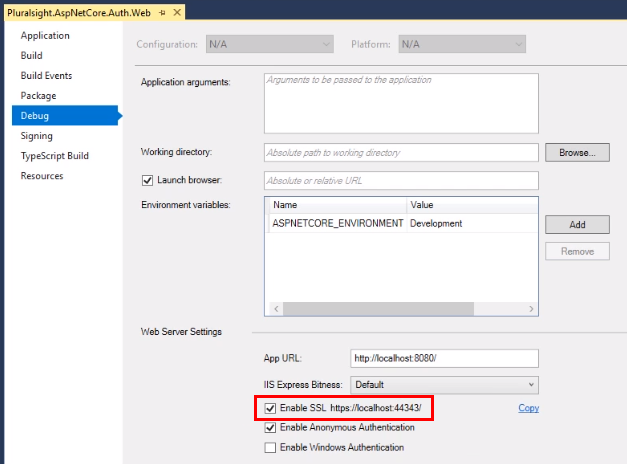
ASP Authentication Cookbook

# Demo Project Highlights

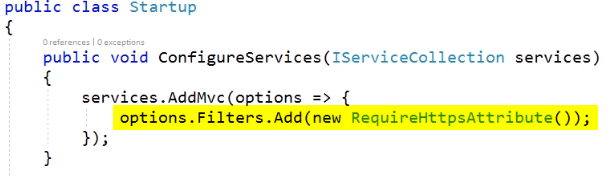
## Enable SSL

First, you’ll want to have SSL enabled in your project:

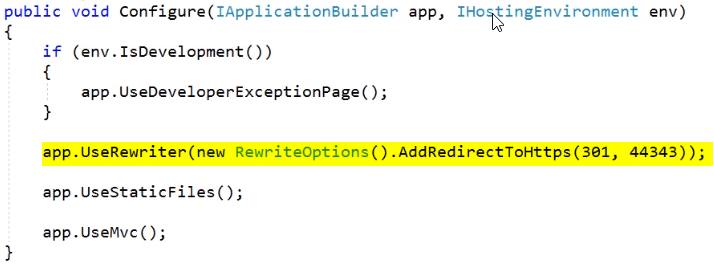


## Require HTTPS

This is done in Startup.cs, in your ConfigureServices method:



You’ll also want to use the URL re-writer to re-direct all HTTP traffic back to HTTPS:



Also notice that we’re adding support to serve up static files, as well as support for MVC.

All controllers use the [Authorize] attribute, and that causes ASP’s security to kick in.

# Cookie Authentication

Ok, so I started with a basic WebAPI project, nothing special.

Side note:

Just so you know, cookie authentication kind of assumes that you are using some kind of HTTP forms authentication. This means that if the server gets a request for a page that requires authentication and the user is not signed in, it will issue a 302 to a login page, with a returnUrl property set on the query string. There’s no real way to this in a SPA using WebAPI. I don’t know what will happen if you hit an API endpoint and get a 3xx response.

If you do want to use forms authentication, the default sign-in form that .NET Core 2 gives you is now hard-coded into the library. This wouldn’t be such a big deal, except for the fact that it is also hard-coded to use Bootstrap.

You can roll your own login page server-side. The process for doing so is pretty much identical to what you see here. The only real difference is that you will be calling HttpContext.SignInAsync() from an MVC controller instead of a WebAPI controller—a difference which is more or less cosmetic.

## Setting up the request pipeline

Ok, so all this stuff needs to be set up in Startup.cs, and it’s all boilerplate code:

In ConfigureServices(), add this pretty much anywhere:

services.AddAuthentication(options =>

{

    // You need all three of these if you want to use cookies!

    // Use cookies to authenticate the user.

    options.DefaultAuthenticateScheme =

CookieAuthenticationDefaults.AuthenticationScheme;

    // After the user is signed in use cookies.

    options.DefaultSignInScheme =

CookieAuthenticationDefaults.AuthenticationScheme;

    // If the user is not signed in, use the cookie authentication scheme to

    // authenticate them.

    options.DefaultChallengeScheme =

CookieAuthenticationDefaults.AuthenticationScheme;

})

    .AddCookie(options =>

    {

// If you’re using Forms authentication, this is how you re-direct

// to your sign-in screen.

        options.LoginPath = "/auth/signin";

    });

Yes, you need all that stuff, no matter what. As I said earlier, cookie authentication more or less assumes that you’re doing some kind of forms-based authentication.

