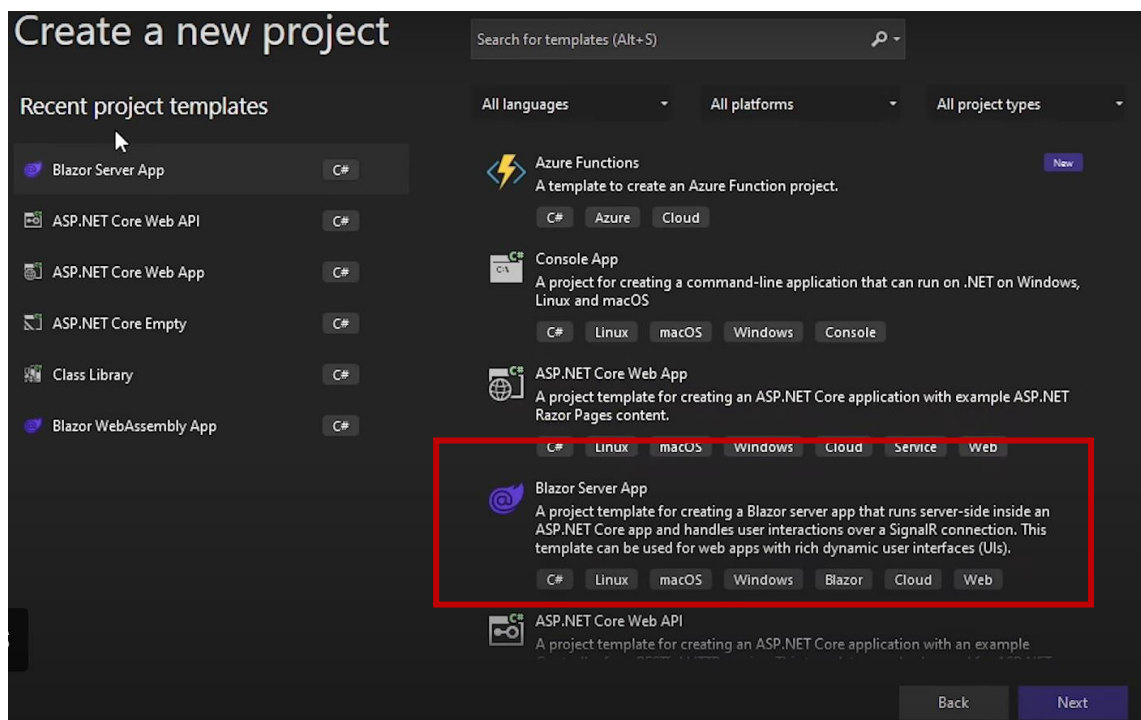


# Authentication & Authorization with OKTA in .NET 6 Blazor Server

## Create a new Blazor Server project –

- First, we need to create New Blazor server application for that open Visual Studio 2022 and chose Blazor server app.



## Change the app url ports –

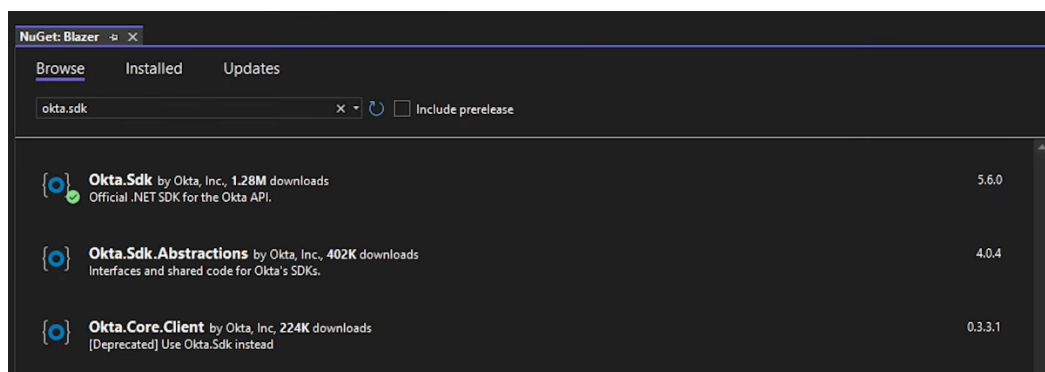
- Under Properties > launchSettings.json change the ports to 5001 (https) & 5000 (http).
- Here's what the Properties/launchSettings.json should look like

```
launchSettings.json*  X
Schema: https://json.schemastore.org/launchsettings.json
1 {
2   "iisSettings": {
3     "windowsAuthentication": false,
4     "anonymousAuthentication": true,
5     "iisExpress": {
6       "applicationUrl": "http://localhost:17549",
7       "sslPort": 44325
8     }
9   },
10  "profiles": {
11    "Blazer": {
12      "commandName": "Project",
13      "dotnetRunMessages": true,
14      "launchBrowser": true,
15      "applicationUrl": "https://localhost:5001;http://localhost:5000",
16      "environmentVariables": {
17        "ASPNETCORE_ENVIRONMENT": "Development"
18      }
19    },
20    "IIS Express": {
21      "commandName": "IISExpress",
22      "launchBrowser": true,
23      "environmentVariables": {
24        "ASPNETCORE_ENVIRONMENT": "Development"
25      }
26    }
27  }
28 }
```

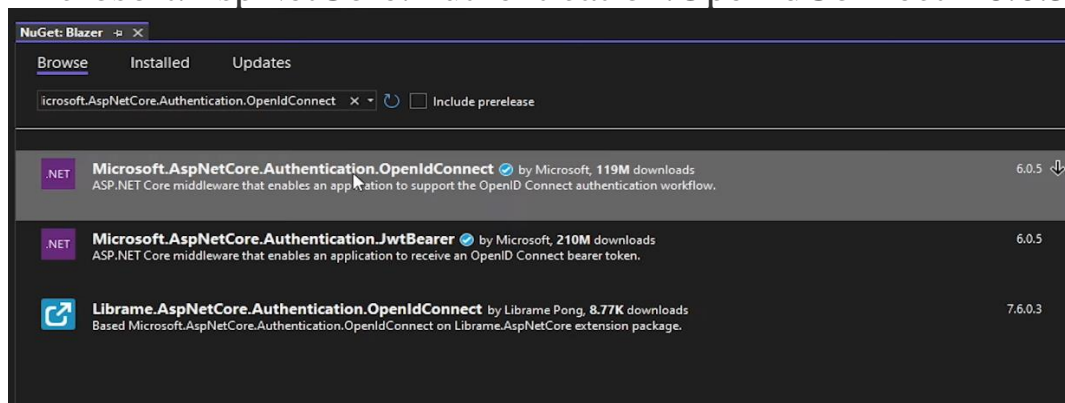
## Install Dependencies

The following dependencies must be installed with the latest versions available at the time of doing this tutorial:

