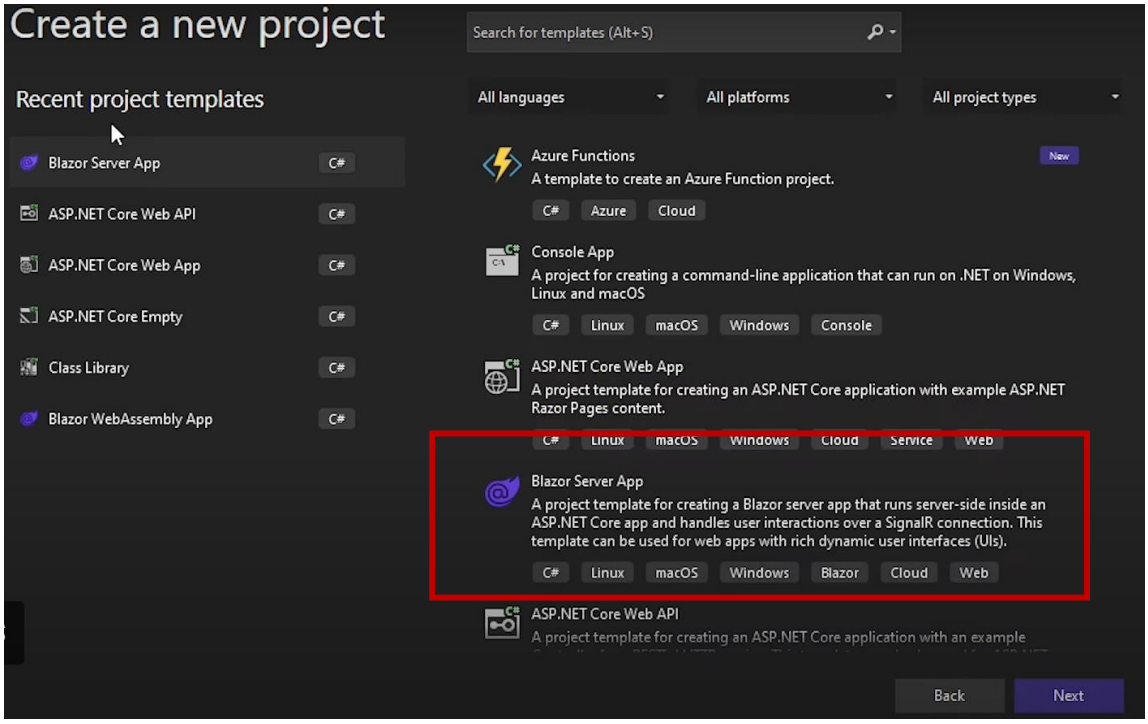
Authentication & Authorization with OKTA in .NET 6 Blazor Server

