecurityController

You want .NET Core to inject the JwtSettings class into the SecurityController so you can have access to the JWT Token settings. Open the **SecurityController.cs** file and add a constructor that accepts an instance of the JwtSettings class. Store this settings object into a field named ***_settings***. Pass this **_settings** field to the SecurityManager constructor.

```
public class SecurityController : Controller
{
  private JwtSettings _settings;
  public SecurityController(JwtSettings settings)
  {
    _settings = settings;
  }

  [HttpPost("login")]
  public IActionResult Login([FromBody]AppUser user)
  {
    IActionResult ret = null;
    AppUserAuth auth = new AppUserAuth();
    SecurityManager mgr = new SecurityManager(_settings);

    auth = (AppUserAuth)mgr.ValidateUser(user);

  // REST OF THE CODE HERE
}
```

Accept Settings in SecurityManager

In the code above, you passed in an instance of the JwtSettings class to the SecurityManager class. Open the **SecurityManager.cs** file and add code to accept this JwtSettings instance. The code is going to look very similar to the constructor and the private field you added in the SecurityController.

```
private JwtSettings _settings = new JwtSettings();
public SecurityManager(JwtSettings settings)
{
  _settings = settings;
}
```

Build a JSON Web Token

You are finally ready to build a bearer token you can add to the *BearerToken* property in the user security object. Add a new method to the SecurityManager class named BuildJwtToken(). Generate a Symmetric security key from the same *Key* property you used in the Startup class. Two claims that are needed by any JWT token, *Sub* and *Jti*, should be created. Into the *Sub* claim put the user name, and into the *Jti* claim create a unique identifier for this user. Generating a Guid is a good identifier for this field.

Next, you add a series of custom claims. In this simple example, create one claim for each property in the AppUserAuth class. I'm sure you can see where you could add claims by looping through a collection of Roles and/or Claims/Permissions from a table. However, I wanted to keep this simple for this article. Notice I use ToLower() to convert the "True" and "False" values to lower case. This keeps things consistent with how JavaScript/TypeScript likes to express true and false.

Create a new JwtSecurityToken object and set the same properties you did in the Startup class using the same values from the JwtSettings class. If you don't use the same values, then they won't be able to be verified when the token is passed in from Angular. Use the WriteToken() method of the JwtSecurityTokenHandler class to base64 encode the resulting string. It is this base64 encoded string that is placed into the *BearerToken* property in your user security object passed back to Angular.

```
protected string BuildJwtToken()
{
  SymmetricSecurityKey key = new SymmetricSecurityKey(
    Encoding.UTF8.GetBytes(_settings.Key));

  // Create standard JWT claims
  List<Claim> jwtClaims = new List<Claim>();
  jwtClaims.Add(new Claim(JwtRegisteredClaimNames.Sub,
      _user.UserName));
  jwtClaims.Add(new Claim(JwtRegisteredClaimNames.Jti,
      Guid.NewGuid().ToString()));

  // Add custom claims
  jwtClaims.Add(new Claim("isAuthenticated",
     _auth.IsAuthenticated.ToString().ToLower()));
  jwtClaims.Add(new Claim("canAccessProducts",
     _auth.CanAccessProducts.ToString().ToLower()));
  jwtClaims.Add(new Claim("canAddProduct",
     _auth.CanAddProduct.ToString().ToLower()));
  jwtClaims.Add(new Claim("canSaveProduct",
     _auth.CanSaveProduct.ToString().ToLower()));
  jwtClaims.Add(new Claim("canAccessCategories",
     _auth.CanAccessCategories.ToString().ToLower()));
  jwtClaims.Add(new Claim("canAddCategory",
     _auth.CanAddCategory.ToString().ToLower()));

  // Create the JwtSecurityToken object
  var token = new JwtSecurityToken(
    issuer: _settings.Issuer,
    audience: _settings.Audience,
    claims: jwtClaims,
    notBefore: DateTime.UtcNow,
    expires: DateTime.UtcNow.AddMinutes(
       _settings.MinutesToExpiration),
    signingCredentials: new SigningCredentials(key,
                SecurityAlgorithms.HmacSha256)
  );

  // Create a string representation of the Jwt token
  return new JwtSecurityTokenHandler().WriteToken(token); ;
}
```

Call the BuildJwtToken() Method

Locate the BuildUserAuthObject() method and add code to call this BuildJwtToken() method and assign the string result to the *BearerToken* property.

```
protected void BuildUserAuthObject()
{
  if (_user != null)
  {
    _auth = CreateMockSecurityObjects()
      .Where(u => u.UserName ==
             _user.UserName).FirstOrDefault();
  }

  if (_auth == null)
  {
    _auth = new AppUserAuth();
  }
  else {
    // Create JWT Bearer Token
    _auth.BearerToken = BuildJwtToken();
  }
}
```

Try it Out

Save all your changes and start debugging on the PtcApi project. If you are logged in to the Angular application, click the Logout menu. Enter either "psheriff" or "bjones" and a password of "**P@ssw0rd**" and you should see a screen that looks similar to Figure 3.

