**DOWNLOADS NECESSÁRIOS**

Visual Studio 2022 (com ASP.NET)

VSCode

Postman

DB Browser for SQLite

Projeto inicial - <https://github.com/alura-cursos/4320-seguranca-aspnetcore>

**CONFIGURAÇÕES INICIAIS**

Instalar gerenciador de BD do .NET - Na pasta MedVoll.Web, pelo CMD

*dotnet tool install --global dotnet-ef*

Aplicar as migrations iniciais do projeto -Pelo CMD, na pasta MedVoll.Web

*dotnet ef database update*

Com o projeto aberto no Visual Strudio - Instalar Identity - Pelo Nuget

Na interface do Visual Studio -> Ferramentas -> Gerenciador de Pacotes Nuget-> Gerenciar Pacotes

Nome: Microsoft.AspNetCore.Identity.EntityFrameworkCore (v. 9.0.0)

Alterar classe ApplicationDbContext para herdar de IdentityDbContext

public **class** **ApplicationDbContext** : IdentityDbContext

Scaffolding ao ASP.NETCoreIdentity - Criar a estrutura necessária para usar o identity na aplicação (nas views Account/Login e Account/Register)

-Clique com o botão direito no projeto MedVoll.Web no Solution Explorer.

-Escolha o menu: Add > Add New Scaffolded Item.

-Na lista, selecione Identity.

-Marque os arquivos necessários:

Account\Login

Account\Register

-No campo DbContext class, selecione:ApplicationDbContext (MedVoll.Web.Data).

Gerar migração para o esquema do ASP.NET Core Identity - Pelo CMD, na pasta MedVoll.Web

dotnet ef migrations add CreateIdentitySchema

Aplicar migração ao BD

dotnet ef database update

\*Para consultar o banco de dados da aplicação

Botão direito em vollmed.db -> abrir com -> DB Browser fo SQLite

CRIAÇÃO DE USUÁRIOS DE TESTE

Criar classe Data-> IdentitySeeder

using Microsoft.AspNetCore.Identity;

namespace MedVoll.Web.Data

{

public class IdentitySeeder

{

//MÉTODO QUE CRIA DOIS USUÁRIOS DE PAPEL User

public static async Task SeedUsersAsync(IServiceProvider serviceProvider)

{

//OBJETOS UserManager E RoleManager - GESTÃO DE USUÁRIOS E PAPÉIS

var userManager = serviceProvider.GetRequiredService<UserManager<IdentityUser>>();

var roleManager = serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();

// Verifica e cria a função "User", se necessário

// CRIA O PAPÉL DE User, CASO AINDA NÃO EXISTA

const string userRole = "User";

if (!await roleManager.RoleExistsAsync(userRole))

{

await roleManager.CreateAsync(new IdentityRole(userRole));

}

// COM A FUNÇÃO CreateUserAsync - CRIA DOIS USUÁRIOS ESPECÍFICOS

await CreateUserAsync(userManager, "alice@smith.com", "Password@123", userRole);

await CreateUserAsync(userManager, "bob@smith.com", "Password@123", userRole);

}

//MÉTODO QUE CRIA USUÁRIOS-DADO O OBJETO UserManager, EMAIL, SENHA E PAPÉL

private static async Task CreateUserAsync(UserManager<IdentityUser> userManager, string email, string password, string role)

{

// SE O USUÁRIO JÁ EXISTE (VERIFICADO PELO E-MAIL) NÃO CRIA

if (await userManager.FindByEmailAsync(email) != null)

{

return;

}

//SE NÃO EXISTE..

var user = new IdentityUser

{

UserName = email,

Email = email,

EmailConfirmed = true // Para evitar a necessidade de confirmação de email

};

// Cria o usuário

var result = await userManager.CreateAsync(user, password);

if (result.Succeeded)

{

// Atribui a função ao usuário

await userManager.AddToRoleAsync(user, role);

}

else

{

throw new Exception($"Erro ao criar usuário {email}: {string.Join(", ", result.Errors)}");

}

}

}

}

Program->Linha22 - builder.Services.AddDefaultIdentity.... Fazer com que a aplicação seja capaz de usar o IdentitySeeder.cs para adicionar papéis aos usuários

builder.Services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)

