**Notes for Blazor (WebAssembly)**

Ref. project in GitHub <https://github.com/aleph0mc/BlazorMoviesSol>

1. Lifecycle of a Blazor Component (check **MoviesListGen.razor** component under BlazorMovies.Client\Shared)

* **OnInitialized** and **OnInitializedAsync** (to initialize the component (ex. to get data) after the HTTP request)
* **OnParametersSet** and **OnParametersSetAsync** (triggered any time a parameter is updated)
* **OnAfterRender** and **OnAfterRenderAsync** (triggered after component has been rendered)
* **ShouldRender** (define if a component has to be rendered again after rendering, ex. after the user performs some actions on the UI)
* **StateHasChanged()** to notify a change on the component

1. Dependency Injection refers to supply dependencies of a class from another class (services), and it can be performed using the **@inject** attribute.  
   Defailt services
   * HttpClient for HTTP requests to server
   * IJSRuntime to interact with JavaScript
   * NavigationManager to manage navigation through code

Lifecycle of a service

* Scoped, the service lives within a context (ex. during HTTP request). In the client side, it is like a singleton.
* Singleton
* Transient, different instances are created each time the service is requested

Let us create a new class for services in the client project called **Services.cs**

public class SigletonServices

{

. . . . . .

}

public class TransientServices

{

. . . . . . . . .

}

then we need to configure the DI in the **Program.cs** class in the following way



Then in the component we can inject the instance with

@inject SigletonService singleton  
@inject TransientService transient  
@inject IRepository repo

1. We can separate HTML code from c# code using partial classes, the partial class must have the same name as the component ex. **Counter.razor** and the class must be **Conter.razor.cs**, in this way we have a sort of code-behind, the class must be declared as partial, as following



And the component html will be



In a component it’s possible to invoke a method from JavaScript, ex.

[JSInvokable]

public static async Task<int> GetCurrentCount()

{

return await Task.FromResult(currentCountStatic);

}

And from JavaScript

function dotnetStaticInvokation() {

DotNet.invokeMethodAsync("BlazorMovies.Client", "GetCurrentCount")

.then(result => {

console.log("count from javascript", result);

});

To invoke an instance method the logic s the same it’s important to mark the c# method with the attribute **[JSInvokable]**.

1. **Using Identity Server  
   Ref.** <https://github.com/aleph0mc/BlazorMoviesSol/tree/IdentityServer4>
   * DataContext class must inherit from IdentityDbContext   
     public class DataContext : IdentityDbContext //DbContext

Import the namespace from NuGet as

Microsoft.AspNetCore.Identity.EntityFrameworkCore

* + In DataContext make sure under the OnModelCreating to have the line  
    base.OnModelCreating(builder);  
    after the builder.Entity statements otherwise there will be an error during migration.
  + Under Startup.cs in ConfigureServices make sure to add the following lines  
     // Using Identity Server

services.AddIdentity<IdentityUser, IdentityRole>()

.AddEntityFrameworkStores<DataContext>()

.AddDefaultTokenProviders();

* + Finally run the command for a new migrations in order to create the table for IdentityServer. Open the terminal window, make sure to be in the Server folder then run  
    **dotnet ef migrations add IdentityTables** (or **Add-Migration IdentityTables 🡸** under Package Manager Console)  
    Then run  
    **dotnet ef database update** (or **Update-Database 🡸** under Package Manager Console)
  + To update db tools:  
    **dotnet tool install --global dotnet-ef --version 3.1.5**