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 –

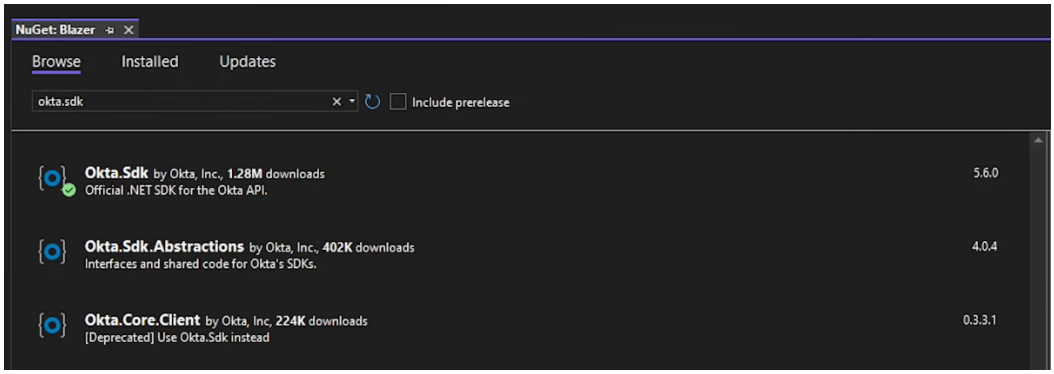
- Under Properties > launchSettings.json change the ports to 5001 (https) & 5000 (http).
- Here's what the Properties/launchSettings.json should look like

```
launchSettings.json*
Schema: https://json.schemastore.org/launchsettings.json
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:17549",
      "sslPort": 44325
    }
  },
  "profiles": {
    "Blazer": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

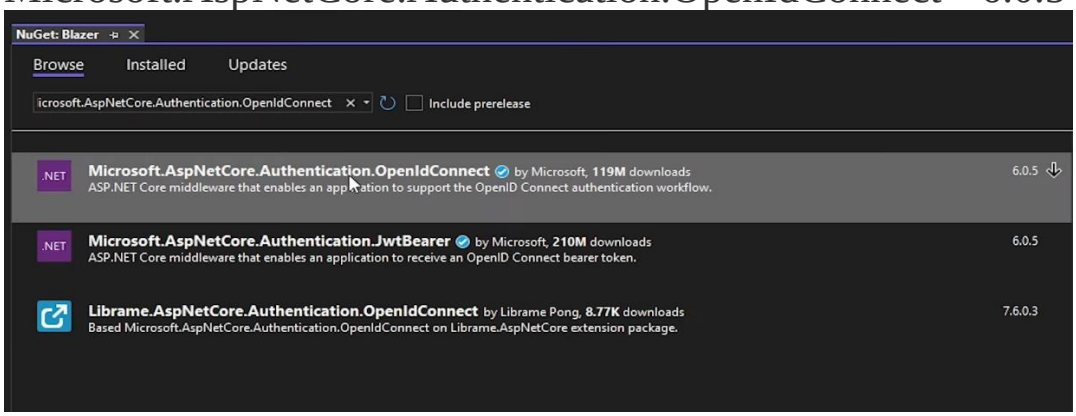
Install Dependencies

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

- Okta.Sdk --version 5.6.0

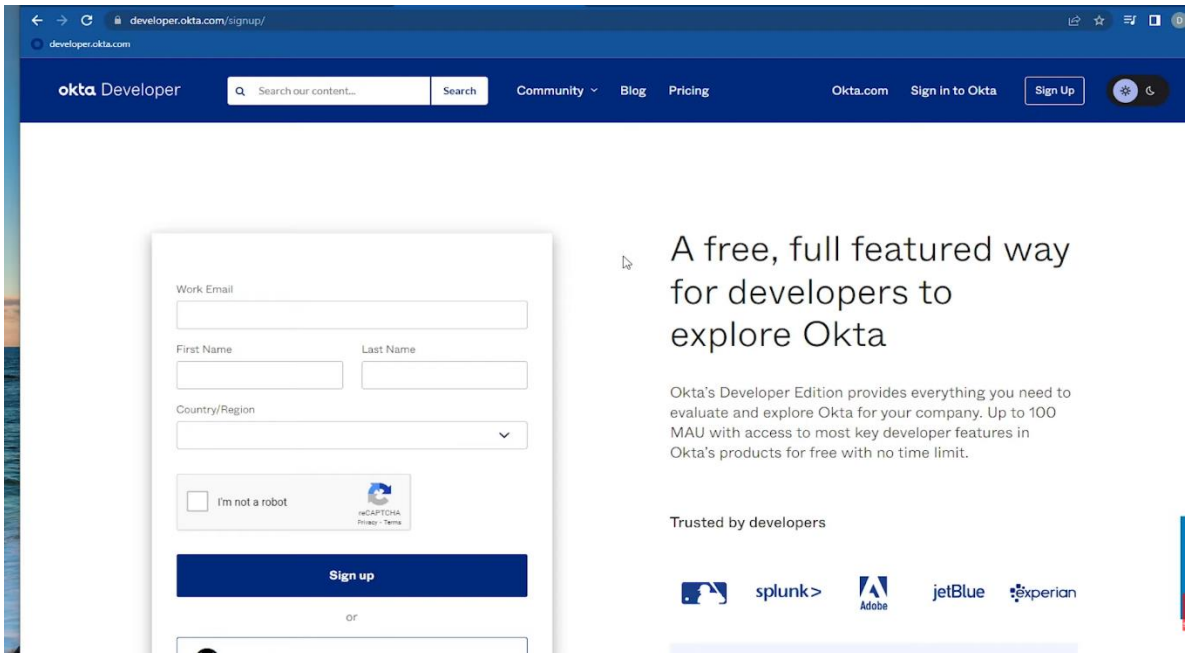


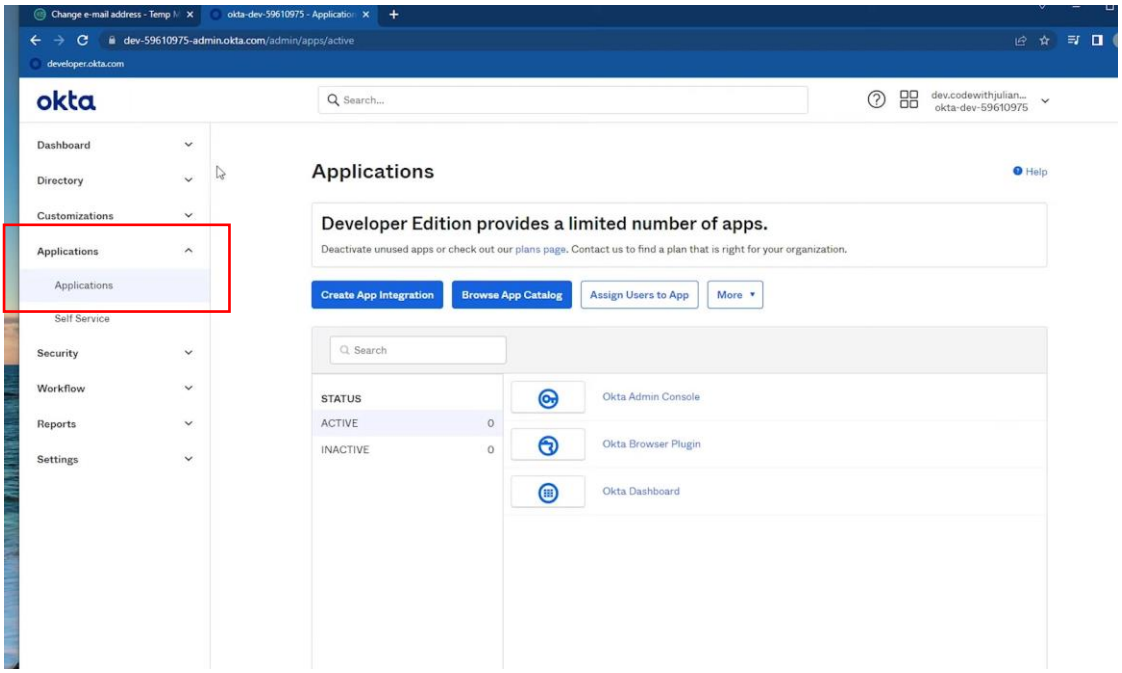
- Microsoft.AspNetCore.Authentication.OpenIdConnect --6.0.5



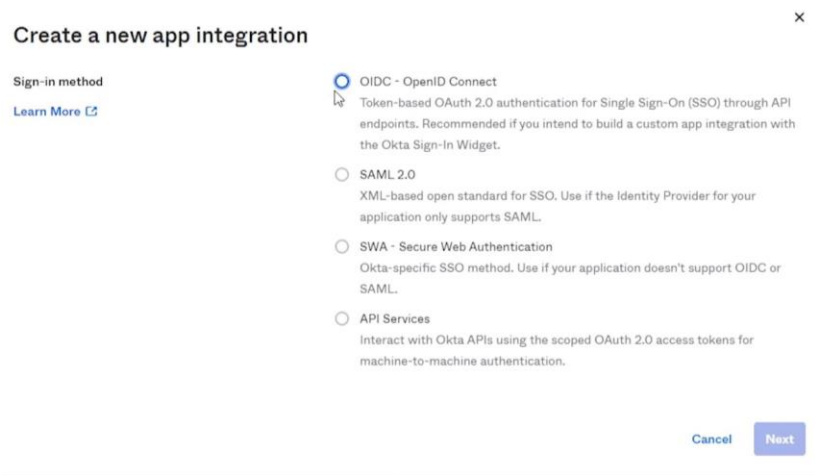
Create a new Application in Okta

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

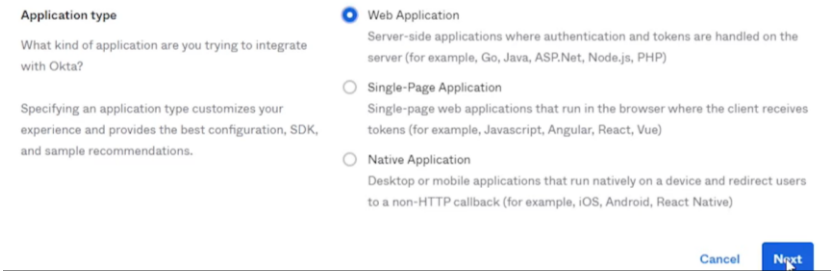




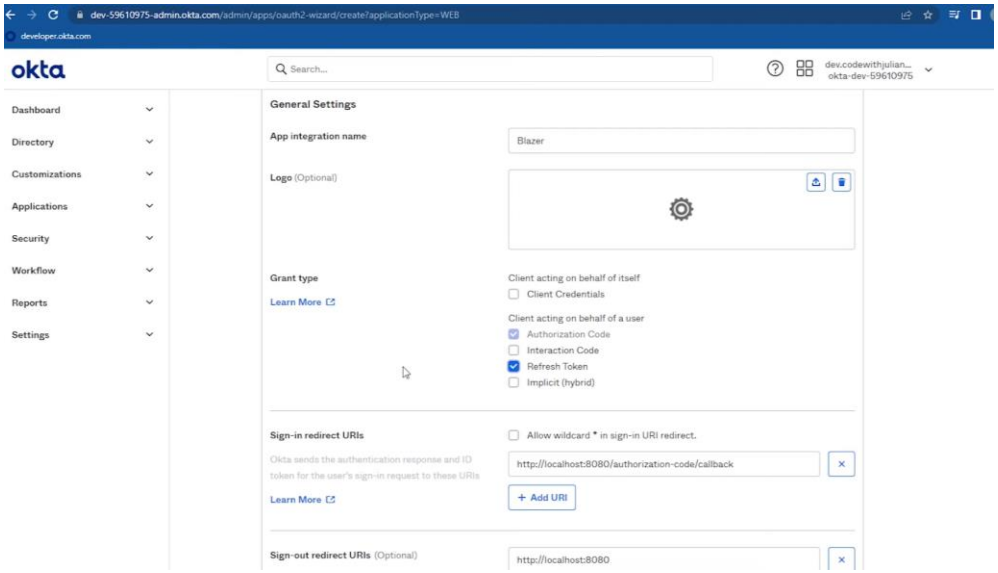
- Sign-in method: OIDC - OpenID Connect