- Okta.Sdk --version 5.6.0

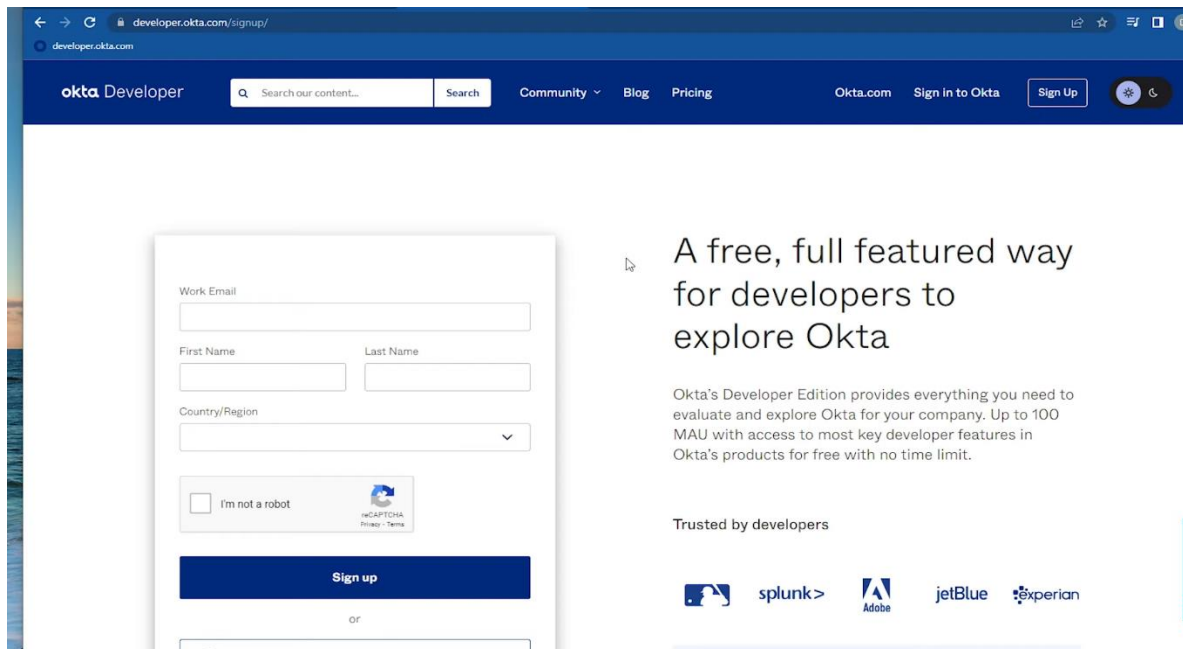


- Microsoft.AspNetCore.Authentication.OpenIdConnect --6.0.5

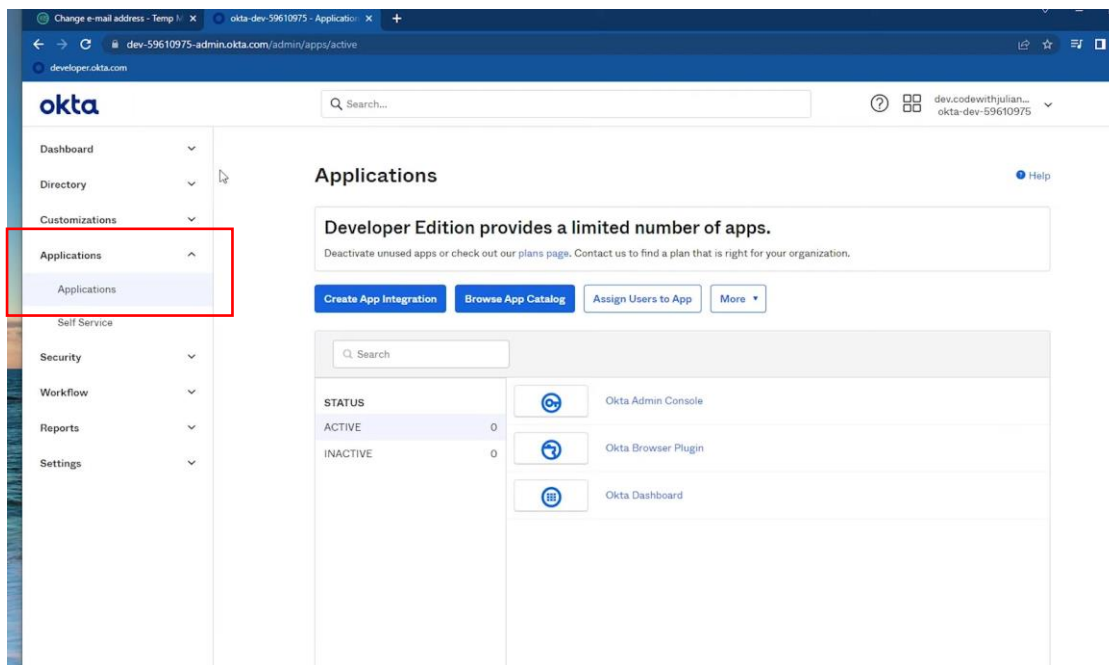


# Create a new Application in Okta

Navigate to [developer.okta.com](https://developer.okta.com) and create a free account. Then go to Applications > Applications submenu and create a new App integration. Select the following parameters:



The screenshot shows the Okta Developer sign-up page. The header includes the Okta Developer logo, a search bar, and links for Community, Blog, Pricing, Okta.com, Sign In to Okta, and a Sign Up button. The main content area features a sign-up form on the left and promotional text on the right. The form includes fields for Work Email, First Name, Last Name, and Country/Region, along with a CAPTCHA and a Sign up button. The promotional text on the right states: "A free, full featured way for developers to explore Okta" and "Okta's Developer Edition provides everything you need to evaluate and explore Okta for your company. Up to 100 MAU with access to most key developer features in Okta's products for free with no time limit." Below this, it says "Trusted by developers" and lists logos for splunk>, Adobe, jetBlue, and Experian.



The screenshot shows the Okta Admin console Applications page. The left sidebar contains a navigation menu with items: Dashboard, Directory, Customizations, Applications (highlighted with a red box), Self Service, Security, Workflow, Reports, and Settings. The main content area is titled "Applications" and includes a message: "Developer Edition provides a limited number of apps. Deactivate unused apps or check out our plans page. Contact us to find a plan that is right for your organization." Below this message are buttons for "Create App Integration", "Browse App Catalog", "Assign Users to App", and "More". A table lists the installed applications:

STATUS			
ACTIVE	0		Okta Admin Console
INACTIVE	0		Okta Browser Plugin
			Okta Dashboard

- Sign-in method: OIDC - OpenID Connect

**Create a new app integration**

Sign-in method

[Learn More](#)

- ☒ **OIDC - OpenID Connect**  
Token-based OAuth 2.0 authentication for Single Sign-On (SSO) through API endpoints. Recommended if you intend to build a custom app integration with the Okta Sign-In Widget.
- ☐ **SAML 2.0**  
XML-based open standard for SSO. Use if the Identity Provider for your application only supports SAML.
- ☐ **SWA - Secure Web Authentication**  
Okta-specific SSO method. Use if your application doesn't support OIDC or SAML.
- ☐ **API Services**  
Interact with Okta APIs using the scoped OAuth 2.0 access tokens for machine-to-machine authentication.

Cancel Next

- Application type: Web Application

**Application type**

What kind of application are you trying to integrate with Okta?

Specifying an application type customizes your experience and provides the best configuration, SDK, and sample recommendations.

- ☒ **Web Application**  
Server-side applications where authentication and tokens are handled on the server (for example, Go, Java, ASP.Net, Node.js, PHP)
- ☐ **Single-Page Application**  
Single-page web applications that run in the browser where the client receives tokens (for example, Javascript, Angular, React, Vue)
- ☐ **Native Application**  
Desktop or mobile applications that run natively on a device and redirect users to a non-HTTP callback (for example, iOS, Android, React Native)

Cancel Next

- Name: Blazer (or your choice) Grant type: default Authorization code. Also select Refresh Token

**okta**

Search...

dev-59610975-admin.okta.com/admin/apps/oauth2-wizard/create?applicationType=WEB

dev.codedwithjulian...  
okta-dev-59610975

**General Settings**

App integration name: Blazer

Logo (Optional)

Grant type

[Learn More](#)

- ☐ Client acting on behalf of itself
- ☐ Client Credentials
- ☐ Client acting on behalf of a user
- ☒ Authorization Code
- ☐ Interaction Code
- ☒ Refresh Token
- ☐ Implicit (hybrid)

Sign-in redirect URIs

Okta sends the authentication response and ID token for the user's sign-in request to these URIs

[Learn More](#)

☐ Allow wildcard \* in sign-in URI redirect.

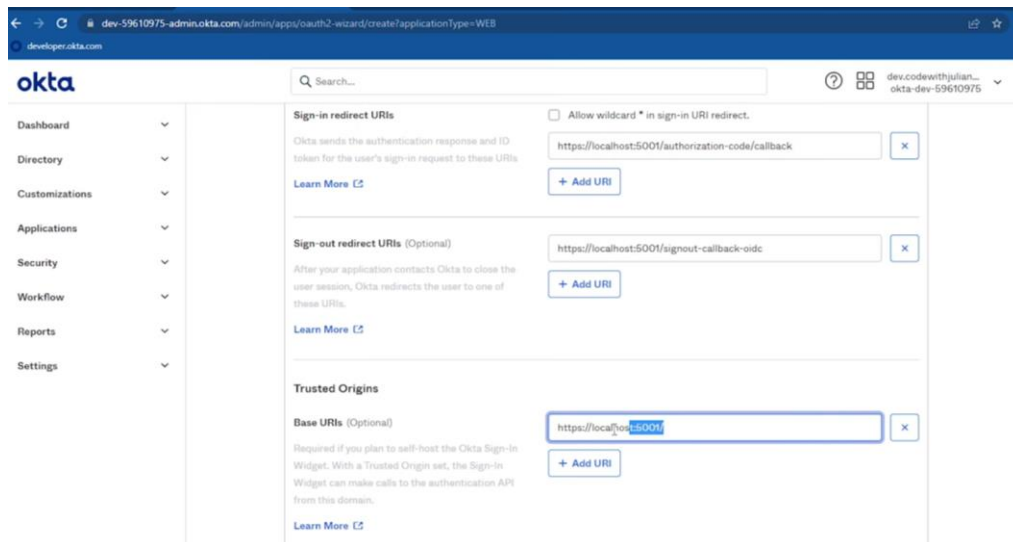
http://localhost:8080/authorization-code/callback