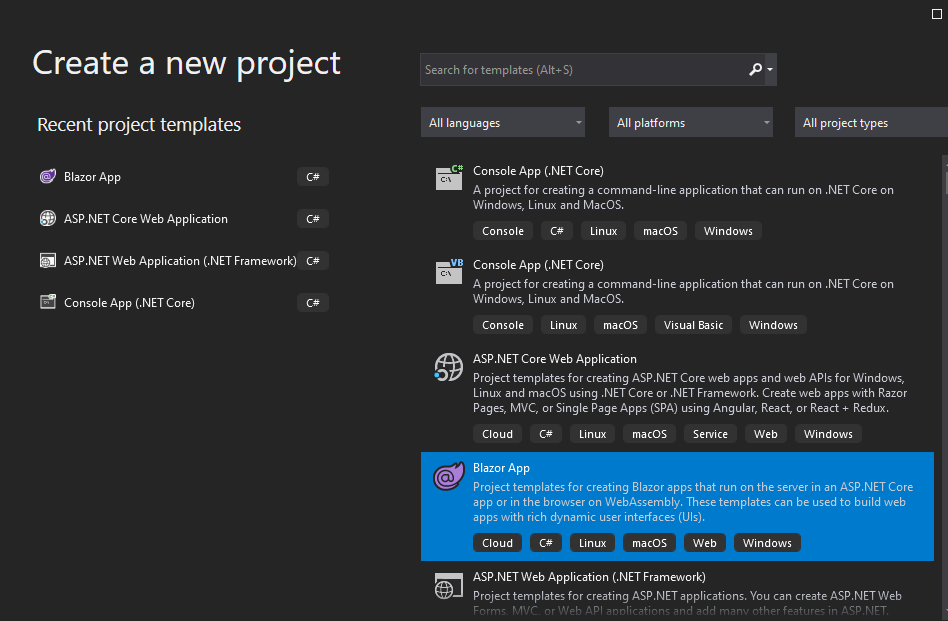
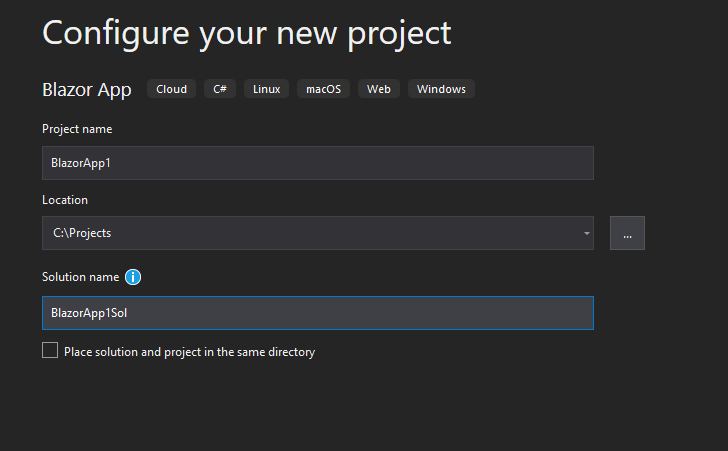
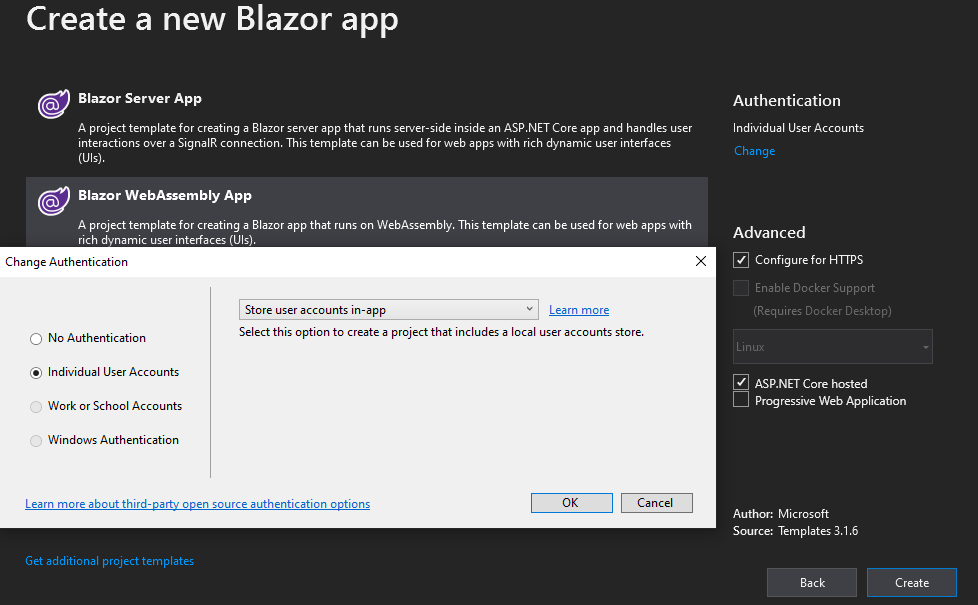
Ref. <https://www.nuget.org/packages/dotnet-ef/>

