FIAP ON

FRAMEWORKS JAVA, .NET & WEBSERVICES

# APLICAÇÕES WEB DE TERNO E GRAVATA

4B

## LISTA DE FIGURAS

## LISTA DE QUADROS

## LISTA DE CÓDIGOS-FONTE

## SUMÁRIO

# 1 APLICAÇÕES WEB DE TERNO E GRAVATA

Agora que você foi apresentado à plataforma .NET e à linguagem C#, vamos, então, criar uma aplicação web já no padrão de projeto MVC.

## 1.1 Introdução

Neste módulo, serão apresentados o conceito do padrão arquitetural MVC (Model-View-Controller) e o framework Microsoft ASP.NET Core Web App, responsável pela implementação do padrão arquitetural em projetos Microsoft .NET.

Assim como nos capítulos anteriores, vamos usar a linguagem C# e a IDE Visual Studio 2022 para implementar aplicações web e Framework Runtime .NET 6.0.

Nos primeiros tópicos deste capítulo, apresentaremos os conceitos básicos do framework ASP.NET MVC, tais como: os componentes, a estrutura do projeto, o fluxo de navegação, as convenções, a criação da primeira aplicação, a persistência de dados e a validação. Em um segundo momento, evoluiremos para a facilidade das bibliotecas de construção de HTML, os Layouts e os Filtros.

## 1.2 Padrão MVC

MVC é um padrão arquitetural que divide uma aplicação em três camadas de componentes: modelo, visão e controlador. Usado por muitos desenvolvedores com a intenção de estruturar melhor o código de grandes aplicativos e determinar a responsabilidade de cada grupo de componente, o framework MVC é utilizado em aplicativos desktop, mobile e web. Veja na Figura “Componentes MVC” o diagrama dos componentes do padrão arquitetural MVC:

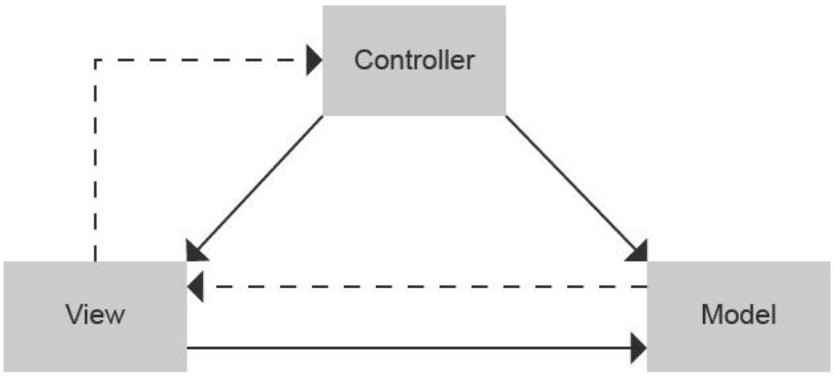


Figura 1 – Componentes MVC
Fonte: Google Imagens (2018)

A lista a seguir traz detalhadamente a função de cada componente:

- **Modelo (*Model*)** – É o componente do aplicativo responsável pela lógica do negócio e pelo modelo de dados. Será responsável por validar, recuperar e armazenar o estado das informações em uma base de dados. O modelo também é usado para notificar sua Visão (*view*) para resposta atualizada ao usuário do aplicativo.
- **Visão (*View*)** – Componente de interface com o usuário, tem a função de exibir dados atualizados dos modelos. Pode usar tabelas para a exibição de grandes conteúdos ou listas ou formulários para a digitação de dados que serão armazenados na base de dados.
- **Controlador (*Controller*)** – É o responsável pelo fluxo da aplicação e gerencia as interações dos usuários, definindo quais modelos devem ser acionados e selecionando qual visão (*view*) será apresentada para o usuário.

No passado, o .NET Framework também ofereceu outro padrão para desenvolvimento de aplicação web, chamado *WebForms*. Esse padrão também é conhecido como tradicional e tem um conceito de *postback*, que não adota os conceitos do framework MVC.

Este capítulo apresentará o Microsoft ASP.NET Core MVC (Web App) do .NET Framework, porém, as linguagens de programação web mais utilizadas (Java, PHP, .NET, Python e Ruby) possuem bibliotecas que facilitam a implementação do padrão MVC. Veja a seguir a lista dos frameworks mais conhecidos:

- PHP – CakePHP, Laravel e Symfony.

- Java – Spring MVC, JSF, VRaptor e Apache Struts.
- .NET – MVC4, MVC5, ASP.NET Core MVC, ASP.NET Core MVC (Web App) e ASP.NET Core Web App.
- Python – Django, Zope.
- Ruby – Rails.

A estrutura MVC tem como principais características a separação de conceitos, tornando cada componente responsável por um assunto e a reutilização de código. Com o uso do padrão MVC, podemos extrair algumas vantagens do desenvolvimento de uma aplicação, tais como:

- Facilidade de desenvolvimento de testes.
- Facilidade de gerenciamento da complexidade, devido à separação das camadas. Esse fator também ajuda na integração de grandes equipes de desenvolvedores.
- Ter controle completo do comportamento do aplicativo. O modelo **WebForms** utiliza o estado da informação armazenado na página e controlado pelo servidor (*ViewState*).
- Processamento centralizado das solicitações em um único controlador.

## 1.3 Criando projeto ASP.NET Core Web App

Para iniciar a criação de um novo aplicativo, precisamos entender o modelo de negócio que será implementado, quais serão seus domínios, quais informações serão armazenadas e manipuladas e quais funcionalidades serão construídas para os usuários alimentarem nosso negócio.

Assim, vamos usar como exemplo para este capítulo um modelo de negócio de conceitos básicos como produtos, fornecedores, clientes e outros. O objetivo deste capítulo não é a criação completa do aplicativo, mas sim apresentar os conceitos que devem ser aplicados para a sua finalização.

Vamos criar nosso projeto?

No Visual Studio 2022, selecione o menu **File > New > Project** (você pode usar a tecla de atalho Ctrl + Shift + N). Vamos selecionar o tipo de projeto ASP.NET Core Web App (Model-View-Controller) e a opção Next. Atente-se para que o modelo selecionado seja o que utiliza a linguagem C#.

A Figura “Projeto ASP.NET Web” mostra os passos para a seleção da linguagem e do tipo de projeto:

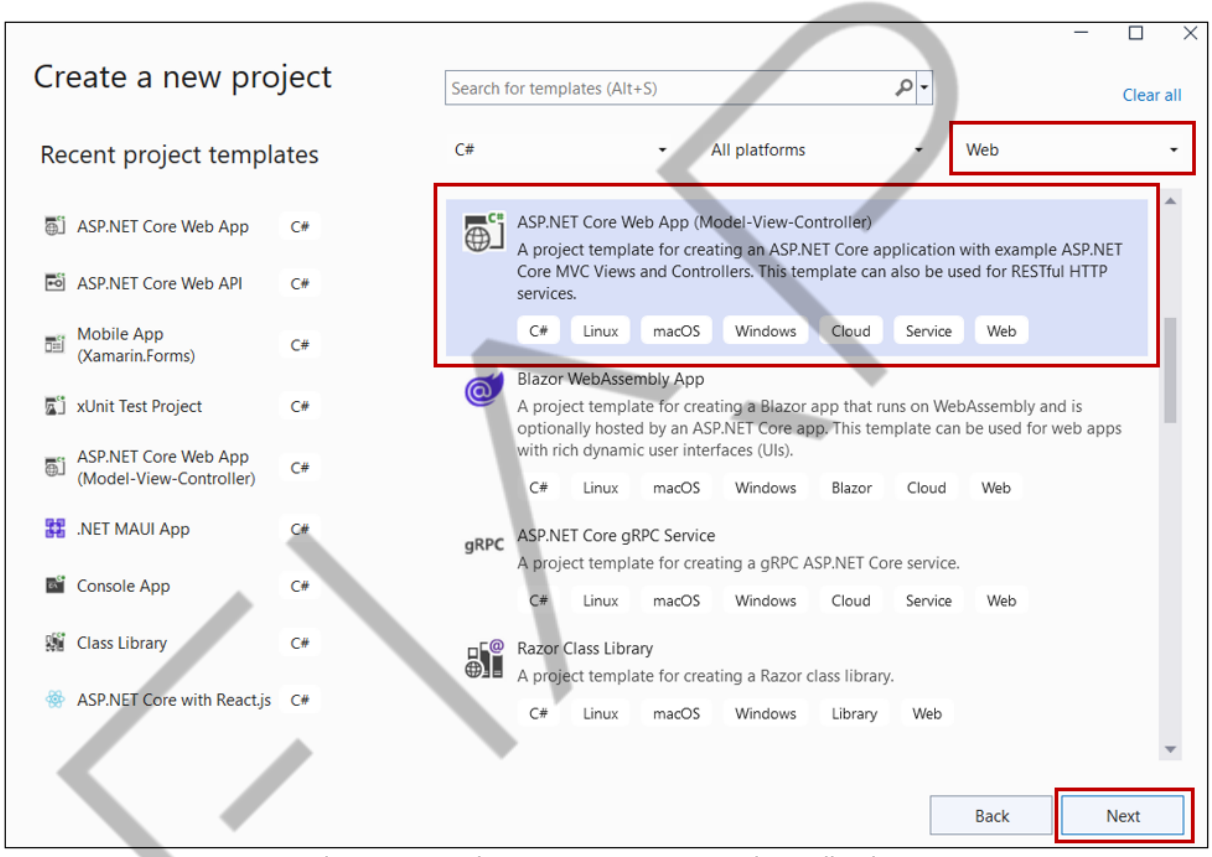


Figura 2 – Projeto ASP.NET Core Web Application
Fonte: Elaborado pelo autor (2022)

Como próximo passo, temos caixas de texto para definir o nome do projeto, o local no sistema de arquivos e o nome da solução. Para nosso exemplo, vamos usar **Fiap.Web.AspNet** como nome do projeto e da solução.

**DICA:** Crie uma pasta específica para o seu projeto. No meu caso, criei D:\workspace-net\ e selecionei essa pasta no *Location* do Visual Studio.

Na próxima janela, devemos escolher a versão do framework .NET 6.0:

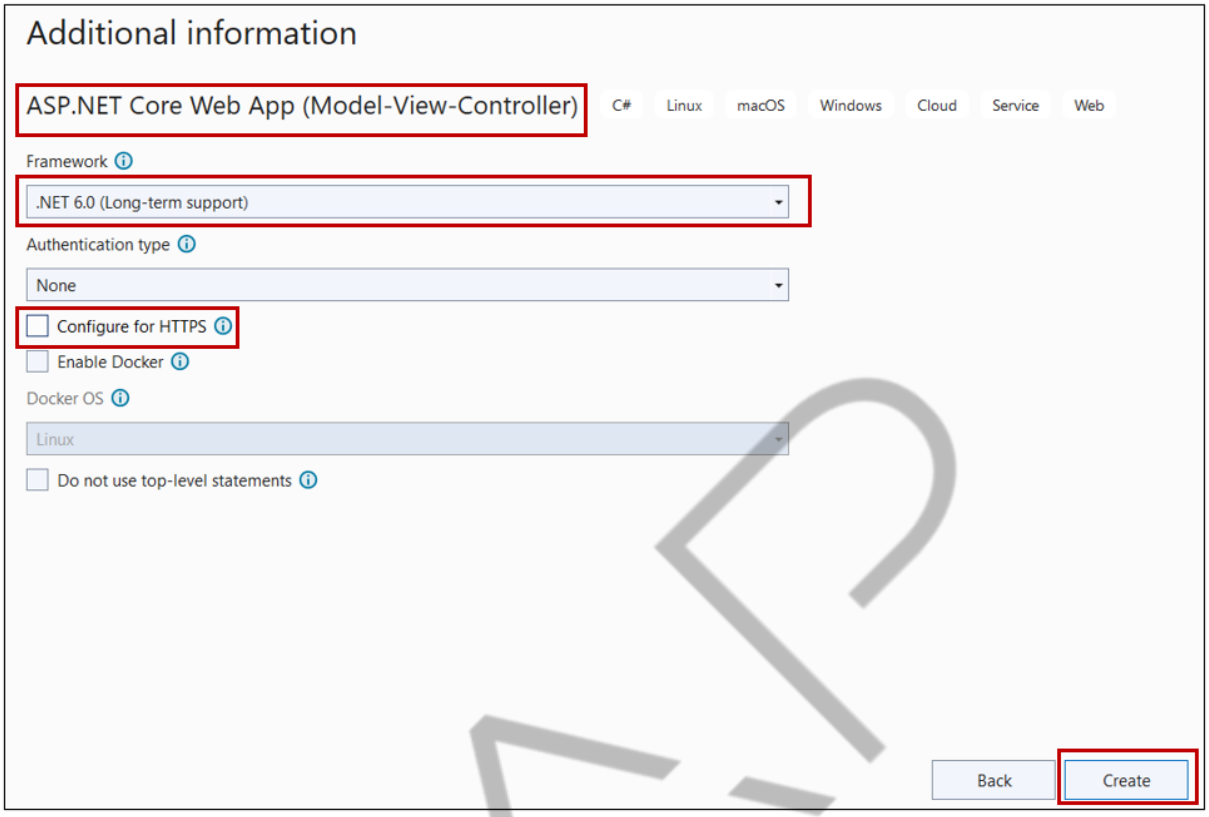


Figura 3 – Template MVC
Fonte: Elaborado pelo autor (2022)

Criado o projeto, conseguimos verificar sua estrutura. Na janela ***Solutions Explorer***, temos nossa solução, novo projeto web, e, na estrutura do projeto, foram criadas as pastas ***Controllers***, ***Models*** e ***Views***, que podem ser observadas na Figura "Estrutura do projeto ASP.NET Core MVC (Web App)":

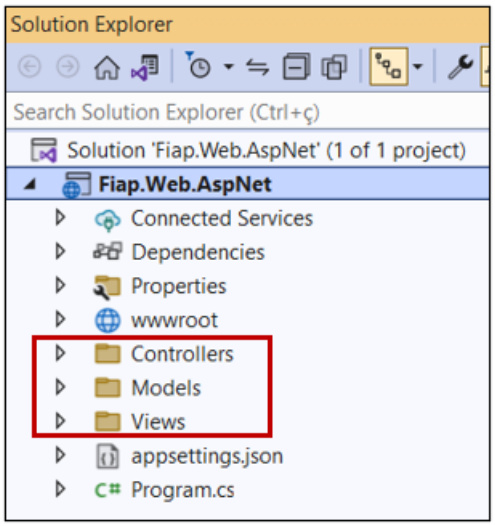


Figura 4 – Estrutura do projeto ASP.NET Core MVC (Web App)
Fonte: Elaborado pelo autor (2022)

## 1.4 Modelos

Com o nosso projeto criado, precisamos iniciar o entendimento dos componentes do MVC e a implementação do nosso conceito de negócio.

