**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?useCookies=true", 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" };

}

}

Importante destacar que o método LoginAsync, faz uso do método PostAsJsonAsync, e nesse método ao acessar a rota *auth/login do identity,* temos um parâmetro useCookies=true. Nesse momento que definimos que o método de login vai usar cookies.

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()

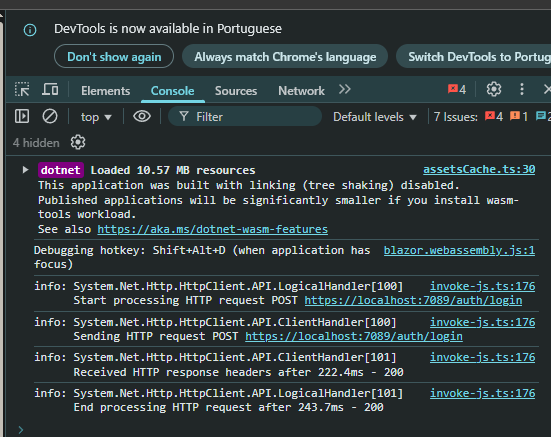
{

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 DelegatingHandler, uma classe que consegue manipular as requisições http

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

**CONFIGURAR A APLICAÇÃO BLAZOR WEB PARA UTILIZAR ESTADO DE AUTENTICAÇÃO**

Agora que temos definido que a aplicação aceita registrar usuários e logar, vamos configurar um estado de autenticação. Isso significa que vamos estruturar a aplicação para se comportar de uma maneira específica quando há um usuário logado (aparecer o avatar com a foto da pessoa, por exemplo).

Pacote para projeto ScreenSound.Web

<PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.Authentication" Version="8.0.0" />

Utilizar o identity na API para configurar esse estado de autenticação, por meio da implementação da classe abstrata ***AuthenticationStateProvider*** no serviço AuthAPI, no método GetAuthenticationStateAsync

public class AuthAPI(IHttpClientFactory factory) : AuthenticationStateProvider

.

.

//CONFIGURAÇÃO ESTADO DE AUTENTICAÇÃO

public override async Task<AuthenticationState> GetAuthenticationStateAsync()

{

//ClaimsPrincipal VAZIO - PESSOA SEM AUTENTICAÇÃO / SEM ESTADO DE AUTENTICAÇÃO

var pessoa = new ClaimsPrincipal();

//Rota "auth/manage/info" DO IDENTITY RETORNA SE USUÁRIO ESTÁ LOGADO OU NÃO

var response = await \_httpClient.GetAsync("auth/manage/info");

//SE HOUVER USUÁRIO LOGADO

if (response.IsSuccessStatusCode)

{

//SALVO INFORMAÇÕES DO USUÁRIO EM OBJETO InfoPessoaResponse info

var info = await response.Content.ReadFromJsonAsync<InfoPessoaResponse>();

//CRIO VETOR DE CLAIMs COM DUS CLAIMs

Claim[] dados =

[

//A PRIMEIRA É PADRÃO DA MICROSOFT Name - EMAIL

//EMAIL NESSE ESTADO É A PRINCIPAL INFORMAÇÃO DE AUTENTICAÇÃO

new Claim(ClaimTypes.Name, info.Email),

//SEGUNDA CLAIM É O PRÓPRIO EMAIL

new Claim(ClaimTypes.Email, info.Email)

];

//OBJETO ClaimIdentity QUE AGRUPA TODOS OS DADOS DA PESSOA

//E INDICA TAMBÉM NUMA STRING QUAL TIPO DE AUTENTICAÇÃO ESTÁ SENDO USADA

var identity = new ClaimsIdentity(dados, "Cookies");

//COM TODAS AS INFORMAÇÕES PREENCHIDAS

//PREENCHO O OBJETO ClaimsPrincipal

//ClaimsPrincipal PREENCHIDO - PESSOA AUTENTICADA / ESTADO DE AUTENTICAÇÃO OK

pessoa = new ClaimsPrincipal(identity);

}

//RETORNO ESTADO DE AUTENTICAÇÃO

//VAZIO OU AUTENTICADO - A DEPENDER SE HÁ USUÁRIO LOGADO NO SISTEMA

return new AuthenticationState(pessoa);

}

Indicar ao sistema a mudança de estado de autenticação no momento do login bem sucedido, na mesma classe (AuthAPI), no método LoginAsync

.

.

//SE A TENTATIVA DE LOGIN FOR BEM SUCEDIDA

if (response.IsSuccessStatusCode)

{

//MUDANDA ESTADO DE AUTENTICAÇÃO

NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());

.

