Blazor Server

Securing Blazor applications

2023

Introduction

- When allowing users to sign in to an application, there are two processes that must happen:
  - Authentication—The process of determining if someone is who they claim to be
  - Authorization—The process of checking if someone has the rights to access a resource

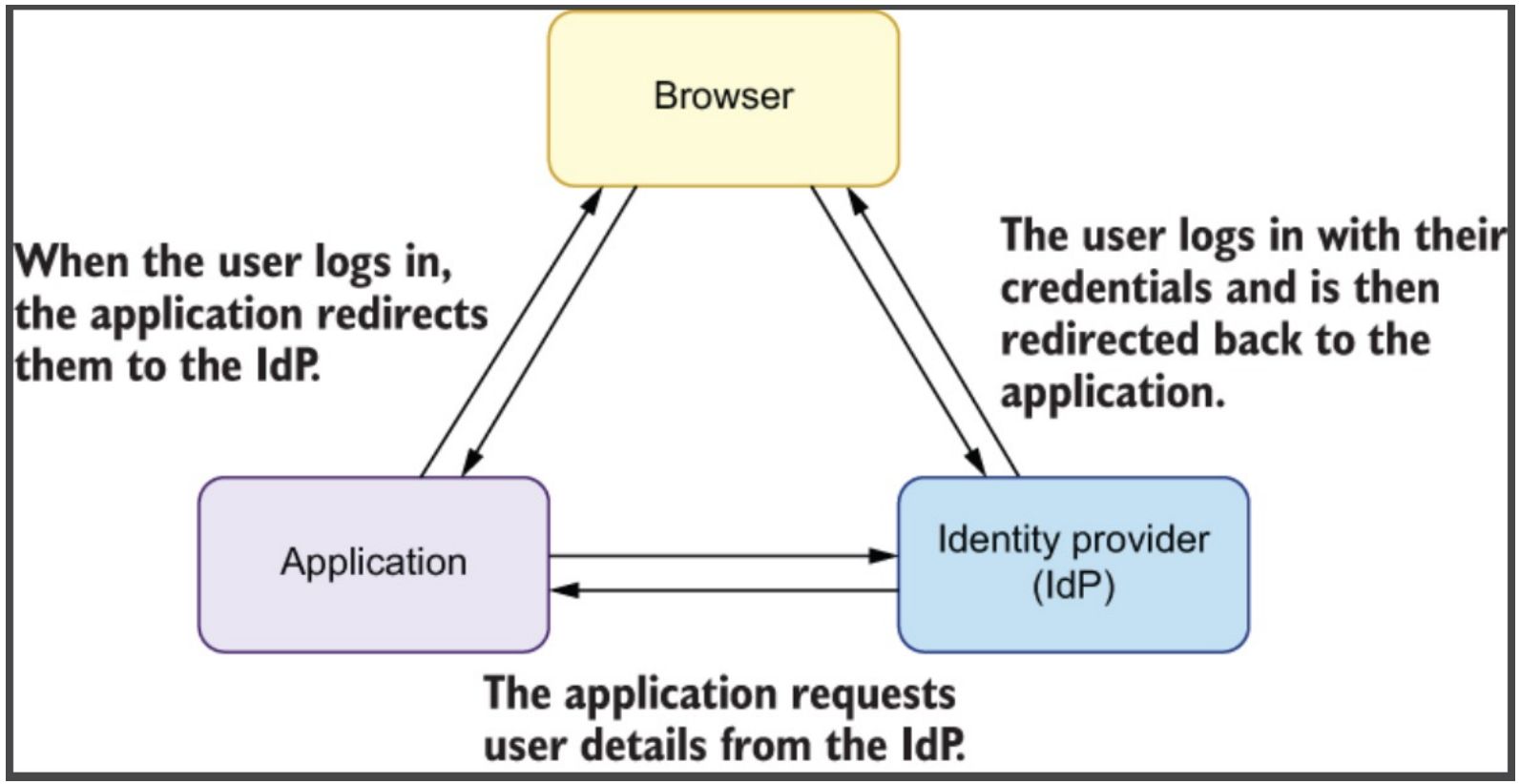
Introduction

- We will secure the app to allow only users who are logged in to the application to create cards.
- We will also restrict users to be allowed to edit only their own card.
- Finally, we will build on this functionality by adding roles.
- Roles are a way of grouping users.
- We will create an administrator role and allow any user in that role to edit any card in the system.

Introduction

- To enable this functionality, we'll need an identity provider (IdP).
- An IdP is responsible for storing and managing a user's digital identity.
- They also offer a way to control access to resources such as APIs, applications, or services.
- Some large IdPs you would have heard of are Microsoft, Google, Facebook, and Twitter.
- If you've ever used your credentials from one of these to sign in to a third-party service, you've used an IdP

Introduction



Integrating with an identity provider: Auth0

- Authentication and authorization are complex issues, and using an IdP abstracts a large chunk of the complexity—and responsibility—away from us
- When using an IdP, we delegate the sign-up and login process to it.
- Users are forwarded from our application to the IdP, where they log in.
- They are then sent back to our application once that process is complete.
- The specifics of what happens depends on what flow (https://auth0.com/docs/authorization/flows) is used