And now you need to config the request pipeline in Configure():

app.UseHttpsRedirection();

app.UseAuthentication();

app.UseMvc();

The call to UseAuthentication() is the important one. It goes someplace before your call to UseMvc(). Side note:

The call to UseHttpsRedirection() is only used if you need HTTPS, which probably isn’t a big deal if you’re running behind a reverse proxy that handles all that for you.

## The user store

With cookie authentication, you’ll want to have some kind of user-store. For this demo I just created a mock service.

Here’s the interface:

public interface IUserService

{

    Task<bool> ValidateCredentials(

string username, string password, out User user);

}

public class User

{

    public User(string username, string password)

    {

        Username = username;

        Password = password;

    }

    public string Username { get; }

    public string Password { get; }

}

Next, I needed to set up the user service as a singleton object in Startup.cs, ConfigureServices():

// Register the dummy user service with the DI container.

var users = new Dictionary<string, string> { { "Chris", "password" } };

services.AddSingleton<IUserService>(new DummyUserService(users));

Now let’s talk about the DummyUserService implementation.

The implementation doesn’t actually authenticate anything. The only thing it does is check whether or not the password supplied by the user matches the password in the database, and it does this by comparing hashed values:

1. See if the username is in the store. If so then
   1. compare the hashed password that was sent in to the hashed password in the store. If it matches then
      1. return the record for the user as an out param
      2. return true
2. return false

You will need to add a NuGet package reference to BCrypt.Net-Next.

Code listing:

public class DummyUserService: IUserService

{

    // Using C#'s new tuple syntax

    private Dictionary<string, (string PasswordHash, User User)> \_users =

        new Dictionary<string, (string PasswordHash, User User)>();

    public DummyUserService(IDictionary<string, string> dummyUsers)

    {

        foreach (var user in dummyUsers)

        {

            // Need to add NuGet package: BCrypt.Net-Next

            // Hash the password with BCrypt's HashPassword function.

// You can use any secure hash function.

            \_users.Add(

                user.Key.ToLower(),

                (BCrypt.Net.BCrypt.HashPassword(user.Value), new User(user.Key, user.Value)));

        }

    }

    public Task<bool> ValidateCredentials(

string username,

string password,

out User user)

    {

        user = null;

        var key = username.ToLower();

        if (\_users.ContainsKey(key))

        {

            var hash = \_users[key].PasswordHash;

            if (BCrypt.Net.BCrypt.Verify(password, hash))

            {

                user = \_users[key].User;

                return Task.FromResult(true);

            }

        }

        // No such user.

        return Task.FromResult(false);

    }

}

## Taking a sign-in request

You’ll need the following usings:

using System.Collections.Generic;

using System.Security.Claims;

using System.Threading.Tasks;

using CookieAuthenticationApi.Models;

using CookieAuthenticationApi.Services;

using Microsoft.AspNetCore.Authentication;

using Microsoft.AspNetCore.Authentication.Cookies;

using Microsoft.AspNetCore.Mvc;

So, in the constructor I need a parameter for IUserService so that my controller can check the incoming user sign-in request against the store:

IUserService \_userService;

public SignInController(IUserService userService)

{

    \_userService = userService;

}

And now I need a POST handler. Note: the route was set on the class as [Route(“api/[controller]”)].

// POST: api/signin

[HttpPost]

public async Task<IActionResult> Post([FromBody] SignInModel model)

{

    if (!ModelState.IsValid)

    {

        return BadRequest(ModelState);

    }

    User user;

    if (await \_userService.ValidateCredentials(model.Username, model.Password, out user))

    {

        await SignInUser(user.Username);

        return Ok();

    }

    ModelState.AddModelError("Password", "Invalid password");

    return BadRequest(ModelState);

}

The key is in the SignInUser method, which is a private method in the controller class:

public async Task SignInUser(string username)

{

    var claims = new List<Claim>

    {

        new Claim(ClaimTypes.NameIdentifier, username),

        new Claim("name", username)

    };

    var identity = new ClaimsIdentity(claims, CookieAuthenticationDefaults.AuthenticationScheme, "name", null);

    var principal = new ClaimsPrincipal(identity);

    await HttpContext.SignInAsync(principal);