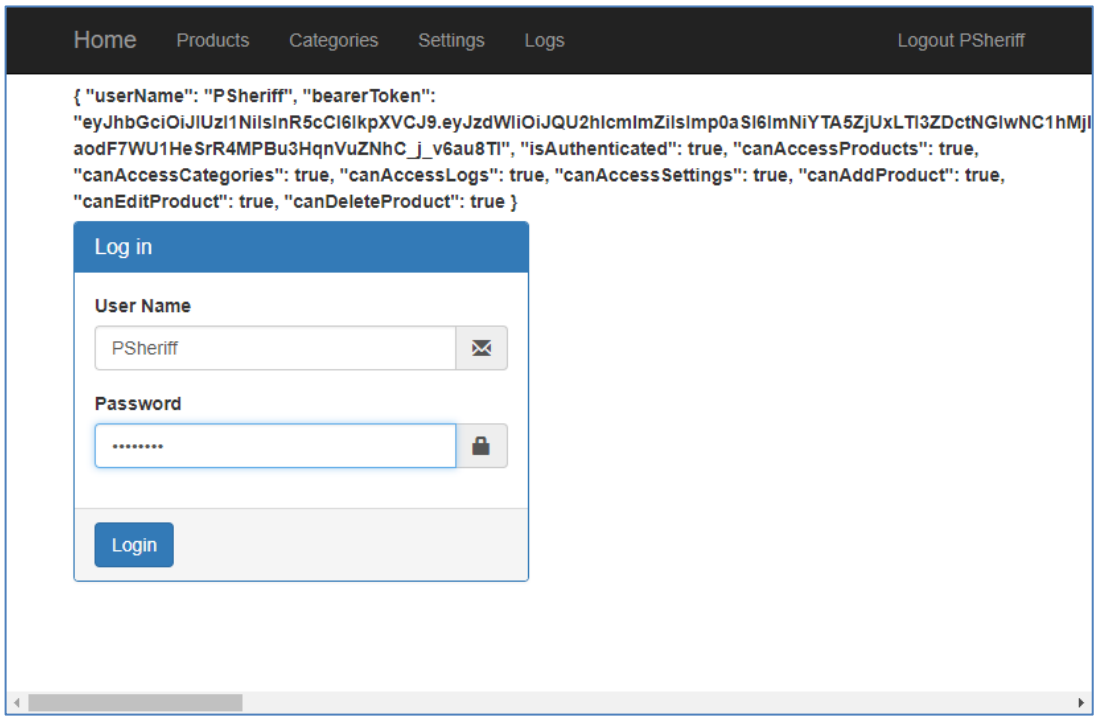

Figure 3: Decode the bearerToken at the jwt.io website.

If you wish to see what makes up the *bearerToken* property, you can decode the value at the www.jwt.io website. Copy the bearerToken on the login page to the clipboard. Open a browser window and go to www.jwt.io. Scroll down on the page until you see a box labeled "Encoded". Delete what is in there and paste in your bearer token. You should immediately see the payload data with all your data as shown in Figure 4.