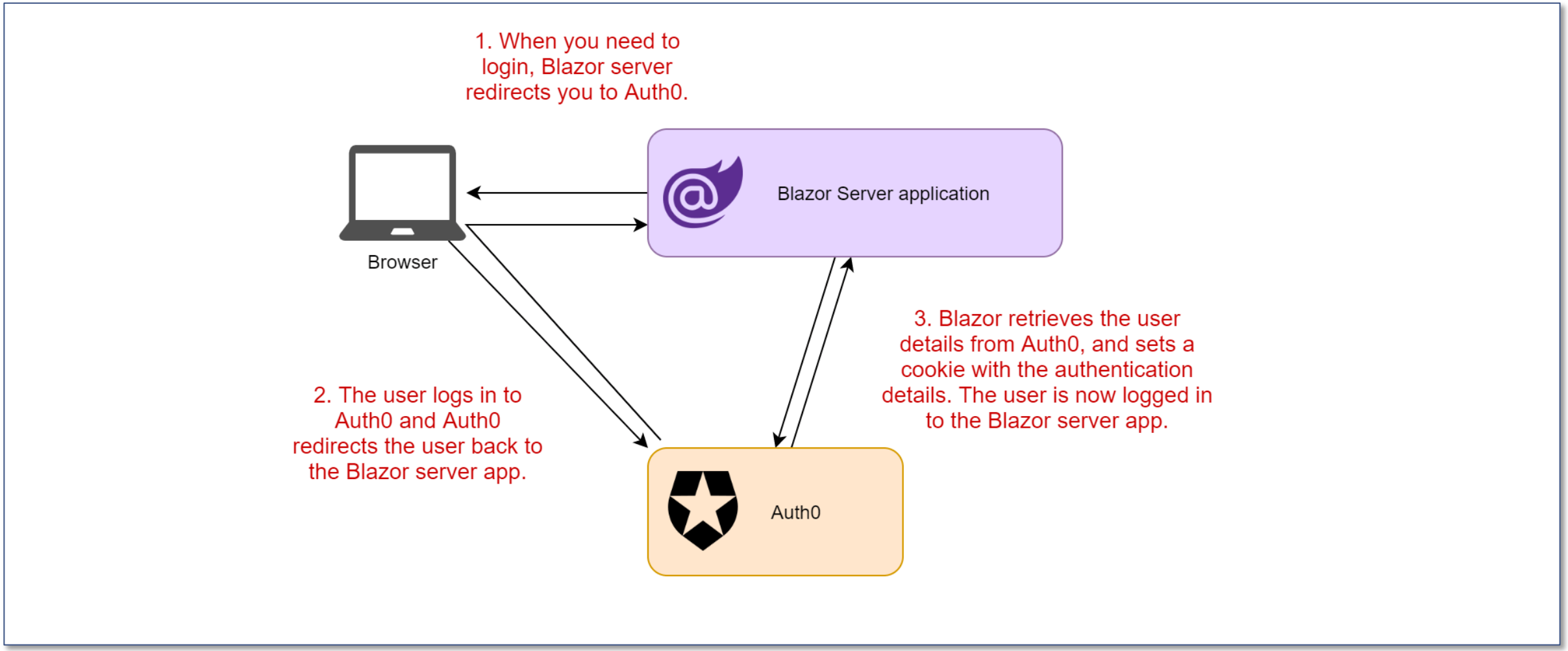
Integrating with an identity provider: Auth0

- Auth0 (https://auth0.com) is an identity provider that you can use to provide user management and authentication for your applications.
- By using an external login provider such as Auto0 (or Azure AD B2C), you delegate responsibility for the "login process" to a third party.
- That means you get benefits such as "paswordless" login, compromised password checks, social logins, and WebAuthn support.
- More importantly, you don't have to worry about losing user passwords, as you don't have them!

Integrating with an identity provider: Auth0

- Using an external identity provider (such as Auth0) is relatively simple with ASP.NET Core, as long as the provider implements OpenId Connect (which most do).
- With this approach, whenever you need to login to your app, you redirect the user to Auth0 to do the actual sign-in.
- Once the user has signed in, they're redirected to a callback page in your app.