}

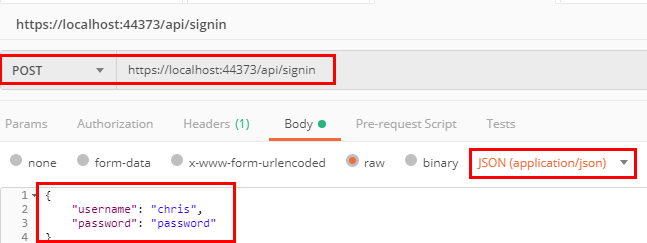
Once the user is signed in, I use the HttpContext object and call the SignInAsync extension

## Postman

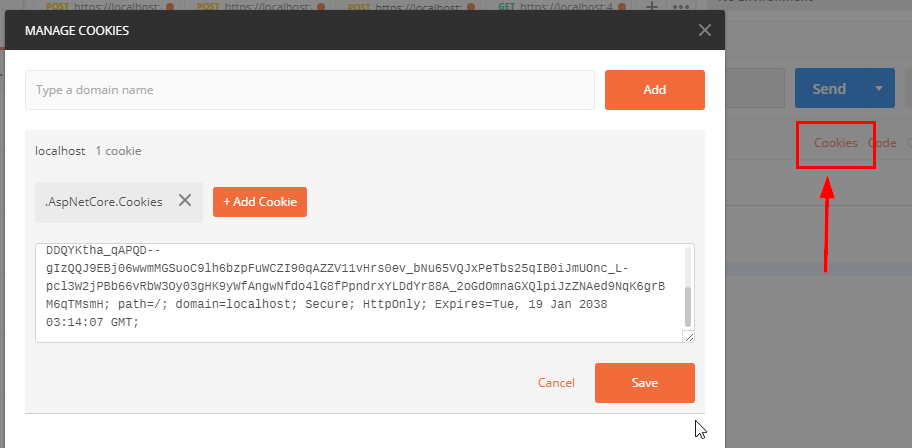
I’ve also got a generic placeholder API controller that VS created for me (the Values controller). I put an [Authorize] attribute at the top of it.

If I use Postman to do a GET <https://localhost:xxx/api/values>, I get a 404. This is kind of weird. I’d rather get a 401. I’m not sure what’s wrong.

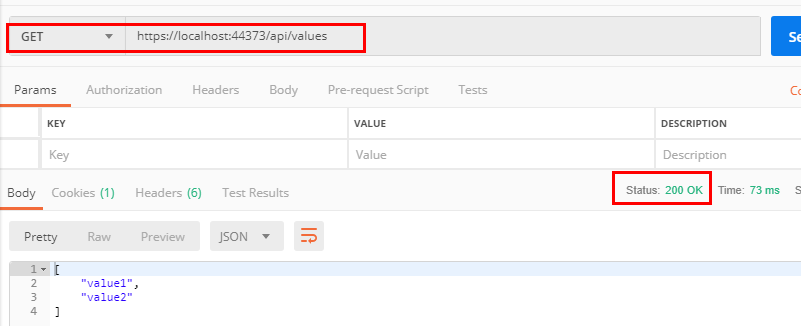
Then I do a POST to <https://.../api/signin>, like so:



After getting back a 200, I can see that I have a cookie. There is a cookies button under Postman’s send button. Click that, and you should see the cookie we got back:



And now you should be able to call the values controller and get a 200:



## Taking a sign-out request

You’ll want a way to cancel the cookie, otherwise the user will be considered logged in indefinitely.

I’ve implemented this as a DELETE request:

DELETE …/api/signin/{userName}

It’s pretty trivial.

* The current user’s ClaimsPrinciple object is given to you in a property called User (which you inherit from ControllerBase).
* The method you want to call is HttpContext.SignOutAsync().

ASP does all the work for you:

// DELETE: api/signin/{userName}

[HttpDelete]

[Route("{userName}")]

public async Task<IActionResult> Delete(string userName)

{

    if (

        string.IsNullOrWhiteSpace(userName) ||

        string.Compare(userName, User.Identity.Name, true) != 0)

    {

        return NotFound();

    }

    await SignOutUser();

    return NoContent();

}

private async Task SignOutUser()

{

    await HttpContext.SignOutAsync(

CookieAuthenticationDefaults.AuthenticationScheme);

}

# Signing in using Facebook (or some other auth-provider)

So before you begin, signing in requires a series of re-directs which in turn means that you can’t really do oauth unless you have some kind of server-side form that you can redirect back to.