+ Add URI

Sign-out redirect URIs (Optional)

http://localhost:8080

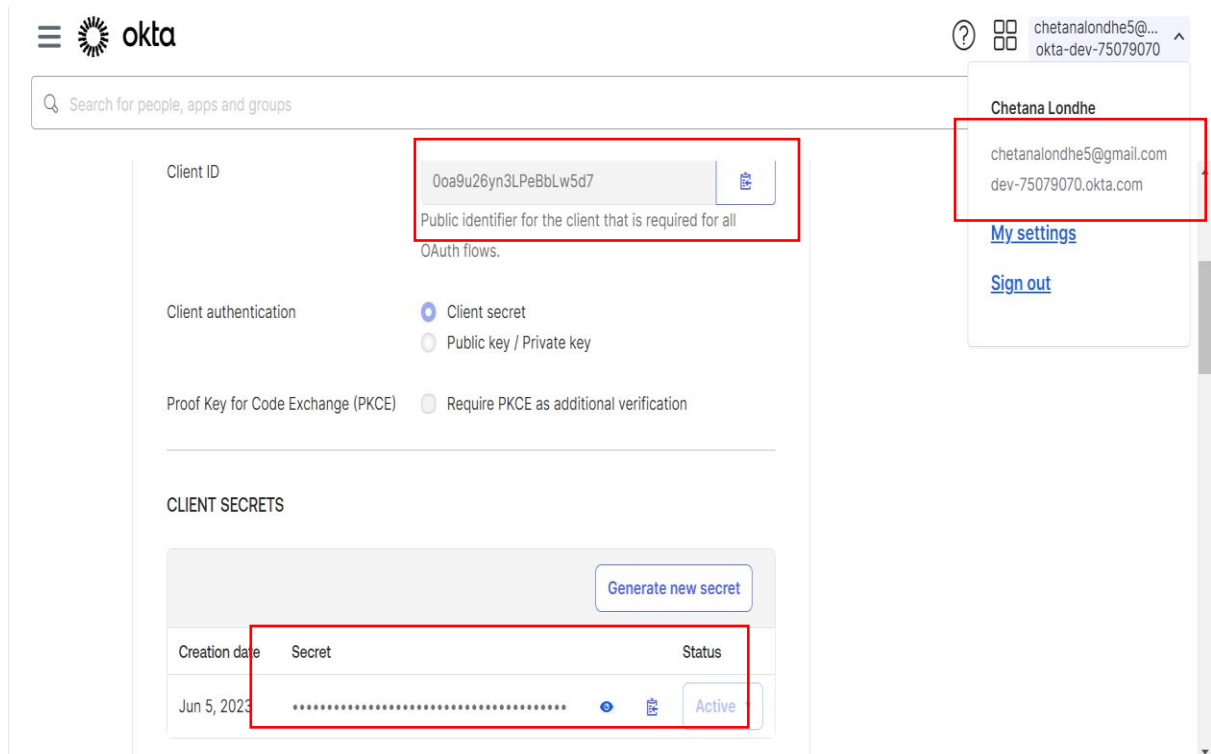
- Sign-in redirect URIs: <https://localhost:5001/authorization-code/callback>
- Sign-out redirect URIs : <https://localhost:5001/signout-callback-oidc>
- Base URIs: <https://localhost:5001/>



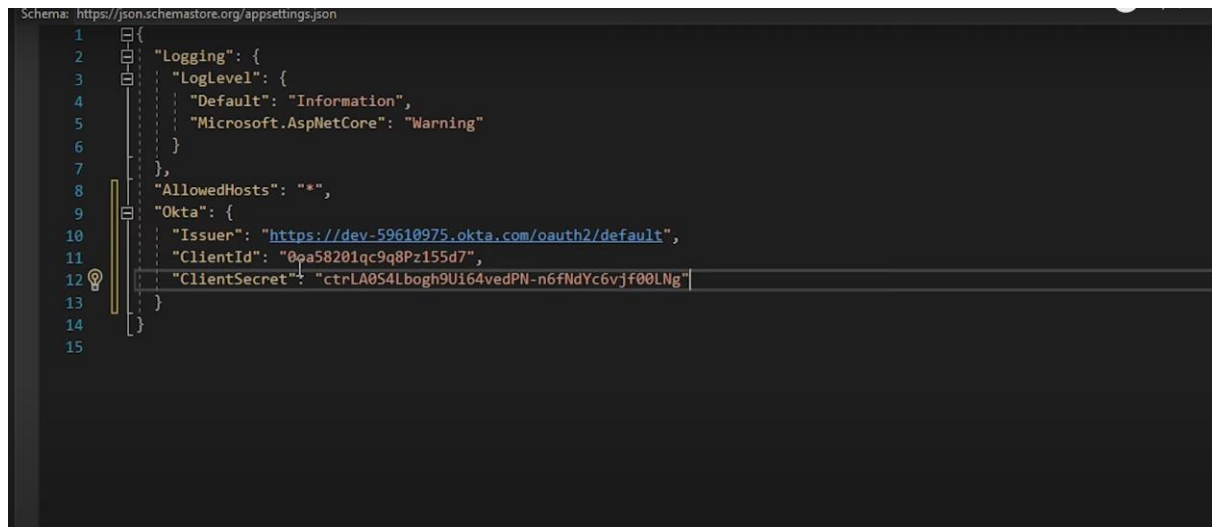
- The Authorization Code Flow an OAuth 2.0 type of grant that allows the client app to swap an authorization code for an Access Token (or JWT).
- The client sends a request up to the Authorization Server (Okta in this case) for an authorization code.
- Then swaps that for an Access Token that can be used to access resources, otherwise inaccessible.

## Modify appsettings.json

- This is where you specify the app integration and Okta account credentials within your app.
- You'll need the Client ID and secret and the okta domain (for the issuer).
- The issuer is composed like following: https:// + Okta\_Domain + /oauth2/default.



- Here is the appsettings.json:

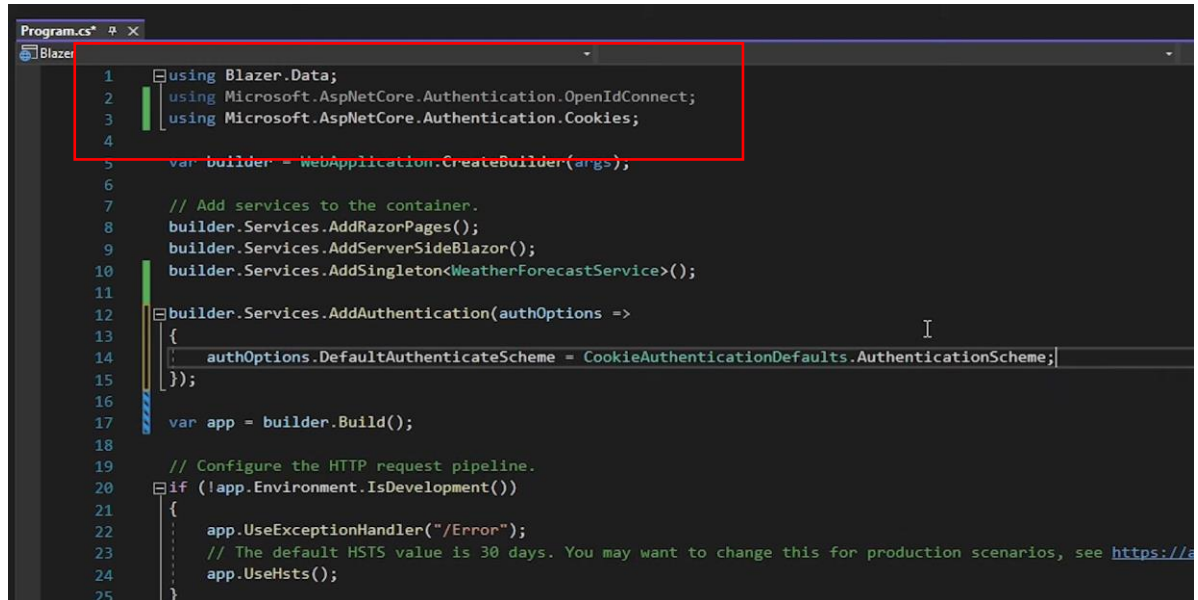


## Configure the Blazor app to use Okta as the External Auth Provider

- This is the part where we setup Authentication and install OpenID Connect within our application.