Integrating with an identity provider: Auth0



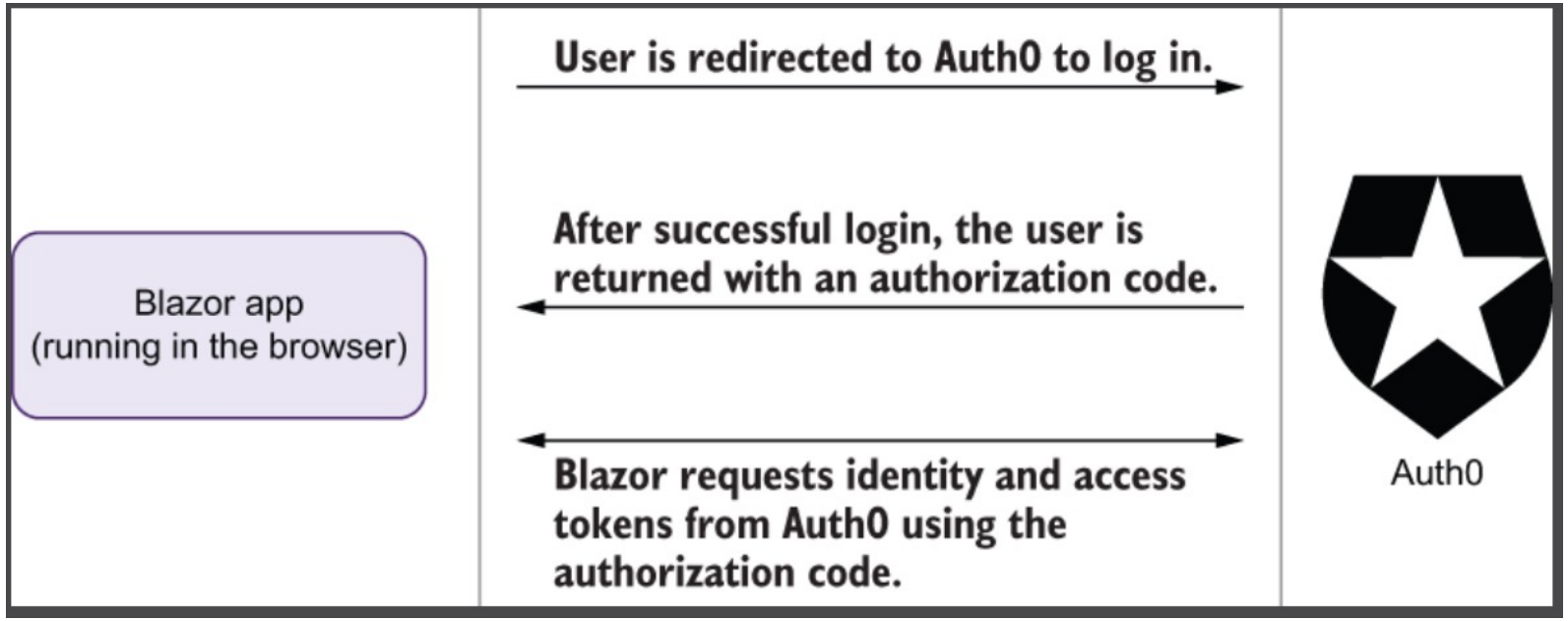
https://andrewlock.net/adding-authentication-to-a-blazor-server-app-using-auth0/

Integrating with an identity provider: Auth0

- We are logging using a Blazor application via Auth0 using the Authorization Code Flow with PKCE (Proof of Key for Code Exchange).
- When they successfully complete the login, they are returned to the application with an authorization code.
- Blazor then makes a call to Auth0 with the code and requests an access token and an ID token, which are kept in the browser's session storage.

Integrating with an identity provider: Auth0

Process illustration using Blazor web assembly

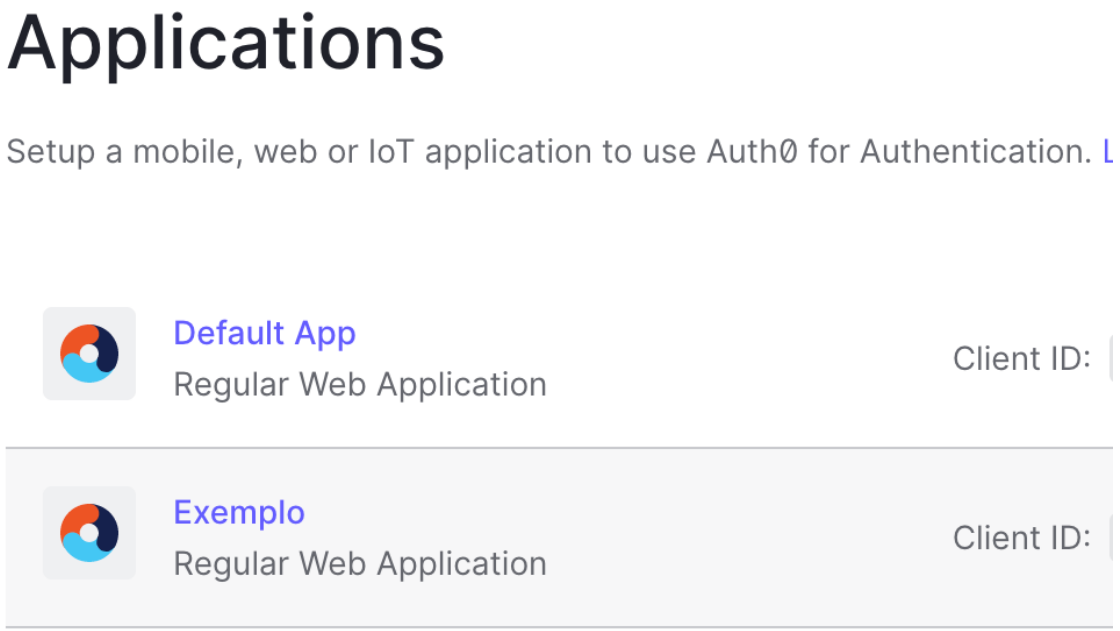


Integrating with an identity provider: Auth0

- Using Blazor Server, the process is a little different.
- When the user is redirected back from the IdP to the Blazor Server app, an additional request is made to Auth0 for the user details, which are then saved into a cookie.
- There are no access tokens or ID tokens in this scenario

Registering applications with Auth0

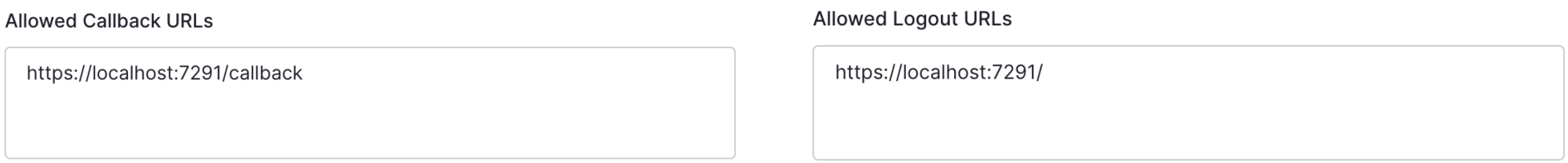
- Create a free account in: https://auth0.com
- The first step to secure your Blazor Server application is to access the Auth0 Dashboard to register your Auth0 application.