If issues to install the latest version, remove the previous version

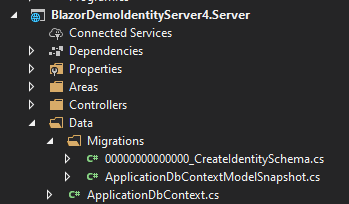
**dotnet tool uninstall -g dotnet-ef**

and then run the install again

The identity tables should be now created in our database.

1. **IdentityServer4** project
   * Create a Blazor App, then Next ****
   * Name the app  
     ****
   * Choose the following config with Authentication on Individual User Account ****
   * In appsettings.json update the DB connection to something like

"DefaultConnection": "Server=.\\SS2017DEV;Database=BlazorDemoIdentityServer4;User Id=aaaaaa;Password=xxxxxx;MultipleActiveResultSets=true"

* + In the Server project under the Data folder there is predefined migration  
    
  + In the terminal, uder the Server folder, run the command  
    **dotnet ef database update** (or **Update-Database** in the Package Manager Console)  
      
    in order to create the identity server database.
  + In order to change the password policy in the Server project, inside Startup.cs, method ConfigureServices it is possible to add some options for identity service, as follows  
      
    services.AddDefaultIdentity<ApplicationUser>(options =>

{

options.SignIn.RequireConfirmedAccount = false; // true if we want the user to confirm his account via email

// The pasword rules can be changed here

options.Password.RequireDigit = false; // no need for the password to contain numbers

options.Password.RequireLowercase = false; // no need of lowercase characters

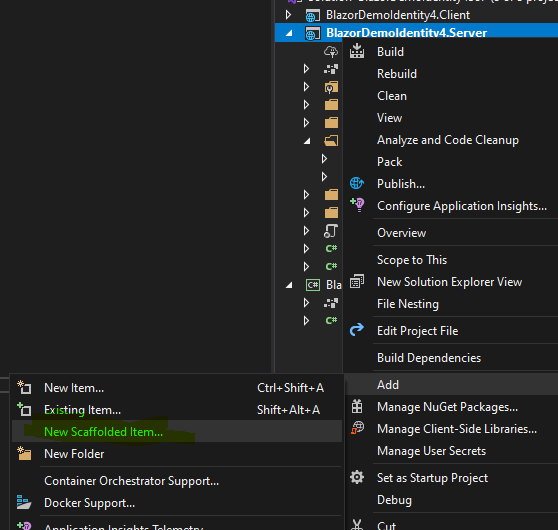
options.Password.RequireUppercase = false; // no need of uppercase characters

options.Password.RequireNonAlphanumeric = false; // no need of non alphanumeric characters (@#?...)