The dialog will probably look like this:

1. User starts in the SPA and either tries to do something that requires authentication. This could be anything from clicking into a protected area of the app or merely trying to load the app.
   1. At this point, the SPA should probably dump any info from the user to window local storage
   2. then send the browser to your app’s sign-in form.
2. The sign-in form should probably be a basic server-side controller that issues a 3xx re-direct to the auth-provider’s sign-in page.
   1. NOTE: ASP can do all this for you. You really don’t need to do ANYTHING.
3. The user signs in using FB (or whatever)
4. If successful, FB redirects back to your sign-in page.
5. Your sign-in controller will then re-direct back to the SPA.

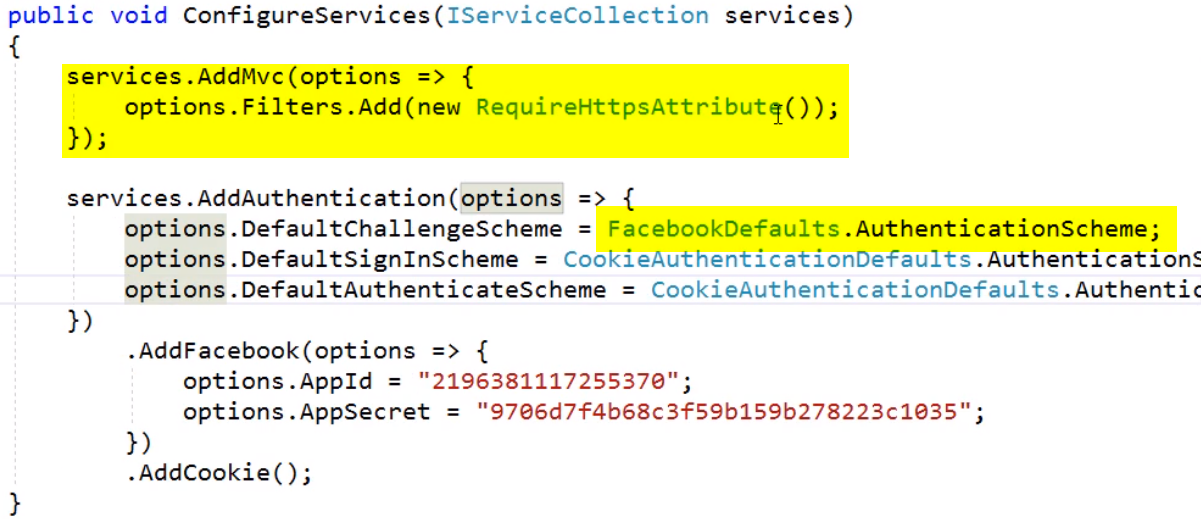
The really nice thing about all this is that the amount of user information your app has to keep is much, much simpler.

You’ll probably want to add a ClaimsIdentity for your with claims that map to roles in your application, to allow users to see

## Setting up the request pipeline

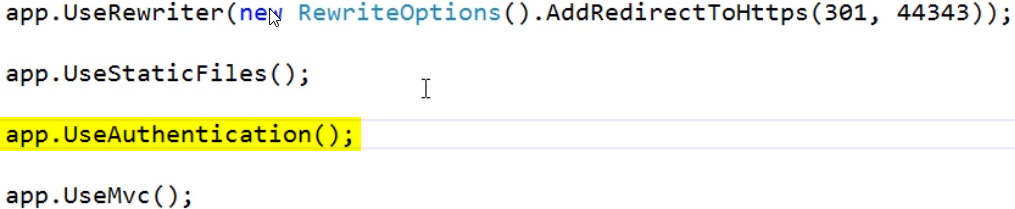
Setting things up for social is very similar to setting things up for cookie authentication, since Oauth + OpenID both do cookies.

So, in Startup.cs ConfigureServices() you’ll want to do this (Note, the lines that were cut off say CookieAuthenticationdefauls.AuthenticationScheme.



NOTE: We’ve gone to FB’s developer portal and set up an “application” with an appId and client secret.