Registering applications with Auth0

- Once in the dashboard, move to the Applications section and follow these steps:
- Click on Create Application.
- Provide a friendly name for your application (for example, Quiz Blazor Server App) and choose Regular Web Applications as an application type.
- Finally, click the Create button.
- These steps make Auth0 aware of your Blazor application and will allow you to control access.

Registering applications with Auth0



- After the application has been registered, move to the Settings tab and take note of your Auth0 domain and client id.
- Then, in the same form, assign the value:
  - https://localhost:<YOUR_PORT_NUMBER>/callback to the Allowed Callback URLs field
  - The value https://localhost:<YOUR_PORT_NUMBER>/ to the Allowed Logout URLs field.
- Replace the <YOUR_PORT_NUMBER> placeholder with the actual port number assigned to your application.

Customizing tokens from Auth0

- As part of the login process, ASP.NET Core constructs a type called ClaimsPrincipal, which contains an Identity property that is of type ClaimsIdentity.
- This represents a user identity for an application.
- One of the properties we'll be accessing is called Name, which we'll do like this: User.Identity.Name.

Customizing tokens from Auth0

- This is going to be automatically populated with the user's email address in our Blazor app
- However, doesn't contain this claim by default.
- This means that we wouldn't have access to the user's email to know what cards they own.

Customizing tokens from Auth0

- Claims are key-value pairs that represent information about the user issued by an identity provider.
- There are a set of standard claims defined (http://mng.bz/XZ0E) , but custom claims can also be added.
- Examples of standard claims are given_ name, family_name, and website.
- Claims are used by an application to decide if a user can perform a certain task or access a certain feature.
- This is known as claims-based authorization.

Customizing tokens from Auth0

- Auth0 allows us to customize the claims that are returned for each token.
- From the main menu in Auth0, select Auth Pipeline > Rules.
- Create button na Empty Rule option.
- Name the rule Customize Tokens.

Customizing tokens from Auth0

- Add the following code:

```
function (user, context, callback) {
    context.idToken.name = user.email;
    callback(null, user, context);
}
```

https://auth0.com/blog/id-token-access-token-what-is-the-difference/

Configuring your Blazor application

- Open the appsettings.json file in the root folder of your Blazor Server project and add the following:

```
"Auth0": {
    "Domain": "YOUR_AUTH0_DOMAIN",
    "ClientId": "YOUR_CLIENT_ID"
}
```

Integrating with Auth0

- Now, install the Auth0 ASP.NET Core Authentication SDK by running the following command in your terminal window:
  - dotnet add package Auth0.AspNetCore.Authentication
- The Auth0 ASP.NET Core SDK lets you easily integrate OpenID Connect-based authentication in your app without dealing with all its low-level details.

Integrating with Auth0

- After the installation is complete, open the Program.cs file and change its content as follows:

```
(...)
using Auth0.AspNetCore.Authentication; // new code

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuth0WebAppAuthentication(options => { // new code
options.Domain = builder.Configuration["Auth0:Domain"]; // new code
options.ClientId = builder.Configuration["Auth0:ClientId"]; // new code
});

builder.Services.AddRazorPages();
(...)
var app = builder.Build();

(...)
app.UseHttpsRedirection(); // new code

(...)
app.UseAuthentication(); // new code
app.UseAuthorization(); // new code
(...)
```

Integrating with Auth0

- Following the highlighted code, you added a reference to the Auth0.AspNetCore.Authentication namespace at the beginning of the file.
- Then you invoked the AddAuth0WebAppAuthentication() method with the Auth0 domain and client id as arguments.
- These Auth0 configuration parameters are taken from the appsettings.json configuration file.
- Finally, you called the UseAuthentication() and UseAuthorization() methods to enable the authentication and authorization middleware.

Securing the server side

- In order to prevent unauthorized users from accessing the server-side functionalities of your application, you need to protect them.
- So, open the Index.razor file in the Pages folder and add the Authorize attribute
- Example:

```
@page "/counter"

@attribute [Authorize]
(...)
```

Creating login and logout endpoints

- In the Blazor Server hosting model, the communication between the client side and the server side does not occur over HTTP, but through SignalR.
- Since Auth0 uses standard protocols like OpenID and OAuth that rely on HTTP, you need to provide a way to bring those protocols on Blazor.
- To solve this issue, you are going to create two endpoints, /login and /logout, that redirect requests for login and for logout to Auth0.

