**SUMÁRIO**

PADRÃO MVC 2

CONTROLLER 3

MODEL 7

VIEWS 8

**PADRÃO MVC**

O padrão MVC **(Model, View e Controller)** juntos eles formam um padrão arquitetural, diferente de estilo arquitetural. No **estilo arquitetural** definimos uma arquitetura por completa. No **padão arquitetural** ela aplica uma aplicação em cima de uma arquitetura.

Nesse padrão temos uma separação de responsabilidades, cada parte da estrutura é responsável por um processo.

Essa arquitetura está presente desde 1976 criado pela Xerox, sendo um padrão aberto aplicando-o em diversas plataformas.

**Controller:** é responsável de fazer as interações com as outras partes do sistema, banco de dados, models e views.

**Model:** é a classe que terá todas as propriedades da classe.

**View:** é a exibição das informações na tela para o usuário.

**CONTROLLER**

**ROTAS**

Rotas são basicamente **estruturas de navegação** personalizadas para que a URL da aplicação possua determinado padrão e atenda às necessidades de passagem de parâmetros.

As rotas possuem um **name**, possui um nome de rota, podendo ter outros. **Template**: mostra o caminho padrão que deve ser seguido na aplicação.

**Attribute Routes**, é uma maneira diferente de trabalhar com rotas, tornando-as mais flexíveis e fáceis de personalizar. Elas são personalizadas diretamente nas controllers. É uma boa recomendação, pois, com ela trabalhamos com personalização isoladas, não para o projeto todo.

**ACTION RESULTS**

No ASP.NET Core, uma **Action Result** é o tipo de retorno da action da Controller, é utilizada a interface IActionResult. que pode retornar alguns tipos de resultados.



**PROTOCOLOS HTTP – VERBOS BÁSICOS**

* **Get:** Um request (pedido) de uma informação ao servidor. É feito através da URL.
* **Post:** Envia informações ao servidor (formulários).
* **Put:** Similar ao Post, envia informações ao servidor. Sua diferença é por ser utilizado para atualização de informações existentes.
* **Delete:** Solicita a exclusão de uma informação no servidor através da URL indicada.
* **LEITURA DAS DEFINIÇÕES DOS METÓDOS HTTPS.**

**CONVENÇÕES DO ASP.NET MVC:**

* Se ao chamarmos uma View, e no nome da nossa Action não possuir o mesmo nome da nossa View, podemos especificar dentro do chamado: (“**return View(nome da view)**”.
* **A rota padrão sempre deve ser a última especificada**, pois, quando desenvolvermos nossa aplicação, o ASP ele criará “pontos”. Nas rotas definidas, ele irá testando uma por uma, a fim de garantir que a rota passada na URL seja compatível. Sendo assim, é lógico deixarmos a nossa rota padrão por último.
* Os nomes passados como parâmetros em um método, **ele deve ser igual** **ao definido na rota**, pois, caso diferentes, ele será reconhecido como um parâmetro diferente.
* Para passar parâmetros que não estão especificados na rota, colocamos? parâmetro**=” Valor**”.
* É possível passar as rotas também pelo controlador, fazendo assim, toda IActionResult (view) ter um padrão ou caminho definido.
* É possivel também fazer uma sobrecarga de rotas. (CUIDADO: Cuidado com os conflitos de rotas).
* Nota: A última sobrecarga sempre será reconhecida como padrão, porém, ela aceita as outras sobrecargas caso ela for chamada.
* Podemos passar parâmetros para a rota também [**Rota(“consumo”/{id}/{preco?}**)]
* Podemos definir o tipo de parâmetro que serão passadas na url. Para isso, basta colocarmos nosso parâmetro e dois pontos o tipo. Exemplo: **{id:int}**

**ACTION RESULTS**

As Actions sempre retornam um resultado. As actions elas podem retornar diversos tipos de arquivos. A sua grande maioria de forma mais explicita dentro da Controller, que possui todas as heranças desses arquivos.

**MODEL**

No MVC um **Model** é a representação de um objeto do mundo real. Na maioria das vezes, este objeto pode representar uma tabela de um banco de dados.

**Annotations** é um importante recurso utilizado principalmente para especificar que tipo de dado a propriedade deve receber. Elas podem definir tamanho, padrões, obrigatoriedade etc.

**VALIDAÇÃO DA MODEL**

Para validarmos a nossa própria model, acessamos o “dicionário” de erros que a Model e o ASP fazem para nós. Com isso, conseguimos saber todos os erros.

