**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.PartialRenderAsync(\_ValidationPartialAsync)} }**

**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.

**CLAIMS**

Claim é uma alternativa ao uso de roles. São declarações que um usuário pode ter, guardando seu nome, suas características e outros. **(ESTUDAR CLAIMSS)!!!!!!!!**

**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.**