Creating login and logout endpoints

- In the pages folder create:
  - Login.cshtml.cs
  - Login.cshtml
  - Logout.cshtml.cs
  - Logout.cshtml

Creating login and logout endpoints

- Login.cshtml.cs

```
using Microsoft.AspNetCore.Authentication;
using Auth0.AspNetCore.Authentication;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace app.Features.Auth{
   public class LoginModel : PageModel{
     public async Task OnGet(string redirectUri){
        var authenticationProperties = new
LoginAuthenticationPropertiesBuilder().WithRedirectUri(redirectUri).Build();

        await HttpContext.ChallengeAsync(Auth0Constants.AuthenticationScheme,
authenticationProperties);
     }
  }
}
```

Creating login and logout endpoints

- Login.cshtml

```
@page
@model app.Features.Auth.LoginModel
@{
}
```

Creating login and logout endpoints

- Logout.cshtml.cs

```
(...)
using Microsoft.AspNetCore.Authentication.Cookies;

namespace app.Features.Auth{
  public class LogoutModel : PageModel{
  [Authorize]
  public async Task OnGet(){
    var authenticationProperties = new
LogoutAuthenticationPropertiesBuilder().WithRedirectUri("/").Build();

    await HttpContext.SignOutAsync(Auth0Constants.AuthenticationScheme, authenticationProperties);
    await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
  }
 }
}
```

Securing the client side

- Now, you have to secure the client side of your Blazor application, so that the users see different content when they are logged in or not.
- Open the App.razor file in the root folder of the project and replace its content:

```
<CascadingAuthenticationState>
  <Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
      <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)">
        <Authorizing><p>Determining session state, please wait...</p> </Authorizing>
       <NotAuthorized> <h1>Sorry</h1> <p>You're not authorized to reach this page. You need to log in.</p> </NotAuthorized>
     < /AuthorizeRouteView>
     <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
   <NotFound> <PageTitle>Not found</PageTitle>
        <LayoutView Layout="@typeof(MainLayout)">
            <p role="alert">Sorry, there's nothing at this address.</p>
        </LayoutView>
  </NotFound> </Router>
</CascadingAuthenticationState>
```

Securing the client side

- Here you are using the AuthorizeRouteView component, which displays the associated component only if the user is authorized.
- In practice, the content of the MainLayout component will be shown only to authorized users.
- If the user is not authorized, they will see the content wrapped by the NotAuthorized component.
- If the authorization is in progress, the user will see the content inside the Authorizing component.
- The CascadingAuthenticationState component will propagate the current authentication state to the inner components so that they can work on it consistently.

Securing the client side

- For testing purposes, put [Authorize] in Counter page, like this:

```
@page "/counter"
@inject AuthenticationStateProvider AuthState

@attribute [Authorize]

<PageTitle>Counter</PageTitle>
(...)
```

- Enter: <localhost>/counter and check!

Securing the client side

- You can also write some code and see the claims returned:

```
protected override async Task OnInitializedAsync(){
  var state = await AuthState.GetAuthenticationStateAsync();

  foreach(var claim in state.User.Claims){
    Console.WriteLine(claim.Type + " = " + claim.Value);
  }
}
```

Securing the client side

- Create a razor component for this purpose by adding an AccessControl.razor file in the Auth folder that you should create at root folder:

```
<div class="container text-end">
    <AuthorizeView>
        <Authorized>
            <div>
                Hello, @context.User.Identity!.Name
                <a href="logout"> Log out</a>
            </div>
        </Authorized>
        <NotAuthorized>
            <div>
                <a href="login?redirectUri=/">Log in/Sign up</a>
            </div>
        </NotAuthorized>
    </AuthorizeView>
</div>
```

Securing the client side

- This component uses the Authorized component to let the authorized users see the Log out link and the NotAuthorized component to let the unauthorized users access the Log in link.
- Both links point to the endpoints you created before.
- In particular, the Log in link specifies the home page as the URI where to redirect users after authentication

Securing the client side

- The final step is to put this component in the top bar of your Blazor application.
- So, replace the content of the MainLayout.razor file with the following content:

```
@using app.Features.Auth

<nav class="navbar mb-5 shadow">
  <a class="navbar-brand" href="/">
    <img src="/images/logo.png" width="200px" height="132px">
  </a>
</nav>

<AccessControl />
```

Securing the client side

- Now, your Blazor application is accessible just to authorized users.
- When users click the *Log in* link, they will be redirected to the Auth0 Universal Login page for authentication.
- After authentication completes, they will be back to the home page of your application.

Accessing the User Profile

- Once you add authentication to your Blazor Server application, you may need to access some information about the authenticated user, such as their name and picture.
- By default, the Auth0 ASP.NET Core Authentication SDK takes care of getting this information for you during the authentication process.

Accessing the User Profile

- Change the content of counter component:

```
@page "/counter"
(...)
<h1>Welcome, @Username!</h1>
You can only see this content if you're authenticated.
<br />
<img src="@Picture">

@code {
  private string Username = "Anonymous User";
  private string Picture = "";
  protected override async Task OnInitializedAsync(){
   var state = await AuthState.GetAuthenticationStateAsync();
   Username = state.User.Identity.Name ?? string.Empty;
   Picture = state.User.Claims.Where(c => c.Type.Equals("picture")).Select(c =>
c.Value).FirstOrDefault() ?? string.Empty;
   await base.OnInitializedAsync();
 }
}
```

Accessing the User Profile

- In this new version of the component, you injected the AuthenticationStateProvider, which provides you with information about the current authentication state.
- You get the actual authentication state in the code block by using its GetAuthenticationStateAsync() method.
- Thanks to the authentication state, you can extract the name and the picture of the current user ad assign them to the Username and Picture variables.
- These are the variables you use in the component's markup to customize this view.