Não existe uma regra para a ordem de criação dos componentes. Algumas equipes iniciam a construção pela camada de modelos, pois possuem uma modelagem de banco de dados preestabelecida. Outras, iniciam pela visão e pelos controladores, pois assim conseguem criar um protótipo e validar o fluxo da aplicação.

Para nossa primeira implementação, vamos iniciar pela camada de modelo, na qual vamos representar nosso modelo de negócio para Clientes e seus representantes. Veja, a seguir, a representação UML para nossas classes de modelo:

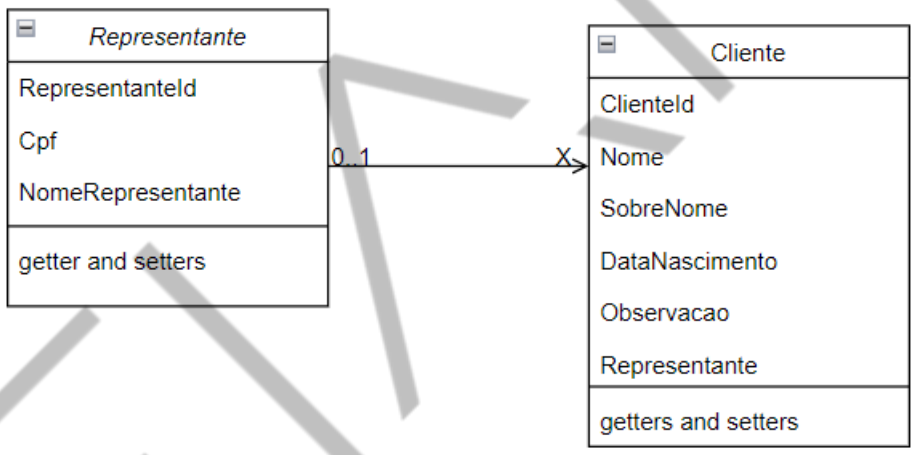


Figura 5 – Diagrama de Classe – Representante e Cliente
Fonte: Elaborado pelo autor (2022)

Os componentes da camada de modelo são simples classes C#, que devem ser adicionadas no *namespace* **Models** do projeto. Para criar o modelo **RepresentanteModel**, clique com o botão direito na pasta Models e escolha a opção **Add > Class**. Defina o nome como **RepresentanteModel.cs**, utilize o Diagrama de Classe da Figura “Diagrama de Classe – Produto e Categoria” e adicione os atributos IdTipo, DescricaoTipo e Comercializado, com seus respectivos tipos. O Código-fonte “Modelo Representante”, a seguir, apresenta a implementação do modelo Representante.

```
namespace Fiap.Web.AspNet.Models
{

    public class RepresentanteModel
    {

        public int RepresentanteId { get; set; }

        public string? Cpf { get; set; }

        public string? NomeRepresentante { get; set; }

        public string? Token { get; set; }

        public RepresentanteModel()
        {

        }

        public RepresentanteModel(int representanteId, string nomeRepresentante)
        {
            RepresentanteId = representanteId;
            NomeRepresentante = nomeRepresentante;
        }
        public RepresentanteModel(int representanteId, string cpf, string nomeRepresentante)
        {
            RepresentanteId = representanteId;
            Cpf = cpf;
            NomeRepresentante = nomeRepresentante;
        }

    }
}
```

Código-fonte 1 – Modelo Representante
Fonte: Elaborado pelo autor (2022)

**DICA:** Após adicionar uma classe no seu projeto, observe o *namespace* declarado na classe e a pasta em que a classe foi adicionada, pois devemos manter sempre o nome. Verifique também se a classe está declarada como *public*.

Seguindo os passos anteriores e o diagrama da Figura “Diagrama de Classe – Representante e Cliente”, vamos criar a classe para o modelo de Representante. O diagrama apresenta uma agregação entre Cliente e Representante, sendo assim, na classe de ClienteModel, precisamos ter uma propriedade do tipo RepresentanteModel e também um atributo do tipo IdRepresentante. Veja o exemplo:

```
namespace Fiap.Web.AspNet.Models
{
    public class ClienteModel
    {
        public int ClienteId { get; set; }
        public string? Nome { get; set; }
        public string? Sobrenome { get; set; }
        public string? Email { get; set; }
        public DateTime DataNascimento { get; set; }
        public string? Observacao { get; set; }


        public int RepresentanteId { get; set; }
        public RepresentanteModel? Representante { get; set; }        //Navigation Object
    }
}
```

Código-fonte 2 – Modelo Produto
Fonte: Elaborado pelo autor (2020)

# 2 IMPLEMENTANDO ASP.NET CORE WEB APP MVC

## 2.1 Funcionalidades

Já temos dois modelos definidos e criados em nosso projeto, precisamos, agora, elaborar os mecanismos para deixar disponível a manipulação pelos usuários. Iniciaremos com nosso modelo de **Representante**, que, para poder ser manipulado, deverá possuir os seguintes comportamentos ou funcionalidades:

- Criação de um novo representante.
- Remoção de um representante existente.
- Alteração dos dados do representante.
- Listagem de todos os representantes do sistema .

### 2.1.1 Controllers e Actions

Em um projeto ASP.NET Core MVC, toda solicitação do usuário feita pelo navegador será recebida e gerenciada por um *Controller.* Este fica responsável por receber o pedido, acionar os componentes necessários e gerar a resposta para o navegador.

Podemos criar um *Controller* para cada funcionalidade da nossa aplicação (por exemplo: CriarProduto, ExcluirProduto, AlterarProduto e ListaProdutos), essa abordagem funciona, mas não é recomendada. Para organizar melhor nossas funcionalidades, temos os conceitos das ***Actions***.

As ações (*Actions*) nada mais são do que métodos adicionados na classe de controle com o objetivo de organizar e padronizar ainda mais o nosso código. Com o uso das ***Actions***, devemos criar um controlador para cada domínio e ações para cada funcionalidade (por exemplo: *Controller* Representante, *Actions* Criar, Excluir, Alterar, Detalhar e Listar).

Todo *Controller* necessita de uma *Action*, pois, caso não seja criada, nada será executado. Além da pasta *Controller* (*namespace*), a criação de *Controllers* e *Actions* deve seguir algumas particularidades:

- O nome da classe do controlador deverá ter o sufixo ***Controller*** (por exemplo: RepresentanteControlller ou FavorecidoController).
- Os métodos que representam as ações devem ser declarados como públicos.
- Os métodos *Actions* não podem ser declarados como *static.*
- Os métodos *Actions* só podem ser sobrecarregados (*overloading*) com uso de Anotações (*Attributes*).
- O mapeamento-padrão adota o nome de *Index* para a *Action* inicial de um *Controller*. Vamos falar sobre mapeamento e rotas em capítulos futuros.
- O retorno mais comum de uma *Action* é um componente *View* em HTML implementado pela classe *ActionResult* ou pela interface *IActionResult*.
- É possível criar uma *Action* sem resposta.
- Uma *Action* tem o mapeamento um para um, ou seja, deve ser implementada para executar apenas uma ação.

As *Actions* podem ser implementadas com algumas responsabilidades diferentes, como de apresentar uma *View* ao usuário, por exemplo. Ou seja, ações que serão responsáveis por retornar um arquivo para download. Na Figura “Tipos de retorno da classe **ActionResult**” segue a especificação dos vários tipos de retorno de uma *Action*, os quais são implementados pela classe ***ActionResult***:

| Resultado de ação | Método auxiliar | Descrição |
|---|---|---|
| ViewResult | *View* | Renderiza uma exibição como uma página da Web. |
| PartialViewResult | PartialView | Apresenta uma visão parcial, que define uma seção de um modo de exibição pode ser processado dentro de outro modo de exibição. |
| RedirectResult | Redirect | Redireciona para outro método de ação usando seu URL. |
| RedirectToRouteResult | RedirectToAction RedirectToRoute | Redireciona para outro método de ação. |
| ContentResult | Content | Retorna um tipo de conteúdo definido pelo usuário. |
| JsonResult | Json | Retorna um objeto serializado do JSON. |
| JavaScriptResult | JavaScript | Retorna um script que pode ser executado no cliente. |
| FileResult | File | Retorna a saída binária para gravar a resposta. |
| EmptyResult | (Nenhuma) | Representa um valor de retorno é usado se o método de ação deve retomar um null o resultado (void). |

Figura 6 – Tipos de retorno da classe **ActionResult**
Fonte: Microsoft MSDN (2020)

### 2.1.2 Implementando *Controllers*

Apresentados os conceitos e as particularidades dos *Controllers*, chegou a hora da criação para a funcionalidade de manutenção dos tipos de produtos. Vamos lá!

Clique com o botão direito do mouse na pasta *Controllers* do projeto e selecione a opção **Add > Controller**, como na Figura “Adicionando *Controller*”, o Visual Studio apresentará a janela *Add Scaffold*. Selecione a opção **MVC Controller – Empty**, como na Figura “Selecionando o *Scaffold* do *Controller*”.

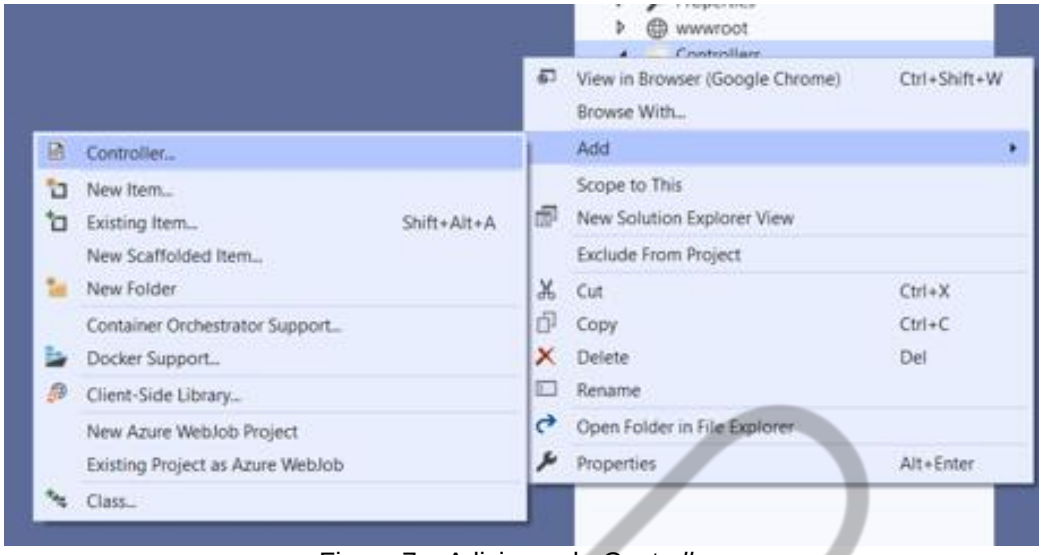


Figura 7 – Adicionando *Controller*
Fonte: Elaborado pelo autor (2022)

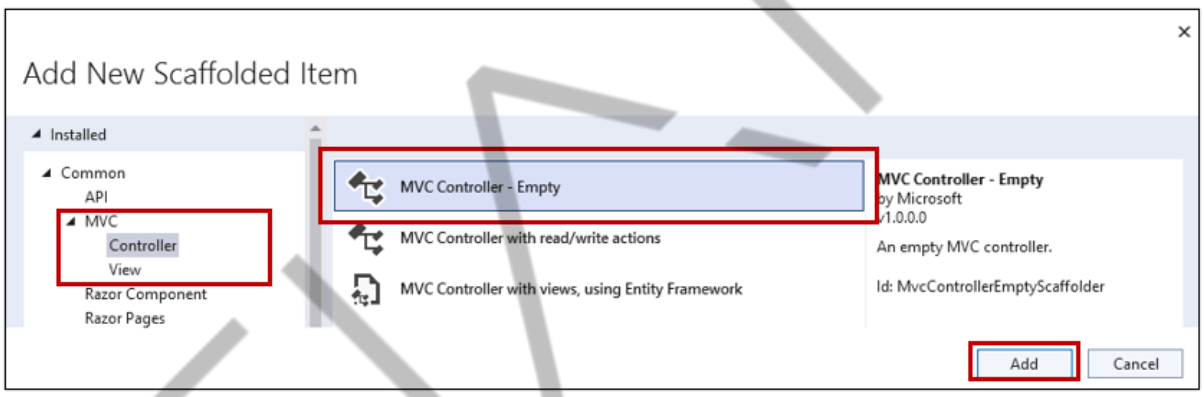


Figura 8 – Selecionando o *Scaffold* do *Controller*
Fonte: Elaborado pelo autor (2020)

O próximo passo é definir o nome do controlador, que será **RepresentanteController** em nosso projeto. Clique no botão *Add* e aguarde a criação. Lembre-se, todo *Controller* deverá ter o sufixo ***Controller*** em seu nome.