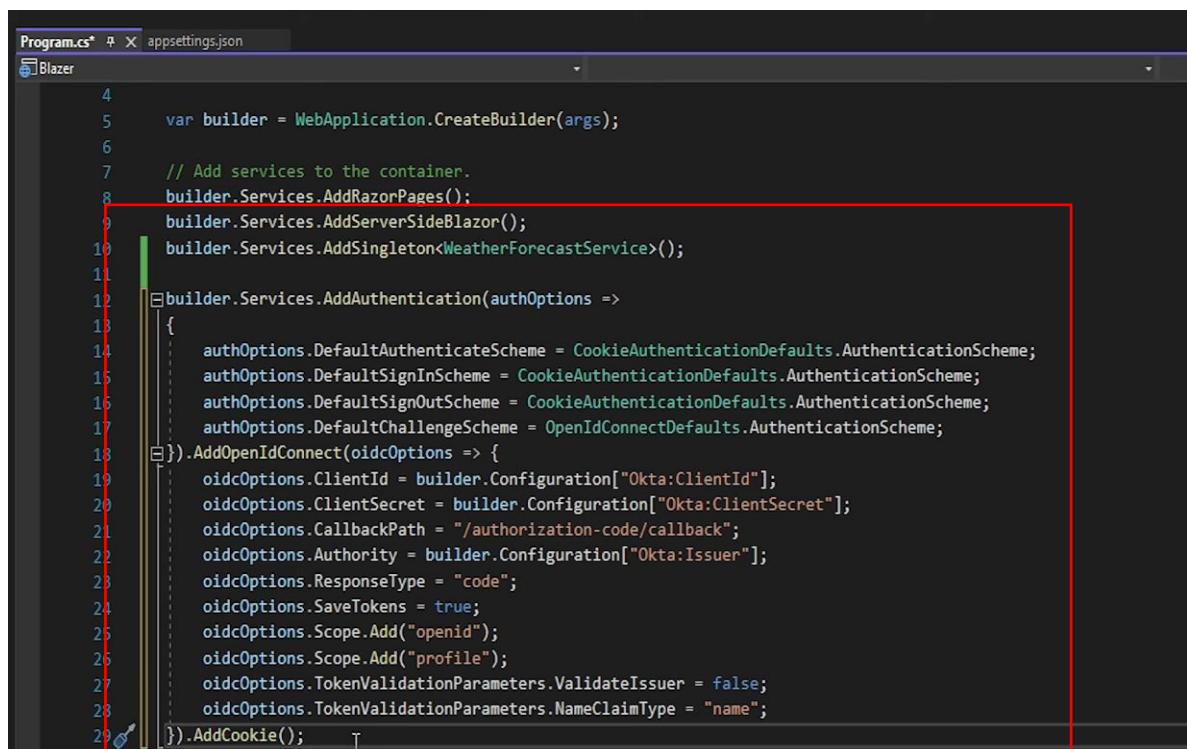


- This is being done in Program.cs. We first add in Authentication and initialise the Default schemes (Authentication, SignIn, SignOut).



```
Program.cs* 4 X
Blazer
1 using Blazer.Data;
2 using Microsoft.AspNetCore.Authentication.OpenIdConnect;
3 using Microsoft.AspNetCore.Authentication.Cookies;
4
5 var builder = WebApplication.CreateBuilder(args);
6
7 // Add services to the container.
8 builder.Services.AddRazorPages();
9 builder.Services.AddServerSideBlazor();
10 builder.Services.AddSingleton<WeatherForecastService>();
11
12 builder.Services.AddAuthentication(authOptions =>
13 {
14     authOptions.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;
15 });
16
17 var app = builder.Build();
18
19 // Configure the HTTP request pipeline.
20 if (!app.Environment.IsDevelopment())
21 {
22     app.UseExceptionHandler("/Error");
23     // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://a
24     app.UseHsts();
25 }
```

- This is then followed by setting up OIDC with Okta. The block is ended by adding cookie authentication in our application.



```
Program.cs* 4 X appsettings.json
Blazer
4
5 var builder = WebApplication.CreateBuilder(args);
6
7 // Add services to the container.
8 builder.Services.AddRazorPages();
9 builder.Services.AddServerSideBlazor();
10 builder.Services.AddSingleton<WeatherForecastService>();
11
12 builder.Services.AddAuthentication(authOptions =>
13 {
14     authOptions.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;
15     authOptions.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
16     authOptions.DefaultSignOutScheme = CookieAuthenticationDefaults.AuthenticationScheme;
17     authOptions.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
18 }).AddOpenIdConnect(oidcOptions => {
19     oidcOptions.ClientId = builder.Configuration["Okta:ClientId"];
20     oidcOptions.ClientSecret = builder.Configuration["Okta:ClientSecret"];
21     oidcOptions.CallbackPath = "/authorization-code/callback";
22     oidcOptions.Authority = builder.Configuration["Okta:Issuer"];
23     oidcOptions.ResponseType = "code";
24     oidcOptions.SaveTokens = true;
25     oidcOptions.Scope.Add("openid");
26     oidcOptions.Scope.Add("profile");
27     oidcOptions.TokenValidationParameters.ValidateIssuer = false;
28     oidcOptions.TokenValidationParameters.NameClaimType = "name";
29 }).AddCookie();
```

- Then, towards the bottom of the file, we add authentication, authorization and map controllers.

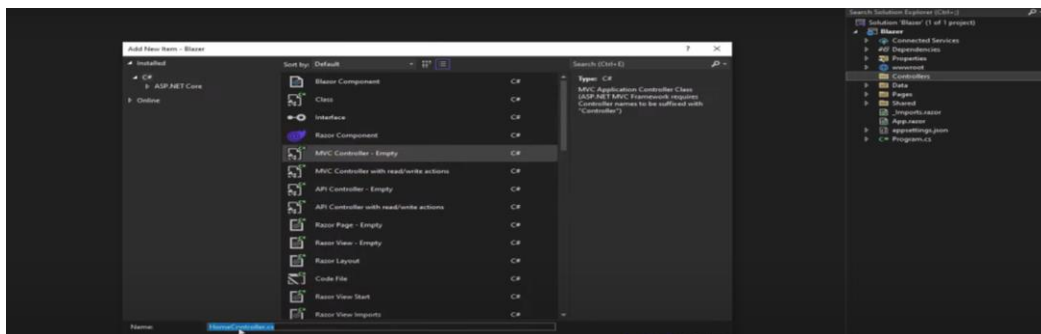
```

39  }
40
41  app.UseHttpsRedirection();
42
43  app.UseStaticFiles();
44
45  app.UseRouting();
46
47  app.UseAuthentication();
48  app.UseAuthorization();
49
50  app.MapControllers();
51  app.MapBlazorHub();
52  app.MapFallbackToPage("/_Host");
53
54  app.Run();
55