Accessing the User Profile

- We are almost there for authenticating our Blazor app.
- We need https support in our app (if you initialize Project with no-https option it haven't).
- So, first create a trust self-signed certificate:
  - dotnet dev-certs https --trust

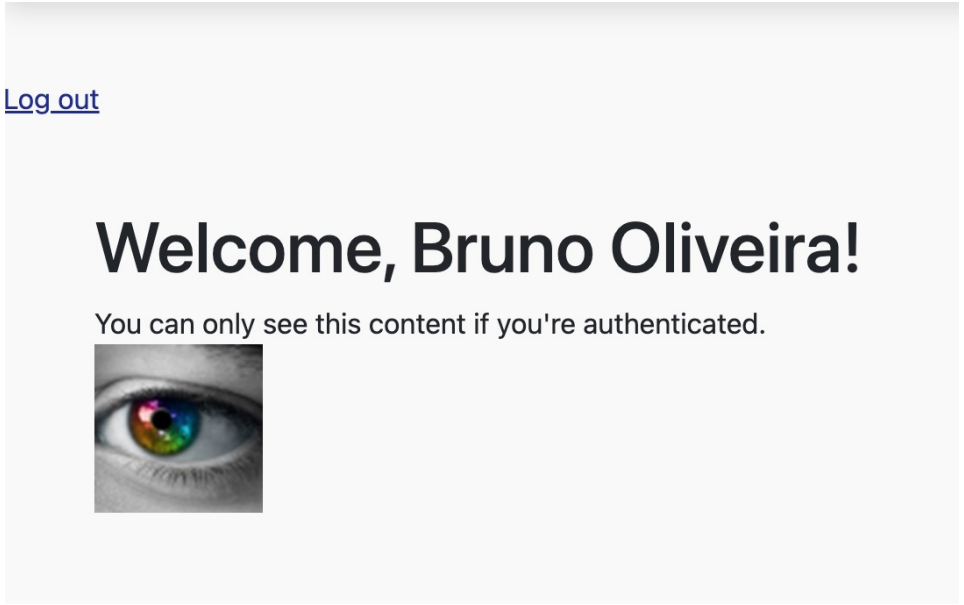
Accessing the User Profile

- Go to Properties/lauchSettings.json and add to applicationUrl field with a port inside 7000-7300 interval;
- Example:

```
"applicationUrl": "https://localhost:7291;http://localhost:5167",
```

Accessing the User Profile

- Test the application:



Updating the Home feature

- Because users can now sign to log in and out, we will make some changes regarding what they will see.
- Up until now, any anonymous user could create or edit a card.
- This needs to change.
- We want only logged-in users to be able to create cards, and we want only the owner of a card to be able to edit it.

Updating the Home feature

- In Features > Home > Shared, we will add an Owner property to Card.cs.

```
public string Owner { get; set; } = default!;
```

Updating the Home feature

- Now we can move up a folder and begin our work on the Index component.
- We're going to update the row template and make a small update to the OnInitializedAsync method

```
<RowTemplate>
    <th scope="col">@card.Name</th>
    (...)
    <td class="text-right">
        <button @onclick="@(() => HandleCardSelected(card))" title="View" class="btn btn-primary">
           (...)</button>
    <AuthorizeView>
    @if (card.Owner.Equals(context.User.Identity?.Name, StringComparison.OrdinalIgnoreCase)){
        <button @onclick="@(() => NavigationManager.NavigateTo($"/edit-card/{card._Id}"))" (...)
    </button>
    }
    </AuthorizeView>
    </td>
</RowTemplate>
```

The existing Edit button is wrapped in an AuthorizeView component.

If the card's owner and the logged-in user are the same, we display the Edit button.

Updating the Home feature

- While we're on the HomePage component, we will make one other change.
- The call to action at the top of the page needs updating.
- It contains a link to the Add Card form, which we want to display only to logged-in users.
- We'll use the AuthorizeView component to display different messages depending on whether the user is logged in or not

Updating the Home feature

```
<AuthorizeView>
    <Authorized>
        <div class="mb-4">
        <p class="font-italic text-center">Do you have an awesome card you'd like to share?
        <a href="add-card">Add ithere</a>.
        </p>
        </div>
    </Authorized>
    <NotAuthorized>
        <div class="mb-4">
        <p class="font-italic text-center">Do you have an awesome card
        you'd like to share? Please <a href="authentication/login">
        log in or sign up</a>.</p>
        </div>
    </NotAuthorized>
</AuthorizeView>
```

Updating the Home feature

- We now need to make a small update to the SearchPage component before moving on to the Card component.
- In the SearchPage component, we need to add the same mapping between the owner properties we did on the HomePage.
- In the OnInitializedAsync method, add the following line to the addCards variable declaration: `Owner = x.Owner`

Updating the Home feature

- In the Card component, we're wrapping the existing button in an AuthorizeView component.
- We then use the context variable to check the current user's email against the owner of the card.
- If they match, the Edit button is displayed.

```
<AuthorizeView>
   @if (card.Owner.Equals(context.User.Identity?.Name, StringComparison.OrdinalIgnoreCase)){
     <button class="btn btn-outline-secondary float-right" title="Edit" @onclick="@(() =>
NavigationManager.NavigateTo($"/edit-card/{card._Id}"))">
     (...)
     </button>
}
</AuthorizeView>
```

Prevent unauthorized users accessing a page

- Restricting access to pages is a common requirement when building applications.
- We can use the @attributes directive to apply the [Authorize] attribute to a page component.
- This works in tandem with the AuthorizeRouteView component in the router to restrict page-level access.