.

Aplicar o estado de autenticação na página de login por meio do componente AuthorizeView, que segue a seguinte estrutura

@using Microsoft.AspNetCore.Components.Authorization

<**AuthorizeView**>

<**Authorized**>

//TRECHO DE CÓDIGO PARA QUEM ESTIVER LOGADO

</**Authorized**>

<**NotAuthorized**>

//TRECHO DE CÓDIGO PARA QUEM NÃO ESTIVER LOGADO

</**NotAuthorized**>

</**AuthorizeView**>

Que aplicada na página, fica o seguinte

@page "/login"

@using Microsoft.AspNetCore.Components.Authorization

@inject AuthAPI authAPI

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

<**AuthorizeView**>

<**Authorized**>

<p>Você está logado como @context.User.Identity.Name</p>

</**Authorized**>

<**NotAuthorized**>

<**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**>

</**NotAuthorized**>

</**AuthorizeView**>

</**MudPaper**>

@code {

.

.

Configurar o serviço de estado de autenticação em program.cs

//SERVIÇOS ESTADO DE AUTENTICAÇÃO

builder.Services.AddAuthorizationCore();

builder.Services.AddScoped<AuthenticationStateProvider, AuthAPI>();

builder.Services.AddScoped<AuthAPI>(sp => (AuthAPI)

sp.GetRequiredService<AuthenticationStateProvider>());

E configurar em app.razor que o serviço de estado de autenticação vai ficar disponibilizado em toda a aplicação, fazendo isso envelopando o conteúdo existente numa tag CascadingAuthenticationState

app.razor

@using Microsoft.AspNetCore.Components.Authorization

<**CascadingAuthenticationState**>

.

.

.

</**CascadingAuthenticationState**>

Agora, ao logar, a aplicação web apresenta a mensagem “Você está conectado como ...”, dado que esse é o estado da página /login para um usuário autenticado.

**SOBRE INJEÇÃO DE SERVIÇO NO ASP.NET Core**

***AddScoped***

*No método AddScoped registramos um serviço com um tempo de vida por escopo. Isso significa dizer que uma instância do serviço será criada e mantida durante todo o ciclo de vida de uma única requisição HTTP (ou escopo) e para cada nova requisição recebe sua própria instância do serviço.*

***AddTransient***

*Já no método AddTransient o serviço é registrado com um tempo de vida transitório, ou seja, uma nova instância do serviço será criada toda vez que ele for solicitado. Isso pode acontecer várias vezes durante a mesma requisição ou em diferentes requisições.*

***AddSingleton***

*Para o método AddSingleton o serviço é registrado com um tempo de vida único em toda a aplicação. Apenas uma instância do serviço será criada e compartilhada por todas as requisições e threads durante a execução do aplicativo.*

**CONFIGURANDO LOGOUT**

Primeiro vamos configurar o endpoint de logout no projeto API, em program.cs

//DEFININDO ROTA DE LOGOUT

app.MapPost("auth/logout", async ([FromServices] SignInManager<PessoaComAcesso>

signInManager) =>

{

await signInManager.SignOutAsync();

return Results.Ok();

}

).RequireAuthorization().WithTags("Autorização");

Agora, ao executar o projeto ScreenSound.API, pelo Swagger, podemos acessar a rota auth/login com um usuário válido, consultar se estamos logados na rota auth/manage/info e efetuar o logout em auth/logout

Feitas as mudanças na API, vamos para o projeto web, na classe de serviço AuthAPI, configurar dois métodos necessários, o LogutAsync, que vai consumir da API e mudar o estado de autenticação, e VerificaAutenticado, que retorna um bool indicando se o usuário está logado ou não. Para que tudo funcione, algumas adições também em GetAuthenticationStateAsync

Services/AuthAPI

.

.

private bool autenticado = false;

.

.

public override async Task<AuthenticationState> GetAuthenticationStateAsync()

{

autenticado = false;

.

.

if (response.IsSuccessStatusCode)

{

.

.

autenticado = true;

.

.

//MÉTODO DE LOGOUT

public async Task LogoutAsync()

{

await \_httpClient.PostAsync("auth/logout", null);

//MUDANDA ESTADO DE AUTENTICAÇÃO

NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());

}

//MÉTODO QUE VERIFICA SE USUÁRIO ESTÁ AUTENTICADO OU NÃO E RETORNA UM bool

public async Task<bool> VerificaAutenticado()

{

await GetAuthenticationStateAsync();

return autenticado;

}

Com o serviço configurado, criamos uma página razor para o logout, que sempre que acessada, providencia o logout, caso haja alguém autenticado no momento.