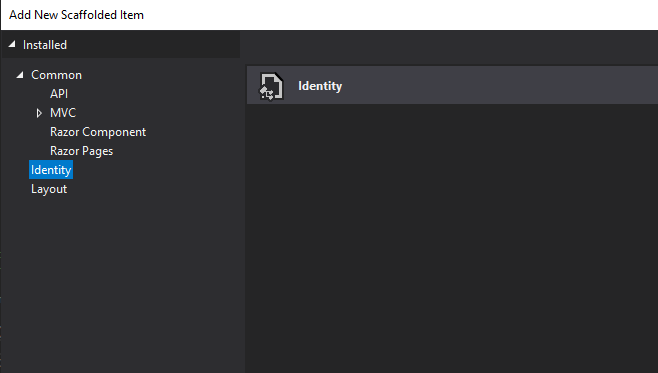
})

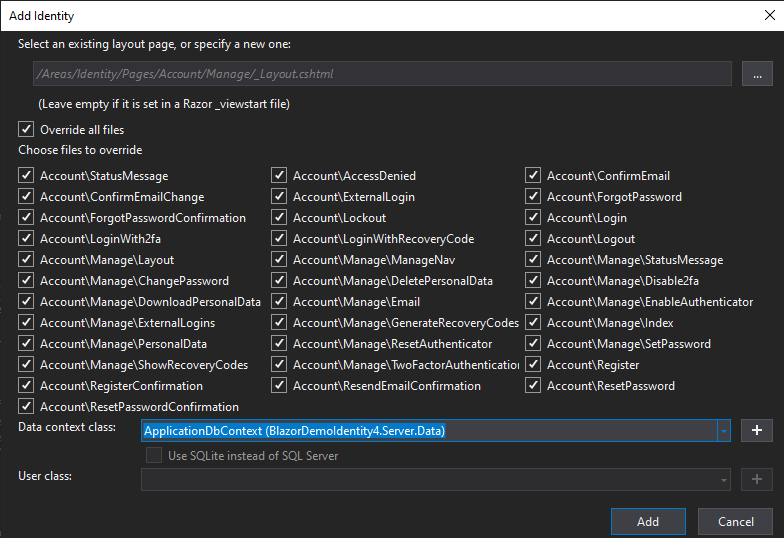
.AddEntityFrameworkStores<ApplicationDbContext>();

* + How modify the Login page or other scaffolded pages  
      
    Right click on the server project and select Add | New Scaffolded item



* + Then select Identity and press Add or double click on Identity item



* + After few seconds it is possible to select a particular layout page or select all the pages, then press Add  
      
    We can find the selected pages under server project folder Areas\Identity\Pages\Account.

In order to make it work, it is **IMPORTANT** to check under the server project, file **\_Layout.cshtml** in folder **Pages\Shared** that the scripts are not required as shown by the following code

. . . .

<body>

<div class="main">

<div class="content px-4">

@RenderBody()

**@RenderSection("Scripts", required: false**) 🡸 **IF THIS LINE IS NOT INCLUDED THEN ERROR**

</div>

</div>

</body>

. . . .

The file **ScaffoldingReadMe.txt** can be deleted either, as we don’t need it anymore.

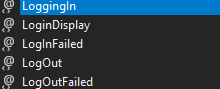
* + After pressing the **Log out button** it is possible to be redirected to any page required, in order to do that we need to perform some change to the file **Authentication.razor** under the Client project, folder Pages, in the following way, adding a RenderFragment under the tag <**RemoteAuthenticatorView** **Action**="@Action" /> therefore we get  
    . . . .  
    <**RemoteAuthenticatorView** **Action**="@Action">

<**LogOutSucceeded**>

**@ On logout succeeded redirect to Home page \*@**

**@{ nm.NavigateTo("/"); }**

</**LogOutSucceeded**>

</**RemoteAuthenticatorView**>  
. . . .  
There are other types of RenderFragment for logging such as  


1. How to add **IdentityServer4** to an existing project.
   * In the **Server** project, perform an update to all the packages, if required, then install the following packages:

Microsoft.AspNetCore.Identity.UI  
Microsoft.VisualStudio.Web.CodeGeneration.Design  
Microsoft.AspNetCore.ApiAuthorization.IdentityServer

* + Open in the Server folder the file **DataContext.cs** (or whatever the name is) and let the class to inherit now from **ApiAuthorizationDbContext**,while for IdentityServer it just was IdentityDbContext

public class DataContext : ApiAuthorizationDbContext<IdentityUser>

as we are using as representation of the user for authentication/authorization the **IdentityUser**.

