**Notes for Blazor (WebAssembly)**

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**
   * 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>()

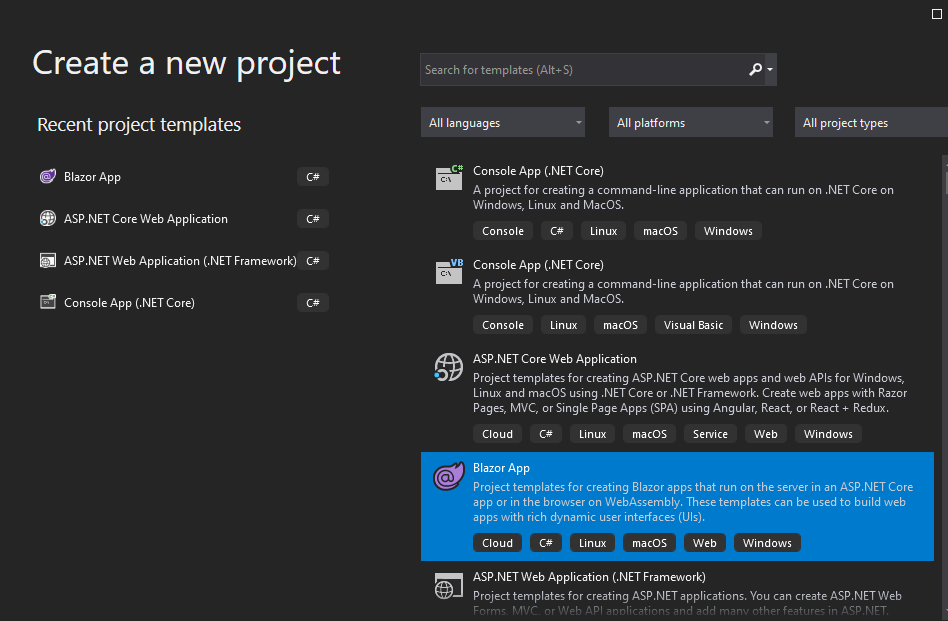
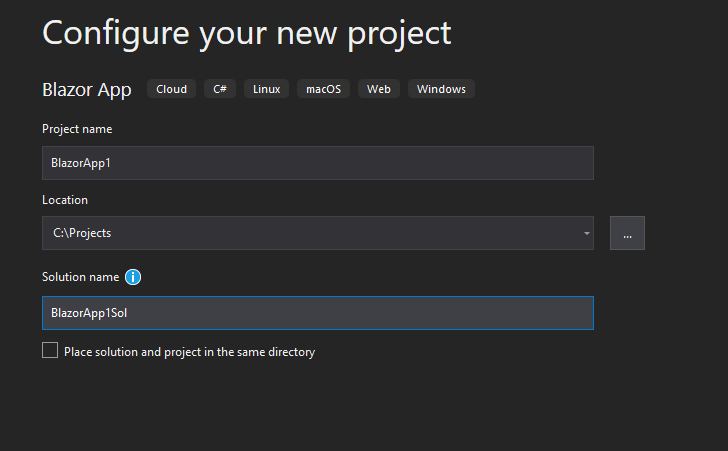
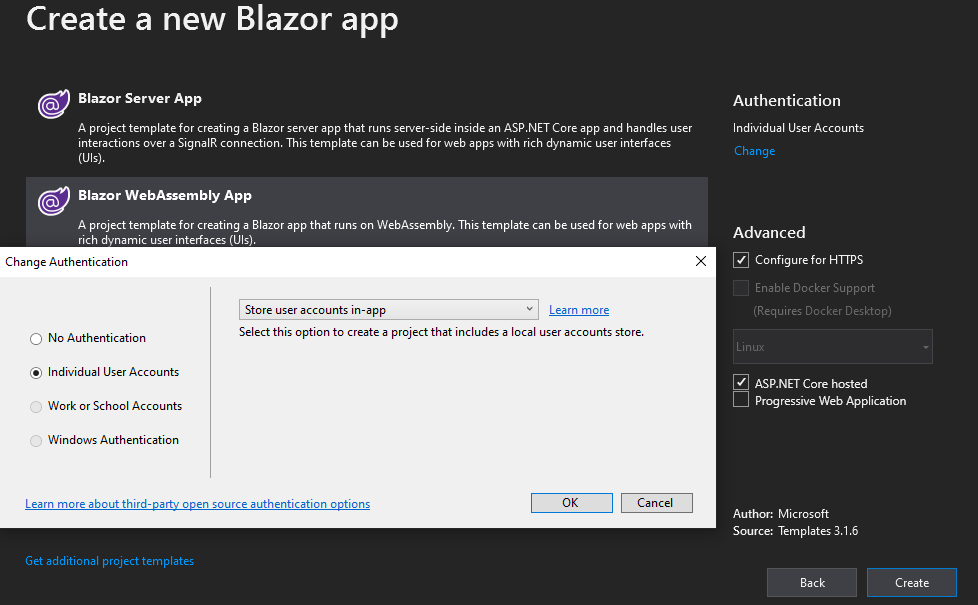
.AddEntityFrameworkStores<DataContext>()