```

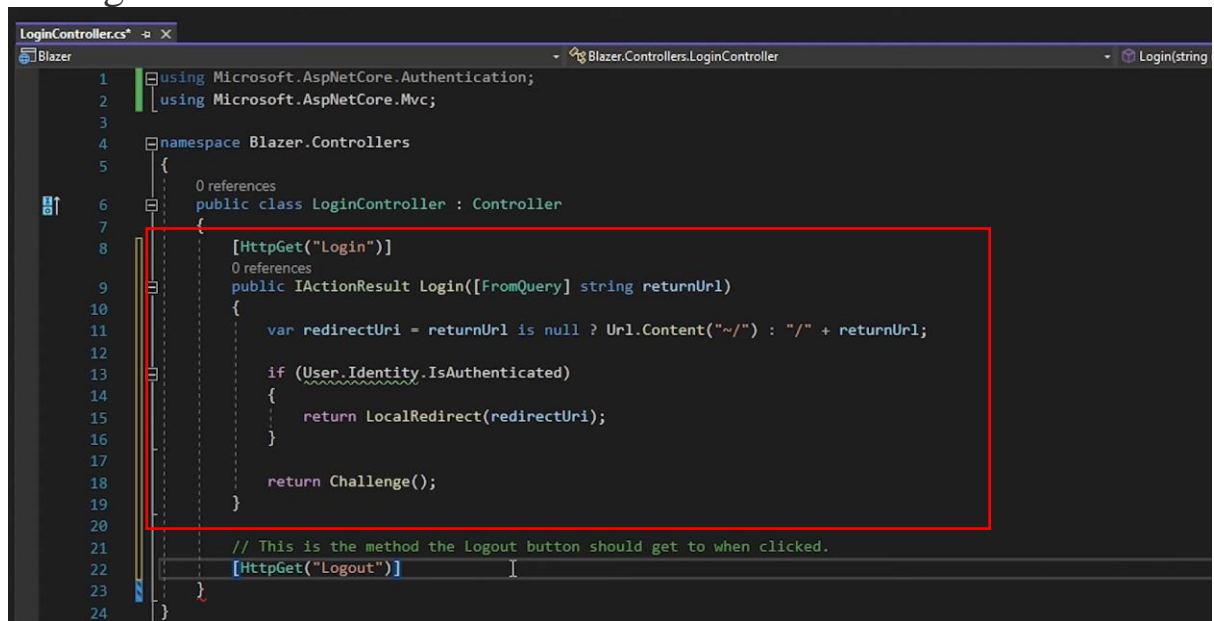
## Add a LoginController

- The LoginController is a simple MVC controller that contains 2 methods, Login and Logout that handle what their name implies.
- This sets up the right actions for when we want to log in our out. In case the user isn't authenticated (or if they click Login), they will be redirected to the login page using the Login endpoint of this controller. If the user clicks Log out, we'll send a GET request to the Logout method.
- For the Login endpoint, we check if the user is already authenticated, if not then return a Challenge (which has them to authenticate).
- The Logout method method ensures the user isn't authenticated before signing them out and redirecting them to the requested URI.
- First add MVC controller





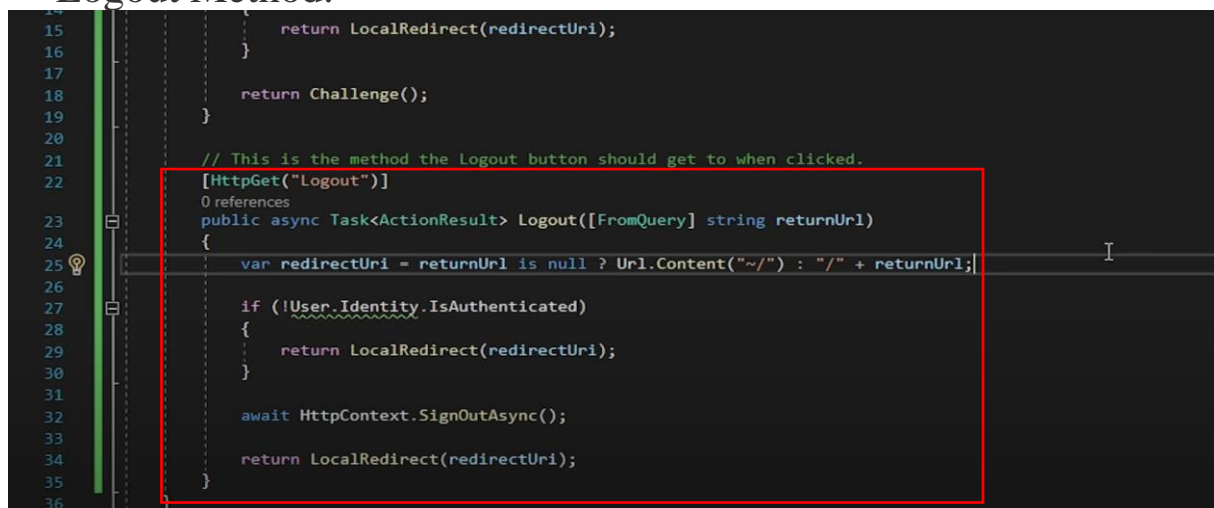
- Login Method:



```

1  using Microsoft.AspNetCore.Authentication;
2  using Microsoft.AspNetCore.Mvc;
3
4  namespace Blazer.Controllers
5  {
6      0 references
7      public class LoginController : Controller
8      {
9          [HttpGet("Login")]
10         0 references
11         public IActionResult Login([FromQuery] string returnUrl)
12         {
13             var redirectUri = returnUrl is null ? Url.Content("~/") : "/" + returnUrl;
14
15             if (User.Identity.IsAuthenticated)
16             {
17                 return LocalRedirect(redirectUri);
18             }
19
20             return Challenge();
21         }
22
23         // This is the method the Logout button should get to when clicked.
24         [HttpGet("Logout")]
    
```

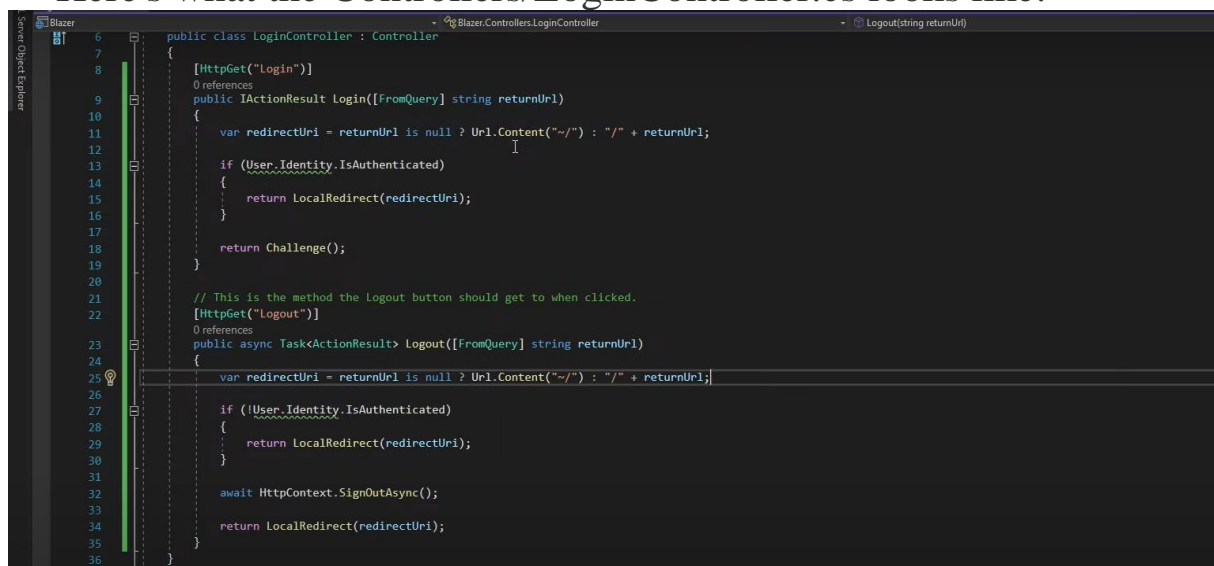
- Logout Method:



```

15         return LocalRedirect(redirectUri);
16     }
17
18     return Challenge();
19 }
20
21 // This is the method the Logout button should get to when clicked.
22 [HttpGet("Logout")]
23 0 references
24 public async Task<ActionResult> Logout([FromQuery] string returnUrl)
25 {
26     var redirectUri = returnUrl is null ? Url.Content("~/") : "/" + returnUrl;
27
28     if (!User.Identity.IsAuthenticated)
29     {
30         return LocalRedirect(redirectUri);
31     }
32
33     await HttpContext.SignOutAsync();
34
35     return LocalRedirect(redirectUri);
36 }
    
```

- Here's what the Controllers/LoginController.cs looks like:

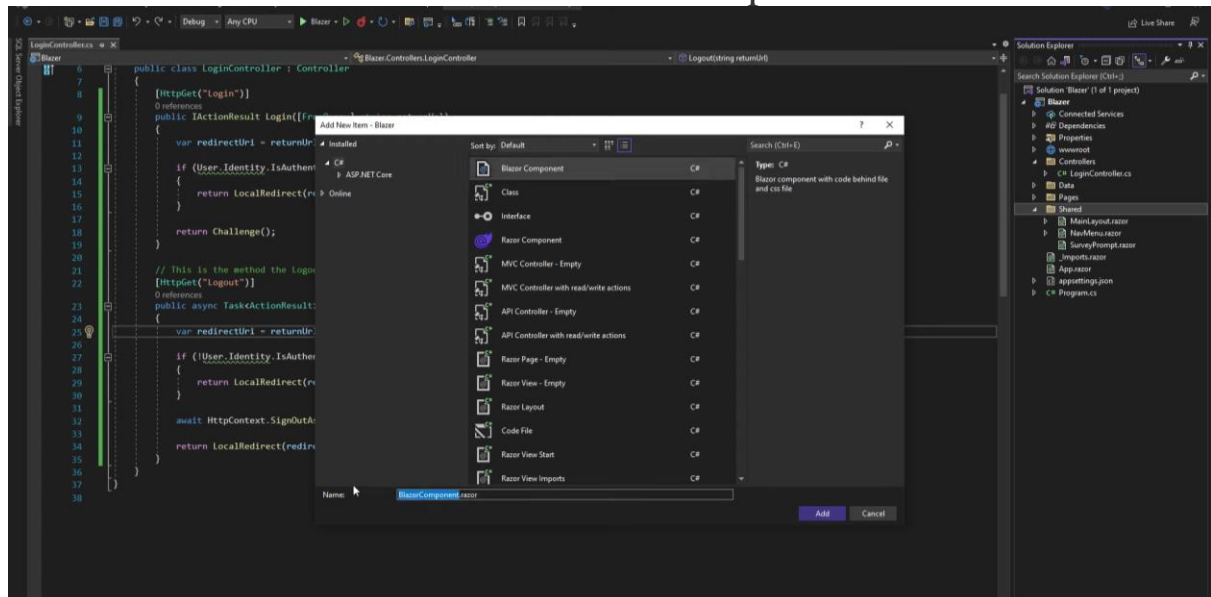


```

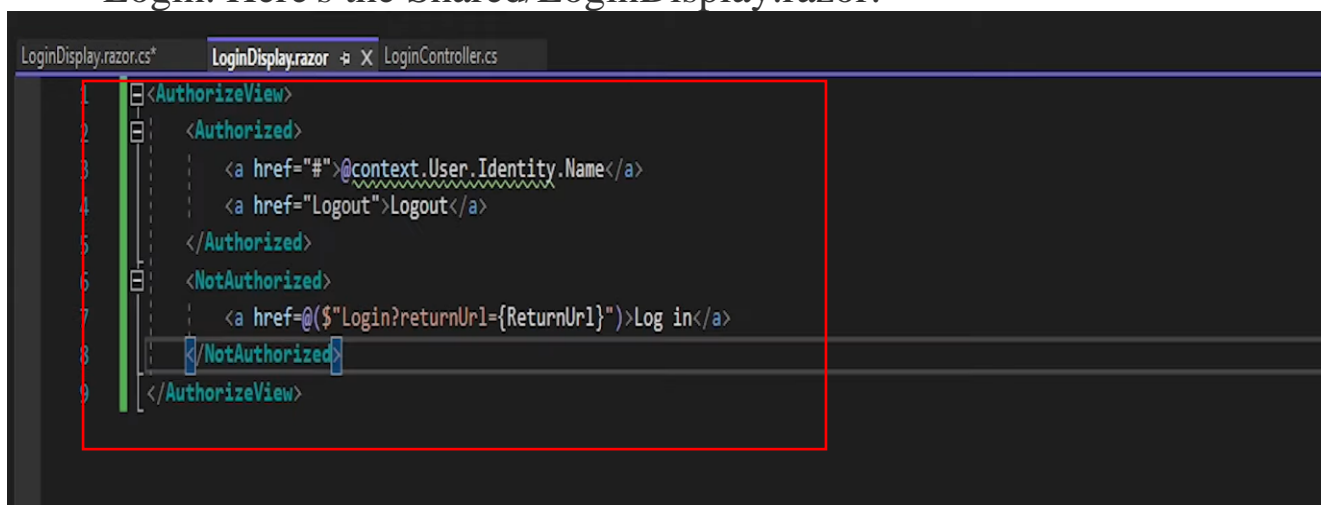
6  public class LoginController : Controller
7  {
8      [HttpGet("Login")]
9      0 references
10     public IActionResult Login([FromQuery] string returnUrl)
11     {
12         var redirectUri = returnUrl is null ? Url.Content("~/") : "/" + returnUrl;
13
14         if (User.Identity.IsAuthenticated)
15         {
16             return LocalRedirect(redirectUri);
17         }
18
19         return Challenge();
20     }
21
22     // This is the method the Logout button should get to when clicked.
23     [HttpGet("Logout")]
24     0 references
25     public async Task<ActionResult> Logout([FromQuery] string returnUrl)
26     {
27         var redirectUri = returnUrl is null ? Url.Content("~/") : "/" + returnUrl;
28
29         if (!User.Identity.IsAuthenticated)
30         {
31             return LocalRedirect(redirectUri);
32         }
33
34         await HttpContext.SignOutAsync();
35
36         return LocalRedirect(redirectUri);
37     }
38 }
    
```

# Add LoginDisplay.razor

- This is a separate component that we're creating to contain the two buttons, log in and log out, based on the user authentication state.
- For that we need to add new Blazor component.



- We're making use of the <AuthorizeView> component which we'll later expose throughout our project from App.razor.
- The <AuthorizeView> component enables two other components that we'll use: <Authorized> and <NotAuthorized>.
- They will show the contents of those components based on the user authentication state (whether or not they're signed in). So if they're authenticated, we show a Logout button, otherwise Login. Here's the Shared/LoginDisplay.razor:



- The returnUrl is defined at init time by grabbing the Uri and turning it into a base relative path, this is the Shared/LoginDisplay.razor.cs:

```
LoginDisplay.razor.cs LoginDisplay.razor LoginController.cs
Blazer Blazer.Shared.LoginDisplay OnInitialized

1 using Microsoft.AspNetCore.Components;
2 using Microsoft.AspNetCore.Components.Rendering;
3 using System;
4 using System.Collections.Generic;
5 using System.Linq;
6 using System.Threading.Tasks;
7
8
9 namespace Blazer.Shared
10 {
11     1 reference
12     public partial class LoginDisplay
13     {
14         2 references
15         [Inject] public NavigationManager Navigation { get; set; }
16         2 references
17         [Parameter] public string ReturnUrl { get; set; }
18
19         0 references
20         protected override async Task OnInitializedAsync()
21         {
22             ReturnUrl = Navigation.ToBaseRelativePath(Navigation.Uri);
23         }
24     }
25 }
```

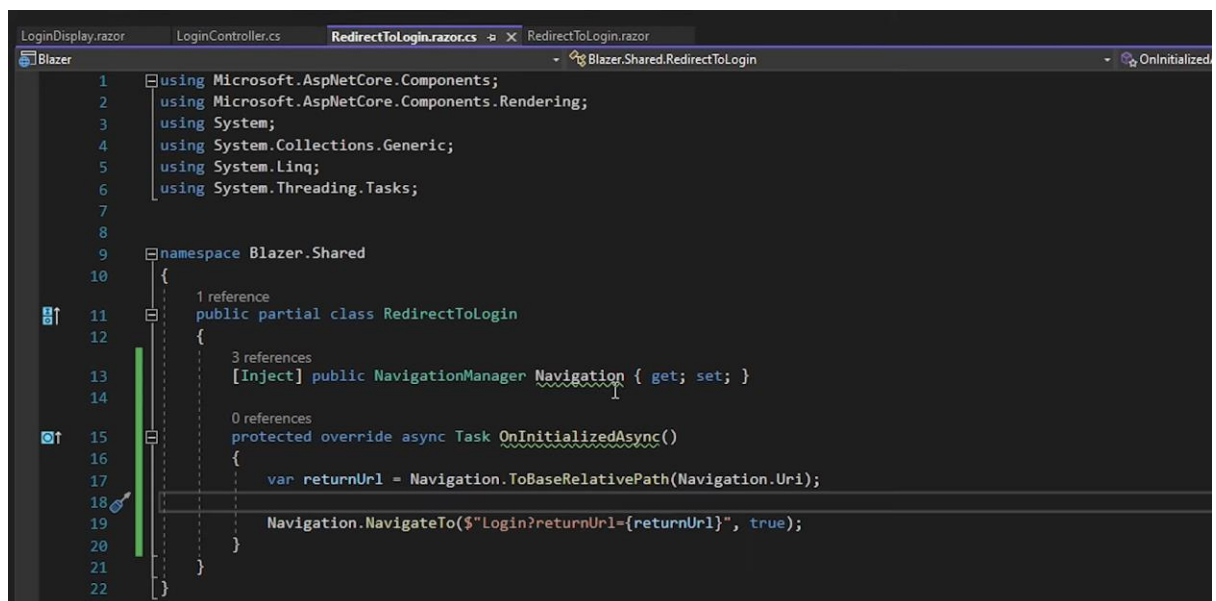
## Add the LoginDisplay component to MainLayout.razor