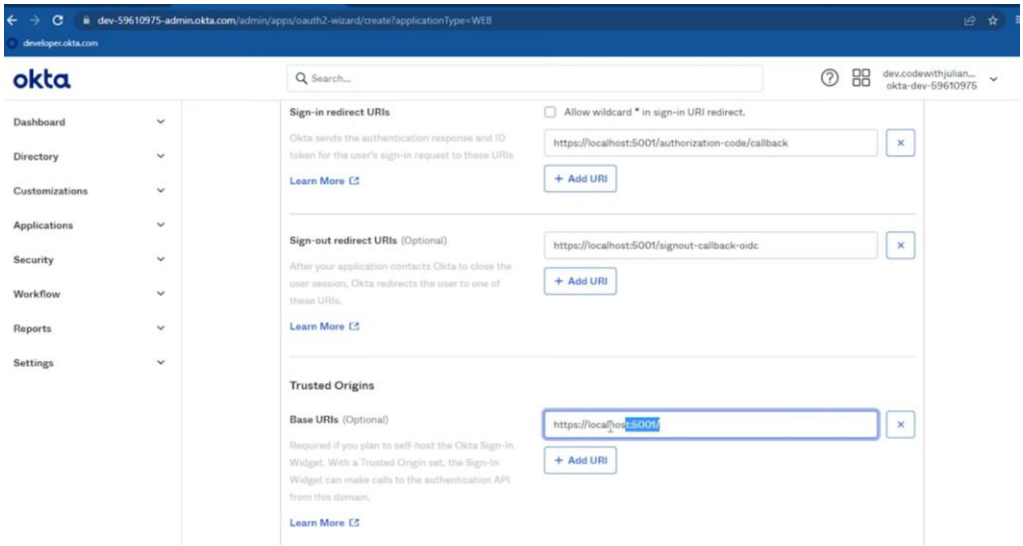
- Application type: Web Application



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



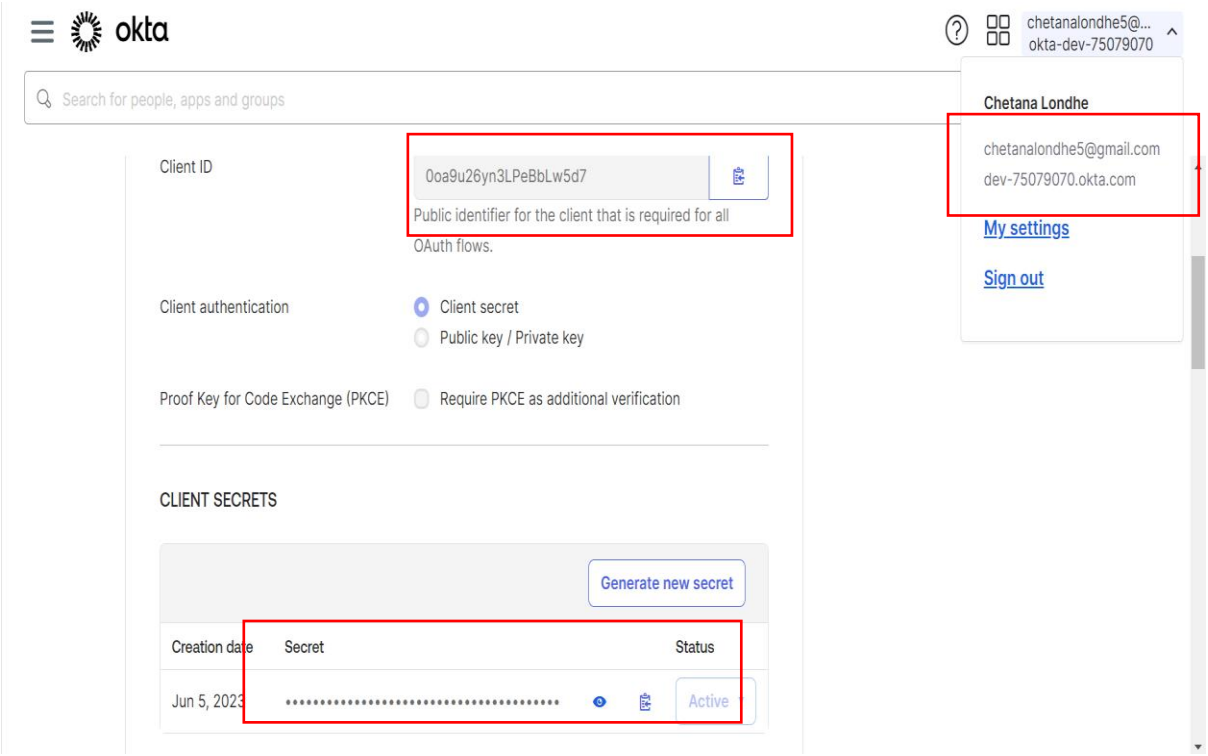
- Sign-in redirect URIs: https://localhost:5001/authorization-code/callback
- Sign-out redirect URIs : https://localhost:5001/signout-callback-oidc
- Base URIs: https://localhost:5001/



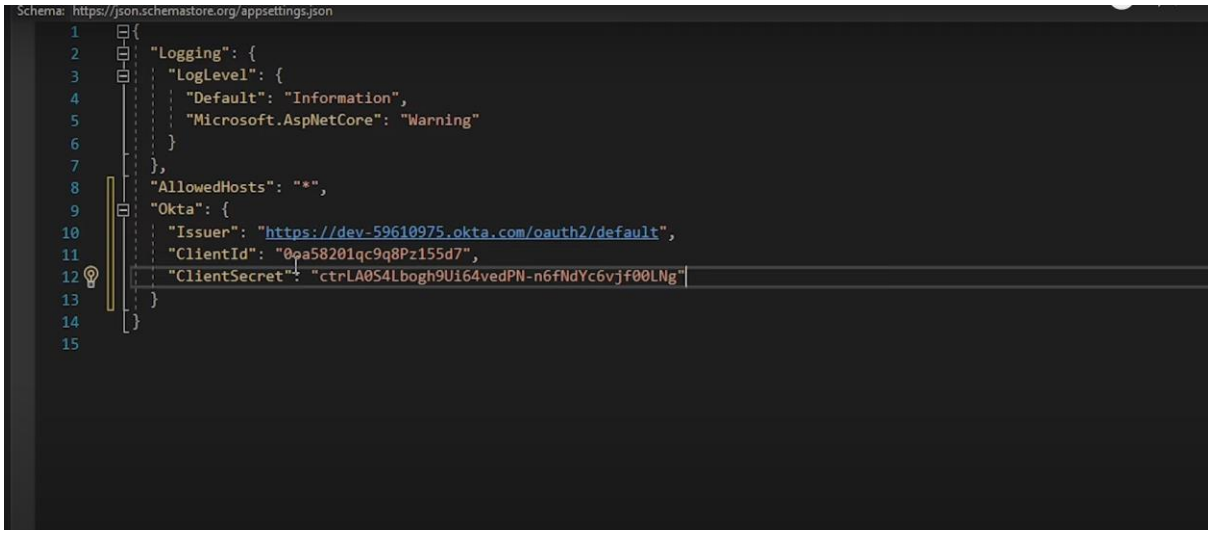
- The Authorization Code Flow an OAuth 2.0 type of grant that allows the client app to swap an authorization code for an Access Token (or JWT).
- The client sends a request up to the Authorization Server (Okta in this case) for an authorization code.
- Then swaps that for an Access Token that can be 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*
Blazer
using Blazer.Data;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();

builder.Services.AddAuthentication(authOptions =>
{
    authOptions.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;
});

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://a
    app.UseHsts();
}
```