Prevent unauthorized users accessing a page

- By default, the Router contains a component called RouteView.
- This component's job is to render the requested page based on the URL.
- The limitation of this component is that it has no awareness of security.
- This is where the AuthorizeRouteView component comes in.

Prevent unauthorized users accessing a page

- It’s a drop-in replacement for the RouteView, and it understands Blazor’s security features.
- Before navigating to a page, it will check that the user is authorized to view that page by checking for the Authorize attribute.
- It will then either load the page or render any markup specified in the NotAuthorized template it exposes.

Prevent unauthorized users accessing a page

- We need to protect our Add and Edit Card pages from unauthorized users.
- So far, we've made changes to render the links to only those pages when a user is logged in.
- However, that won't protect us from a user who knows the direct URL.

Prevent unauthorized users accessing a page

- To protect against this, we need to add the following line of code to each page:

```
@page "/edit-card/{Card_Id}"
@attribute [Authorize]
```

- Add this line to EditCard and AddCard pages

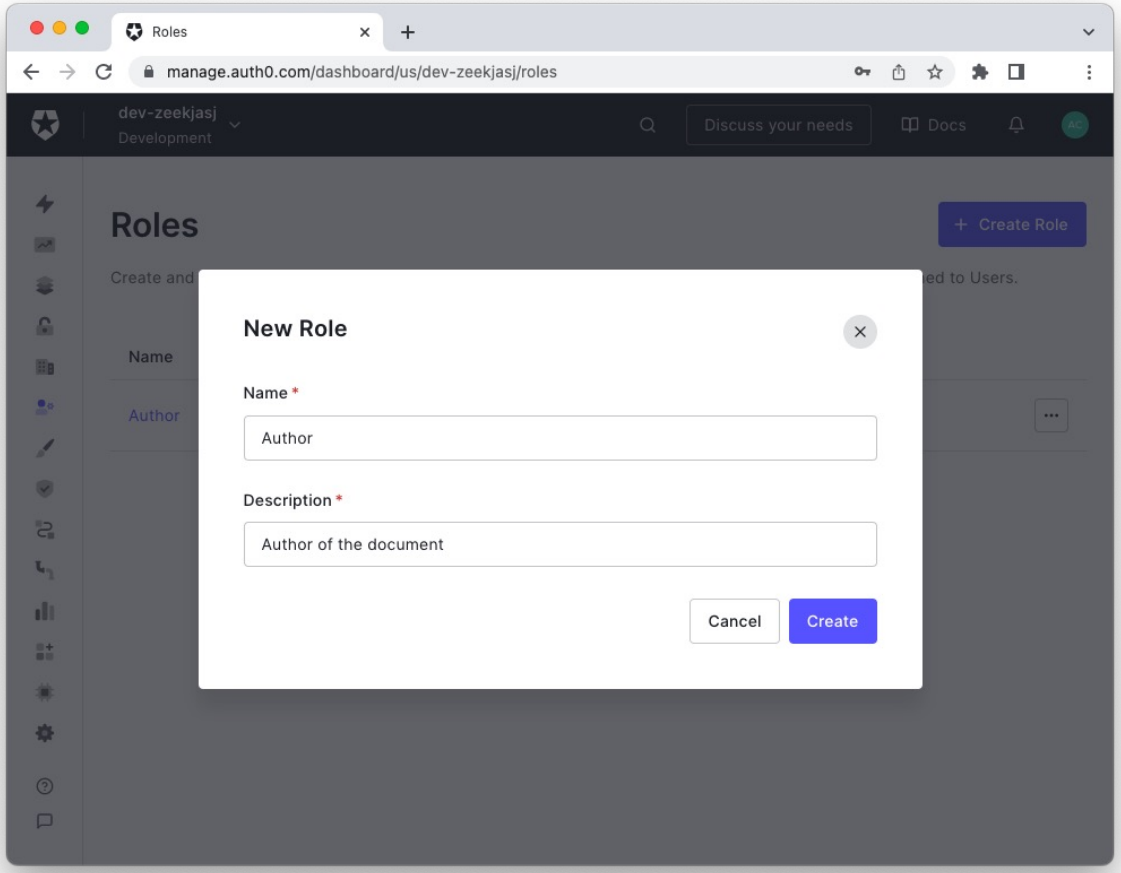
Authorizing users by role

- While some applications may need a user to be logged in only to perform actions, often a further level of permissioning is required.
- One option to enable this is called roles.
- How roles are created and managed will depend on the underlying identity provider.
- For Blazor Server, roles may be encoded into a cookie.

Authorizing users by role

- Before we make any updates to our Blazor app, we're going to head over to Auth0 and create the administrator role and assign a user to it.
- Once you're in the Auth0 dashboard, click the User Management option in the left-hand menu, then select Roles.
- From there, click the Create button to add a new role

Authorizing users by role



Authorizing users by role

- For the role name, enter Administrator.
- You can then enter a description for the role; this can be anything you wish.
- Auth0 will then create the new role and show the role settings page.
- Assign a user to the Administrator role

Making Roles Available to the Application

- The roles you have created and assigned to users are only visible in the Auth0 dashboard.
- To make them available to your application, you must include them in the ID or access token that will be issued by Auth0 at login time
- Role inclusion doesn't happen automatically, so you need to create an Auth0 Action to include the current user's roles in the ID token at login time.

