**CURSO ASP NET CORE - BLAZOR WEB, AUTENTICAÇÃO E AUTORIZAÇÃO EM APIs**

Projeto inicial do curso: <https://github.com/alura-cursos/ScreenSound-Curso4>

**MUDANÇAS NECESSÁRIAS**

Antes mesmo de começar o projeto, mudar dois pequenos equívocos

1 - Em ScreenSound.API/Appsettings.Development.json - mudar o banco de dados de ScreenSoundV1 (que não existe) para ScreenSoundV0 (que é o usado nos cursos anteriores).

1. Em ScreenSound.Dados/Migrations/ Na migration PopularMusicas, há uma linha

migrationBuilder.Sql("update Musicas set ArtistaId = (select Id from Artistas where Nome = 'Djavan')");

Comentar ela ou excluir.

Esse trecho da migration usa uma coluna que ainda nem foi criada (ArtistaId), cuja criação vai ser feita somente na migration posterior.

**INSTALAÇÃO E CONFIGURAÇÃO DO IDENTITY**

Para que seja possível ter controle de usuários, autenticação e autorização, usaremos o identity.

Instação do Identity no projeto ScreenSound.Shared.Dados

<PackageReference Include="Microsoft.AspnetCore.Identity.EntityFrameworkCore" Version="7.0.13" />

Criação da pasta modelos em ScreenSound.Shared.Dados, com as classes PessoaComAcesso e PerfilDeAcesso

PerfilDeAcesso - um *role*

using Microsoft.AspNetCore.Identity;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ScreenSound.Shared.Dados.Modelos

{

public class PerfilDeAcesso : IdentityRole<int>

{

}

}

PessoaComAcesso

using Microsoft.AspNetCore.Identity;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ScreenSound.Shared.Dados.Modelos

{

public class PessoaComAcesso : IdentityUser<int>

{

}

}

Mudança em Banco/ScreenSoundContext para incluir classes PessoaComAcesso e PerfilDeAcesso

public class ScreenSoundContext: IdentityDbContext<PessoaComAcesso, PerfilDeAcesso, int>

.

.

Gerar a migration que vai o banco de dados com as tabelas do identity

Ferramentas -> Gerenciador de pacotes Nuget-> Console do gerenciador

***Add-Migration IdentityTabelas***

***Update-Database***

**ENDPOINTS DA API PARA USUÁRIOS**

Para fazer uso do identity na interface do sweagger, vamos primeiro injetar o serviço do identity em program.cs

ScreenSound.API/program.cs

.

.

//INJEÇÃO SERVIÇO IDENTITY PARA GESTÃO DE ENDPOINTS DE ACESSO

builder.Services

.AddIdentityApiEndpoints<PessoaComAcesso>()

.AddEntityFrameworkStores<ScreenSoundContext>();

.

.

.

//MAPEAMENTO DOS ENDPOINST DO IDENTITY - GESTÃO DE ACESSO

app.MapGroup("auth").MapIdentityApi<PessoaComAcesso>().WithTags("Autorização");

//MapGroup("auth") - TODAS ROTAS MAPEADAS COMEÇARÃO COM ESSE CAMINHO

//.WithTags("Autorização"); - ORGANIZAÇÃO. NO SWAGGER, APARECERÃO JUNTAS NESSA TAG

Com essas configurações definidas, na rota auth/register podemos registrar usuários. Registraremos dois:

{

"email": "luciano@mail.com",

"password": "Senha!123"

}

{

"email": "luciano2@mail.com",

"password": "Senha!123"

}

Para verificar se os usuários foram registrados, usamos a rota auth/login, com *useCookies* true

{

"email": "luciano2@mail.com",

"password": "Senha!123"

}

**DETERMINANDO AUTORIZAÇÃO PARA ACESSO AOS DEMAIS ENDPOINTS DA APLICAÇÃO**

Para exemplificar uma aplicação de controle de acesso a um endpoint específico, vamos estabelecer que, para todos endpoints de artistas, será necessário estar identificado na API, para acessar. Como fazer isso?

Injetar o serviço de autorização em program.cs

//INJEÇÃO SERVIÇO DE AUTORIZAÇÃO

builder.Services.AddAuthorization();

.

.

//VERIFICAR REQUISIÇÕES HTTPS ANTES DE USAR ENDPOINTS

app.UseAuthorization();

Feito isso, vamos aplicar o uso dessa autorização aos endpoints de artista, que estão todos em Endpoinsts/ArtistaExtension

public static void AddEndPointsArtistas(this WebApplication app)

{

//VARIÁVEL groupBuilder QUE AGRUPA TODAS ROTAS COMEÇANDO COM artistas

//JÁ REQUERINDO AUTORIZAÇÃO

//E ORGANIZANDO COM TAG artistas

var groupBuilder = app.MapGroup("artistas")

.RequireAuthorization()

.WithTags("artistas");

#region Endpoint Artistas

groupBuilder.MapGet("", ([FromServices] DAL<Artista> dal) =>

{

.

.

groupBuilder.MapGet("{nome}", ([FromServices] DAL<Artista> dal, string nome) =>

.