- The MainLayout is the default layout of our blazor app and we'll add the newly created <LoginDisplay /> component in the top navbar area (above the About anchor tag). This is the MainLayout.razor:

```
MainLayout.razor* LoginDisplay.razor.cs LoginDisplay.razor LoginController.cs
1 @inherits LayoutComponentBase
2
3 <PageTitle>Blazer</PageTitle>
4
5 <div class="page">
6     <div class="sidebar">
7         <NavMenu />
8     </div>
9
10    <main>
11        <div class="top-row px-4">
12            <LoginDisplay />
13            <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
14        </div>
15
16        <article class="content px-4">
17            @Body
18        </article>
19    </main>
20 </div>
21
```

## Setup unauthorized redirect

- This component will be rendered in case of an unauthorized access of a particular page/resource. If the user isn't authenticated, or hasn't got the correct permissions to view/edit that page/resource, this component will be used to redirect them to the login (Login endpoint from the LoginController.cs).
- This will be a new component within the Shared namespace. This will be a Blazor component but we'll only use the backend for this. Call it RedirectToLogin. All this is doing is grabbing the Uri (parsed as a base relative path) and navigating to the Login 'page'. So in essence sending a GET request to the Login endpoint when initialised. The front-end .razor file is empty. Here's Shared/RedirectToLogin.razor.cs:



```
1 using Microsoft.AspNetCore.Components;
2 using Microsoft.AspNetCore.Components.Rendering;
3 using System;
4 using System.Collections.Generic;
5 using System.Linq;
6 using System.Threading.Tasks;
7
8
9 namespace Blazer.Shared
10 {
11     1 reference
12     public partial class RedirectToLogin
13     {
14         3 references
15         [Inject] public NavigationManager Navigation { get; set; }
16
17         0 references
18         protected override async Task OnInitializedAsync()
19         {
20             var returnUrl = Navigation.ToBaseRelativePath(Navigation.Uri);
21             Navigation.NavigateTo($"Login?returnUrl={returnUrl}", true);
22         }
23     }
24 }
```

## Setup the App.razor

- Here's where we'll enable the authentication state within our project. This is done in App.razor and we must wrap the <Router> component in two components: <CascadingValue> this is used to expose the AccessToken to all of our components. And the <CascadingAuthenticationState> which is the one to enable the authentication state (so we can make use of those <Authorized> / <NotAuthorized> components).

- You will notice we're also making use of the later one, and in it, we're rendering the just created `<RedirectToLogin />` component. This way, if a user is not logged in, the app redirects them to the login page (in there, the authentication flow is started and the user will be sent to Okta to login, then redirected back to us. Those redirect URLs help us do that).
- This is App.razor:

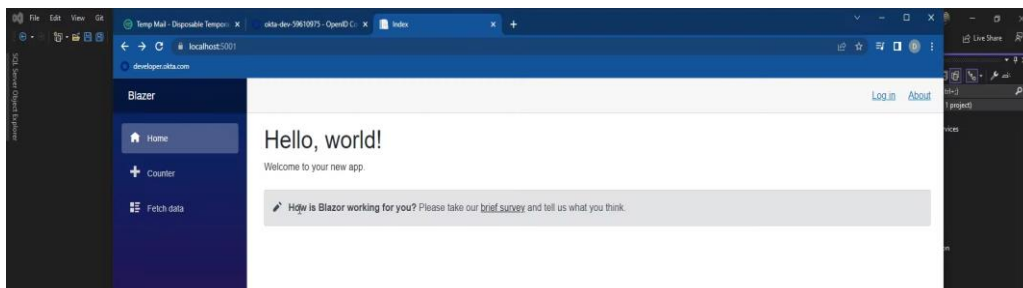
```

App.razor*  RedirectToLogin.razor.cs
1  <CascadingValue Name="AccessToken" Value="AccessToken">
2      <CascadingAuthenticationState>
3          <Router AppAssembly="@typeof(App).Assembly">
4              <Found Context="routeData">
5                  <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)">
6                      <NotAuthorized>
7                          <RedirectToLogin />
8                      </NotAuthorized>
9                      <Authorizing>
10                         Authorizing...
11                     </Authorizing>
12                 </AuthorizeRouteView>
13             </Found>
14             <NotFound>
15                 <PageTitle>Not found</PageTitle>
16                 <LayoutView Layout="@typeof(MainLayout)">
17                     <p role="alert">Sorry, there's nothing at this address.</p>
18                 </LayoutView>
19             </NotFound>
20         </Router>
21     </CascadingAuthenticationState>
22 </CascadingValue>
23
24
25 @code{
26     [Parameter] public string AccessToken { get; set; }
27 }

```

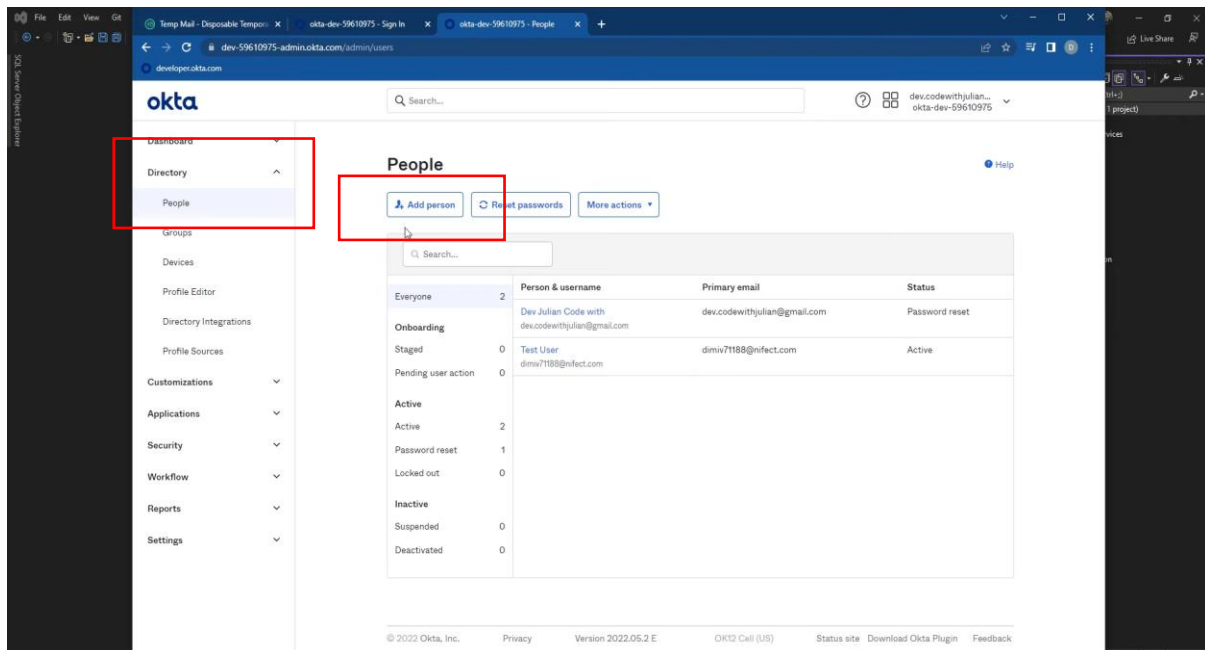
## Run the app and test it

- When you run the app and click Login, most likely you will be logged in without having to input your username and password.
- This is because you are already logged into Okta, from when you've created your account and app integration. If you want, log out of Okta to test this properly.