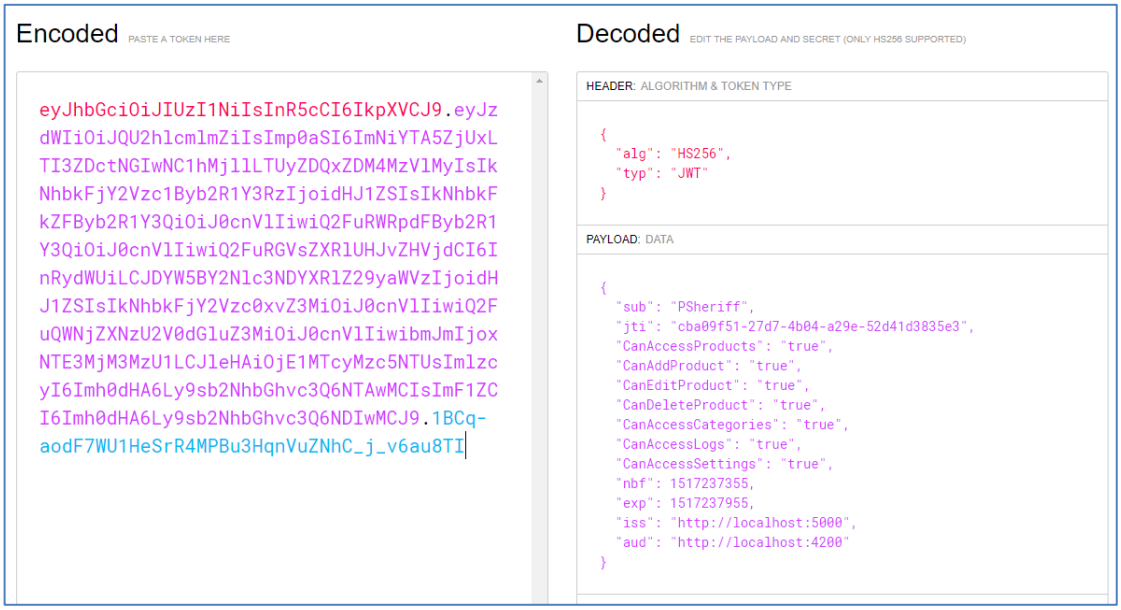

Figure 4: Decode your bearer token at www.jwt.io.

Try it Out

Click on the Products menu and you should now see a generic 401 Unauthorized message in the developer tools console window that looks like Figure 5. The reason for this error is the server does not know that you are the same person that just logged in. You must pass back the bearer token on each Web API call to prove to the server that you have permission to call the API.

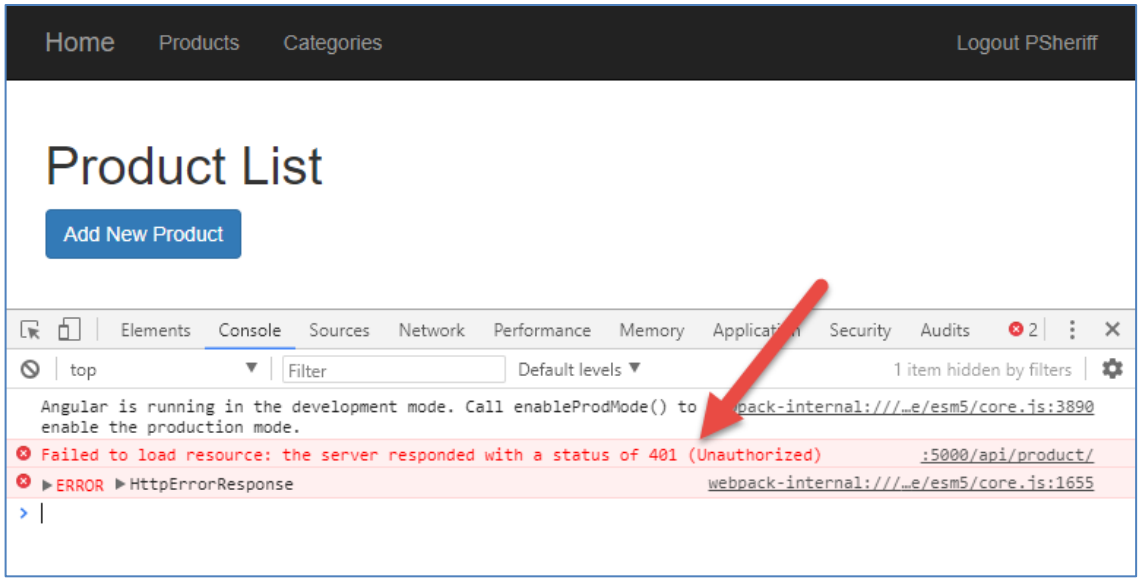

Figure 5: Unless you pass the bearer token back to the server, you will get a 401 error from any secured Web API method.

Add Headers to Product Service

To avoid the status code 401, you must pass the bearer token on each Web API call from your Angular code. You are going to learn how to automatically add the token to every call, but first, just modify the Get() method in the product service class. Open the **product.service.ts** file and modify the constructor to inject the SecurityService.

```
constructor(private http: HttpClient,
  private securityService: SecurityService) { }
```

Add code in the getProducts() method to create a new HttpHeaders object. Once you have instantiated this object, call the set() method and pass in "Authorization" as the header name. The data to pass in this header is the word 'Bearer ' followed by a space, then the bearer token itself. Add a second parameter to the HttpClient's get() method and pass an object with a property named *headers* and the HttpHeaders object you created as the value for that property.

```
getProducts(): Observable<Product[]> {
  let httpOptions = new HttpHeaders()
  .set('Authorization', 'Bearer ' +
      this.securityService.securityObject.bearerToken);

  return this.http.get<Product[]>(API_URL,
                                  { headers: httpOptions });
}
```

Try it Out

Save your changes, go to the browser and login as "psheriff", click on the Products menu and you should see the product data displayed. This is because the server has authenticated your token and now knows who you are. Thus, you are granted access to the Get() method in the ProductController.