* + The construct must become as follows

public DataContext(DbContextOptions<DataContext> options,

IOptions<OperationalStoreOptions> operationalStoreOptions

) : base(options, operationalStoreOptions) { }

* + At this point we can add a new migration, therefore under the Server folder, we can run the commanddotnet ef migrations add NewIdentitySystem (or **Add-Migration NewIdentitySystem 🡸** under Package Manager Console)

then, after checking the migration went well, run the command

dotnet ef database update (or **Update-Database 🡸** under Package Manager Console)

* + Then we need to perform some changes to the **Startup.cs** file in the Server project, in method ConfigureServices

// Using Identity Server

//services.AddIdentity<IdentityUser, IdentityRole>()

// .AddEntityFrameworkStores<DataContext>()

// .AddDefaultTokenProviders();

// Using IdentityServer4

**services.AddDefaultIdentity<IdentityUser>(options =>**

**{**

**options.SignIn.RequireConfirmedAccount = false; //Probably true in production**

**})**

**.AddRoles<IdentityRole>()**

**.AddEntityFrameworkStores<DataContext>();**

Then we can comment out the code for JWT token for IdentityServer will take care of that

//JWT token - authentication scheme

//services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

// .AddJwtBearer(options =>

// options.TokenValidationParameters = new TokenValidationParameters

// {

// ValidateIssuer = false,

// ValidateAudience = false,

// ValidateLifetime = true,

// ValidateIssuerSigningKey = true,

// IssuerSigningKey = new SymmetricSecurityKey(

// Encoding.UTF8.GetBytes(Configuration["jwt:key"])),

// ClockSkew = TimeSpan.Zero

// });

**services.AddAuthentication()**

**.AddIdentityServerJwt();**

Then in method **Configure**, we need to use the **IdentityServer** therefore we have the following

//Add authentication and authorization for IdentityServer

app.UseAuthentication();

app.UseIdentityServer();

app.UseAuthorization();

* + Then we can delete the **AccountController.cs** file undwer Controllers folder for we don’t need it anymore and we can remove from each **[Authorize]** attribute the reference to the JWT bearer

[Authorize(~~AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme~~, Roles = "Admin")]

Just becomes

[Authorize(Roles = "Admin")]

In the **appsettings.json** we can remove the item

~~"jwt": {~~

~~"key": "KLASMDKL3M4KLMSDLKM3LKM4KLMSK543643341332S5DA3S5D453"~~

~~}~~

Then we need to add info for the Client is using IdentityServer, therefore we can add the following

. . . .

"IdentityServer": {

"Key": {

"Type": "Development"

},

"Clients": {

"BlazorMovies.Client": {

"Profile": "IdentityServerSPA"

}

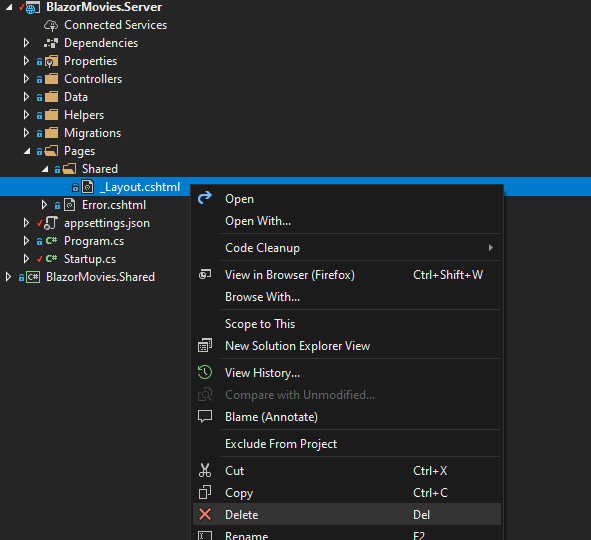
}

}

. . . .