.

Agora, ao tentar executar um endpoint de artista, sem que seja feito um login antes, o resultado é um code 401 (não autorizado)

**FORMULÁRIO DE LOGIN NA APLICAÇÃO WEB**

Agora iremos criar um formulário de login na aplicação web. Para que os dois projetos possam ser executados aos mesmo tempo





Antes de construir a página de formulário de login vamos primeiro criar uma classe AuthResponse, que vai guardar a resposta da API para a tentativa de login

ScreenSound.Web/Response/AuthResponse

public class AuthResponse

{

public bool Sucesso { get; set; }

public string Erro { get; set; }

}

Agora vamos criar o serviço que vai usar a autenticação da API

ScreenSound.Web/Services/AuthAPI

public class AuthAPI(IHttpClientFactory factory)

{

private readonly HttpClient \_httpClient = factory.CreateClient("API");

public async Task<AuthResponse> LoginAsync(string email, string senha)

{

var response = await \_httpClient.PostAsJsonAsync("auth/login", new

{

email,

password = senha

});

//SE A TENTATIVA DE LOGIN FOR BEM SUCEDIDA

if (response.IsSuccessStatusCode)

{

return new AuthResponse{Sucesso = true};

}

//SE NÃO

return new AuthResponse{Sucesso = false, Erro="Erro no login" };

}

}

Para que esse serviço funcione, temos que adicionar ele em program.cs

builder.Services.AddScoped<AuthAPI>();

E por fim, o formulário de login

ScreenSound.Web/Pages/Login

@page "/login"

@inject AuthAPI authAPI

<**MudPaper** **Class**="px-8 pt-2 pb-4 mx-16 my-8" Justify="Justify.Center">

<**MudForm**>

<**MudTextField** **T**="string" **Label**="Email" @bind-Value="email"

**Variant**="Variant.Outlined" **Class**="my-4"

**Required**="true" **RequiredError**="Email obrigatório!"

**OnlyValidateIfDirty**="true" />

<**MudTextField** **T**="string" **Label**="Senha" @bind-Value="senha"

**InputType**="InputType.Password"

**Variant**="Variant.Outlined" **Class**="my-4"

**Required**="true" **RequiredError**="Senha obrigatória!"

**OnlyValidateIfDirty**="true" />

<**MudButton** **Variant**="Variant.Filled" **Color**="Color.Primary"

**Class**="my-6" @onclick="FazerLogin">

Login

</**MudButton**>

</**MudForm**>

</**MudPaper**>

@code {

private string? email;

private string? senha;

private async Task FazerLogin()

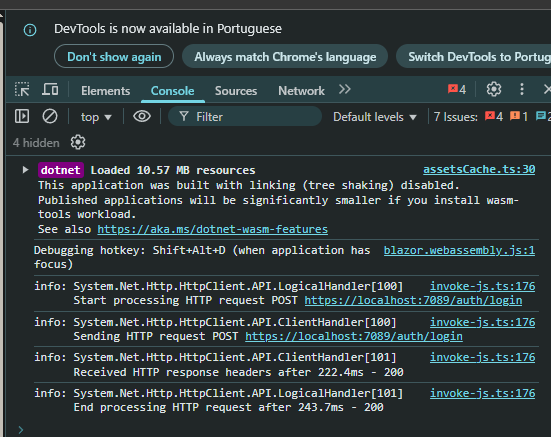
{

await authAPI.LoginAsync(email, senha);

}

}

Para observar se o login foi bem sucedido através do formulário, podemos consultar, no navegador, as ferramentas do desenvolvedor, a aba console.



Importante destacar que, para que esse login efetuado seja mantido, precisamos ainda configurar o cookie que armazena essa informação de login para que ele possa ser utilizado nas requisições futuras, ou seja, o usuário loga uma vez e seu acesso continua para as requisições futuras. Para isso, usaremos o DelegatinsHandler, uma classe que consegue manipular as requisições https

Services/Cookie Handler

using Microsoft.AspNetCore.Components.WebAssembly.Http;

namespace ScreenSound.Web.Services

{

public class CookieHandler : DelegatingHandler

{

//SOBRESCRITA DE MÉTODO QUE PASSA TODAS AS CREDENCIAIS DO NAVEGADOR

//PARA A PRÓXIMA REQUISIÇÃO

//PESSOA LOGA UMA VEZ, PERMANECE LOGADA

protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)

{

request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);

return base.SendAsync(request, cancellationToken);

}

}

}

Program.cs

builder.Services.AddScoped<CookieHandler>();

.

.

//CONFIGURANDO CLIENTE HTTP CHAMADO API

builder.Services.AddHttpClient("API",client => {

client.BaseAddress = new Uri(builder.Configuration["APIServer:Url"]!);

client.DefaultRequestHeaders.Add("Accept", "application/json");

}).AddHttpMessageHandler<CookieHandler>();//CONFIGURAÇÃO DE COOKIE

Não deu certo! - Pelo Swagger, com o useCookie true funciona

Pelo navegador, o cookie não aparece. Pq?