HTTP Interceptor

Instead of having to add the same headers in front of every Web API call, create an HTTP Interceptor class to place custom headers into each Web API call in Angular. Right mouse-click on the **\security** folder and add a file named **http-interceptor.module.ts**. I am not going to explain this code as it is well documented at http://bit.ly/2GYt1H3, and you should find many blog posts about this on the internet.

The one thing I do want to point out is that you are retrieving the bearer token from local storage. You might be wondering why you don't inject a SecurityService object into this class and retrieve it from the SecurityService object like in all the other components. You can't have an HTTP interceptor injected with the SecurityService since the security service class also uses the HttpClient. This causes a recursion error.

If you remember from earlier you added code in the login() method of the SecurityService class to store the bearer token into local storage. Now you know the reason for that code.

```
import { Injectable, NgModule } from '@angular/core';
import { Observable } from 'rxjs';
import { HttpEvent, HttpInterceptor, HttpHandler,
         HttpRequest } from '@angular/common/http';
import { HTTP_INTERCEPTORS } from '@angular/common/http';

@Injectable()
export class HttpRequestInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
    var token = localStorage.getItem("bearerToken");

    if(token) {
      const newReq = req.clone(
        {
           headers: req.headers.set('Authorization',
                    'Bearer ' + token)
        });

        return next.handle(newReq);
    }
    else {
      return next.handle(req);
    }
  }
};

@NgModule({
  providers: [
    { provide: HTTP_INTERCEPTORS,
      useClass: HttpRequestInterceptor,
      multi: true }
  ]
})
export class HttpInterceptorModule { }
```

In order to use this HTTP Interceptor class, register it with your AppModule class. Open the **app.module.ts** file and add this new module to the *imports* property.

```
imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule,
  AppRoutingModule,
  HttpInterceptorModule
],
```

Open the **product.service.ts** file and change the getProducts() method back to what you had originally. Remove the code that creates the HttpHeaders, and passing this as a second parameter to the get() method. The getProducts() method should now look like the following.

```
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(API_URL);
}
```

Try it Out

Save your changes, go to your browser, login as "psheriff" and try accessing the Products menu. You should still be getting data from the product controller. This verifies that the HTTP Interceptor class is working.

Add Security Policy

The [Authorize] attribute just ensures that a user is authenticated. However, for some Web API methods, you may wish to restrict access based on roles or claims/permissions. This is accomplished by adding Authorization to the services of the .NET Core Web API project.

Open the **Startup.cs** file and add the following code in the ConfigureServices method, just under the code you added earlier for adding authentication. For each property you have in your AppUserAuth object you can add policy objects and specify the value the user must have.

```
services.AddAuthorization(cfg =>
{
   // NOTE: The claim key and value are case-sensitive
   cfg.AddPolicy("CanAccessProducts", p =>
      p.RequireClaim("CanAccessProducts", "true"));
});
```

Once you create your claims, you may add the Policy property to the Authorize attribute to check for any of the claim names. For example, on the ProductController.Get() method you can add the following code to the Authorize attribute to restrict usage on this method to only those users whose *CanAccessProducts* property is set to true.

```
[Authorize(Policy = "CanAccessProducts")]
```

Try it Out

Open the **SecurityManager.cs** file in the PtcApi project and locate the CreateMockSecurityObjects() method. Modify the claim "*CanAccessProducts*" to be a "false" value for the "PSheriff" user.

Open the **app-routing.module.ts** file and remove the route guard for the 'products' path.

```
{
  path: 'products',
  component: ProductListComponent
},
```

Save your changes and start the debugging on the PtcApi project. Go back to the browser, login as "psheriff" and type in http://localhost:4200/products directly into the address bar and you should now get a 403-Forbidden error. Put the route guard back and reset the *CanAccessProducts* property to a "true" value in the SecurityManager class.

Summary

In this article you built Web API calls to authenticate users and provide an authorization object back to Angular. In addition, you configured your .NET Core Web API project to use the JSON Web Token system to secure your Web API calls. You learned to send bearer tokens to Angular and have Angular send those tokens back using an HTTP Interceptor class.

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 2" from the drop-down.