Pronto! O primeiro controlador no projeto foi criado. Agora podemos observar a classe criada no namespace *Controllers*; no código da classe *Controller,* é possível ver a importação do namespace **Microsoft.AspNetCore.Mvc** e a extensão da classe **Microsoft.AspNetCore.Mvc.Controller**. Como padrão da criação de todo *Controller*, a *action* **Index** foi adicionada na classe, por meio do método de mesmo nome, e o retorno é um objeto do tipo **IActionResult**. A Figura “Detalhes de um *Controller*” traz todas as informações do *Controller*.

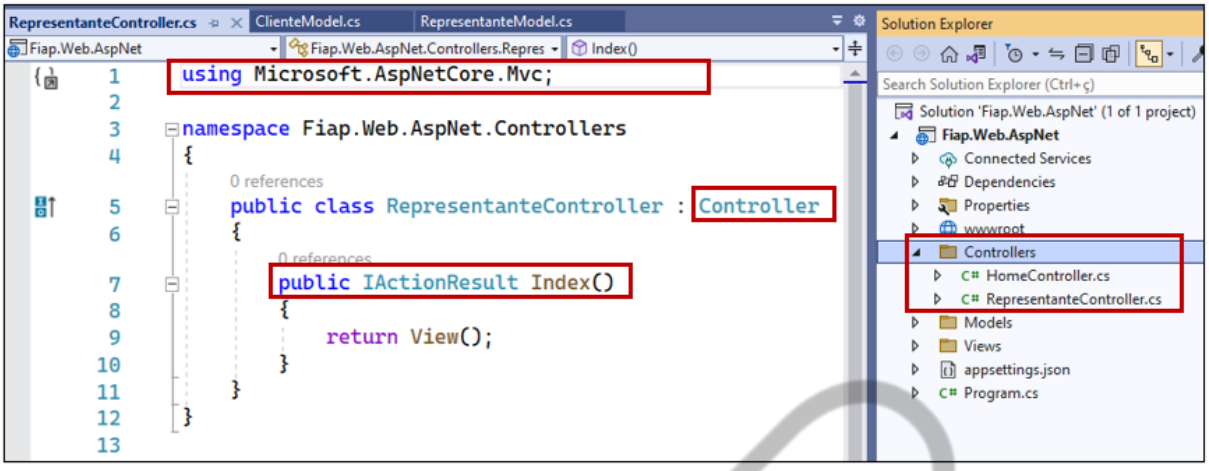


Figura 9 – Detalhes de um *Controller*
Fonte: Elaborado pelo autor (2022)

*Controller* criado, agora podemos fazer o primeiro teste. Pressione a tecla **F5** e aguarde o navegador-padrão do seu computador ser aberto. Com o navegador aberto, complemente o endereço com o caminho **/Representante** e pressione Enter. O navegador vai exibir uma tela de erro, informando que nenhuma *View* com o nome de Index foi encontrada. Apesar de apresentar uma mensagem de erro, significa que nosso teste foi bem-sucedido.

A Figura "Testando o *Controller*" apresenta o endereço completo para a execução do *Controller* e a tela de erro que indica que nenhuma *View* foi encontrada para ser exibida, pois, afinal, não criamos a *View*.

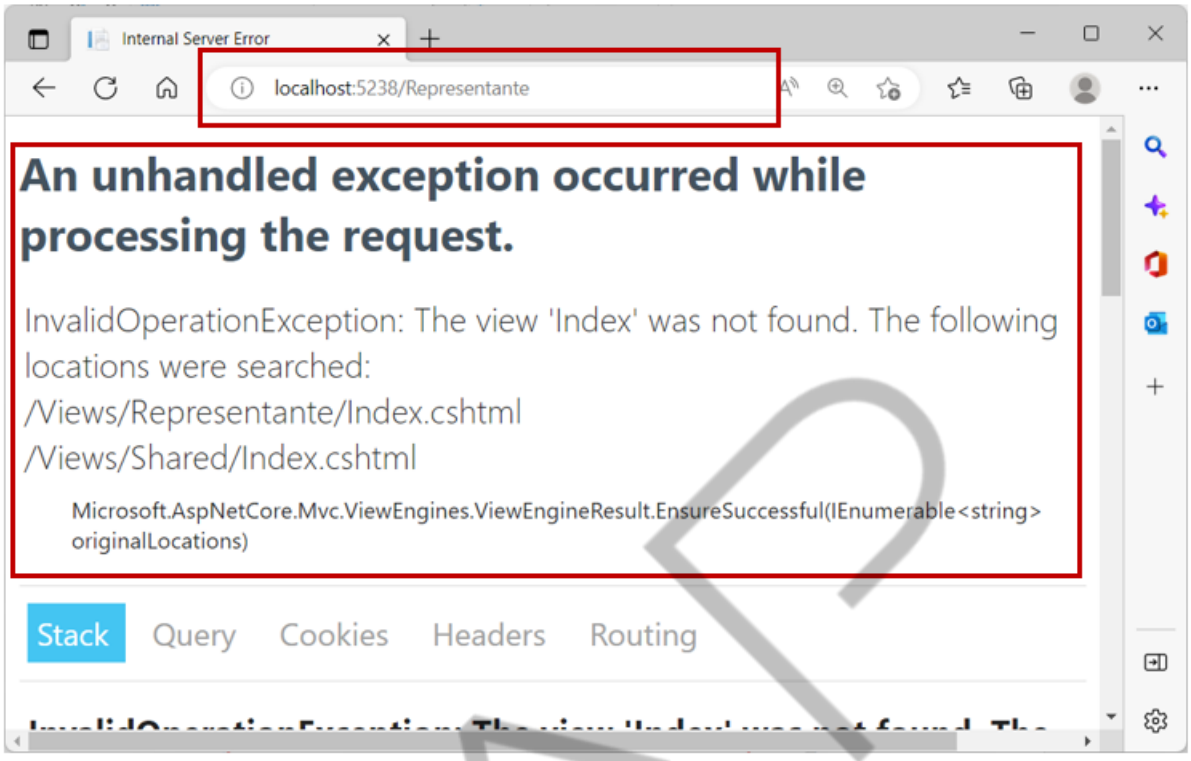


Figura 10 – Testando o *Controller*
Fonte: Elaborado pelo autor (2022)

**IMPORTANTE:** A porta usada no endereço do navegador pode ser variável. No exemplo utilizado na Figura "Testando o *Controller*", o Visual Studio definiu a porta 5238. Na execução em outro computador, podemos ter um novo número de porta. Atente-se para o número da porta gerada na execução do projeto.

### 2.1.3 Associando uma *View* e um *Controller*

Anteriormente, criamos e testamos nosso *Controller*, porém, a validação da execução foi feita por meio da tela de erro, informando que não havia uma *View* para ser exibida. Vamos, então, criar a primeira *View* e validar a execução do nosso *Controller*. A *View* será uma página HTML com uma mensagem de texto informando o nome do *Controller* e da *Action*.

Com o *Controller* **RepresentanteController** aberto na janela de edição, clique com o botão direito sobre o nome da *Action* Index e selecione a opção "***Add View***" (Adicionar edição) (uma janela com detalhes da *View* será apresentada). Selecione o modelo vazio.

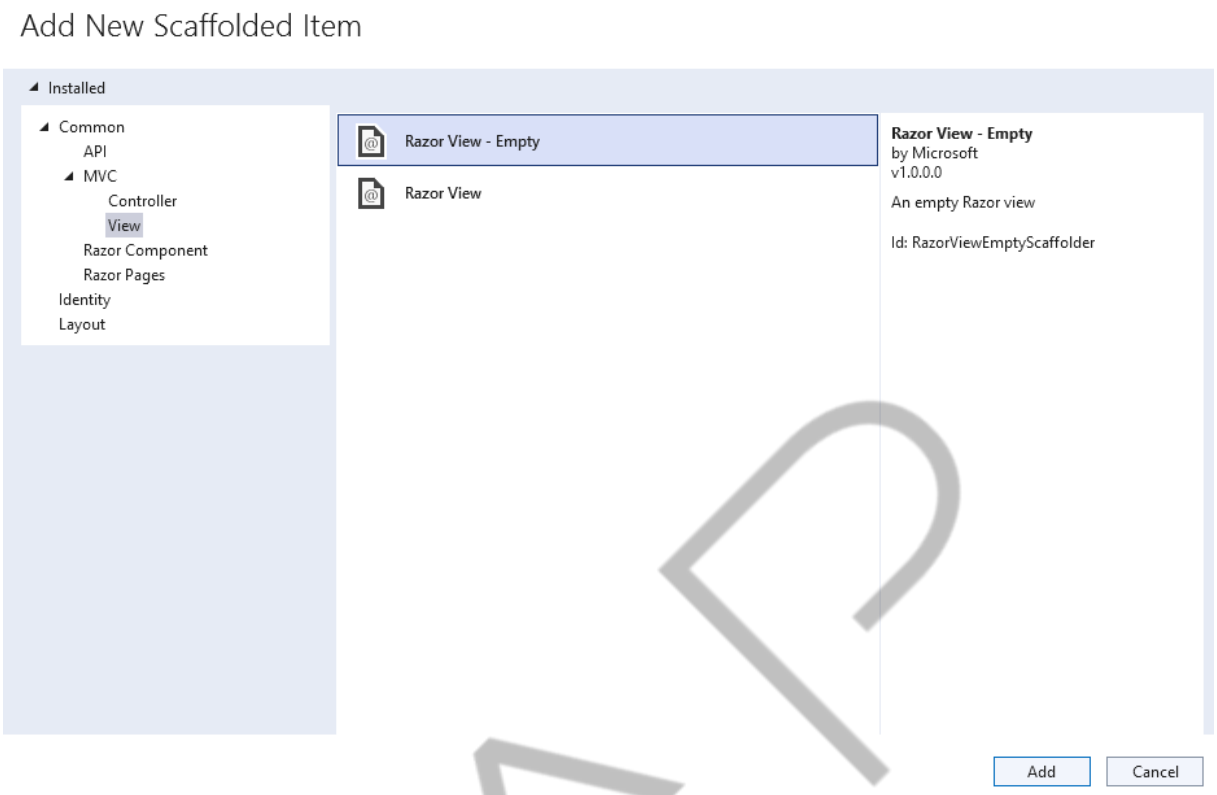


Figura 11 – Criando uma *View (Scaffold)*
Fonte: Elaborado pelo autor (2022)

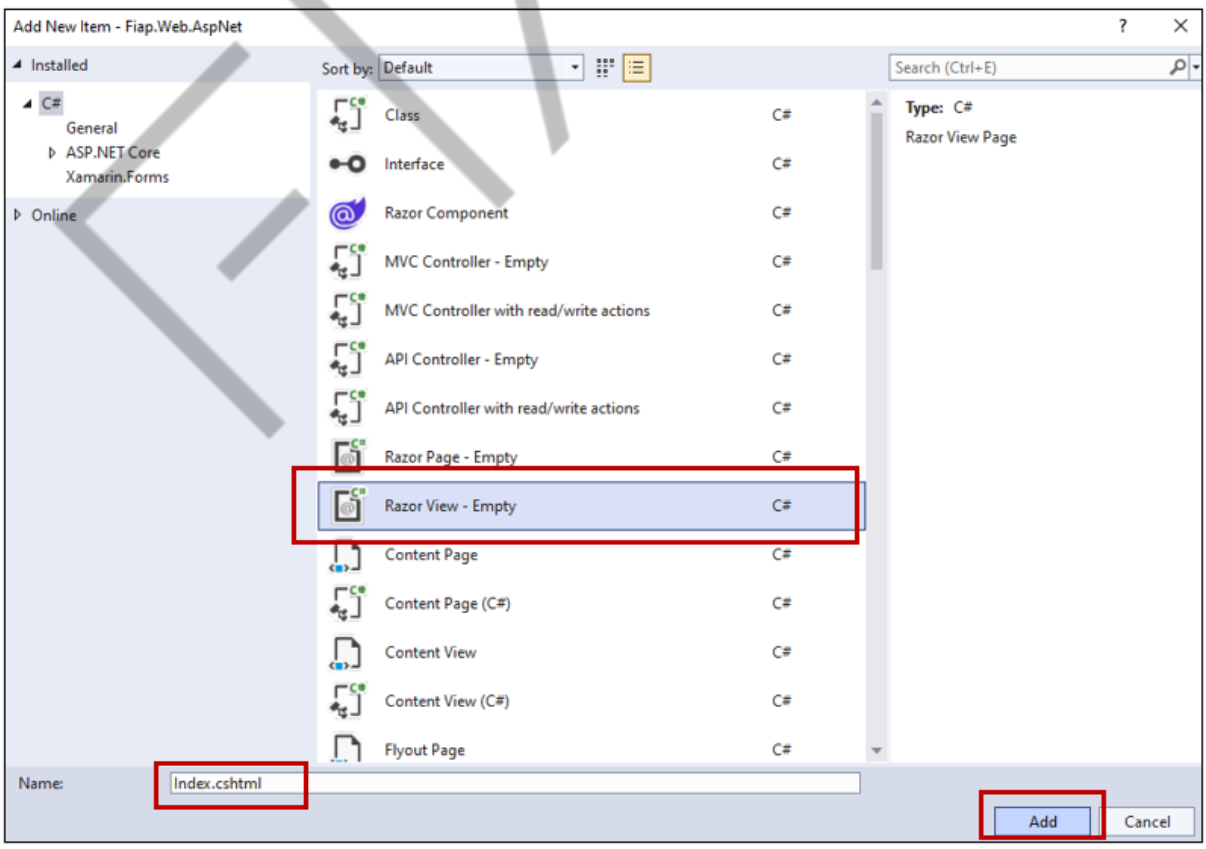


Figura 12 – Criando uma *View* (Tipo e nome do arquivo)
Fonte: Elaborado pelo autor (2022)