.AddRoles<IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>();

Program-> Linha 53 e abaixo - Criar um escopo ServiceProvider que vai ser capaz de injetar dependências entre as classes do projeto

using (var scope = app.Services.CreateScope())

{

var services = scope.ServiceProvider;

try

{

await IdentitySeeder.SeedUsersAsync(services);

}

catch (Exception ex)

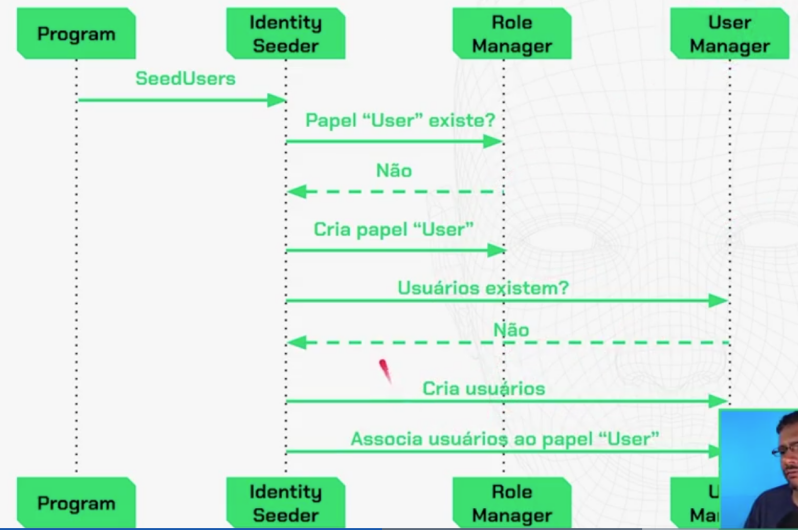
{

Console.WriteLine($"Erro ao executar o Seeder: {ex.Message}");

}

}

Diagrama de sequência - Criação de usuários teste



Como consultar as informações inseridas na BD?

Botão direito em vollmed.db -> abrir com DB Browser for SQLite-> Browse Data->

Assim é possível consultar nas tabelas AspNetUsers e AspNetRoles os usuários e papéis criados pela última execução.

Lembrando que: considerando como o código foi escrito, caso seja executado novamente, não vai duplicar os usuários/papéis ou gerar erro, dado que, caso já existam os registros, ele simplesmente não faz nada.

**CSRF - CROS-SITE REQUEST FORGERY**

Tipo de ataque onde o hacker se aproveita da sessão em aberto do usuário para acessar métodos que precisem de autenticação. No momento, a aplicação não tem proteção contra este tipo de ataque, tanto que, se o método de adicionar médico for acionado, pelo Postman, ele funciona (o que simula um uso indevido de sessão)

Como evitar?

Em Program->Linha 32 - var app = builder.Build(); - escrever acima

//MIDDLEWARE CONTRA ATAQUES CSRF

//

builder.Services.AddAntiforgery(options =>

{

options.Cookie.Name = "VollMed.AntiForgery"; // Nome personalizado do cookie

options.Cookie.HttpOnly = true; // Evitar acesso via JavaScript

options.HeaderName = "X-CSRF-TOKEN"; // Cabeçalho personalizado para APIs

});

Em Controllers->MedicoController - Mudar o método SalvarAsync para usar o token de autenticação do cookie da sessão, ou seja, evitar acessos externos não autenticados

//MÉTODO QUE ADICIONA UM MÉDICO NA APLICAÇÃO

//VALIDAÇÃO DO COOKIE COM TOKEN DE AUTENTICAÇÃO

//COOKIE CONFIGURADO EM PROGRAM->builder.Services.AddAntiforgery(options....

[ValidateAntiForgeryToken]

[HttpPost]

[Route("")]

public async Task<IActionResult> SalvarAsync([FromForm] MedicoDto dados)

{...

CONTROLE DE AUTORIZAÇÃO

Como controlar o acesso aos métodos de MedicoController?

Por exemplo, na homepage da aplicação, clicando em médicos, qualquer pessoas, sem autenticação, consegue visualizar e editar os dados dos médicos.