Uma das formas de validar a model é o **ModelState.IsValid**

Uma outra forma é acessando o **ModelState.Values.SelectMany.**  foreach (var error in ModelState.Values.SelectMany(m => errors))

{

Console.WriteLine(error.ErrorMessage);

}

**OBS:** A model com **“m” minúsculo**, define com o tipo que essa model está fazendo referência. **Com “M” maiúsculo**, utilizamos a model direta de fato, com suas possíveis propriedades.

**VIEWS**

* **Razor Views:** as Razor Views é o motor de renderização das views no MVC. Ele transforma as views em HTML puro, que serão interpretadas pelo browser. Elas também podem ser tralhadas com models, **CADA VIEW PODE TRABALHAR APENAS COM UMA MODEL.**
* **Tag Helpers:** as tag helpers são um auxílio que o asp.net dá as páginas HTML. Ela nos ajuda com diversas coisas, desde uma validação de um input (**asp-validation)**, trabalhar com conexões com models (**asp-for**). Assim, com as tag helpers, as nossas páginas HTML ficam muito mais inteligentes.
* **\_ViewStart Page:** é dizer qual página que será o layout, ou, a página de padronização das views.
* **\_ViewImports Page:** Importar componentes que serão utilizadas de forma global.
* **Shared:** Aqui é onde ficam as views que serão compartilhadas em outras views. Por exemplo: página de erro, página de avisos etc.
* **Home:** Por convenção do ASP.NET MVC, todas as views elas ficaram em pastas de acordo com sua Controller. Assim, cada pasta corresponde as views de determinadas Controllers.
* **\_Layout.cshtml:** nessa página é onde determinamos como será o nosso layout, formato, padrão e estilos.
* **Partial Views:** As partial views são pedaços de views que podem ser reaproveitadas em qualquer outra view. **Exemplo:** quando tivermos um pedaço de html que sempre se repete, podemos colocar em **Partial Views**, que poderão ser implementadas dentro de outras views. As partial views **dependem** de um model, limitando assim o seu uso.
* **View Components:** Parecido com uma Partial View, as Views Components são componentes independentes que auxiliam no desenvolvimento das views. Elas possuem processamento **server-side** e são **exibidos como um pedaço de HTML**, assim como as partial view. Além disso, elas são **independentes de models**, podendo trabalhar de forma mais eficiente. **Exemplo**: São excelentes para componentizar recursos da página, como o carrinho de compras.

**UTILIZANDO AS PARTIAL VIEWS**

* Os underlines na frente dos nomes das **Views,** por convenção,representam que aquela view trabalha em conjunto com outras views.
* Para criar uma Partial View, podemos utilizar o Scaffold, porém, ele é um pouco mais lento. Nada impede de criar na mão.
* Para utilizar essa Partial criada em uma outra **view,** da forma atual, podemos utilizar a **tag helper: <partial name=”\_NomeDaPartial”>**
* Caso queiramos criar uma partial assíncrona, podemos usar o **HTML helper**: **@async Html.PartialAsync(“\_NomeDaPartial”)**.
* **OBS:** Podemos utilizar sempre que quisermos reaproveitar um código. DESDE QUE, essa view, seja compatível com sua model.

**UTILIZANDO AS VIEW COMPONENT**

* Criar uma pasta chamada “**ViewComponents”** no seu projeto.
* Criar uma classe.
* Essa classe deve herdar de **ViewComponent. (using Microsoft.AspNetCore.Mvc).**
* Podemos dar o nome para essa componente com uma annotation própria**: [ViewComponent(Name =””)**].
* Ela precisa de um método invoke para funcionar.
* Ela pode retornar uma **View**, porém, devemos cria-la de forma especial. Em nossa pasta Shared, criamos uma pasta chamada **“Components”**
* Dentro da pasta Components, criamos uma pasta com o nome do nosso componente.
* Por fim, dentro teremos uma view chamada **“Default”**, na qual, nossa view chamará por padrão.
* Colocamos esse componente em nosso layout **<vc:NOMEDOCOMPONENTE><vc:NOMEDOCOMPONENTE>**, e por fim, colocamos nosso tag helper em **\_ViewImports.**

**VALIDAÇÕES DE FORMULÁRIO**

* Uma vez ao fazer a validação na model, não precisamos validar no formulário, apenas reutilizamos a mesma validação.
* Caso tenha que fazer uma amostra ao usuário de um resumo de todas as suas validações, você pode utilizar uma **validation summary**. Caso não queira que apareça para o usuário todas as informações, apenas colocamos na **summary="ModelOnly",** assim será passado ao usuário apenas o que voltou do banco de dados.
* **É IMPORTANTE, AO VALIDAR UM FORMULÁRIO**, que a validação seja feita antes mesmo de entrar na controller, sendo assim, em nossa page view **via client**, adicionamos uma