Com a *View* concluída, verifique na janela *Solution Explorer* se na pasta “Views” foram adicionados uma subpasta Representante e um arquivo Index.cshtml (arquivo da *View*), conforme a Figura “Estrutura da pasta *View*” abaixo:

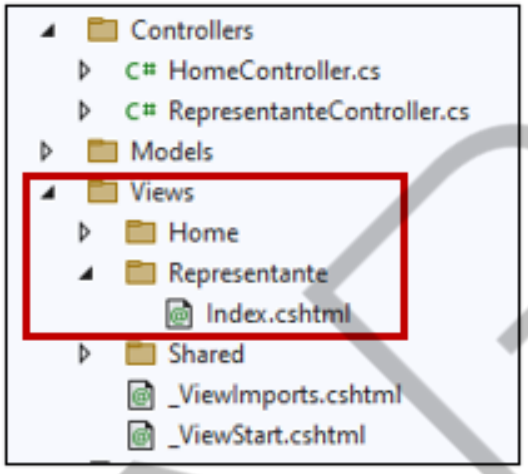


Figura 13 – Estrutura da pasta *View*
Fonte: Elaborado pelo autor (2020)

Nosso próximo passo é editar o arquivo Index.cshtml e, no bloco “<body>”, adicionar uma mensagem com o nome do *Controller* e a *Action* à qual a *View* pertence. Segue o exemplo do HTML da *View* no Código-Fonte “*View* Index para tipo de produto”:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
     Executando Controller <b>Representante</b> e a Action
<b>Index</b>
    </div>
</body>
</html>
```

Código-fonte 3 – *View* Index para representante
Fonte: Elaborado pelo autor (2022)

Arquivo editado, voltamos a testar nosso *Controller*. Pressione F5, aguarde o navegador ser carregado, informe o caminho /Representante/Index e pressione Enter.

Assim, nosso *Controller* será executado novamente e a *View* Index, que acabamos de construir, será retornada para o navegador. Segue um exemplo de *View* Index na Figura “*View* Index apresentada para usuário”.

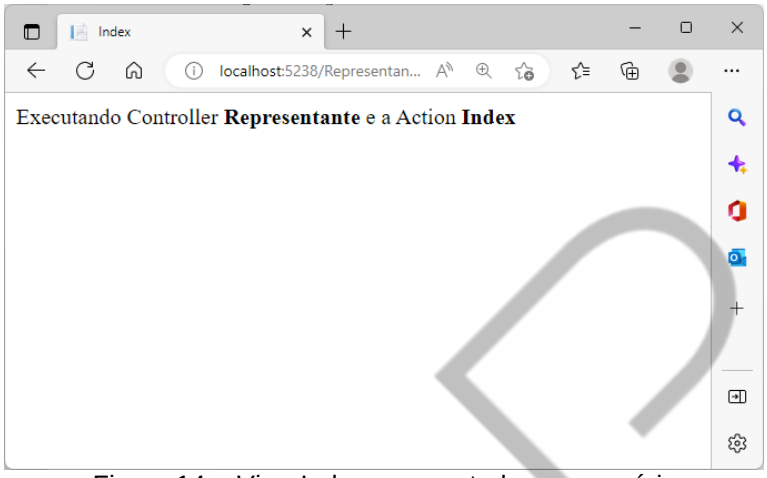


Figura 14 – *View* Index apresentada para usuário
Fonte: Elaborado pelo autor (2022)

Até aqui conseguimos criar o *Controller*, associar uma *View* e testar o aplicativo. O próximo capítulo vai apresentar como essa associação acontece e de que modo os componentes estão vinculados.

### 2.1.4 Método de retorno – *View*()

O *Controller* e a *Action* criados até este ponto retornam para a requisição a visão do mesmo nome da ação por meio método **View()**.

O método **View()** apresenta algumas sobrecargas, as quais permitem passagem de parâmetros para informar resultados diferentes, como outra *View*. Podemos alterar a *View-padrão,* passando uma *string* como parâmetro ou informando um objeto que será usado para a renderização da *View*. Veja, na Figura “Sobrecargas do método View()”, todas as sobrecargas permitidas para o retorno do método **View()**:

| | Nome | Descrição |
|---|---|---|
| | View() | Cria um ViewResult objeto que processa um modo de exibição para a resposta. |
| | View(Object) | Cria um ViewResult o objeto usando o modelo que processa um modo de exibição para a resposta. |
| | View(String) | Cria um ViewResult o objeto usando o nome de exibição que processa um modo de exibição. |
| | View(IView) | Cria um ViewResult que processa especificado do objeto IView objeto. |
| | View(String, Object) | Cria um ViewResult o objeto usando o nome de exibição e o modelo que processa um modo de exibição para a resposta. |
| | View(String, String) | Cria um ViewResult objeto usando o nome e o nome da página mestra que processa um modo de exibição para a resposta. |
| | View(IView, Object) | Cria um ViewResult que processa especificado do objeto IView objeto. |
| | View(String, String, Object) | Cria um ViewResult objeto usando o nome de exibição, o nome da página mestra e o modelo que processa um modo de exibição. |

Figura 15 – Sobrecargas do método View()
Fonte: Microsoft MSDN (2022)

# 3 ROTAS E NAVEGAÇÃO

## 3.1 Convenções

O framework ASP.NET Core MVC usa uma simples convenção para associar *Actions* dos *Controllers* às *Views*. Para o nosso exemplo do *Controller* RepresentanteController, foi criada uma subpasta com o nome “Representante” dentro da pasta “Views”, e, para o *Action* “Index” foi criado o arquivo “Index.cshtml”. A convenção de nomes e estrutura das pastas é associar as *Views* aos *Controllers*.

Essas convenções são simples e fáceis. Seguindo a padronização de nomes, já temos boa parte do trabalho reduzido e delegado para o framework. Tiramos a responsabilidade de o nosso código definir essas associações e deixamos isso a cargo do framework.

Com isso, ficam claras a facilidade e a simplicidade de seguir as recomendações de uso das nomenclaturas e as estruturas criadas e sugeridas pelo ASP.NET Core MVC.

## 3.2 Rotas da URL

Já mostramos como é feita a associação entre *Controller* e *View*, agora veremos como nossa aplicação entende a URL digitada e como ela consegue identificar qual *Controller* e qual *Action* deve executar.

Analisamos a URL da aplicação <http://localhost:5238/Representante/Index>, o primeiro bloco apresenta o protocolo, o nome do servidor e a porta de comunicação; o segundo bloco apresenta:

Representante – *Controller* responsável por gerenciar a execução.

Index – *Action* que atenderá à requisição.

A composição entre *Controller* e *Action* é conhecida como Rota e todo projeto ASP.NET Core MVC (Web App) possui uma classe C# responsável por essa configuração. Na janela da *Solution Explorer*, navegue até a classe **“Program.cs”** e

até a chamada ao método **“MapControllerRoute”**. A Figura “Configuração de Rotas”, a seguir, exibe o conteúdo da classe **Program.cs**:

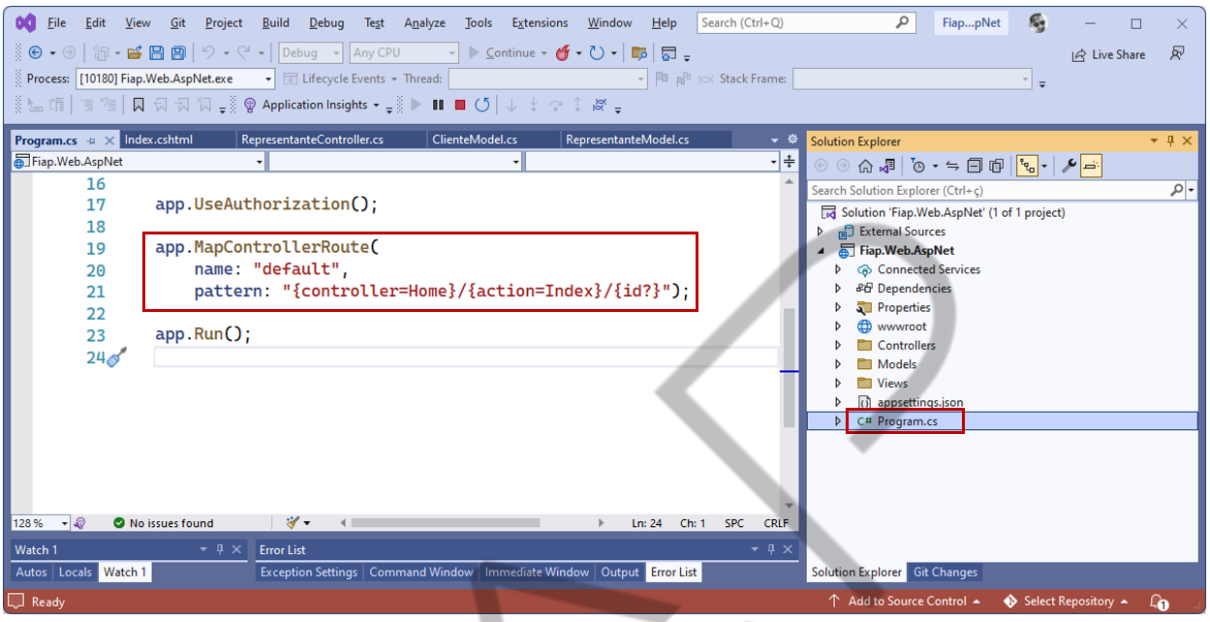


Figura 16 – Configuração de Rotas
Fonte: Elaborado pelo autor (2022)

O bloco de código dá chamado ao método **MapControllerRoute,** que é o responsável por interceptar todas as chamadas do aplicativo, analisar o caminho da URL requisitada e mapear para o *Controller* e para a *Action* correspondentes. Verifique o código da implementação da Figura “Configuração de Rotas”. Temos um padrão na propriedade url “**{controller}/{action}/{id}”** definindo que os caminhos deverão ser compostos pelo nome do controle, ação e id (valores opcionais).

Ainda no **MapControllerRoute**, temos uma definição “***default***”, que determina quais *Controller* e *Action* deverão ser executados. Caso nenhuma informação seja indicada na url. O padrão é o *Controller* chamado Home e a *Action,* Index.

Podemos alterar o controlador padrão de **Home** para **Representante** e executar o aplicativo novamente. Desse modo, será possível acessar nossa funcionalidade usando apenas a url: http://localhost:5238/.

Embora essas configurações possam ser alteradas, recomenda-se manter o padrão. Por esse motivo, manteremos o *Controller* **Home** como padrão.

Para não deixar nossa aplicação sem uma apresentação inicial, o Visual Studio disponibiliza um *Controller* chamado (**HomeController**) e sua *View* (Index.cshtml). Na

*View*, devemos escrever uma mensagem para identificar que estamos navegando pela *homepage*. Veja na Figura “Estrutura da *homepage*” a estrutura das pastas *Controllers* e *View*, e seus respectivos códigos-fonte:

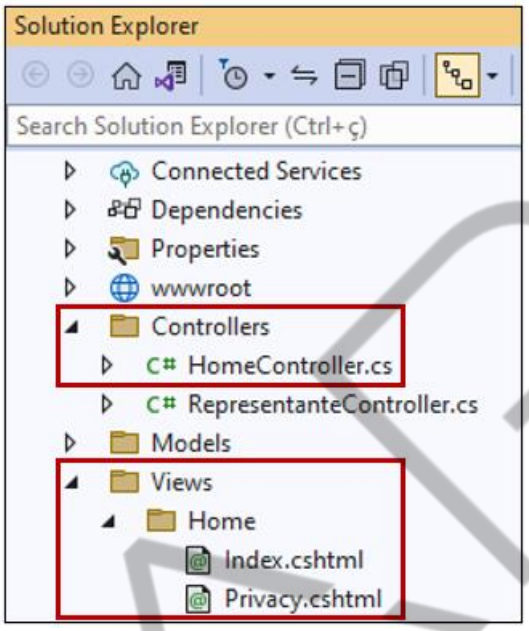


Figura 17 – Estrutura da *homepage*
Fonte: Elaborado pelo autor (2022)

```
using Fiap.Web.AspNet.Models;
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;

namespace Fiap.Web.AspNet.Controllers
{
   public class HomeController : Controller
   {
      private readonly ILogger<HomeController> _logger;

      public HomeController(ILogger<HomeController> logger)
      {
         _logger = logger;
      }

      public IActionResult Index()
      {
         return View();
      }

      public IActionResult Privacy()
      {
         return View();
      }

      [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
```

```
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier
});
        }
    }
}
```

Código-fonte 4 – *Controller* da *homepage*
Fonte: Elaborado pelo autor (2022)

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Fiap On <a href="https://docs.microsoft.com/aspnet/core">HomePage with ASP.NET
Core</a>.</p>
</div>
```

Código-fonte 5 – *Script* para a criação da tabela Tipo Produto
Fonte: Elaborado pelo autor (2022)

Execute novamente o aplicativo e note que a nossa *homepage* será apresentada como página inicial. Acesse os endereços abaixo no navegador e verifique se todos vão exibir a mesma visão (*homepage*):

- http://localhost:5238/
- http://localhost:5238/Home
- http://localhost:5238/Home/Index

**IMPORTANTE:** Verifique a porta que está sendo usada pela aplicação. Se for diferente de **5238**, altere os endereços acima adicionando o número da nova porta.