Isso pode ser remediado com a definição do atributo [Authorize] para todos os métodos da classe pertinente

No exemplo do projeto, será definido o atributo [Authorize] para todos métodos das classes MedicoController e ConsultaController

Controllers->MedicoController

[Authorize]

[Route("medicos")]

public class MedicoController : BaseController ...

Controllers->ConsultaController

[Authorize]

[Route("consultas")]

public class ConsultaController : BaseController...

Program->Linha 38 - Acima de var app = builder.Build()

//CONTROLE DE AUTORIZAÇÃO - [Authorize] NO INÍCIO DA CLASSE

builder.Services.AddAuthorization();

Program->Linha 56 - Acima de app.UseAuthorization

//USO DE UM MIDDLEWARE DE AUTENTICAÇÃO NO PIPELINE DE REQUISIÇÕES

app.UseAuthentication();

Program-> Linha 63- Abaixo de app.MapControllerRoute....

//HABILITA AS PÁGINAS DE LOGIN E CADASTRO - FEITO NO SCAFOLDING

//E HABILITA O USO DESSES RECURSOS NOS ARQUIVOS ESTÁTICOS DO ASP.NET CORE IDENTITY - CSS, JS

app.MapRazorPages().WithStaticAssets();

Feito isso, agora quando, a partir da homepage, o usuário acessar a aba médicos, ao invés de permitir o acesso, ele será redirecionado para a página de login.

\*Caso o login seja feito, como ainda não foi implementado o logout, para fazer o logout, acessar [https://localhost:7058/Identity/Account/Logout](https://localhost:/7058/Identity/Account/Logout)

Ainda pensando em controlar acesso, para que a homepage do site seja aberta a todos, sem uso de autenticação, caso seja necessário explicitar isso, um atributo [AllowAnonymous] pode ser utilizado no início da classe

[AllowAnonymous]//PERMITE ACESSO AOS MÉTODOS DESSA CLASSE SEM AUTENTICAÇÃO

public class HomeController : Controller

{...

\*Sem definir, por padrão, já permite acesso sem autenticação

CONFIRMAÇÃO DE E-MAIL PARA CADASTRO

Na aplicação, pode ser configurado a exigência do usuário confirmar o cadastro na sua caixa de entrada de e-mail. Como a aplicação não tem emissor de e-mail, ela simula

Program->Linha 25 - abaixo de builder.Services.AddDefaultIdentity<IdentityUser...

//CONFIRMAÇÃO DE E-MAIL PARA CADASTRAR USUÁRIO

//EM TEORIA, O USUÁRIO PRECISARIA ACESSAR SUA CAIXA DE ENTRADA PARA CONFIRMAR O E-MAIL E CADASTRAR

//MAS COMO A APLIAÇÃO NÃO TEM UM EMISSOR DE E-MAIL, ELA MESMA OFERECE A OPÇÃO DE CONFIRMAR

builder.Services.Configure<IdentityOptions>(options =>

{

options.SignIn.RequireConfirmedEmail = true; // Exigir e-mails confirmados para login

options.SignIn.RequireConfirmedPhoneNumber = false; // Não exigir confirmação de número de telefone

});

BLOQUEIO DE ACESSO POR 2MIN APÓS ERRAR A SENHA 3X

Program-> Linha 25 - abaixo de ....options.SignIn.RequireConfirmedEmail...

//BLOQUEIO DE TENTATIVA DE ACESSO POR 2 MINUTOS APÓS ERRAR A SENHA 3X

builder.Services.Configure<IdentityOptions>(options =>

{

options.Lockout.AllowedForNewUsers = true;

options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(2);

options.Lockout.MaxFailedAccessAttempts = 3;

});

Areas->Identity->Pages->Account->Login.html->Login.cshtml.cs

Linha 114 - dentro do método OnPostAsync, que é responsável por efetuar o login, mudar um atributo da função PasswordSignInAsync, que é lockoutOnFailure, passando valor true

var result = await \_signInManager.PasswordSignInAsync(Input.Email, Input.Password, Input.RememberMe, lockoutOnFailure: true);

Sendo assim, se o usuário errar a senha 3x, é bloqueado por 2min.

É possível confirmar até quando o usuário fica bloqueado na tabela AspNetUsers, no campo LockOutEnd.