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

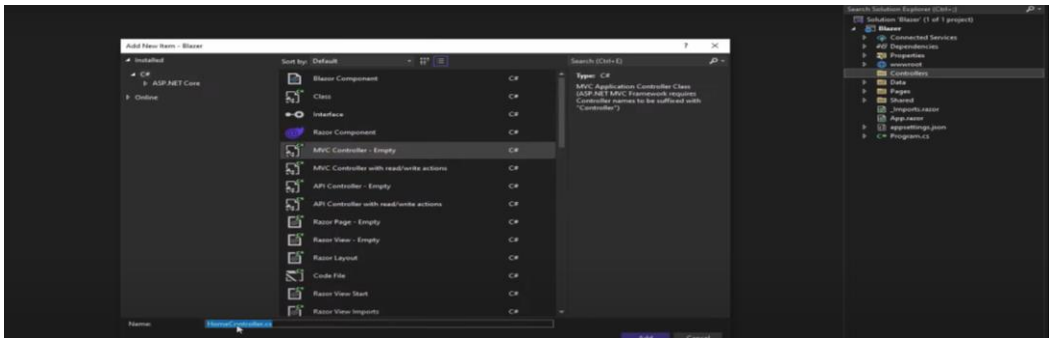
```
Program.cs*   appsettings.json
Blazer

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();

builder.Services.AddAuthentication(authOptions =>
{
    authOptions.DefaultAuthenticateScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    authOptions.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    authOptions.DefaultSignOutScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    authOptions.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
}).AddOpenIdConnect(oidcOptions => {
    oidcOptions.ClientId = builder.Configuration["Okta:ClientId"];
    oidcOptions.ClientSecret = builder.Configuration["Okta:ClientSecret"];
    oidcOptions.CallbackPath = "/authorization-code/callback";
    oidcOptions.Authority = builder.Configuration["Okta:Issuer"];
    oidcOptions.ResponseType = "code";
    oidcOptions.SaveTokens = true;
    oidcOptions.Scope.Add("openid");
    oidcOptions.Scope.Add("profile");
    oidcOptions.TokenValidationParameters.ValidateIssuer = false;
    oidcOptions.TokenValidationParameters.NameClaimType = "name";
}).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:

```
LoginController.cs*
Blazer                                  Blazer.Controllers.LoginController                    Login(string
1   using Microsoft.AspNetCore.Authentication;
2   using Microsoft.AspNetCore.Mvc;
3
4   namespace Blazer.Controllers
5   {
        0 references
6       public class LoginController : Controller
7       {
8           [HttpGet("Login")]
            0 references
9           public IActionResult Login([FromQuery] string returnUrl)
10          {
11              var redirectUri = returnUrl is null ? Url.Content("~/") : "/" + returnUrl;
12
13              if (User.Identity.IsAuthenticated)
14              {
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      }
24  }
```

- Logout Method:

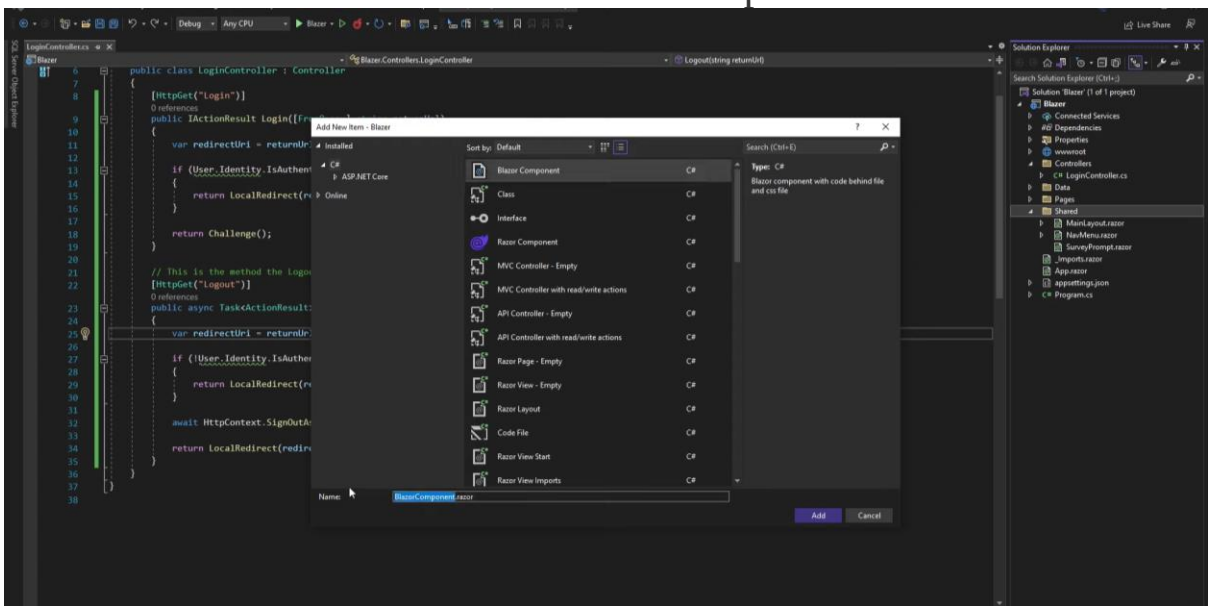
```
15                  return LocalRedirect(redirectUri);
16              }
17
18              return Challenge();
19          }
20
21          // This is the method the Logout button should get to when clicked.
22          [HttpGet("Logout")]
            0 references
23          public async Task<ActionResult>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

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

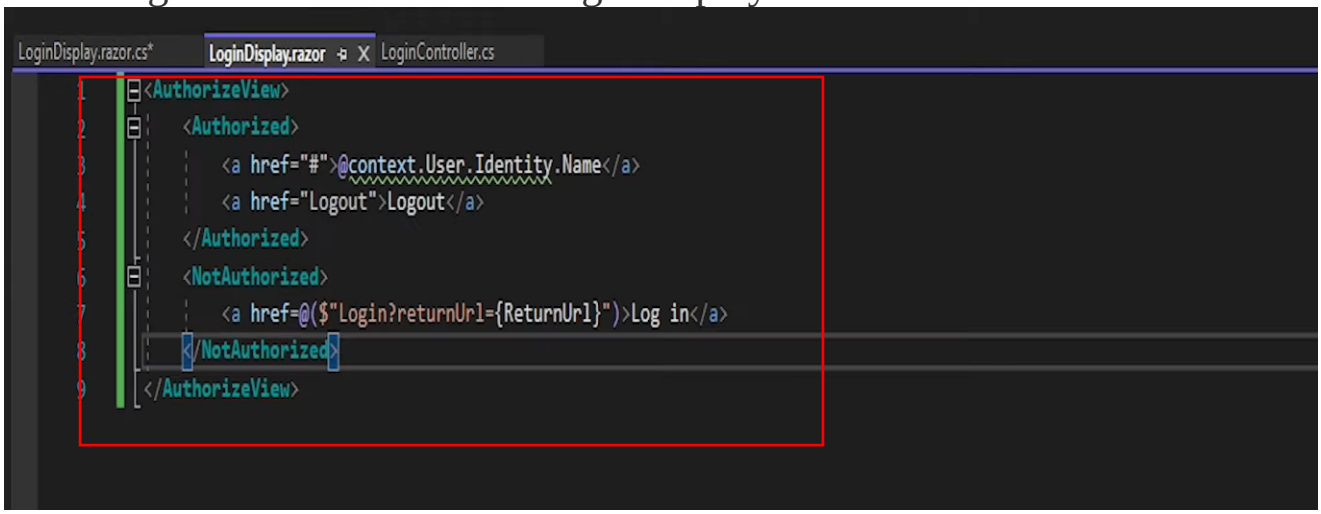
```
Blazer                                  Blazer.Controllers.LoginController                    Logout(string returnUrl)
6       public class LoginController : Controller
7       {
8           [HttpGet("Login")]
            0 references
9           public IActionResult Login([FromQuery] string returnUrl)
10          {
11              var redirectUri = returnUrl is null ? Url.Content("~/") : "/" + returnUrl;
12
13              if (User.Identity.IsAuthenticated)
14              {
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")]
            0 references
23          public async Task<ActionResult> Logout([FromQuery] string returnUrl)
24          {
25              var redirectUri = returnUrl is null ? Url.Content("~/") : "/" + returnUrl;
26
27              if (!User.Identity.IsAuthenticated)
28              {
29                  return LocalRedirect(redirectUri);
30              }
31
32              await HttpContext.SignOutAsync();
33
34              return LocalRedirect(redirectUri);
35          }
36      }
```