### 3.3 Views

Até este ponto do nosso curso, vimos que as *Views* no framework ASP.NET Core MVC (Web App) são arquivos .cshtml com base em HTML e que, por convenção, elas são salvas na pasta *Views* e na subpasta com o nome do *Controller* associado.

Fazendo uso apenas de HTML sabemos que não é possível ter dinamismo para manipular e persistir informações em nossa base de dados. Para isso, o ASP.NET Core MVC (Web App) possui o mecanismo de *view engine*, que usa a linguagem C#

com a marcação **Razor**. Podemos fazer uma relação com JSP da linguagem Java e a *Expression Language* (EL), que facilita os famosos *scriptlets*.

### 3.4 ASP.NET Razor

O Razor é um dos mecanismos do ASP.NET Core MVC (Web App) responsáveis por construir nossas *Views* dinâmicas; antes de seu lançamento, o mecanismo-padrão era o ASPX, que usava como base *scriptlets* ASP.NET puros. Ainda disponível para a criação de projetos MVC, não é recomendado pelo framework.

Em 2011, integrado com a versão do ASP.NET MVC 3, foi lançado o *view engine* **Razor,** com o objetivo de simplificar a codificação na camada *View*. O Razor trouxe alguns benefícios significativos para os desenvolvedores, segue uma lista deles:

- Usa a linguagem C# como base de seus *scriptlets*.
- Apresenta sintaxe limpa, reduzindo o código.
- Simplifica o acesso aos componentes Model.
- Permite escrever testes unitários apenas para a camada *Views*.
- Uso do *autocomplete* (*IntelliSense*) para completar sintaxe de código no Visual Studio.
- Facilita o uso de layouts predefinidos para todo o site.

Para identificar uma expressão Razor em um arquivo .cshtml, basta observar blocos de código iniciados pelo caractere @. O Código-Fonte “Exemplo de bloco de código Razor” apresenta um bloco de código **Razor** para contar de 0 até 10 e exibir na tela do navegador:

```
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Homepage</title>
</head>
```

```
<body>
    <div>
        <h1>Nossa home-page.</h1>
    </div>

    @{
        for (int i = 0; i < 10; i++)
        {
            <p>@i</p>
        }
    }
</body>
</html>
```

Código-fonte 6 – Exemplo de bloco de código Razor
Fonte: Elaborado pelo autor (2022)

### 3.5 Tag *Helpers*

O framework ASP.NET Core MVC (Web App) disponibiliza os componentes auxiliadores para o desenvolvimento dos componentes *Views*.

Os Auxiliadores de Marcação fazem com que o código do lado do servidor participe da criação e renderização de elementos HTML. O ImageTagHelper interno pode acrescentar um número de versão ao nome da imagem, assim, sempre que a imagem é alterada, o servidor gera uma nova versão exclusiva para a imagem, de modo que os clientes tenham a garantia de obter a imagem atual. Existem auxiliadores para todos os elementos HTML comuns (por exemplo: formulários, links, imagens, botões e outros).

Veja o Quadro “Tag Helpers” com os Tag Helpers disponíves no ASP.NET Core MVC (Web App):

| Tag Helper | Método Tipado |
|---|---|
| Anchor tag helper | Link tag helper |
| Cache tag helper | Option tag helper |
| Environment tag helper | Partial tag helper |
| Form Action tag helper | Script tag helper |
| Form tag helper | Select tag helper |
| Image tag helper | Textarea tag helper |
| Input tag helper | Validation Message tag helper |
| Label tag helper | Validation Summary tag helper |

Quadro 1 – Tag Helpers
Fonte: Elaborado pelo autor (2022)

Para demonstração, a Figura “Exemplo de uso de TagHelpers” apresenta a sintaxe para a criação de uma caixa de texto usando o *helper* e o código HTML gerado depois que o *view engine* renderiza o código da *View*. Veja:

```
<!-- Tag Help -->
<input asp-for="DescricaoTipo" placeholder="Digite a Descrição" />

<!-- Tag Help -->
<input placeholder="Digite a Descrição" type="text" id="DescricaoTipo" name="DescricaoTipo" value>
```

Figura 18 – Exemplo de uso de Tag Helpers
Fonte: Elaborado pelo autor (2022)

## 3.6 Listando dados na tela (*View*)

No bloco anterior, fomos apresentados ao *view engine* Razor e aos *helpers* para criar componentes HTML, além de conhecermos os conceitos de rotas, *Controllers* e convenções. Agora, precisamos colocar em prática e implementar nosso projeto.

O nosso *Controller* Representante possui apenas uma simples ação para exemplificar o funcionamento da associação *Controller* > *Action* > *View*. É preciso adicionar os comportamentos: cadastro, alteração, exclusão e consulta (CRUD).

Para não perdermos tempo criando e configurando nosso banco de dados, vamos partir para uma estratégia de simulação, também conhecida como *Mock*. Essa

estratégia simula os comandos de integração com as tabelas da base de dados. Dessa forma, será possível testar os componentes do MVC e o fluxo de navegação a fim de, posteriormente, criar apenas o código de integração com o banco de dados.

A ideia desta seção é criar uma listagem de dados para os tipos de produtos da **Fiap.Web.AspNet**. Para cada informação listada, será necessário criar uma ação que será implementada posteriormente para consultar, editar e excluir, bem como uma opção para criar um novo tipo. Vamos usar a ***Action*** e a ***View*** já criadas e adicionar nosso código.

No RepresentanteController, vamos criar um atributo do tipo lista e no construtor vamos adicionar três objetos do modelo Representante. No método de retorno da *Action* Index, vamos passar como parâmetro o atributo lista. O Código-Fonte "Criando a lista de representantes no *Controller*", a seguir, mostra a criação do atributo lista e o retorno do método **View()**:

```
using Fiap.Web.AspNet.Models;
using Microsoft.AspNetCore.Mvc;

namespace Fiap.Web.AspNet.Controllers
{
   public class RepresentanteController : Controller
   {

      private IList<RepresentanteModel> representantes;

      public RepresentanteController()
      {
         // De uma forma rudimentar, podemos dizer que esse bloco de código
         // simula o retorno de uma consulta no banco de dados
         representantes = new List<RepresentanteModel>();
         representantes.Add(new RepresentanteModel(1, "444.143.658-05", "José Carlos Silva"));
         representantes.Add(new RepresentanteModel(2, "062.572.723-19", "Maria José Fernandes"));
         representantes.Add(new RepresentanteModel(2, "920.680.661-06", "Carlos Silva"));
      }

      public IActionResult Index()
      {
         // Retornando para View a lista de Representantes
         return View(representantes); // <-- Atenção
      }
   }
}
```

Código-fonte 7 – Criando a lista de representantes no *Controller*
Fonte: Elaborado pelo autor (2022)

Com a lista de representantes criada de forma simulada e retornada para a *View*, agora, precisamos implementar o mecanismo de exibição e a criação das futuras ações. O objetivo para o componente *View* é criar uma tabela que apresente a lista dos dados. Para cada item da lista, serão criados três (3) hiperlinks (Editar, Excluir e Consultar) e, por fim, um (1) hiperlink para cadastrar um novo tipo.

A codificação para as tags **Razor** da nossa implementação deverá compreender: a declaração **@model** para definir o tipo do objeto modelo, um bloco **@foreach** para listar os elementos da lista e as declarações **asp-controller, asp-action** e **asp-route-id** para os *hiperlinks* de edição, exclusão, cadastro e consulta. Nosso objeto modelo é uma lista, com isso, devemos especificar na declaração ***@model*** o tipo ***IEnumerable***.

O Código-Fonte "Criando a lista de representantes na *View*" mostra o resultado do nosso componente *View* com alguns comentários explicativos do uso dos Tag Helpers, veja:

```
@model IEnumerable<Fiap.Web.AspNet.Models.RepresentanteModel>

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Representantes</title>
</head>
<body>
  <h1>Representantes</h1>
  <p>
    <!-- uso de TagHelpers para definir o Controller e a Action -->
    <a asp-controller="Representante" asp-action="Cadastrar">Novo Representante</a>
  </p>
  <table class="table" border="1">
    <tr>
      <th>Id</th>
      <th>CPF</th>
      <th>Nome</th>
      <th></th>
    </tr>

    @foreach (var item in Model)
    {
      <tr>
        <td>
          <label>@item.RepresentanteId</label>
        </td>
        <td>
          <label>@item.Cpf</label>
```

```
            </td>
            <td>
              <label>@item.NomeRepresentante</label>
            </td>
            <td>

              <!-- asp-route-id é usado para informar o Id do Item selecionado. -->
              <a asp-controller="Representante"
                asp-action="Editar"
                asp-route-id="@item.RepresentanteId">Editar</a>

              <a asp-controller="Representante"
                asp-action="Consultar"
                asp-route-id="@item.RepresentanteId">Consultar</a>

              <a asp-controller="Representante"
                asp-action="Excluir"
                asp-route-id="@item.RepresentanteId">Excluir</a>
            </td>
          </tr>
      }
    </table>
</body>
</html>
```

Código-fonte 8 – Criando a lista de representantes na *View*
Fonte: Elaborado pelo autor (2022)

Vamos executar a aplicação. Pressione a tecla **F5** e, no navegador, informe o caminho /Representante. Depois, aguarde o carregamento da lista de representantes. Veja o resultado na Figura “Resultado da tela de Tipo de produtos”:

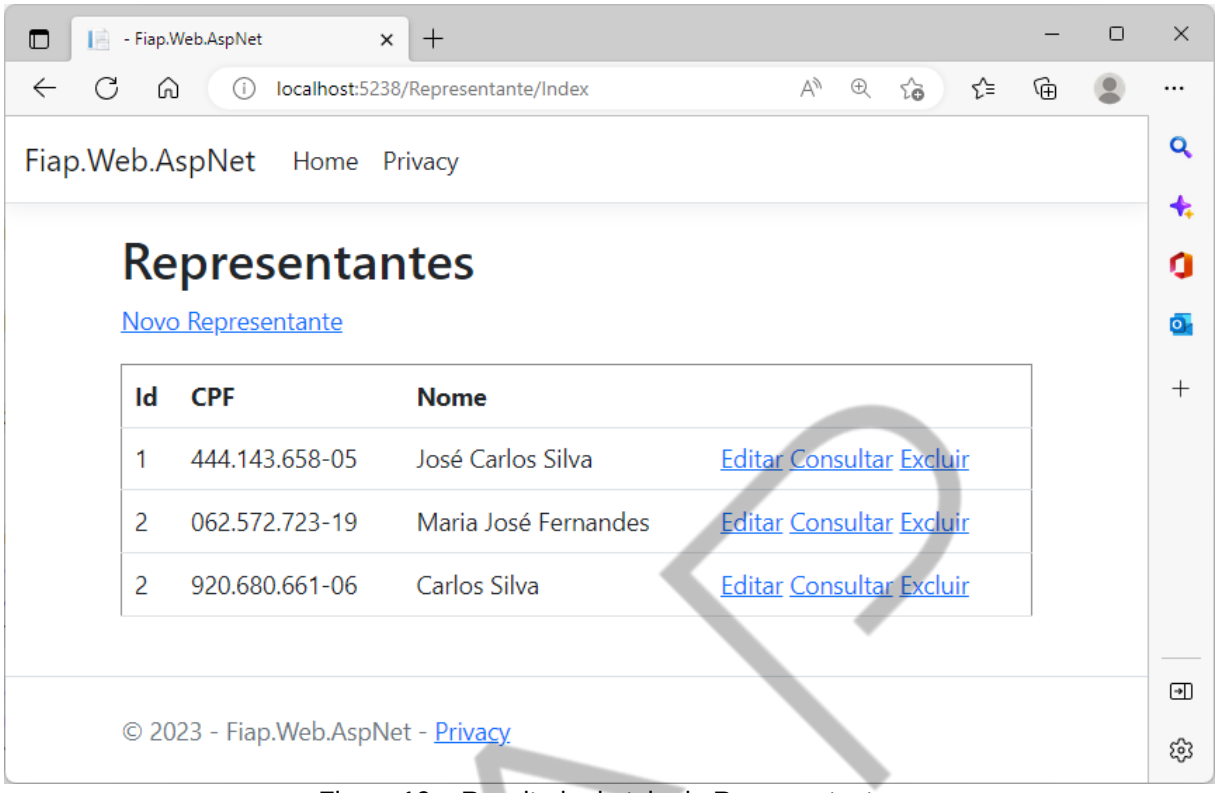


Figura 19 – Resultado da tela de Representantes
Fonte: Elaborado pelo autor (2022)

**IMPORTANTE:** Ao clicar em qualquer link da tela, não será executada nenhuma ação, pois as ações ainda não foram implementadas em nosso *Controller*.

