Resumo da introdução ao **AspNetCore**

ASP.NET Core é um framework aberto e customizável, saímos de um bloco monolítico (System.Web) para uma plataforma aberta e flexível.

O novo pipeline do ASP.NET Core é uma coleção de Middlewares trabalhando em conjunto no estilo “pay as you go” que significa que usaremos somente aquilo que for necessário e nada mais.

O resultado de tudo isto é uma performance e uma ótima experiencia no desenvolvimento.

**Dominando o ASP.NET MVC Core**

**Padrão MVC**

Consiste em 3 elementos: **Model, View e Controllers.**

**Controller –** controla o fluxo do mvc, cada aspecto da nossa aplicação possui uma controller. A controler acessa o banco de dados e traz os dados necessários. Umas vez que acessamos esses dados, precisamos retorna-los e preencher a Model

**Model –** model é uma **classe** que possui as propriedades dos dados que desejamos manipular. É ela quem passará os dados recebidos pelo controller para a view.

**View –** é quem exibe a interface e os dados, basicamente a interface da aplicação.

Resumindo: A controller sendo chamada (identificada pela rota), buscará os dados e irá popular a view.

**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.

**Rotas por atributo.**

Rotas por atributos é uma maneira alternativa de trabalhar com rotas, são muito mais flexíveis e fáceis de personalizar. As rotas valem a apenas para a Controller qual foi configurada.

**Models:**

**O que são?**

- No MVC um modelo é a representação de um objeto do mundo real.

Na maioria das vezes este objeto do mundo real pode representar uma tabela de um banco de dados.

No MVC um model pode ser um conjunto de informações de diversos objetos em um só, esse conceito nós chamamos de DTO (Data Transfer Objects) que são muito utilizados para diminuir o número de requisições no servidor.

**Anotações na model**

Annotations é um importante recurso utilizado principalmente para especificar que tipo de dados a propriedade deve receber.

Também é utilizada para definir tamanho, padrão e obrigatoriedade de preenchimento.

Uma model com anotações pode passar por uma validação para garantir que as especificações foram atendidas.

As mesmas anotações podem servir para mapear tipos e tamanho de colunas no banco de dados.

Outra usabilidade comum é fazer as anotações trabalharem como validação de formulários nas Razor Views.

**DRY – Dont Repeat Yourself**

O aspnetcore nos incentiva a especificar a funcionalidade ou comportamento somente uma vez e, em seguida, refleti-lo em qualquer lugar de um aplicativo. Isso reduz a quantidade de código que precisamos escrever e faz com que o código escrito seja menos propenso a erros e mais fácil de testar e manter.

Especificando as regras de validação de forma declarativa em um único lugar (na classe de modelo) e as regras são impostas em qualquer lugar no aplicativo.

**Atributos de validaçãos mais comuns:**

[Required(ErrorMessage)] – define se um campo do modelo é obrigatório ou não.

[StringLength(60, MinimumLength = 3)] – define um tamanho máximo e mínimo para o campo. Também é possível atribuir uma mensagem de erro.

[DataType.*tiposdedados*] – garante o tipo de dado que será passado para o campo.

[Display(Name = “Data de lançamento”)] – define um nome a ser exibido na propriedade.

[RegularExpression(*padra que sera utilizado*)] – Define uma expressão regular para validar o campo.

[Range(*inicio, fim*)] – Valor entre mínimo e máximo.

[Column(TypeName = “decimal(18,2)”)] – define como será o comportamento do campo no banco de dados.

[Key] – já trata o campo como chave primaria da tabela que será utilizada.

**Como efetuar as validações de modelos pela controller.**

Criamos uma instancia do objeto da model.

Para a model ser validada de alguma maneira, ela primeiro precisa ser recebida na controller.

**ModelState =** estado da model

.isValid = true or false.

É uma propriedade que validará a model.

**Views – MVC**

No MVC, o motor de renderização das Views chama-se Razor, logo os temos as Razor Views que são arquivos HTML mesclados com recursos do razor.

Os recursos do Razor servem para criar uma experiencia mais rica no desenvolvimento de páginas HTML. Podemos pensar em Views fortemente tipadas.

O mecanismo Razor transforma as Views em arquivos HTML puros para a interpretação do Browser.

**Tag Helpers**

Tag Helpers são os recursos do Razor para a geração de HTML. No ASP.NET MVC 5 este resurso chama-se HTML Helpers e são muito similares.

A grande melhoria dos Tag Helpers é a nova sintaxe muito mais agradável e próxima do HTML.

Diagrama

Descrição gerada automaticamente

**Pastas Views**

Dentro da pasta views possuímos duas pastas.

Shared – pasta de views que são compartilhadas por todo o projeto,por ex: views de erro, avisos etc.

Temos tambpem as viewsque são de alguma controler especifica, por ex:

Na pasta Home, teremos todas as views que farão parte da controler Home. Na Filmes, todas asViews que fazem parte da contoler Filmes.