**@section Scripts {**

**@{**

**await Html.RenderPartialAsync("\_ValidationScriptsPartial");**

**}**

**}**

**PROJETO DO ZERO**

1. Começando uma solution vazia (**blank solution**).
2. Na pasta raiz, onde foi criado o projeto, **criamos uma pasta chamada "source"**
3. De volta em nosso projeto, em cima da solution -> add -> project -> Web App -> salva na pasta **src** -> nome = é interessante colocar nome da empresa. app ou projeto.
4. Add pastas Models, Views e Controller.
5. Em configureServices -> services.AddMvc().SetCompatibilityVersion.
6. Em configure, configuro a minha rota padrão.
7. Criar uma pasta chamada Shared.
8. Dentro da pasta shared, criar a nossa \_Layout padrão.
9. Fora da pasta, em “Views” criamos a \_LayoutStart, que será responsável por dizer qual será a layout em uso.
10. Ao configurar a \_Layout, é preciso que coloquemos em \_LayoutImports, todos os recursos que serão essenciais para o funcionamento da page.
11. Um dos recursos que iremos adicionar é o @addTagHelper @, Microsoft.AspNetCore.Mvc.TagHelpers.
12. Para melhorar a estilização é importante adicionarmos os pacotes que são referentes ao front-end.
13. **Primeiro**: Criamos a pasta wwwroot, que assumirá esses arquivos.
14. **Segundo:** Adicionar Client-side Library. Podemos adionar algumas bibliotecas **Bootstrap** (bootrstrap.js, boots.css.grid, boots.css, boots.reebot), **Jquery, Jquery-Validation, Jquery-Validation-unobstrusive.**
15. **Terceiro:** Adicionar fontes awesome.
16. Adicionar as referências <environment include e exclude> na página Layout.
17. Os arquivos javaScript deverão ser carregados por último em nossa página. Isso devido a esses arquivos geralmente serem mais demorados para serem recarregados.
18. **Em Program.cs** adiciono o app.UseStaticFiles.
19. **Bundleconfig.json:** Servem para minificar os componentes dos arquivos como por exemplo, os arquivos js, css e Bootstrap.
20. Para criar novos ambientes de produção, no arquivo lauchsettings.json, definimos em “profiles” um IIS Express – Dev e um IIS Express Prod- produção, mudando ambos os ASPNETCORE\_ENVIRONMET.

**CRIANDO A PRÓPRIA TAGHELPER**

1. Criar uma pasta chamada “Extensions”, não é obrigatório criar essa pasta, mas como forma de organização irei criar.
2. Dentro dessa pasta criamos um arquivo .class chamado “EmailTaghelper” que herda de TagHelper.
3. Dentro da classe sobrescrevemos um método com o **override**, que é o ProcessAsync (duas sobrecargas).
   1. **Output.TagName = “a”**, ele diz em qual saída será em que formato, nesse caso html. Será um link**.**
   2. **Var content = await output. GetChildContentAsync();** ele buscará o conteúdo dentro da <tag> </tag>
   3. **Var target = contet.GetContent() + “@” + “desenvolvedor.io”,** o que será gerado, contato + seu domínio.
   4. **Output.Attributes.SetAtributes(**“href”, “mailto: ” + target**), ou seja,** uma <tag href para + alvo>.
   5. **Output.Content.SetContent(target);**
4. Para o VisualStudio reconhecer minha taghelper, tenho que adicioná-la em \_ViewImports uma @addTagHelpers “\*, caminho onde que a tag helper esta)
5. A sintaxe das taghelpers segue o QbabCase, na qual os primeiros nomes das tags são com letras minúsculas, caso tenhamos um complemento do nome com letra maiúscula como “EmailComercial”, separamos com letras minúsculas com **-** :email-comercial.
6. Por convenção, o ASP.Net reconhece as taghelpers pelo pelo seu nome, assim como as controllers. EmailTagHelper, ContatoTagHelper etc.
7. **Como definir parâmetros para a taghelper?**
   1. Posso criar uma propriedade string que irá receber como parâmetro algo novo. Ex: public string EmailContato {get;set;} = “gmail.com”.
   2. Na minha taghelper no html, apenas coloco na tag <email email-contato>, que irá reconhecer.

**AREAS**

As áreas proporcionam uma maneira de organizar uma aplicação ASP.NET MVC em grupos funcionais menores, cada um com seu próprio conjunto de Models, Views e Controllers.