Pages/Logout.razor

@page "/logout"

@inject AuthAPI authAPI

@code {

//SEMPRE QUE A PÁGINA FOR CARREGADA

protected override async Task OnInitializedAsync()

{

//SE HOUVER USUÁRIO AUTENTICADO

if(await authAPI.VerificaAutenticado())

{

//LOGOUT É FEITO

await authAPI.LogoutAsync();

}

//SE NÃO, FAÇO SÓ OQUE É DEFINIDO PELA CLASSE BASE

await base.OnInitializedAsync();

}

}

E para refletir essas funções de login e logout mais facilmente na aplicação, mudamos a barra de cima do mudblazor para mostrar botão de login ou botão de logout, a depender do estado de autenticação autorizado ou não.

Layout/MainLayout/Trecho MudAppBar

<**MudAppBar** **Color**="Color.Surface" **Fixed**="true" **Elevation**="2">

<**MudImage** **Src**="images/screensound-logo.png"></**MudImage**>

<**MudSpacer**/>

<**AuthorizeView**>

<**Authorized**>

<**MudAvatar** **Color**="Color.Default">

<**MudIcon** **Icon**="@Icons.Material.Filled.Person"></**MudIcon**>

</**MudAvatar**>

<**MudButton** **Class**="ml-4" **Href**="logout"

**Variant**="Variant.Outlined"

**Color**="Color.Default">Logout

</**MudButton**>

</**Authorized**>

<**NotAuthorized**>

<**MudButton** **Class**="ml-4" **Href**="login"

**Variant**="Variant.Outlined"

**Color**="Color.Default">Login</**MudButton**>

</**NotAuthorized**>

</**AuthorizeView**>

</**MudAppBar**>

Agora, a interface Blazor já processa login e logout junto com a API.

**CONTROLE DE ACESSO À PÁGINAS ESPECÍFICAS**

Para definir que o acesso a uma página específica só pode ser feito com autenticação, o que fazer? Pensando no exemplo da página que exibe os artistas, seria feito

Adição do atributo Authorize na página Pages/Artistas

@attribute [Authorize]

Indicar no app.razor que as páginas de rotas vão usar autorização

.

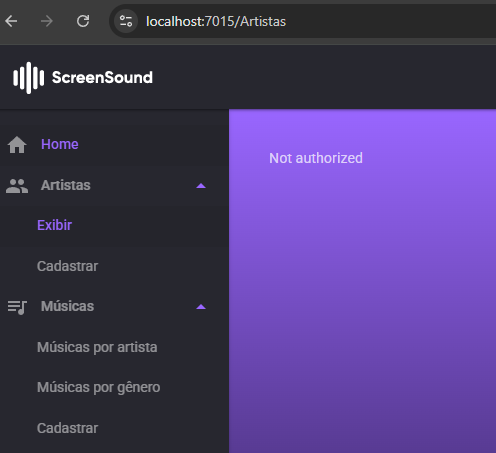
.

<**AuthorizeRouteView** **RouteData**="@routeData" **DefaultLayout**="@typeof(MainLayout)" />

.

.

Assim, ao tentar a página de exibição de artistas sem estar autenticado, aparece uma mensagem “Not authorized”



Mas essa abordagem implica que seria necessário ir em todas as páginas da aplicação que necessitam de autenticação para acesso e adicionar o atributo authorize, o que pode não ser viável. Nesse caso, é melhor definir que todas as páginas precisam de autenticação para serem acessadas e somente nas páginas selecionadas, permitir o acesso sem autenticação.

Exigir acesso a todas as páginas da aplicação

Imports.razor

@attribute [Authorize]

E nas páginas cujo acesso seja livre

@attribute [AllowAnonymous]

Na nossa aplicação, o acesso livre foi definido para as páginas home, login e logout

Podemos ainda personalizar o funcionamento do sistema para que, sempre que um usuário não autenticado tentar acessar uma página que precisa de autorização, ele será redirecionado para login. Para isso, cria-se um componente RedirectToLogin

Pages/RedirectToLogin.razor

@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

@inject NavigationManager navigation

@code {

protected override void OnInitialized()

{

navigation.NavigateToLogin("/login");

}

}

E muda-se o arquivo app.razor para definir que, sempre que for feito um acesso não autorizado, o componente RedirectToLogin é chamado, fazendo o redirecionamento para a página de login.

.

.

<**AuthorizeRouteView** **RouteData**="@routeData" **DefaultLayout**="@typeof(MainLayout)">

<**NotAuthorized**>

<**RedirectToLogin** />

</**NotAuthorized**>

</**AuthorizeRouteView**>

.