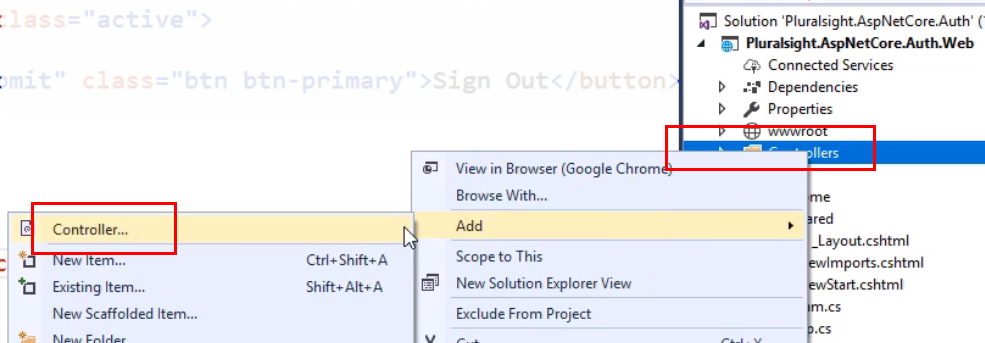
And in your Configure() method, you’ll want to have these calls:



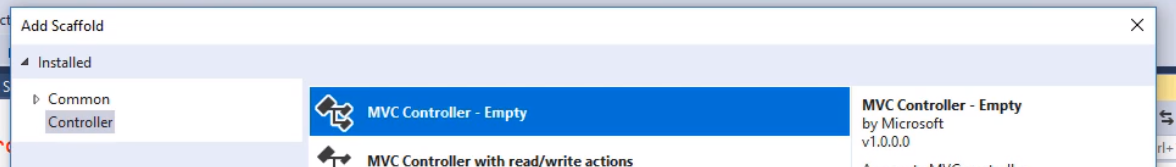
## Making a sign-in page

You can’t fake this stuff using a WebAPI controller. You have to make a page, but it doesn’t have to be full-blown page with styling and all that. It’s just an MVC controller that has some endpoints that the SPA can hit.

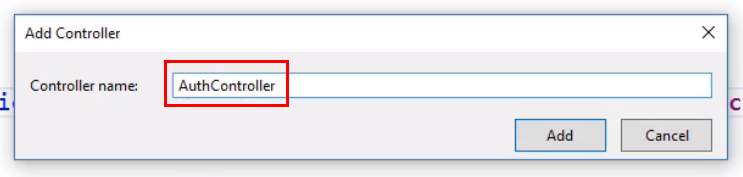
So first off let’s create a new empty controller:



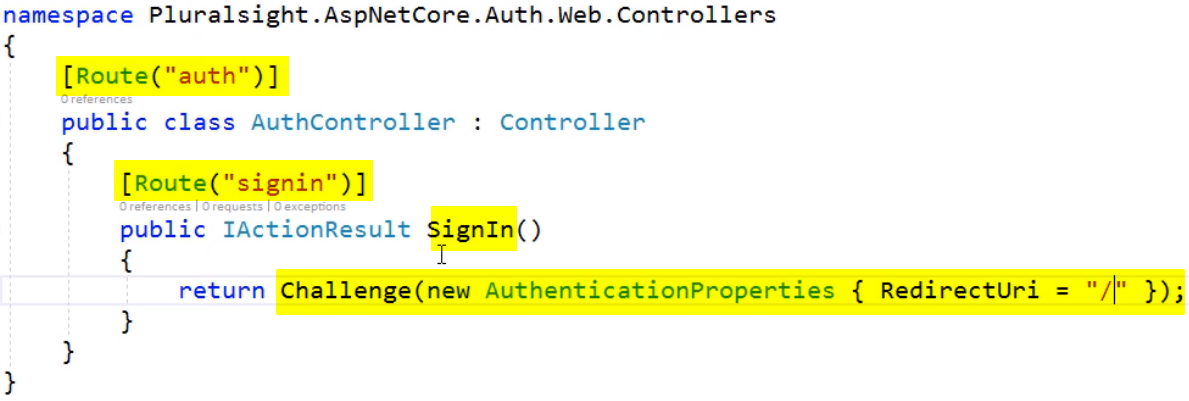
Then we’ll select MVC Controller – Empty:



Now let’s give it a name, and click the Add button:



And this is what we want:



And that is literally it. ASP will know what to do. The Challenge() method does it all.

So, how does the sign-in process work?

1. You have a sign-in button on your page, which points to this controller that you just made.
2. Clicking it takes the browser to the sign-in controller, which hands control over to ASP’s sign-in process via the call to the Challenge() method.
3. ASP re-directs the browser to FB’s sign-in page.
4. User signs in.
5. If successful, FB re-directs back to our controller
6. Our controller re-directs to the URI that we passed in to the Challenge() method (see above). This can point back to our SPA if we need it to. If our SPA uses application routing we can even have that route passed in on the query string or some-such.

## Signing out

Signing out is just as easy, and it’s literally the same thing that we did with cookie authentication.