Elas servem para caso queiramos uma Controller, uma view ou uma model fique em uma pasta diferente dos outros objetos.

Para criar uma **Area -> Project -> add scaffold item ->** MVC AREA

Ao criar uma área, em nosso programa em \_Layout, nos itens dos links do site, teremos o **asp-area**=””. Ela serve exatamente para marcamos as nossas áreas.

Nessas áreas precisamos criar todas a views, models e controllers responsáveis dentro dela.

Mas para essa view funcionar, precisamos cadastrar ela como uma área. Então em nossa controller da área, precisamos colocar a annotation [Area(“nome da área aqui”)].

Porém, mesmo com essa convenção, ela não funcionará, então precisamos entrar na Startup e marcar uma nova rota.

template: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

O Exists é apenas um padrão que mostra que há uma área.

* + - Para definirmos um padrão para essas áreas podemos fazer de duas formas:

**Primeira:** por criar uma \_ViewImports, \_ViewStart e uma \_ViewLayout nova, definindo assim um padrão diferente para essa page

* + - * 1. **Segunda:** Por utilizar a \_Layout padrão do projeto, para isso, precisamos apenas da \_ViewStart
    - Caso queiramos definir um outro nome para a pasta “Areas” que não seja essa convenção do ASP, em nossa Startup.cs em ConfigureServices, precisamos definir um serviço dizendo ao ASP, um novo padrão de Area.

**services.Configure<RazorViewEngineOptions>(options =>**

**{**

**options.AreaViewLocationFormats.Clear();**

**options.AreaViewLocationFormats.Add(item: "/Modulos/{2}/Views/{1}/{0}.cshtml");**

**options.AreaViewLocationFormats.Add(item: "/Modulos/{2}/Views/Shared/{0}.cshtml");**

**options.AreaViewLocationFormats.Add(item: "/Views/Shared/{0}.cshtml");**

**});**

* + - O objetivo de utilizar áreas é para segregarmos componentes, módulos de nossa aplicação. Caso, contrário, devemos trabalhar com rotas, pois, elas exercem o mesmo papel com perfeição.
    - Ao trabalhar com as áreas, não se esquecer da annotation da Controller. [Area(nome da area)]
    - Caso eu queira que meu usuário veja minha aplicação escrevendo pela url manualmente, eu tenho que fazer minha rota manualmento padrão. Com isso, temos a **AreaRoutes**. (mapeamento de rotas de áreas).

**INJEÇÃO DE DEPENDENCIA (DI)**

É um padrão de design de codificação que faz parte dos princípios do SOLID.

A ideia principal é obter a inversão de controle (IoC) para simplificar as responsabilidades de uma classe, tornando o código mais simples de ter uma manutenção e ser testado.

A inversão de dependência faz parte dos 5 princípios SOLID.

**DICAS**:

* + - Uma das dicas ao se trabalhar com DI, é utilizar as interfaces pois assim você segrega as responsabilidades sem depender de uma classe diretamente.
    - Em startup configuramos a DI em **ConfigureServices**.
    - Podemos adicionar nossa dependência por **services.AddTransient <Interface, objeto que será implementado>().** Nosso MVC, quando iniciarmos nossa aplicação, já deixará instanciado esse pedido para quando utilizarmos.
    - No ASP.CORE MVC, é recomendável que trabalhemos com as injeções de dependência em nosso construtor/controller. **(RECOMENDÁVEL)**
    - Podemos também fazer uma DI diretamente na view utilizando o **@inject**. **(NÃO RECOMENDÁVEL)**
    - Caso não consigamos fazer a injeção de dependência no construtor, seja por qual motivo, conseguimos através do atributo FromServices. Dessa maneira, nos colocamos ele como se fosse um parâmetro dentro do método (no caso exemplo: IActionResult). **(NÃO RECOMENDÁVEL)**

**public IActionResult Index([FromServices] IPedidoRepository \_pedidoRepository)**

**{**

**var pedido = \_pedidoRepository.ObterPedido();**

**return View();**

**}**

**CICLO DE VIDA DA DI**

É uma das partes mais importantes da DI, indo além da importância de desenvolver uma DI.

**TIPOS DE VIDAS:**

* + - **Transiente:** obtém uma nova instância do objeto a cada solicitação, fazendo uma nova alocação para cada vez que injetar o objeto em uma classe.
    - **Scoped:** ele faz uma reutilização do objeto durante toda a request. (WEB). **(RECOMENDADO)**.
    - **Singleton:** utiliza a mesma instância para toda aplicação. (CUIDADO).