**Partial Views**

Partial Views são pedaços de uma View que possuem dados e que podem ser reaproveitados em N Views, assim proporcionando mais reaproveitamento de código.

As Partial Views dependem do modelo implementado na View principal, gerando certa limitação no seu uso.

As partial Views são muito utilizadas também para renderizar dinamicamente parte de uma View através de requisições AJAX.

**View Components**

View Components é um novo recurso do ASPNET MVC Core, é uma poderoso aliado para desenvolvimento de componentes independentes das views.

Os VIew Components possuem processamento server-side independente e podem realizar ações como obter dados de uma tabela e exibir valores manipulados.

É uma excelente funcionalidade para componentizar recursos de página como um carrinho de compras por ex.

**Trabalhando com ViewComponents**

**1.** Em primeiro lugar criamos uma pasta no projeto chamada – “**ViewComponents**”, sendo uma classe C#.

- adicionamos o namespace do MVC –

using Microsoft.AspNetCore.Mvc;

2. Adicionamos o método InvokeAsync() –

public async Task<IViewComponentResult> InvokeAsync()

{

}

3. Simulando uma chamada de um banco de dados e criamos uma classe e um construtor:

public int ItensCarrinho { get; set; }

public CarrinhoViewComponent()

{

ItensCarrinho = 3;

}

- nesse caso em especifico a informação no carrinho de compras é um int, mas também poderia ser uma Model para popular o carrinho.

4. Para retornar a View do Carrinho de compras, é preciso criar uma dentro da pasta de Views “Shared”, nomeada “Components” e dentro dessa pasta iremos criar mais uma pasta nomeada “Carrinho”, dentro dessa pasta ficarão as views do carrinho, ou seja, para cada componente, teremos uma pasta.

5. Dentro de cada pasta de componentes necessitamos ter uma View bem especifica, nomeada “Default.cshtml”

**Trabalhando com Formulários.**

Uma das coisas mais comuns em aplicações é a entrada e saída de dados, geralmente a entrada de dados se da através de formulários.

No nosso exemplo criaremos um formulário para adicionar um novo filme no nosso projeto “MinhaDemoMVC”,

Utilizando dos campos que criamos na nossa model de filme.

- **Criação do HTML**

Em primeiro lugar, precisamos nos lembrar que temos no View de de layout configurada para todas as nossas outras views, então criaremos uma chamada do nosso formulário na barra de navegação do nosso layout.

Em seguida criamos uma nova pasta dentro da pasta de Views, nomeada “Adicionar” que futuramente sera no nome da nossa ação na controller desse formulário.

Na nossa viu Adicionar, criamos o formulário, utilizando o bootstrap.



Como podemos ver, primeiro importamos o model que iremos utilizar com a diretiva @model.

**Html helpers utilizados.**

<**form** **asp-action**="Adicionar" > Essa html helper funciona na mesma maneira que o atributo ‘’action=””’’ da tag <form>, é nela que definimos a ação do nosso formulário, no nosso caso colocamos “Adicionar” que será a ação dentro da nossa controller.

Nos campos de <input /> tempos a htmml helper **asp-for**="" – ela nos permite referenciar no input, qual campo da model queremos preencher naquele campo especifico.

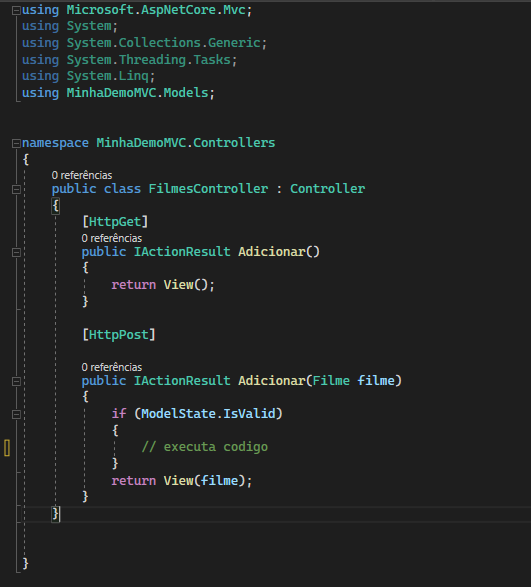
**Validação do formulário**

O primeiro ponto a ser reforçado quando trabalhamos com formulários é definição dos métodos HTTP.

Apesar dos formulários já estarem funcionando, precisamos aponta-lo para algum lugar, ou seja, utilizar o método **POST** [**HttPost**] – sendo o método que recebe o formulário do cliente.

Quando precisamos solicitar os dados, utilizamos o método **GET [HttGet]**

A controller do filme ficaria assim a principio:



É possível ver que, quando chamamos o método post, estamos passando um parâmetro “Filme filme”, que é nossa model.

Logo após, vem a validação da model (ModelState) como condicional, ou seja, se a model state for valida, faremos algo, senão retornamos a View com os dados do filme.