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 {
       1 reference
11     public partial class LoginDisplay
12     {
           2 references
13         [Inject] public NavigationManager Navigation { get; set; }
           2 references
14         [Parameter] public string ReturnUrl { get; set; }
15
           0 references
16         protected override async Task OnInitializedAsync()
17         {
18             ReturnUrl = Navigation.ToBaseRelativePath(Navigation.Uri);
19         }
20     }
21 }
22
```

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 used to redirect them to the login (Login endpoint form 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:

```
LoginDisplay.razor   LoginController.cs   RedirectToLogin.razor.cs   RedirectToLogin.razor
Blazer                                    Blazer.Shared.RedirectToLogin                    OnInitializedA
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  {
        1 reference
11      public partial class RedirectToLogin
12      {
            3 references
13          [Inject] public NavigationManager Navigation { get; set; }
14
            0 references
15          protected override async Task OnInitializedAsync()
16          {
17              var returnUrl = Navigation.ToBaseRelativePath(Navigation.Uri);
18
19              Navigation.NavigateTo($"Login?returnUrl={returnUrl}", true);
20          }
21      }
22  }
```

Setup the App.razor

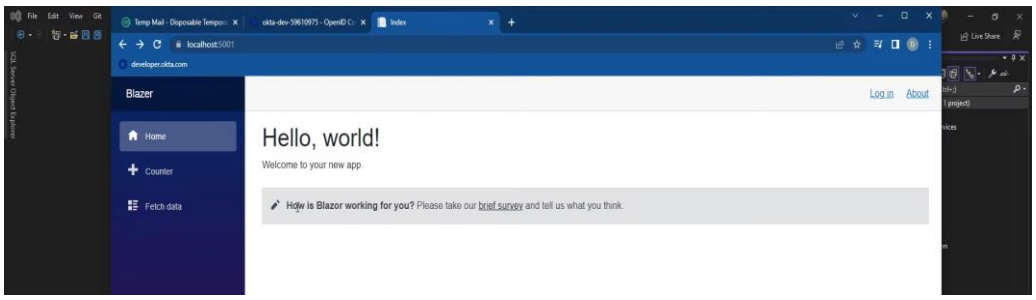
- Here's where we'll enable the authentication state within our project. This is done in App.razor and we must wrap the <Router> component in two components: <CascadingValue> this is used to expose the AccessToken to all of our components. And the <CascadingAuthenticationState> which is the one to enable 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
23 </CascadingValue>
24
25 @code{
26     [Parameter] public string AccessToken { get; set; }
27 }
```

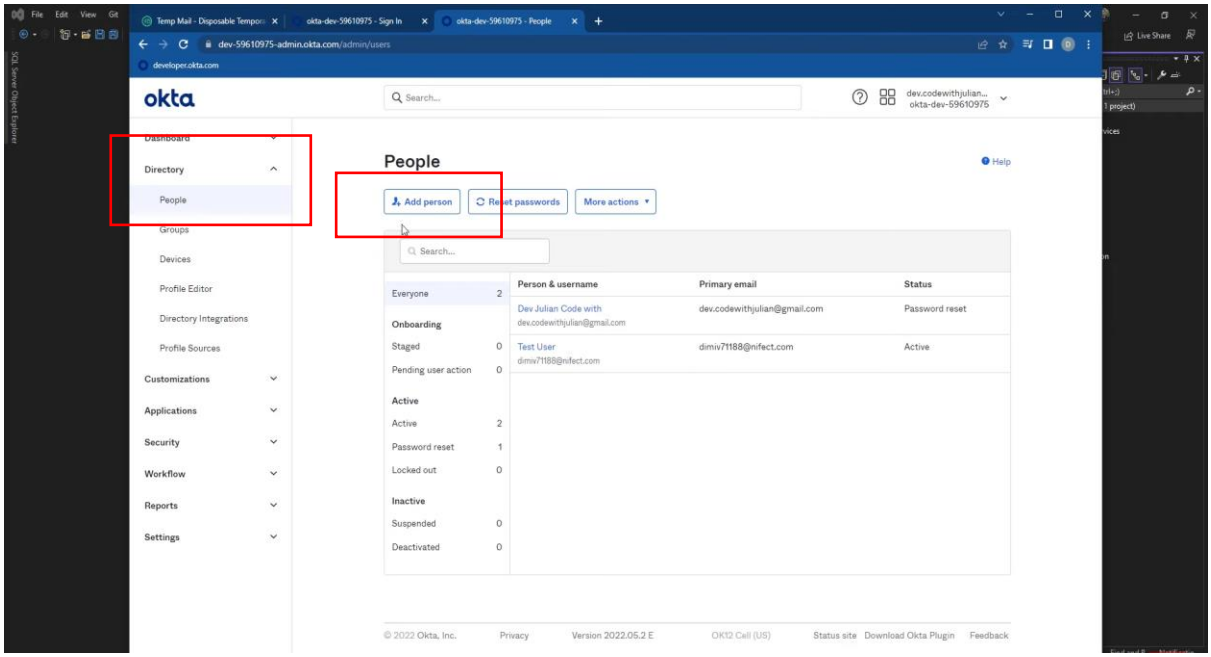
Run the app and test it

- 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.