The Client will use the profile called IdentityServerSPA, now we need to configure this profile

In order to do that we need the scaffolding, but first we need to remove the \_Layout.cshtml file from the Server project



After that, we can proceed with the scaffolding as mentioned above.

* + Under the Client project we need to add a script to the **index.html** file under the **wwwroot** Client folderin the following way

. . . .

</div>

<script src="\_content/Microsoft.AspNetCore.Components.WebAssembly.Authentication/AuthenticationService.js">

</script> <script src="\_framework/blazor.webassembly.js"></script>

<script src="https://cdn.jsdelivr.net/npm/sweetalert2@9"></script>

<!-- CUSTOMS JS FILES GOES HERE -->

<script src="js/utilities.js"></script>

. . . .

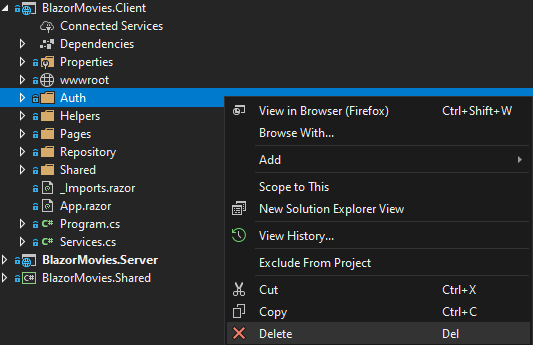
* + Then in the Server project we need to add a new controller under Controllers that we can call **OidcConfigurationController** containing the following code



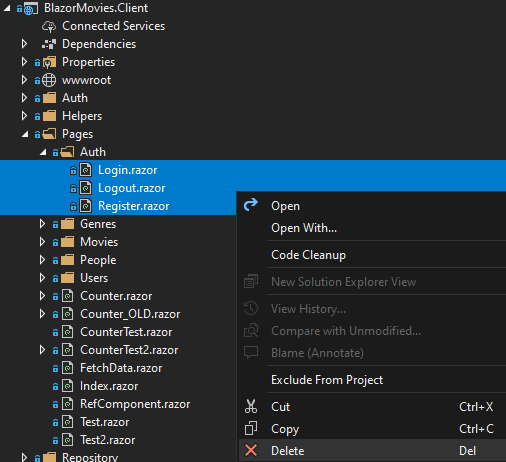
* + Now we need to change the **LoginLink.razor** component in Client\Shared foldcer in order to better manage the **login** and **logout** , the code is as follows



* + We can now remove the Auth folder in the Client project



Then we can delete as well the files under Client\Pages\Auth folder



And add a new component under that folder called **Authentication.razor** with the following code



* + Finally in the Client **Program.cs** we can add some code for the DI system where we can delete the following code

~~services.AddAuthorizationCore(); //Authorization component~~

~~//In JWTAuthenticationStateProvider there is the ILoginService and to use the DI we need to add this code~~

~~//Create an instance of JWTAuthenticationStateProvider~~

~~services.AddScoped<JWTAuthenticationStateProvider>();~~

~~//Use same instance of JWTAuthenticationStateProvider for both AuthenticationStateProvider~~

~~services.AddScoped<AuthenticationStateProvider, JWTAuthenticationStateProvider>(~~

~~provider => provider.GetRequiredService<JWTAuthenticationStateProvider>()~~

~~);~~

~~//And ILoginService~~

~~services.AddScoped<ILoginService, JWTAuthenticationStateProvider>(~~

~~provider => provider.GetRequiredService<JWTAuthenticationStateProvider>()~~

~~);~~

~~//Used to renew the token automatically (if required) as a background task~~

~~services.AddScoped<TokenRenewer>();~~

And add the following line

//IdentityServer4 authorization

**services.AddApiAuthorization();**

* + In order to work with claims we need to add a new Service, call it IdentityProfileService under the Server\Helpers folder with the following code



Then we need to add this service in the Server **Startup.cs** file as follows

services.AddIdentityServer()

.AddApiAuthorization<IdentityUser, DataContext>()

.AddProfileService<IdentityProfileService>();

The highlighted line is important as it is the service required to manage the claims in IdentityServer4.

