**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 as 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 server. 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 parametro diferente.
* Para passar parâmetros que não estão especificados na rota, colocamos **?parametro=”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.