**ENTITY FRAMEWORK**

O Entity framework é uma camada que fica entre a aplicação e o banco de dados. Ele é o responsável por fazer a comunicação entre a sua aplicação (models, entidades de negócios) com os dados do banco. Ele é um ORM (Object Relational Mapper).

**COMO TRABALHAR COM O ENTITY FRAMEWORK:**

* + - Em minha pasta **“Data”**, crio uma classe que fará a comunicação com meu banco de dados. “NomeDbContext”.
    - Preciso configurá-lo no meu **“Startup”**, em ConfigureServices coloco: **services.AddDbContext<**NomeDbContext**>();**
    - Porém é preciso passar um connection string que será onde vamos encontrar o nosso banco de dados. Então, em **appsetting.jSON** vou criar um outro laço:
    - “ConnectionStrings”: {

“NomeDbContext”: “Local onde encontro meu banco de dados;Database=NomeDoBancoDeDados;Trusted\_Connection=True;MultipleActiveResultSets=true”

**Trusted\_Connection:** É dizer para o framework que ele acredita que o usuário que está executando a ação é confiável.

**MultipleActiveResultSets:** Múltiplos resultados da pesquisa.

}

* + - De volta ao ConfigureServices vou criar uma option dentro parâmetro, passando a conexão da string criada. Essa option será criada essa expressão lambda. Antes disso, caso esteja criando um projeto do zero, é importante que crie um método IConfiguration Configuration, esse que será responsável por pegar a string de conexão por meio de injeção de dependência.

Public IConfiguration Configuration {get;}

Public Startup (IConfiguration configuration)

{ Configuration = configuration}

options => options.UseSqlServer(Configuration.GetConnectionString(name:”NomeDbContext))

* + - De volta a classe de contexto, “NomeDbContext”, vou criar um construtor. Esse construtor deve receber a “options” que trabalhamos em nossa classe de serviço, passando para a :base o valor dela.

public MeuDbContext(DbContextOptions<MeuDbContext> options)

:base(options)

{

}

**OPERAÇÕES DE CRUD**

* + - Em nosso desenvolvimento, ao criar uma classe devemos nos preocupar sempre com a **CHAVE PRIMÁRIA.**
    - Após criar a classe, devo registrar em meu contexto(Data) que irá mapear em nosso banco.

public Dbset <NomeDaModel> NomeDaModel(no plural) {get;set;}

* + - Invés de instanciar o objeto diretamente na classe, faço uma propriedade privada do meu contexto.

private readonly MeuDbContext \_meuDbContext;

public TesteCrudController(MeuDbContext context)

{

\_meuDbContext = context;

}

**MIGRATIONS**

Para persistirmos nossos dados dentro do banco de dados. A **migration** **sempre** olhará nossa **DbContext** e irá comparar com nosso **banco.**

**ASP.NET IDENTITY**

Interface gráfica do usuário, Aplicativo, Teams

Descrição gerada automaticamenteÉ um componente que faz a **autenticação** e **autorização** de usuários com segurança. O Identity está na versão 3.0 (versão Core) e é basicamente uma portabilidade da versão 2.0.

Pilha de como funciona o Identity:

**Data Source**: é a nossa base de dados (SQL, Azure, Mongo).

**Data Access Layer**: é uma camada de abstração dos bancos de dados, que tem um pacote de extensão que faz a leitura e persistência dos dados.

**Identity Store**: Ela faz a persistência dos dados dos próprios identity.

**Identity Manager**: Camada de negócios, onde fazemos a interação com o identity (UserManager, RoleManager).

**ASP.NET Core App**: Nossa aplicação.

**TRABALHANDO COM O IDENTITY (PROJETO QUE NÃO TENHA ELE)**

* + - Primeiro passo a fazer é instalar o suporte ao identity, através do Package Manager Console.

**Install-Package Microsoft.AspNetCore.Identity.Ui.** (Cuidado com a versão do Identity com a versão do seu asp.net core).

* + - Add Scaffold Item **-> Identity ->** Posso selecionar os arquivos de extração do identity.
    - Criará uma pasta “Area”, em **IdentityHosting**, ele criará um build da aplicação, na qual, ela deveria ficar em **Startup**. (após isso posso excluir). O **IdentityUser** é uma biblioteca que trabalha como o usuário conectado na aplicação.
    - Algo importante que não foi colocado por padrão é em **DefaultIdentity** adicionar = Add.DefaultUI(UIFramework.Bootstrap4).
    - **Após isso,** para realmente trabalhar com identity e autenticação, em Configure(), adiciono o add.UseAuthentication();
    - Posso fazer uma **migration** para após dar um **update** para o meu banco de dados.
    - A nossa classe de negócio ela está dentro da view, porém, essa view do identity não é uma view comum, ela é uma **Page.**
    - Para mudar algo das views que são criadas pelo Identity, **É PRECISO** acessar o **SCAFFOLD,** e selecionar as views que serão utilizadas

**AUTENTICAÇÃO**

Autenticação é garantir que o usuário esteja autenticado(login) para acessar as áreas de nossa aplicação.