* + Finally we need a HTTPClient to send the web token to the server, in order to do that we have to install a Nuget package under the **Client** project called

Microsoft.Extensions.Http

Then we need to create two classes under the Client\Helpers folder called

HttpClientWithToken and HttpClientWithoutToken having the code as follows





Then we need to instantiate these services in the Client Program.cs file under the method Main as follows

. . . .

//builder.Services.AddTransient(sp => new HttpClient { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });

builder.Services.AddHttpClient<**HttpClientWithToken**>(

client => client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress))

.AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

builder.Services.AddHttpClient<**HttpClientWithoutToken**>(

client => client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress));

. . . .

Now in files **HttpService.cs** we want to inject the HTTPClient with token therefore the constructor must become as follows

. . . .

private readonly **HttpClientWithToken** \_httpClient;

. . . . .

public HttpService(**HttpClientWithToken** httpClient)

{

\_httpClient = httpClient;

}

Then we need to change the signature in I**HttpService.cs** for the following two methods

Task<HttpResponseWrapper<T>> Get<T>(string url**, bool includeToken = true**);

Task<HttpResponseWrapper<TResponse>> Post<T, TResponse>(string url, T data**, bool includeToken = true**);

and the following line in **HttpService.cs** for those methods

~~var httpClient = GetHttpClient();~~

must become

var httpClient = GetHttpClient(includeToken);

Then the code for the class must be arranged accordingly (check file in GitHub repository).

Then there is an issue with the mapping from JWT token and Identity token for Admin users (or other roles), in order to fix this we must perform the following changes in Server **Startup.cs** file, adding the highlighted lines

. . . .

using System.IdentityModel.Tokens.Jwt;

. . . .

public Startup(IConfiguration configuration)

{

JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

Configuration = configuration;

}

. . . .

In this way we clear the JWT token and the mapping is correct.

Finally IHttpServiceExtensionMethods must be changed to include the new option

public static async Task<T> GetHelper<T>(this IHttpService httpService, string url**, bool includeToken = true**)

public static async Task<PaginatedResponse<T>> GetHelper<T>(this IHttpService httpService, string url, PaginationDTO paginationDTO**, bool includeToken = true**)

and **GenreRepository.cs** and **MovieRepository.cs** accordingly (see solution in GitHub for these two files).

1. We can generate the database script (queries) from one migraton to another using the command

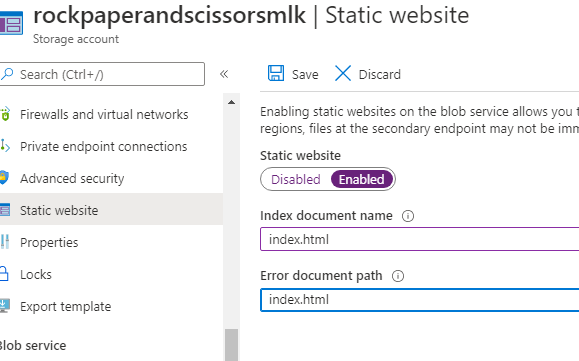
dotnet ef migrations script AdminRole AdminUser