### 3.6.1 Inserindo dados (*View* e *Controller*)

Avançando na implementação do nosso projeto, precisamos, então, criar os elementos do framework MVC que permitam ao usuário preencher os dados de tipo de produto e simular a gravação na base de dados.

Ainda no mesmo *Controller*, adicionaremos dois novos métodos (*Actions*). Os dois métodos receberão o nome **"Cadastrar"**. Pode parecer estranho, mas adotaremos o mesmo nome para testar a forma particular de sobrecarga de métodos em *Controllers*.

Tendo os dois métodos com o mesmo nome, a diferenciação será feita de duas formas: a primeira é com o uso de uma anotação que define qual o verbo HTTP (Get ou Post) que a *Action* vai aceitar em execução. A segunda forma é por meio de um

parâmetro, um dos métodos receberá como *model* Representante. O Quadro "*Actions* de cadastro do *Controller* Representante" detalha os dois métodos a serem criados:

| Nome | Verbo HTTP | Parâmetro | Funcionalidade |
|---|---|---|---|
| Cadastrar | Get | N/A | Tem como objetivo abrir um formulário com os dados do tipo de produto em branco. O *Controller* deverá passar para a *View* uma instância do objeto *model* com as propriedades em branco. |
| Cadastrar | Post | Model TipoProduto | Receber o modelo do parâmetro, simular a gravação dos dados no banco de dados e redirecionar o usuário para a lista de tipos. |

Quadro 2– *Actions* de cadastro do *Controller* Representante
Fonte: Elaborado pelo autor (2022)

Para usar as anotações que indicam qual verbo HTTP é usado no método, é necessário declarar acima da implementação do método, com as seguintes expressões: [HttpGet], [HttpPost]. A simulação de gravação dos dados no banco de dados será feita pelo comando **Console.WriteLine()**.

Veja, no Código-Fonte "*Actions* de cadastro representante" como ficou a implementação das *Actions* Cadastrar:

```
// Anotação de uso do Verb HTTP Get
        [HttpGet]
        public IActionResult Cadastrar()
        {
            // Imprime a mensagem de execução
            Console.WriteLine("Executou a Action Cadastrar()");

            // Retorna para a View Cadastrar um
            // objeto modelo com as propriedades em branco
            return View(new RepresentanteModel());
        }

        // Anotação de uso do Verb HTTP Post
        [HttpPost]
        public IActionResult Cadastrar(RepresentanteModel representante)
        {
            // Imprime os valores do modelo
            Console.WriteLine("Descrição: " + representante.Cpf);
            Console.WriteLine("Comercializado: " + representante.NomeRepresentante);

            // Simila que os dados foram gravados.
            Console.WriteLine("Gravando o Representante");
```

```
        // Substituímos o return View()
        // pelo método de redirecionamento
        return RedirectToAction("Index", "Representante");
    }
```

Código-fonte 9 – *Actions* de cadastro de representante
Fonte: Elaborado pelo autor (2022)

Implementado o nosso *Controller*, o próximo passo é criar uma *View* para fornecer um formulário e os elementos para a digitação dos dados.

Seguindo as convenções do framework, devemos criar a nova *View* com o mesmo nome da Action: **"Cadastrar" como na figura abaixo**. Essa *View* deverá estar dentro de Views > Representante, e deverá fazer uso dos *tag helpers* **asp-controlle**r e **asp-action** para a criação do formulário, além dos elementos HTML puros para posicionamento e formatação da tela.

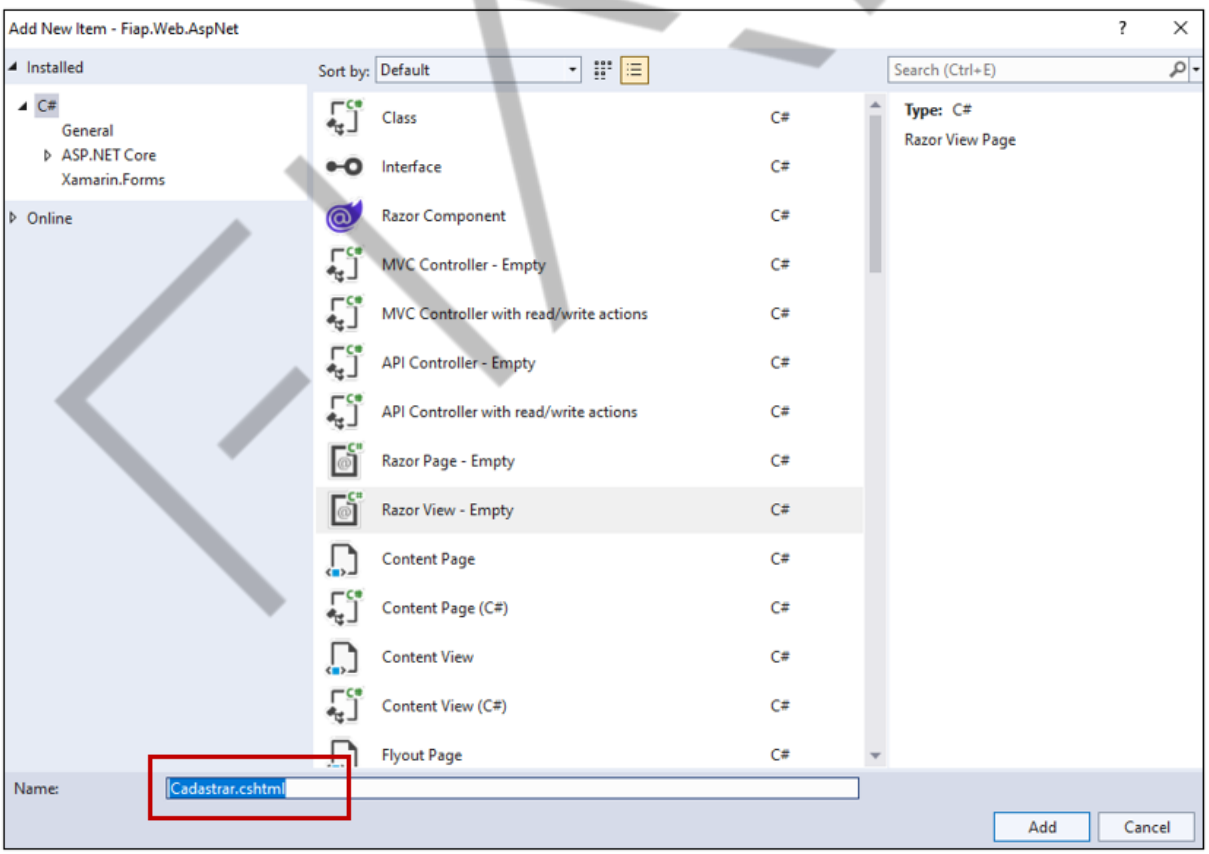


Figura 20 – Adicionando a View Cadastrar
Fonte: Elaborado pelo autor (2022)

**DICA:** Preste atenção nos elementos da *View*, os tag helpers (Razor) ficarão muito parecidos com propriedades dos elementos HTMLs.

Veja, no Código-Fonte “*View* de cadastro de representante”, o resultado da *View* Cadastrar:

```
@model Fiap.Web.AspNet.Models.RepresentanteModel

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Representante - Cadastrar</title>
</head>
<body>
  <h1>Representante - Cadastrar</h1>

  <!-- formulário HTML com Tag Helpers-->
  <form asp-action="Cadastrar" asp-controller="Representante" method="post">
    <div class="form-horizontal">
      <hr />

      <div class="form-group">
        <label>Cpf:</label>
        <div class="col-md-10">
          <!-- Caixa de Texto -->
          <input asp-for="Cpf" />
        </div>
      </div>

      <div class="form-group">
        <label>Nome:</label>
        <div class="col-md-10">
          <!-- Caixa de Texto -->
          <input asp-for="NomeRepresentante"  />
        </div>
      </div>
      <hr />
      <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
          <a asp-controller="Representante" asp-action="Index" class="btn btn-warning">Voltar</a>
          <input type="submit" value="Cadastrar" class="btn btn-primary" />
        </div>
      </div>
      <hr />
    </div>
  </form>


</body>
</html>
```

Código-fonte 10 – *View* de cadastro de representante
Fonte: Elaborado pelo autor (2022)

Podemos usar duas estratégias para validar na implementação. Uma delas é adicionando *breakpoints* nos trechos de código do *Controller* e, com a tecla F10, percorrer linha a linha para acompanhar a execução. A outra forma é observar pela janela do console do projeto mensagens que são impressas pelo comando Console.WriteLine(). Veja, na Figura “Janela *Console do projeto*”, a janela Output e algumas mensagens da execução do fluxo de cadastrar:

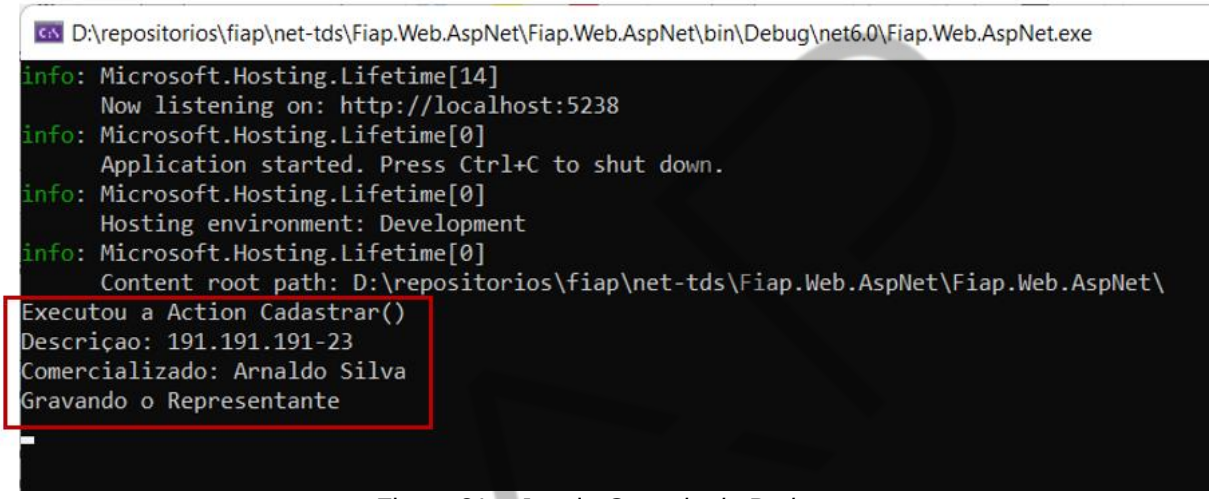


Figura 21 – Janela *Console do Projeto*
Fonte: Elaborado pelo autor (2022)

Execute a aplicação e acesse a lista de tipos. No link “Novo Representante”, simule um cadastro de tipo use *breakpoints* ou a janela ***Console*** para acompanhar os dados digitados. Lembre-se, como estamos usando trechos de código para simulação, os dados da lista não serão alterados.

### 3.6.2 Editando dados (*View* e *Controller*)

O fluxo de edição possui algumas semelhanças com o de cadastro. Podemos nos basear no código criado na seção anterior e, com poucas alterações, será possível implementar a edição do representante.