Making Roles Available to the Application

- In your Auth0 dashboard, go to the Actions section, select Flows, and then select the Login flow. Add the following custom Action to the flow:

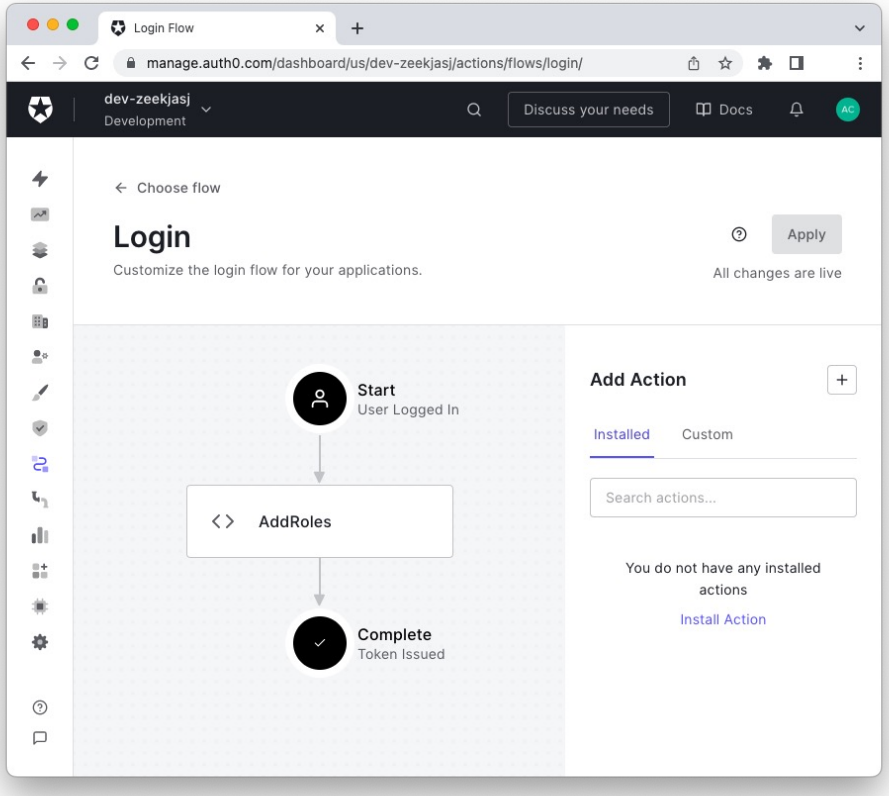
```
exports.onExecutePostLogin = async (event, api) => {
    const roleClaim = 'http://schemas.microsoft.com/ws/2008/06/identity/claims/role';

    if (event.authorization) {
        api.idToken.setCustomClaim(roleClaim, event.authorization.roles);
    }
};
```

More info: https://auth0.com/blog/role-based-access-control-in-blazor-apps/

Making Roles Available to the Application

- The final step to attach this Action to the user login flow is to drag and drop it into the Login Action's editor.



Consuming Auth0 roles - RBAC within the UI

- Auth0 will now return an array of roles for each user when they log in.
- To enable users with a given role to see and access a specific portion of the user interface, use the <AuthorizeView> component with the Roles attribute.
- For example, to allow only users with the *Administrator* role to see the content button on your UI, add the markup shown in the following snippet:

```
<AuthorizeView Roles="Editor">
    <button>Update</button>
</AuthorizeView>
```

Consuming Auth0 roles - RBAC on pages

- The <AuthorizeView> component is useful when you need to control access to portions of your application's UI.
- If you need to control access to an entire page, you should use the Authorize attribute.
- For example, to allow only authors and editors to access the editing page of a document, you should use the Authorize attribute as follows:

```
@page "/edit"
@attribute [Authorize(Roles = "Author, Editor")]

<h1>Editing page</h1>
```

Implementing role-based logic

- Of course, not all access control is done at the markup level.
- Sometimes, you need to check whether a user has a particular role within your code.
- In this case, you need to access the current authentication state object to get the user's role.

```
protected override async Task OnInitializedAsync()
{
    var state = await AuthState.GetAuthenticationStateAsync();

    if (state.User.IsInRole("Administrator")){
        user = "This user is an Administrator";
    }else{
        user = "This user is not an User";
    }
}
```

Implementing role-based logic

- We need to update the Index component (Features > Home) and the Card component (Features > Home > Shared).
- In both components, we need to update the owner check, to show the Edit Card button when a user is in the administrator role.
- Then, we can perform the same update to the Card component
- Exercise: Do it!

Summary

- Blazor Server apps commonly use cookie-based authentication.
- The token passed to blazor contains information about the user's identity, while the access token contains information about what they are allowed to access.
- However, custom claims can be added to either one.
- The AuthorizeView component is used to show different pieces of UI based on the user's authorization status.

Summary

- The AuthorizeRouteView component replaces the RouteView component in the Router to allow access to pages to be restricted using the [Authorize] attribute.
- Blazor supports restricting access based on roles or policies.

Summary

- A role check can be done in code using the User.Identity.IsInRole() method.
- The AuthorizeView component supports both roles and policies via its Roles and Policy parameters.
- The Authorize attribute also supports roles and policies via its Roles and Policy properties.

Blazor Server

Securing Blazor applications

2023