.AddDefaultTokenProviders();

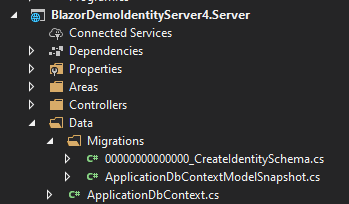
* + Finally run the command for a new migrations in order to create the table for IdentityServer. Open the terminal window, make sure to be in the Server folder then run  
    dotnet ef migrations add IdentityTables (or **Add-Migration IdentityTables 🡸** under Package Manager Console)  
    Then run  
    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/>

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

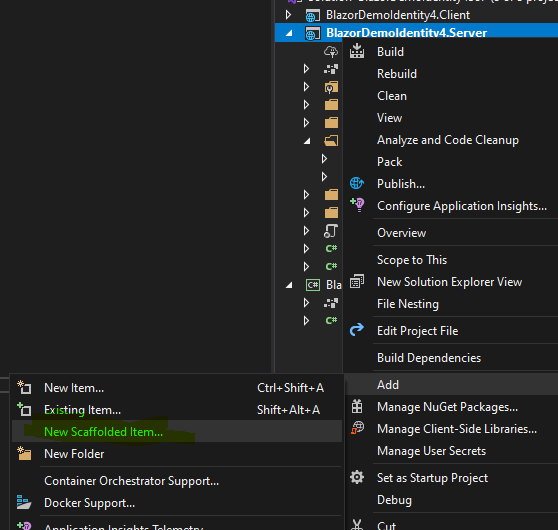
options.Password.RequireUppercase = false; // no need of uppercase characters

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

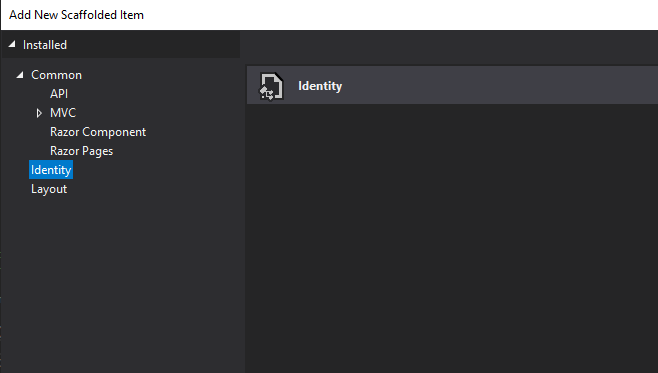
})

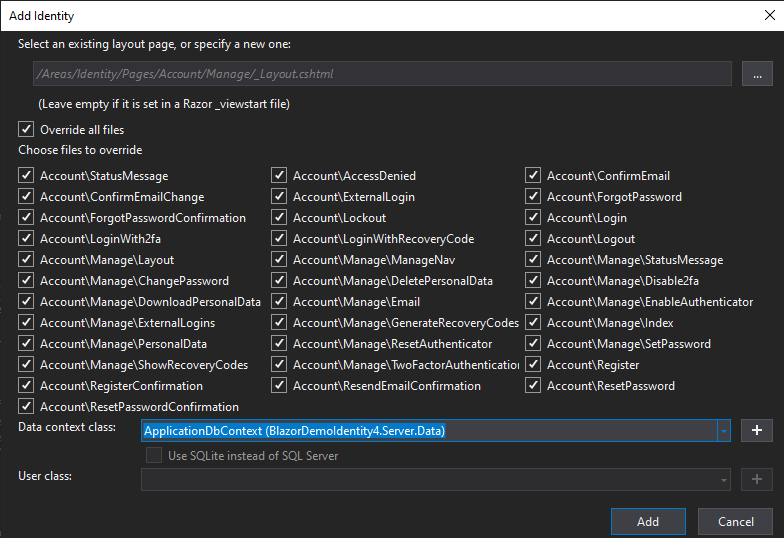
.AddEntityFrameworkStores<ApplicationDbContext>();

* + 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



* + 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**

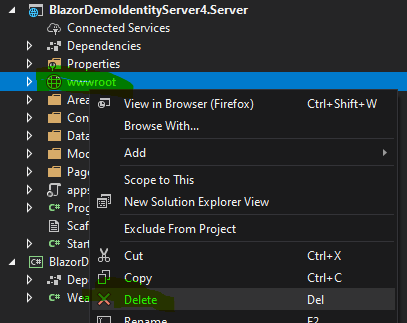
</div>

</div>

</body>

. . . .

* + After the scaffolding under the server project is created a **wwwroot** folder that can be deleted



* + 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  