Um Atributo muito utilizado pela autenticação, apesar de seu nome ser diferente, é o **[Authorize].** Esse atributo faz com que somente usuários logados consigam acessar partes de nossa aplicação.

Esse atributo pode ser aplicado somente em um método específico da Controller ou na Controller toda, colocando-a fora do escopo. Porém, pode ser que algumas áreas do site não sejam acessíveis. Para permitir exceções, utilizamos o atributo **[AllowAnonymous].**

**AUTORIZAÇÃO**

A autorização funciona como uma camada a mais de proteção. Nessa camada, além de estar autenticado para poder acessar á área, é preciso que o usuário seja autorizado para acessar tal parte.

É comum nós utilizarmos o mesmo **Atributo [Authorize],** porém, com o parâmetro de **(Role = ”name”).** É uma autorização de nível de papel.

Para configurar a role diretamente no SQL, vou em minha **table (ASP.NetRoles)** => **view data =>** e adiciono meu papel. Em meu **Asp.NetUsers** copio o id ou gild do meu usuário. Por fim em **AspNetUserRoles**, coloco o id do meu usuário e o id do papel. Isso em caso de falta da página de costumização.

Porém, para funcionar, é preciso que em **STARTUP** setamos o serviço de Roles em ConfigureServices **addRoles<IdentityRole>()**

**CLAIM**

A claims é uma alternativa ao uso de roles. As claims são as declarações do usuário, já as roles é um papel que o usuário assume. As claims podem ser utilizadas para atribuir declarações como, uma área de exclusão, de adição etc. O ponto positivo que como uma claim é uma **permissão,** “dificilmente” haverá uma alteração. Já em **papeis**, geralmente se tem uma troca de papeis (gestor, vendedor, administrador).

* + - Dentro do nosso banco de dados, o **dbo.AspNetRoleClaims** é uma role associada a claims, em nosso caso, vamos usar a **dbo**.**AspNetUseClaims**, para vermos como funciona.
    - Em nossa **AspNetUser**, vamos pegar o id do usuário para cadastrar em nossa claim.
    - Em nossa Controller, tenho uma action/método, que utilizara a ainda a annotation **[Authorize(Policy=”Valor atribuída a claim”)]**
    - Porém, ainda é preciso registar nossa claim em Startup. Dentro de ConfigureServices:

services.AddAuthorization(options =>

{

options.AddPolicy("Valor da Claim", configurePolicy: policy => policy.RequireClaim("PodeExcluir"));});

**Também há a possibilidade de atribuir várias claims em uma só tipo (Claim Value (PodeLer, PodeEscrever)). Porém, para implementar se torna um pouco mais difícil.**

* + - Em minha aplicação => Add => New Folder => “**Extensions**”
    - Adiciono uma classe que será responsável por receber as permissões. AuthorizationHelper. (Nome da classe não precisa ser essa) Essa classe tem que herdar de I**AuthorizationRequirement**.
    - Nessa classe adiciono uma **propriedade** junto com o seu **construtor** string que vai receber a permissão.
    - Em nossa ConfigureServices, vou adicionar um outro serviço referente a minha claim, dentro da primeira que fizemos.

Options.AddPolicy(“Valor da Claim”, configurePolicy: policy => policy.Requirements.Add(new NomeDaClasseQueVocêCriou(“Valor da Claim)))

* + - De volta em nossa classe, é necessário que criamos um Handler, pois em nossa ConfigureServices adicionamos uma Requirement. Esse handler vai testar se nosso usuário possui a claim, com o tipo e com seu valor.