in Package Manager Console it is possible to use the command

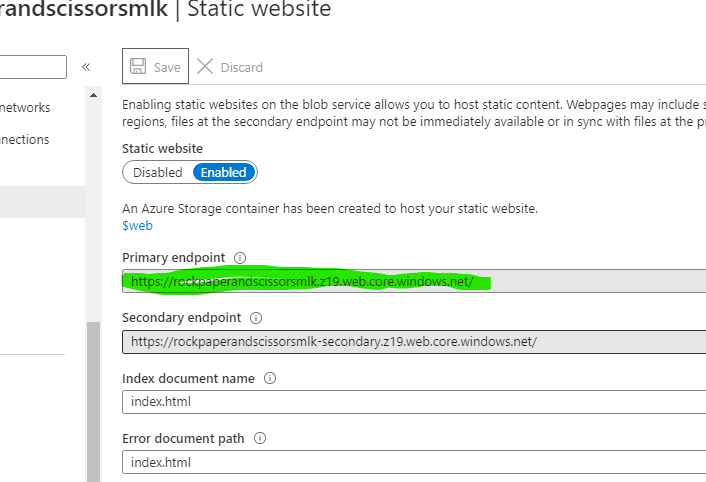
Script-Migration -from AdminRole –to AdminUser

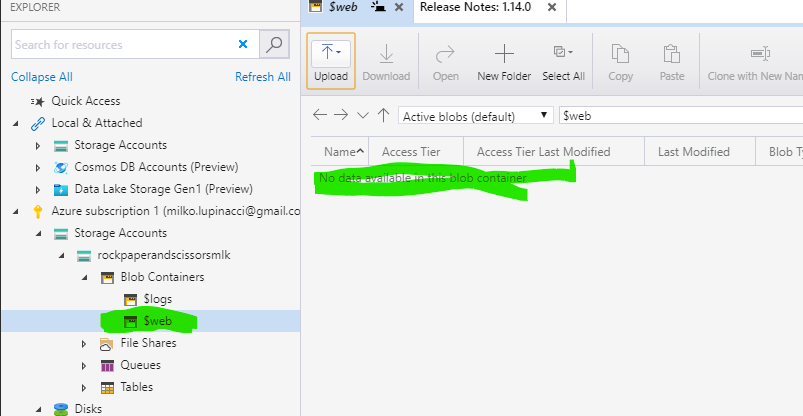
Where AdminRole and AdminUser are migration names.

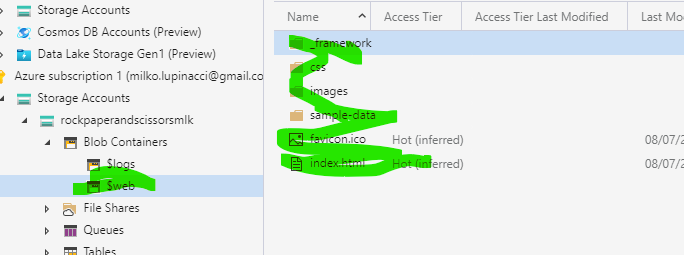
This create a standard SQL database script.

1. In Azure Storage is possible to deply only static sites, not dependent on the server side, ex is RockPaperAndScissors game (ref. <https://github.com/aleph0mc/BlazorRockPaperScissorsGameSol>)
   * Open Azure portal and select Storage Account and create a new one
   * Inside the storage account just created select Static Website
   * Then click on Enabled   
     

Then press Save and we get the Primary Endpoint



* + In orer to upload our app we can get a free tool called Azure Storage Explorer that can be downloaded from <https://azure.microsoft.com/en-us/features/storage-explorer/>
  + Using the tool we can select   
      
    That is the folder where wew have to upload our app. In order to do that we publish our static app then we drag and drop the file from the source folder to container shown above, the files to be upload are



Now we can start our app going to the above endpoint

<https://rockpaperandscissorsmlk.z19.web.core.windows.net/>

and That’s it!

1. Continuous delivery – DevOps  
   It refers to the automatic process of compilation, test run and deployment of a project into a determined environment, whether testing or production.