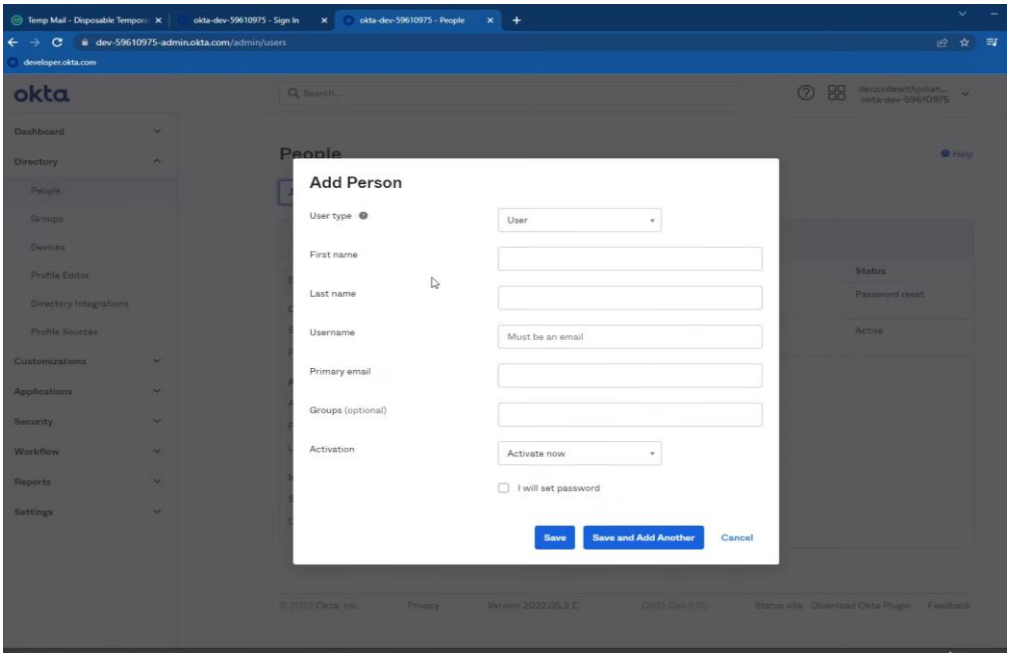


Add a New User to our Application

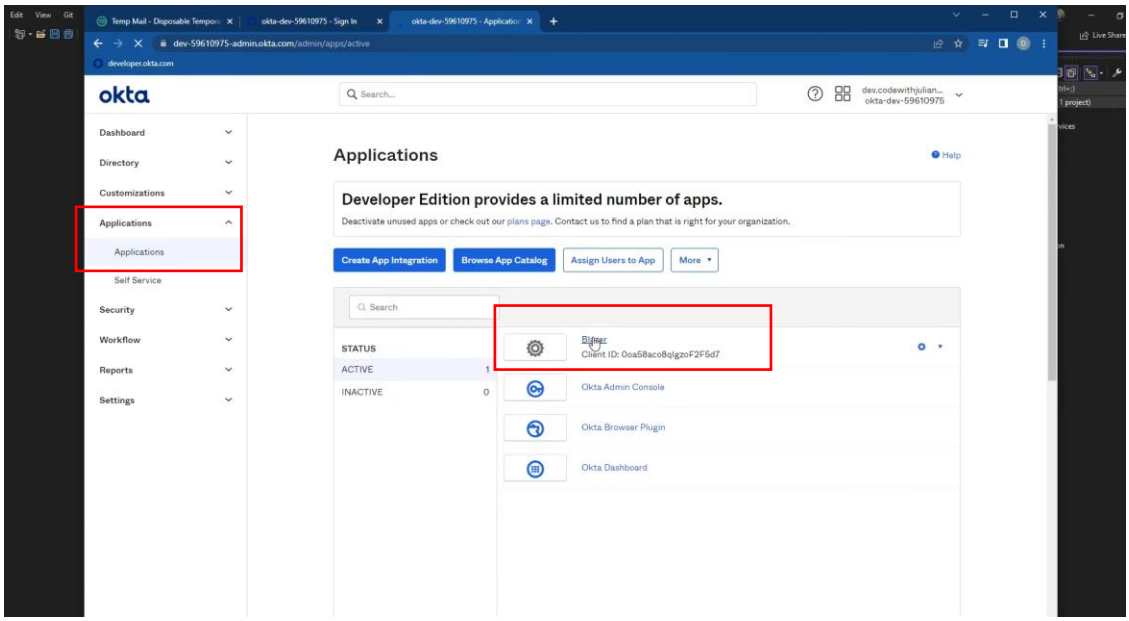
- Go to Directory >People.And then Add Person.



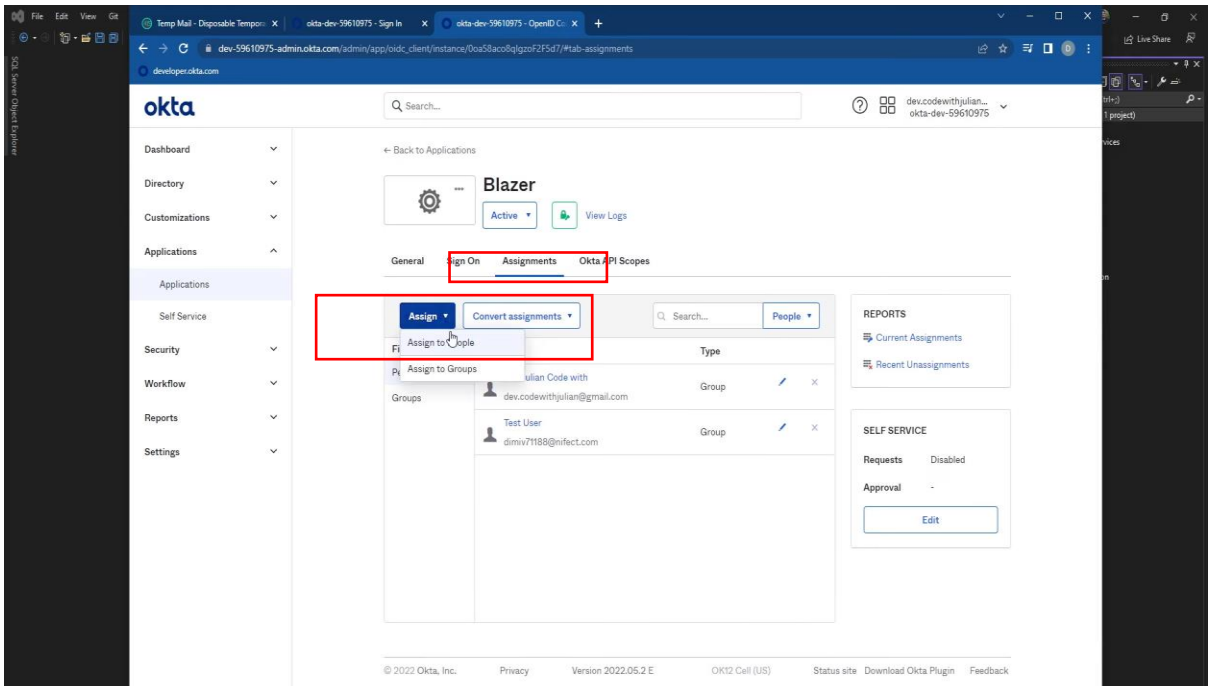
- Add details of the user.