Seguem os métodos que devem ser criados para a edição:

| Nome | Verbo HTTP | Parâmetro | Funcionalidade |
|---|---|---|---|
| Editar | Get | int Id | Objetivo: abrir um formulário com os dados do tipo selecionado na lista. O parâmetro Id é responsável por receber o código do tipo selecionado na lista e será usado para consulta da chave primária na tabela no banco de dados.O *Controller* deverá passar para a *View* uma instância do objeto *model* preenchido com o resultado da pesquisa no banco de dados. |
| Editar | Post | Model TipoProduto | Receber o modelo do parâmetro, simular a gravação das alterações no banco de dados e redirecionar o usuário para a lista de tipos. |

Quadro 3 – *Actions* de edição do *Controller* Representante (1)
Fonte: Elaborado pelo autor (2022)

Veja a implementação no Código-Fonte "*Actions* de edição do representante":

```
        [HttpGet]
        public IActionResult Editar(int id)
        {
            // Imprime a mensagem de execução
            Console.WriteLine("Consultando pelo Id = " + id);

            // Cria o modelo que SIMULA a consulta no banco de dados
            Models.RepresentanteModel Representante = new Models.RepresentanteModel(id,"191.191.191-
91","Almir Moura");


            // Retorna para a View o objeto modelo
            // com as propriedades preenchidas com dados do banco de dados
            return View(Representante);
        }

        [HttpPost]
        public IActionResult Editar(RepresentanteModel representante)
        {
            // Imprime os valores do modelo
            Console.WriteLine("CPF: " + representante.Cpf);
            Console.WriteLine("Id: " + representante.RepresentanteId);
            Console.WriteLine("Nome: " + representante.NomeRepresentante);

            // Simila que os dados foram gravados.
            Console.WriteLine("Gravando o Tipo Editado");

            // Substituímos o return View()
            // pelo método de redirecionamento
            return RedirectToAction("Index", "Representante");
        }
```

Código-fonte 11 – *Actions* de edição do representante
Fonte: Elaborado pelo autor (2022)

Para a *View "*Editar", podemos reaproveitar todo o código-fonte criado na *View* Cadastrar. Com muito cuidado, revise os caminhos usados para o post do formulário. Altere o título da página e adicione um componente do tipo ***hidden***, que vai armazenar o Id representante.

É preciso armazenar o Id do representante, pois, na execução do comando de atualização no banco de dados (*Update*), devemos informar a chave primária. Observe, no Código-Fonte "*View* de edição do representante", os detalhes da *View* **Editar**:

```
@model Fiap.Web.AspNet.Models.Representante

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Tipo de Produto - Editar</title>
</head>
<body>
    <h1>Tipo de Produto - Editar</h1>

    <!-- formulário HTML com Tag Helpers-->
    <form asp-action="Editar" asp-controller="Representante"
method="post">
        <!-- Campo oculto para guardar qual Id (chave) do
registro alterado -->
        <input type="hidden" asp-for="IdTipo" />
        <div class="form-horizontal">
            <hr />

            <div class="form-group">
                <label>Descrição</label>
                <div class="col-md-10">
                    <!-- Caixa de Texto -->
                    <input asp-for="DescricaoTipo" />
                </div>
            </div>

            <div class="form-group">
                <label>Comercializado</label>
                <div class="checkbox">
                    <!-- CheckBox -->
                    <input asp-for="Comercializado" />
```

```
                </div>
            </div>

            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <input type="reset" value="Limpar"
class="btn btn-default" />
                    <!-- HTML Simple para envio dos dados do
formulário -->
                    <input type="submit" value="Gravar
Alteração" class="btn btn-default" />
                </div>
            </div>
            <hr />
        </div>
    </form>

    <div>
        <a asp-controller="Representante" asp-
action="Index">Voltar</a>
    </div>

</body>
</html>
```

Código-fonte 12 – *View* de edição do representante
Fonte: Elaborado pelo autor (2022)

Antes de testar, é preciso garantir que, na tela de lista de representantes (View Index), o link para edição esteja correto. Abra a View Index e verifique se o código do link editar está como o código a seguir.

```
<!-- asp-route-id é usado para informar o Id do Item selecionado. -->
<a asp-controller="Representante"
  asp-action="Editar"
  asp-route-id="@item.RepresentanteId">Editar</a>
```

Código-fonte 13 – Link para funcionalidade de edição do representante
Fonte: Elaborado pelo autor (2022)

Execute a aplicação, acesse a lista de tipos e clique link Editar qualquer representante. Simule uma editação e use *breakpoints* ou a janela Console para acompanhar os dados digitados. **Lembre-se, como estamos usando trechos de código para simulação, os dados exibidos para edição sempre serão iguais e também os dados da lista não serão alterados**.

### 3.6.3 Consultando dados (*View* e *Controller*)

Para criar o fluxo de consulta de dados, podemos replicar parte do trabalho do fluxo de edição. No *Controller*, devemos usar apenas o método que utiliza o verbo HTTP *Get*; e, para a *View*, podemos remover a criação de formulário e substituir os elementos de edição (input) por simples labels.

Segue o método para a consulta dos dados:

| Nome | Verbo HTTP | Parâmetro | Funcionalidade |
|---|---|---|---|
| Consultar | Get | int Id | Objetivo: abrir um formulário com os dados do tipo selecionado na lista. O parâmetro Id é responsável por receber o código do tipo selecionado na lista e será usado para consulta da chave primária na tabela no banco de dados.O *Controller* deverá passar para a *View* uma instância do objeto *model* preenchido com o resultado da pesquisa no banco de dados. |

Quadro 4 – *Actions* de consulta do *Controller* Representante
Fonte: Elaborado pelo autor (2022)

Veja, a seguir, a implementação no Código-Fonte "*Action* de consulta do representante":

```
        [HttpGet]
        public IActionResult Consultar(int id)
        {
            // Imprime a mensagem de execução
            Console.WriteLine("Consultando pelo Id = " + id);

            // Cria o modelo que SIMULA a consulta no banco de dados
            Models.RepresentanteModel Representante = new Models.RepresentanteModel(id, "191.191.191-91", "Almir Moura");


            // Retorna para a View o objeto modelo
            // com as propriedades preenchidas com dados do banco de dados
            return View(Representante);
        }
```

Código-fonte 14 – *Action* de consulta do representante
Fonte: Elaborado pelo autor (2022)

Criando a *View* **Consultar**, reaproveite o código da *View* **Editar** para ter funcionalidade de apenas exibir os dados. Relembrando, devemos remover o bloco do **form** e substituir os elementos de **input**.

Veja a alteração aplicada na *View* **Consultar**:

```
@model Fiap.Web.AspNet.Models.RepresentanteModel

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Representante - Consultar</title>
</head>
<body>
  <h1>Representante - Consultar</h1>

  <div class="form-horizontal">
    <hr />
    <div class="form-group">
      <label><b>Id</b></label>
      <div class="col-md-10">
        <span>@Model.RepresentanteId</span>
      </div>
    </div>

    <div class="form-group">
      <label class="col-md-2"><b>Cpf</b></label>
      <div class="col-md-10">
        <span>@Model.Cpf</span>
      </div>
    </div>

    <div class="form-group">
      <label><b>Nome</b></label>
      <div class="col-md-10">
        <span>@Model.NomeRepresentante</span>
      </div>
    </div>
    <hr />
    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
        <a asp-controller="Representante" asp-action="Index" class="btn btn-warning">Voltar</a>
      </div>
    </div>
    <hr />
  </div>

</body>
</html>
```

Código-fonte 15 – *View* de consulta do representante
Fonte: Elaborado pelo autor (2020)

### 3.6.4 Removendo dados (*View* e *Controller*)

Diferentemente dos demais fluxos, a remoção será feita apenas por uma *Action*, não vamos utilizar *View*.

Segue o método para a consulta dos dados:

| Nome | Verbo HTTP | Parâmetro | Funcionalidade |
|---|---|---|---|
| Excluir | Get | int Id | Receber o valor do parâmetro Id, simular o comando de exclusão do banco de dados e redirecionar o usuário para lista dos Tipos. |

Quadro 5 – Action de excluir do Controller Representante
Fonte: Elaborado pelo autor (2020)

Veja, a seguir, a implementação no Código-Fonte “Action de exclusão do representante”:

```
[HttpGet]
        public IActionResult Excluir(int id)
        {
            // Imprime a mensagem de execução
            Console.WriteLine("Excluir o Representante Id = " + id);

            // Substituímos o return View()
            // pelo método de redirecionamento
            return RedirectToAction("Index", "Representante");
        }
```

Código-fonte 16 – *Action* de exclusão do representante
Fonte: Elaborado pelo autor (2022)

Execute a aplicação e acompanhe as mensagens na janela Console a fim de validar todo o fluxo das operações.

**ATENÇÃO:** A Execução desse fluxo pode parecer sem funcionalidade. O recomendando é acompanhar pela janela de Console e inserir alguns break-points no código.

### 3.7 Layout pages e identidade visual

Nos capítulos anteriores, criamos nosso aplicativo com o objetivo de testar fluxo, comportamento e componentes do MVC. Apesar de criar componentes *View,* não implementamos recursos visuais mais profissionais, usamos a estratégia de manter o funcionamento apenas.

Passaremos a incrementar nossa camada visual, dando um tom mais profissional com componentes do framework ASP.NET Core MVC (Web App) para facilitar a evolução do aplicativo. Para isso, usaremos a biblioteca **Bootstrap**, pois, além de ser uma biblioteca bem difundida, foi utilizada em módulos anteriores.

#### 3.7.1 Instalando Bootstrap

Nas versões anteriores do framework ASP.NET MVC, era necessário realizar a instalação da biblioteca **Bootstrap** utilizando a ferramenta **Nuget**, disponível no Visual Studio.

O **Nuget** é um gerenciador de pacotes da tecnologia .NET que possibilita usar pacotes de bibliotecas externas ou construir bibliotecas para serem usadas por outros desenvolvedores. O **Nuget** possui um repositório central que armazena todos os pacotes, os quais podem ser utilizados por qualquer desenvolvedor da plataforma .NET. Para mais informações, consulte: <https://www.nuget.org/>.

A versão ASP.NET Core MVC já disponibiliza o **Bootstrap** na criação no projeto; portanto não é necessário realizar a instalação. É possível encontrar as pastas e os arquivos da biblioteca na pasta **wwwroot**, disponível na *Solution Explorer* do Visual Studio.

Veja, a seguir, a estrutura do projeto na Figura “Estrutura do projeto com Bootstrap”:

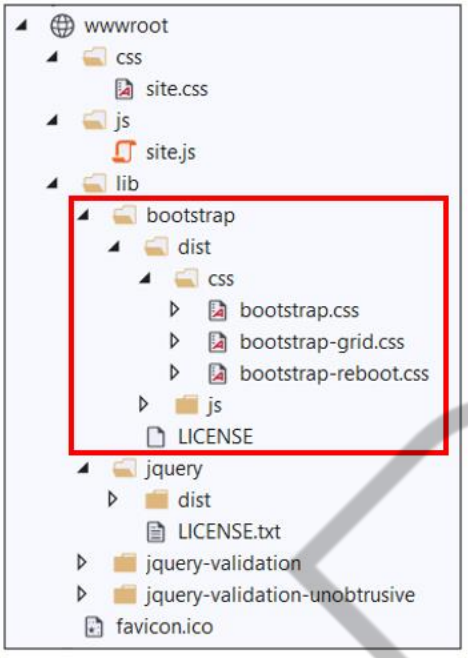


Figura 22 – Estrutura do projeto com Bootstrap
Fonte: Elaborado pelo autor (2022)

### 3.7.2 Criando Layouts

O uso do Bootstrap obriga todas as páginas HTML do site a importar as referências para os arquivos da biblioteca (.css e .js). Com o uso dos recursos de Layouts, centralizaremos as importações em um único ponto do projeto. E mais, todos os websites possuem padrões e áreas comuns em todas as páginas, como: cabeçalho, logotipo, menu, rodapé e outros que a criatividade permitir. Os recursos de Layouts do MVC permitem criar uma única vez os padrões e as partes comuns e usar em todo o projeto sem muito esforço de código.

Como estamos trabalhando com nossa camada de visualização, devemos trabalhar bastante no *namespace Views* do projeto. Por convenção, nossos layouts devem ficar em uma subpasta chamada *Shared*, dentro da pasta *Views*.

Na pasta *Shared*, vamos abrir o arquivo **_Layout.cshtml** e adaptar o código HTML para o nosso projeto.