.

Para melhorar esse funcionamento do redirecionamento para login, podemos estabelecer ainda que sempre que um usuário não autenticado tentar acessar uma página restrita específica, ela seja redirecionado para o login e que caso consiga logar, seja redirecionado novamente para a página específica que estava tentando acessar.

Isso é feito primeiramente salvando a página que o usuário tentou acessar, no momento em que o componente RedirectToLogin é acessado.

@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

@inject NavigationManager navigation

@code {

protected override void OnInitialized()

{

string rotaParaRetornarAposLogin = Uri.EscapeDataString(navigation.Uri);

navigation.NavigateToLogin($"/login?ReturnUrl={rotaParaRetornarAposLogin}");

}

}

E na página de login, esse valor do parâmetro ReturnUrl é passado para a propriedade de mesmo nome, que é usada para redirecionar a aplicação.

Pages/login

.

.

//PROPRIEDADE RECEBE VALOR QUE VEM NO PARÂMETRO ReturnUrl DA URL

//CRIADA NO COMPONENTE RedirectToLogin

[SupplyParameterFromQuery]

public string? ReturnUrl { get; set; }

private async Task FazerLogin()

{

var resposta = await authAPI.LoginAsync(email, senha);

if (resposta.Sucesso)

{

if(ReturnUrl is not null)

{

//APLICAÇÃO É REDIRECIONADA PARA A PÁGINA QUE ESTAVA SENDO ACESSADA

//ANTES DO LOGIN

navigation.NavigateTo(ReturnUrl);

}

}

}

Assim, a aplicação tem um uso mais prático e intuitivo.

**SISTEMA DE AVALIAÇÃO DE ARTISTAS - MODELOS**

Para criar um sistema de avaliação dos artistas, primeiros criamos a classe AvaliacaoArtista no projeto ScreenSound.Modelos

ScreenSound.Shared.Modelos/Modelos/AvaliacaoArtista

public class AvaliacaoArtista

{

//PROPRIEDADES - CAMPOS

public int ArtistaId { get; set; }

public virtual Artista? Artista { get; set; }

//RELAÇÃO COM PESSOA SEM REFERÊNCIA DIRETO AO OBJETO

//PARA NÃO CRIAR DEPENDÊNCIA ENTRE O PROJETO DE MODELOS

//E O PROJETO DE DADOS (ONDE FICA A CLASSE PESSOA COM ACESSO)

public int PessoaId { get;set; }

public int Nota { get; set; }

}

}

E mudar no modelo Artista para que englobe a nova classe

ScreenSound.Shared.Modelos/Modelos/Artista

public virtual ICollection<AvaliacaoArtista> Avaliacoes { get; set; } = new List<AvaliacaoArtista>();

.

.

public void AdicionarNota(int pessoaId, int nota)

{

//NOTA ENTRE INTERVALO 1 A 5

nota = Math.Clamp(nota, 1, 5);

//nota = Math.Min( Math.Max(nota, 1), 5);

Avaliacoes.Add(new AvaliacaoArtista()

{

ArtistaId = this.Id,

PessoaId = pessoaId,

Nota = nota

});

}

**PERSISTIR A AVALIAÇÃO DE ARTISTAS NA BD COM ENTITY FRAMEWORK**

Para que o novo modelo AvaliacaoArtista seja persistido na BD já com sua relação com a tabela Artistas, faz-se o seguinte:

-Considerando que na tabela AvaliacoesArtistas teremos as colunas ArtistaId e PessoaId conjuntamente como chave primária, ou seja, não haverá mais de uma avaliação para uma banda vinda de uma pessoa;

-Relação entre Artistas e AvaliacoesArtistas é 1:N, ou seja, um artista com várias avaliações.

ScreenSound.Shared.Dados/Banco/ScreenSoundContext

.

.

public DbSet<AvaliacaoArtista> AvaliacoesArtistas { get; set; }

.

.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

.

.

//CHAVE PRIMÁRIA DA TABELA AvaliacoesArtistas

//É O CONJUNTO DAS CHAVES ArtistaId e PessoaId

modelBuilder.Entity<AvaliacaoArtista>()

.HasKey(a => new { a.ArtistaId, a.PessoaId });

}

Feitas as mudanças, construir a migração e realizar a atualização do banco

Ferramentas -> Console Gerenciador de Pacotes do Nuget -> Console Gerenciador de pacotes -> Com projeto ScreenSound.Shared.Dados selecionado

Add-Migration AvaliacoesArtistas

Update-Database