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, PermissaoNecessaria requisito)

{

if (context.User.HasClaim(match: c => c.Type == "Permissao" && c.Value.Contains(requisito.Permissao)))

{

context.Succeed(requisito);

}

return Task.CompletedTask;

}

* + - Por fim, preciso startar essa Handle

**Há também a possibilidade de trabalhar com Claims de maneira um pouco mais fácil. Essa que invés de adicionar uma policy, adicionamos diretamente na propriedade.**

* + - Essa que utiliza a annotation [ClaimsAuthorize(“TipoDeClaim”, “ValorClaim”). A **primeira parte** como: “módulos/funcionalidades”, a segunda parte “ que se podem ter/fazer, permissões”.
    - Primeiro a se fazer é criar uma classe na qual ela fará a verificação da claim por meio de um método estático.
    - Essa classe terá um **método estático** que testará se o usuário está autenticado e se o usuário possui alguma claim com o nome que o método receber e o valor contido nela.

public static bool ValidarClaimsUsuario(HttpContext context, string claimName, string claimValue)

{

return context.User.Identity.IsAuthenticated &&

context.User.Claims.Any(c => c.Type == claimName && c.Value.Contains(claimValue));

}

* + - Vou criar mais uma classe, essa que será um filtro da minha claim. Ela vai testar se as claim vindas do usuário pelo contexto não são verdadeiras.
    - Por se tratar de um filtro, não posso diretamente utilizando como um atributo. Preciso criar uma classe de atributo, essa que herdará de **TypeFilterAttribute**.
    - Exemplo:

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

using System.Linq;

using System.Security.Claims;

namespace AspNetCoreIdentity.Extensions

{

public class CustomAuthorization

{

public static bool ValidarClaimsUsuario(HttpContext context, string claimName, string claimValue)

{

return context.User.Identity.IsAuthenticated &&

context.User.Claims.Any(c => c.Type == claimName && c.Value.Contains(claimValue));

}

}

public class ClaimsAuthorizeAttribute : TypeFilterAttribute

{

public ClaimsAuthorizeAttribute(string claimName, string claimValue) : base(type: typeof(RequisitoClaimFilter))

{

Arguments = new object[] { new Claim(claimName, claimValue) };

}

public class RequisitoClaimFilter : IAuthorizationFilter

{

readonly Claim \_claim;

public RequisitoClaimFilter(Claim claim)

{

\_claim = claim;

}

public void OnAuthorization(AuthorizationFilterContext context)

{

if (!CustomAuthorization.ValidarClaimsUsuario(context.HttpContext, claimName: \_claim.Type, claimValue: \_claim.Value))

{

context.Result = new ForbidResult();

}

}

}

}

}

* + - Por fim, em minha Controller, em meu método que utilizo a [ClaimsAuthorize(“Permissão”, “ValorPermissao”)] e atualizo a permissão que eu quero que aquela área possua.

PAREI MINUTO 24 AULA DE CLAIMS CUSTOMIZADAS

**Trabalhando com Configurações**

* + - Boa prática, limpar comentários que não são úteis no programa.
    - Uma das opções de organizar o meu Configure é o **Region**, mas existe opções melhores. Na opção que vamos utilizar, criaremos uma classe que herda de **IServicesCollection** que tem as bibliotecas referentes ao services e nós as exportaremos da nossa **ConfigureServices**.
    - Criar um método estático que retorna o próprio objeto, essa por sua vez tornará a classe estática.

namespace AspNetCoreIdentity.Configuration

{

public static class DependencyInjectionConfig

{

public static IServiceCollection ResolveDependencies(this IServiceCollection services)

{

services.AddSingleton<IAuthorizationHandler, PermissaoNecessariaHandler>();

return services;

}

}

}

* + - Agora, apenas devemos voltar em nossa classe Startup, em **ConfigureServices** adicionar o meu **services.NomeDaClasse**
    - Ao tentar passar a string de conexão com o banco de dados, pode dar erro no **Configuration**. Isso acontece porque ele precisa da injeção de dependência do Configuration, necessitando assim, passá-lo também a nossa classe. Essa eu posso passar diretamente como parâmetro.

**AMBIENTES DE EXECUÇÃO**

* + - Em **LauchSettings.json** dentro de **Properties** é onde criamos os ambientes.
    - É possível criarmos ambientes, mas não são totalmente necessários criar mais do que três padrões: “**Staging**” que é o pré-produção, “**Prod**” produção e o “**Dev**” o de desenvolvimento.
    - Qual a finalidade? Com isso podemos atribuir connections strings diferentes para cada ambiente. Em nosso **appsetting.jSON** temos um nó global, que serve para todos os ambientes. Como exemplo, retiramos a conection string e dentro do appsetting passamos para o **ambiente de desenvolvimento**.
    - Mas, para executarmos os ambientes automaticamente, não devemos apenas criar os ambientes, mas sim configurarmos em nossa Startup.
    - Em nossa startup, vamos recriar nossa Configure(). Em nosso método Startup vamos usar como parâmetro o **IhostingEnvironmet**, que é o nosso próprio ambiente de aplicação.
    - Nela vamos criar um builder que irá startar nossa configuração, utilizando o .**SetBasePath(hostingEnvironment.ContentRootPath)** que passa para o Asp.net que o ambiente está a partir do local da aplicação, sem um caminho raiz. Adicionamos também o **.AddJsonFile** (onde temos o **path**, **optional** e o **reloadOnChange**).
    - Por fim, adicionamos um caminho parecido do .AddJsonFile, porém, com uma interpolação ($) dizendo para utilizar o objeto json que tiver o nome em nosso ambiente. **Ex**:

.AddJsonFile(path: $"appsettings.{hostingEnvironment.EnvironmentName}.json", optional: true, reloadOnChange: true).

* + - E adicionar as variáveis de ambiente .**AddEnvironmentVariables**().

**USER SECRETS**

* + - As **UserSecrets** são necessárias para escondermos a string de conexão com o banco de dados no ambiente de produção, evitando que outras pessoas tenham o acesso a esses dados, seja por meio online etc. (**ISSO NA MÁQUINA);**
    - Em nosso appsettings, vou tirar minha conection string do meu ambiente de produção.
    - Em nossa Startup, dentro do startup eu vou testar **se** o ambiente é o de produção, se for, vamos criar nosso builder como builder.AddUserSecrets<Startup>().
    - Para utilizarmos as secrets, eu clico com o botão direito em nosso projeto e vou em “Manage User Secrets”, e lá coloco minha connectionString.

**TRATAMENTO DE ERROS**

* + - A ideia por segurança é que não devemos exibir os erros ou “estourar” os erros em **produção**.
    - O ideal, ao menos recomendável, é que em **Error View Model**, dentro das models, substituímos as propriedades existentes por nossas propriedades.
    - Após termos criado as nossas próprias propriedades, vamos em nossa Controller e fazemos a nossa lógica para tratamento de erro, passando como parâmetro o id do erro.

[Route("erro/{id:length(3,3)}")]

public IActionResult Error(int id)

{

var modelError = new ErrorViewModel();

if (id == 500)

{

modelError.Message = "Ocorreu um erro! Tente novamente mais tarde ou contate nosso suporte.";

modelError.Title = "Ocorreu um erro!";

modelError.CodeError = id;

}

else if (id == 404)

{

modelError.Message = "A página que está procurando não existe! <br/>Em caso de dúvidas entre em contato com nosso suporte";

modelError.Title = "Ops! Página não encontrada.";

modelError.CodeError = id;

}

else if (id == 403)

{

modelError.Message = "Você não tem permissão para acessar isto.";

modelError.Title = "Acesso Negado";

modelError.CodeError = id;

}

else

{

return StatusCode(404);

}

return View("Error", modelError);

}

}

* + - Faço a forma de exibição também em na nossa view **Error**, nela podemos fazer uma lógica que testará se nossa model está vazia ou não.
    - Em startup, também arrumamos o ExceptionHandler, ela por padrão, toda exception tem como o erro 500.
    - Por fim, adiciono mais um middleware. Chamado app.UseStatusCodePagesWithRedirects(“/caminho do erro ou rota/{0}”) o {0} é por padrão que passará o identificador, o id do erro pela rota.

**COMO CRIAR “LOGS”**

Log é uma forma de fazermos um tratamento e um controle sobre possíveis erros, falhas e avisos que podemos ter em nossa aplicação, registrando-as e facilitando no entendimento e na correção de um possível problema.

* + - Pago (elma.io). Para estudar e ver o funcionamento dos Logs, vamos usar o **Kisslog**, lista com provedores de logs terceiros: <https://docs.microsoft.com/pt-br/dotnet/core/extensions/logging-providers#third-party-logging-providers>
    - SITE COM A DOCUMENTAÇÃO: [.NET Core Web App — KissLog.net documentation](https://kisslog.net/Docs/SDK.install-instructions.netcore-webApp.html)
    - Utilizar a conta cadastrada: email e senha.
    - Instalar por meio do **nugget package manager** os pacotes do Kisslog.
    - Em **appsettings** colocamos as conexões referentes ao Kisslog.

"KissLog.OrganizationId"**:** "\_OrganizationId\_"**,**

"KissLog.ApplicationId"**:** "\_ApplicationId\_"**,**

"KissLog.ApiUrl"**:** <https://api.kisslog.net>

* + - Registrar as dependências dentro do Startup.
    - Em configure, adicionar o **Kisslog.Middleware** e o **RegisterKissLogListeners();**
    - Criar uma classe separada que ficara com as configurações do log.
    - MINUTO 12