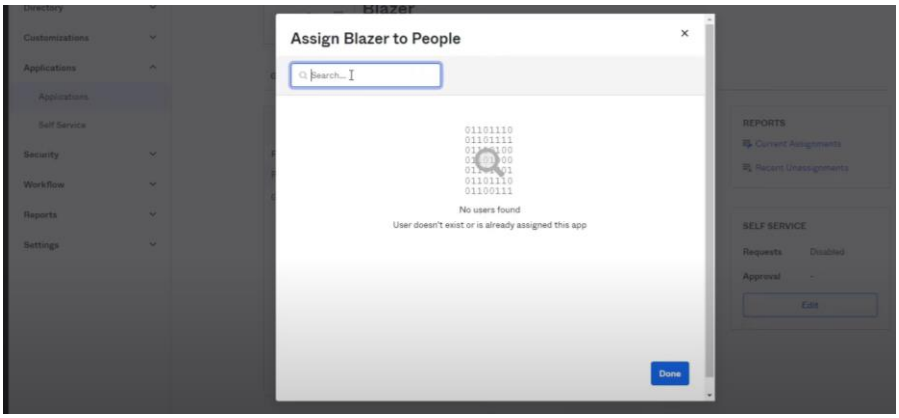
- To assign user to our application go to Application >Blazer



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

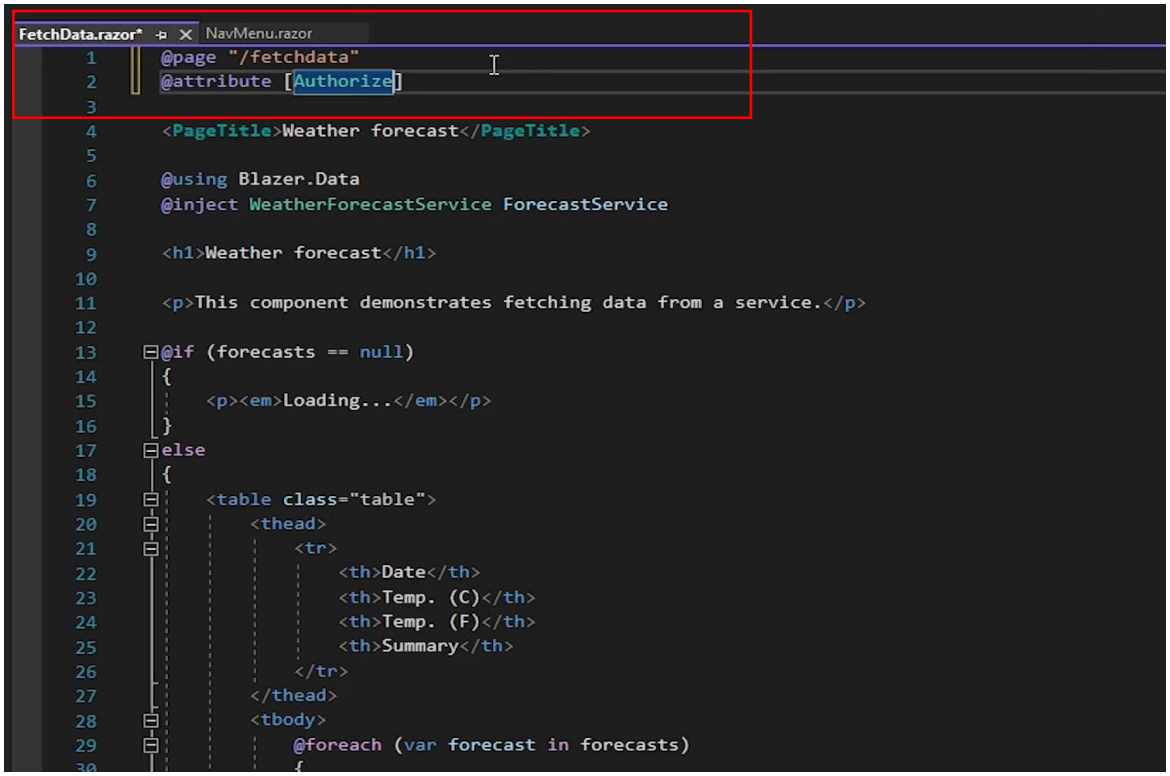


- Search for the user in Assign Blazer to people and click on Done, which will allow that user to login to that application



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.



- And to hide the menu button from Shared/NavMenu.razor, just wrap it in an <AuthorizeView> component, like so:

NavMenu.razor*

```
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">Blazer</a>
        <button title="Navigation menu" class="navbar-toggler" @onclick="ToggleNavMenu">
            <span class="navbar-toggler-icon"></span>
        </button>
    </div>
</div>

<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="counter">
                <span class="oi oi-plus" aria-hidden="true"></span> Counter
            </NavLink>
        </div>
        <div class="nav-item px-3">
            <AuthorizeView>
            <NavLink class="nav-link" href="fetchdata">
                <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data
            </NavLink>
            </AuthorizeView>
        </div>
    </nav>
</div>
```