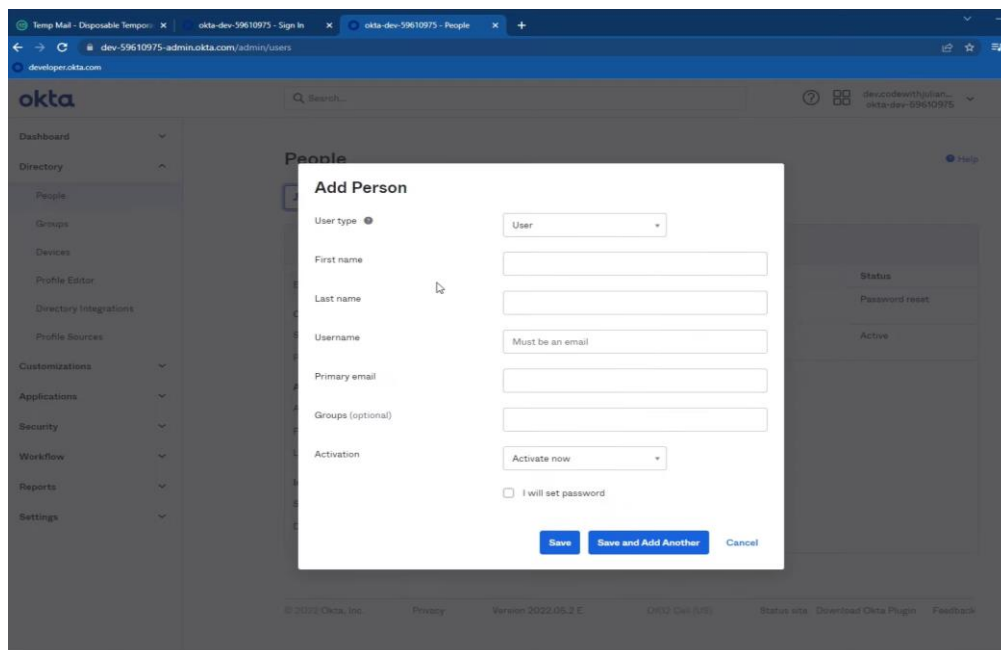


# Add a New User to our Application

- Go to Directory > People. And then Add Person.

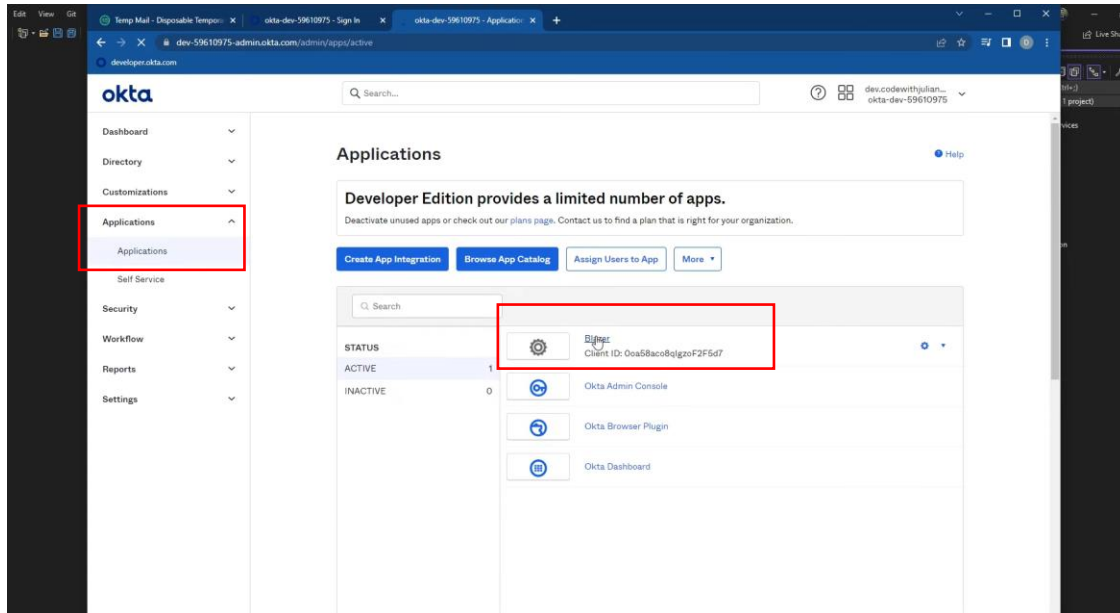


- Add details of the user.

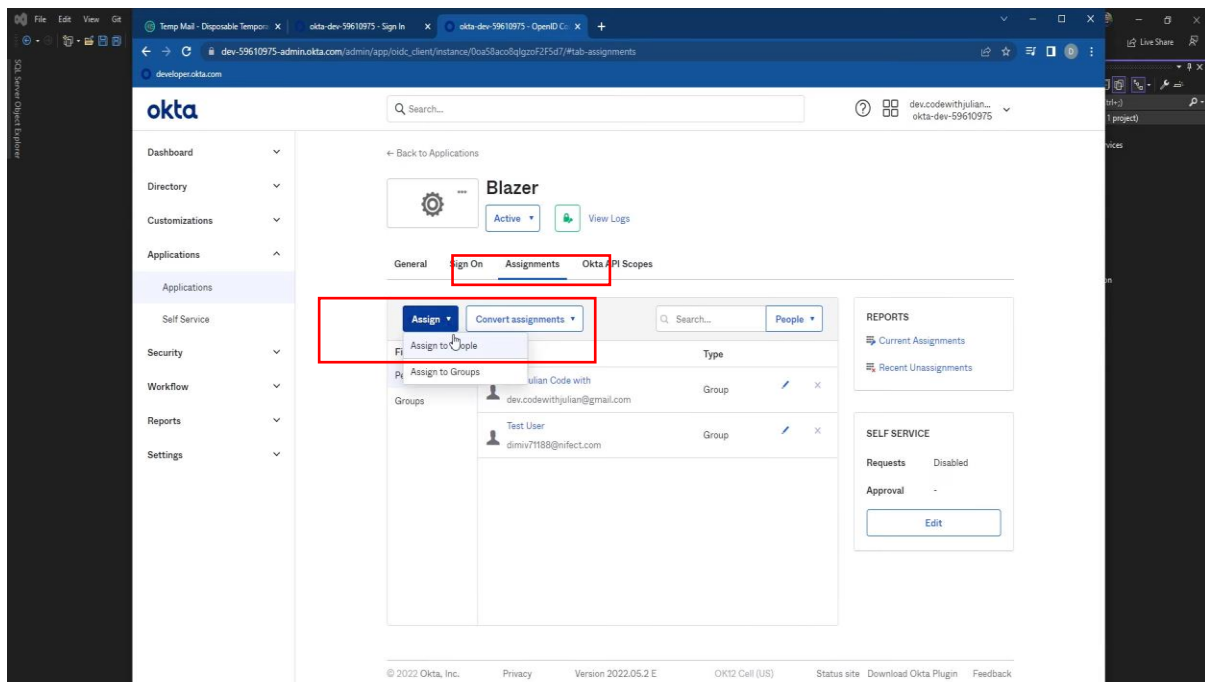




- To assign user to our application go to Application >Blazer

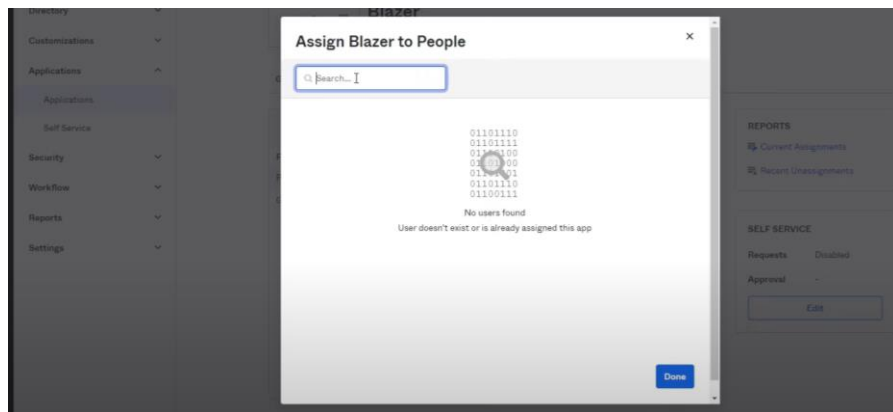


- In our Blazor application go to Assignments >Assign>Assign to people.





- Search for the user in Assign Blazer to people and click on Done, which will allow that user to login to that application



## Setup authorization inside pages

- If we want to secure the default FetchData component, as in, no unauthenticated users could access it.
- Go to that component's razor page (Pages/FetchData.razor), and add an `[Authorize]` attribute at the top of the page.

```

FetchData.razor* x NavMenu.razor
1  @page "/"fetchdata"
2  @attribute [Authorize]
3
4  <PageTitle>Weather forecast</PageTitle>
5
6  @using Blazer.Data
7  @inject WeatherForecastService ForecastService
8
9  <h1>Weather forecast</h1>
10
11 <p>This component demonstrates fetching data from a service.</p>
12
13 @if (forecasts == null)
14 {
15     <p><em>Loading...</em></p>
16 }
17 else
18 {
19     <table class="table">
20         <thead>
21             <tr>
22                 <th>Date</th>
23                 <th>Temp. (C)</th>
24                 <th>Temp. (F)</th>
25                 <th>Summary</th>
26             </tr>
27         </thead>
28         <tbody>
29             @foreach (var forecast in forecasts)
30

```