A Figura “Arquivo de Layout” apresenta o arquivo de layout criado e a estrutura de pastas do projeto:

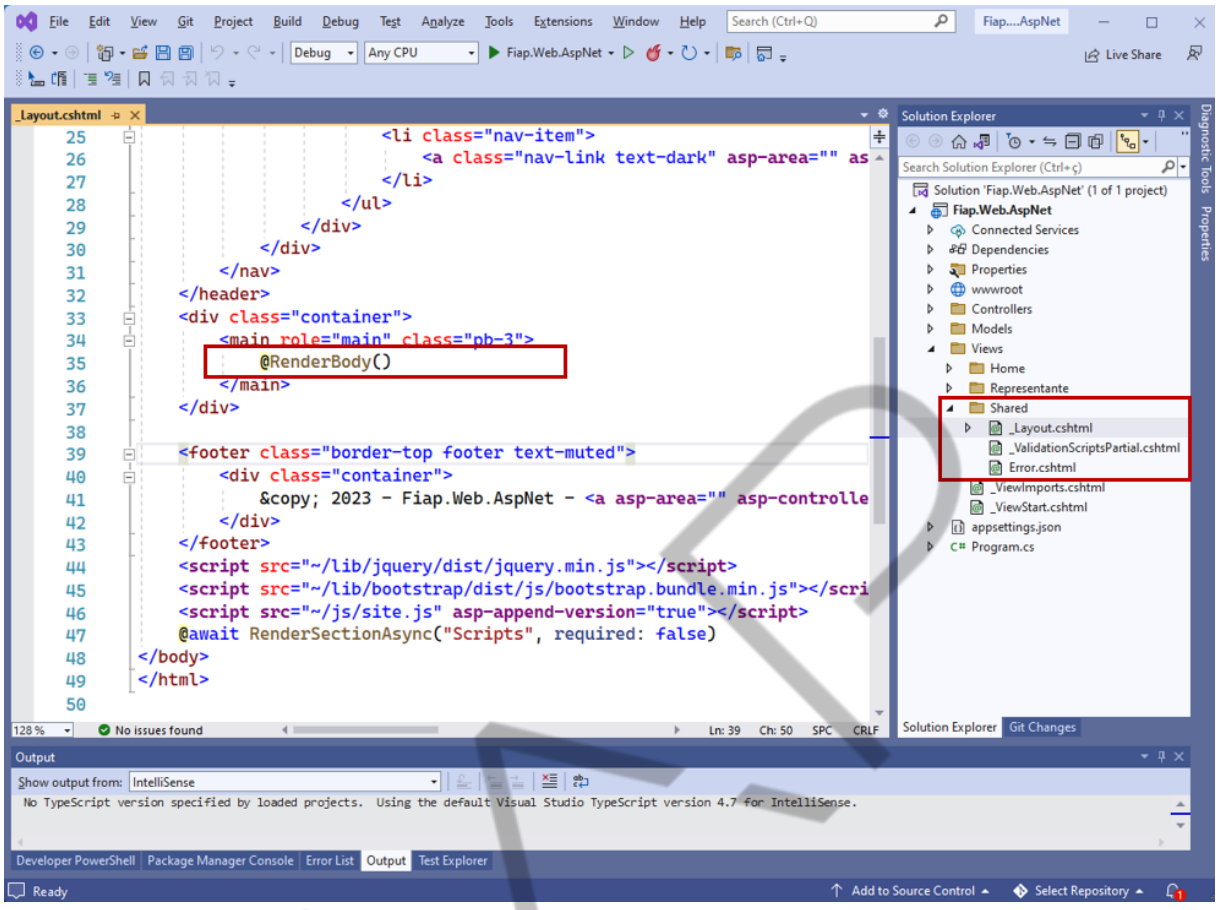


Figura 23 – Arquivo de Layout
Fonte: Elaborado pelo autor (2022)

É possível notar que o arquivo de layout tem seu conteúdo muito similar a um HTML ou a uma *View* .cshtml e também possui algumas tags Razor declaradas inicialmente.

Falaremos sobre as tags @ViewBag e @RenderBody logo mais, antes, precisamos inserir as tags HTML para o uso do Bootstrap. No corpo da tag <head>, as tags <link> com referência ao arquivo .css do Bootstrap. É possível usar a tag HTML pura, mas, para explorar os recursos do framework, é recomendado usar o recurso do símbolo “~”, que permite transformar caminhos de arquivos relativos para caminhos semiabsolutos. Ou seja, não importa o endereço em que a *View* é exibida, a tag apontará para o caminho correto dos arquivos de estilo, javascript e imagem. Segue o código-fonte que devemos usar no <head> do nosso layout:

```
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - Fiap.Web.AspNet</title>
```

```
    <!-- importando bootstrap e o css do nosso site-->
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/Fiap.Web.AspNet.styles.css" asp-append-version="true" />
</head>
```

Código-fonte 17 – Importando CSS com @Url.Content()
Fonte: Elaborado pelo autor (2022)

Depois do arquivo de estilo, vamos verificar a importação dos arquivos de *script*; para isso, é adotada a composição da tag HTML <script> e, novamente, o recurso do símbolo "~". No Código-Fonte "Importando JavaScript com @Url.Content()" é apresentado o bloco de importação das bibliotecas em **JavaScript**.

```
<!-- importando as libs JavaScript -->
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
```

Código-fonte 18 – Importando JavaScript
Fonte: Elaborado pelo autor (2022)

Para incrementar um pouco mais o nosso aplicativo, modificaremos a seção de cabeçalho e rodapé. O cabeçalho será composto por mais um item de menu com a opção para a funcionalidades de Representantes e Clientes (implementado futuramente) e o rodapé terá uma seção de contato *fake* do projeto **Fiap.Web.AspNet**.

Esse conteúdo será inserido dentro da tag <body>, pois agora ele faz parte do conteúdo visível ao usuário. O Código-Fonte "*View* de Layout", a seguir, mostra a versão final do nosso layout, que contém nosso cabeçalho e rodapé todo construído em HTML e Bootstrap.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Fiap.Web.AspNet</title>

    <!-- importando bootstrap e o css do nosso site-->
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/Fiap.Web.AspNet.styles.css" asp-append-version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom
box-shadow mb-3">
            <div class="container-fluid">
```

```
            <a class="navbar-brand" asp-area="" asp-controller="Home" asp-
action="Index">Fiap.Web.AspNet</a>
            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target=".navbar-collapse" aria-controls="navbarSupportedContent"
                    aria-expanded="false" aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                <ul class="navbar-nav flex-grow-1">
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                    </li>
                    <!-- Link funcionalidade de representante -->
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Representante" asp-
action="Index">Representante</a>
                    </li>
                    <!-- Link funcionalidade de cliente -->
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Cliente" asp-
action="Index">Cliente</a>
                    </li>

                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>
</header>
<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>

<footer class="border-top footer text-muted">
    <div class="container">
        <b>Fiap On - Copyright &copy; 2022</b> - <a asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
    </div>
</footer>

<!-- importando as libs JavaScript -->
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
@await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

Código-fonte 19 – *View* de Layout

Fonte: Elaborado pelo autor (2022)

Entre nosso cabeçalho e nosso rodapé, existe a tag Razor @RenderBody(); pois bem, aqui está nosso segredo!

A tag @RenderBody() é a responsável por especificar o ponto em que o conteúdo da View será renderizado, ou seja, o conteúdo HTML da View será inserido no espaço da tag @RenderBody().

Para juntar o quebra-cabeça do Layout e da *View*, é necessário entender como nossas *Views* utilizam o bloco **@{ Layout }** nos arquivos .cshtml, mas, observando nosso arquivo **Views\Representante\Index.cshtml,** é possível notar que não temos nenhum bloco de código com @{Layout}, pois esse trecho está declarado no arquivo View_ViewStart.cshtml. O arquivo ViewStart é uma forma de declarar configurações visuais no nível da aplicação, ou seja, algo declarado nesse arquivo será usado em toda aplicação. Como sugestão, edite o arquivo _ViewStart.cshtml deixe a linha que define o layout comentada, logo após executa a aplicação como na figura “Site sem Layout” que a tela inicial não tem mais menu e nenhuma formatação de CSS e Bootstrap.

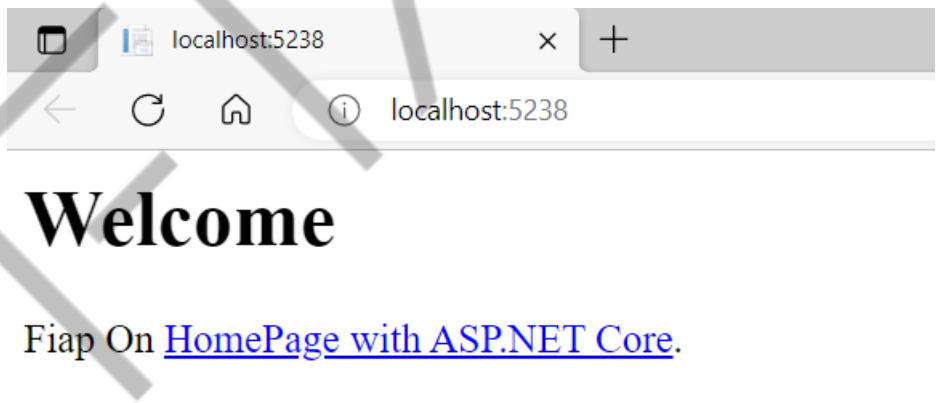


Figura 24 – Site sem Layout
Fonte: Elaborado pelo autor (2022)

Dado o teste anterior, notamos que perdemos o layout loga na página Home, então, agora vamos declarar apenas para a página Home o uso do layout. Abra o arquivo **Views\Home\Index.cshtml** e declare o layout logo após a tag @model como no código-fonte “Importando Layout na *View* Home\Index”.

```
@{
    ViewData["Title"] = "Home Page";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Fiap On <a href="https://docs.microsoft.com/aspnet/core">HomePage with ASP.NET
Core</a>.</p>
</div>
```

Código-fonte 20 – Importando Layout na *View* Home\Index
Fonte: Elaborado pelo autor (2022)

Execute o projeto para confirmar que a nossa tela inicial voltou ao formato normal e sem problemas de layout. Porém, clique no item de menu Representantes para verificar que toda a funcionalidade apresenta problemas de layout e formatação.

Para resolver o problema de layout na funcionalidade de representantes, podemos abrir todas os arquivos de Views e adicionar a linha @{Layout} semelhante a linha que foi aplicada na View \Home\Index, porém não é uma abordagem recomendada devido ao trabalho de duplicação de código, ou seja, para esse caso o melhor é continuar com a definição do layout do projeto no arquivo _ViewStart.cshtml.

**IMPORTANTE:** O layout criado neste capítulo possui menu de acessos às funcionalidades do site. Seu uso não é indicado para uma página de login, pois, mesmo sem o usuário ter efetuado o login, seria possível acessar as funções do site. Para contornar esse problema, é possível criar outro Layout, que será usado em áreas não logadas, ou criar todo o conteúdo na própria página de Login.

DICA: Remova o comando @Layout de todas as views deixando apenas no arquivo _ViewStart.cshtml como na primeira versão que encontramos antes das alterações.

### 3.7.3 Validações

Até o momento, criamos um fluxo de navegação, adicionamos um cabeçalho e um rodapé padrão para o site por meio de Layout e usamos algumas facilidades do framework ASP.NET Core MVC (Web App), porém não inserimos nenhum tipo de validação de dados, deixando que qualquer informação digitada pelo usuário seja aceita no website.

É preciso criar bloqueios que não permitam a digitação de quaisquer dados nos formulários do sistema e, para isso, serão apresentadas algumas técnicas com o uso de recursos do framework para a implementação de validações.

### 3.8 Validação pelo *Controller*

Para as validações no *Controller*, tomaremos como base a *Action* Cadastrar() do RepresentanteController. Nela, será adicionada a validação que não permitirá o cadastro de representante sem que nome e cpf sejam digitados.

Antes de apresentar a codificação, precisamos saber que todos os *Controllers* do framework possuem uma propriedade chamada **ModelState**, em que podemos adicionar uma coleção de mensagens de erro e usá-la para controlar nosso fluxo ou deixar as mensagens disponíveis para nossas *Views*. A regra aplicada para o nosso exemplo deverá ser implementada com os seguintes passos:

- Validar o conteúdo do nome e cpf digitado.
- Adicionar uma mensagem de erro ao ModelState.
- Validar se existe algum erro no ModelState.
- Caso um erro seja encontrado no ModelState – manter o usuário na tela do formulário e exibir a mensagem de erro.
- Caso um erro não seja encontrado no ModelState – simular o cadastro no banco de dados e direcionar o usuário para a tela de lista.

Segue o código-fonte do *Controller* para a regra de validação:

```
[HttpPost]
public IActionResult Cadastrar(RepresentanteModel representante)
{
   // Validando o campo CPF
   if ( string.IsNullOrEmpty(representante.Cpf) )
   {
      ModelState.AddModelError("Cpf", "O campo Cpf é obrigatório");
   }

   // Validando o campo NOME
   if (string.IsNullOrEmpty(representante.NomeRepresentante))
   {
      ModelState.AddModelError("NomeRepresentante", "O campo Nome é obrigatório");
   }
```

```
        // Se o ModelState não possuir nenhum erro
        if ( ModelState.IsValid ) {
            // Imprime os valores do modelo
            Console.WriteLine("Descrição: " + representante.Cpf);
            Console.WriteLine("Comercializado: " + representante.NomeRepresentante);

            // Simila que os dados foram gravados.
            Console.WriteLine("Gravando o Representante");

            // Substituímos o return View()
            // pelo método de redirecionamento
            return RedirectToAction("Index", "Representante");

        // Caso o ModelState tenha algum erro
        } else
        {
            // retorna para tela do formulário
            return View(representante);
        }
    }
```

Código-fonte 21 – Validando dados pelo *Controller*
Fonte: Elaborado pelo autor (2022)

Nosso *Controller* já valida a entrada de dados, porém, ainda não informa para o usuário a mensagem de erro. O Razor, com a tag **asp-validation-summary**, ajudará nossa aplicação com isso. A tag **asp-validation-summary** renderiza ou exibe em nosso HTML todas as mensagens que foram adicionadas na propriedade ModelState, assim, precisamos inseri-la em nossa *View*. Veja no Código-Fonte “Mensagens de erro com asp-validation-summary”, o trecho HTML e o exemplo de uso da tag:

```
<!-- formulário HTML com Tag Helpers-->
  <form asp-action="Cadastrar" asp-controller="Representante" method="post">
    <div class="form-horizontal">
      <hr />
      <!-- Trecho de validação para se existe mensagem a ser exibida -->
      @if (!Html.ViewData.ModelState.IsValid)
      {
        <!-- Tag para exibição da lista de erros -->
        <div asp-validation-summary="All" class="alert alert-danger"></div>
      }
```

Código-fonte 22 – Mensagens de erro com asp-validation-summary
Fonte: Elaborado pelo autor (2022)

Veja o fluxo em execução e a mensagem de erro exibida na tela:

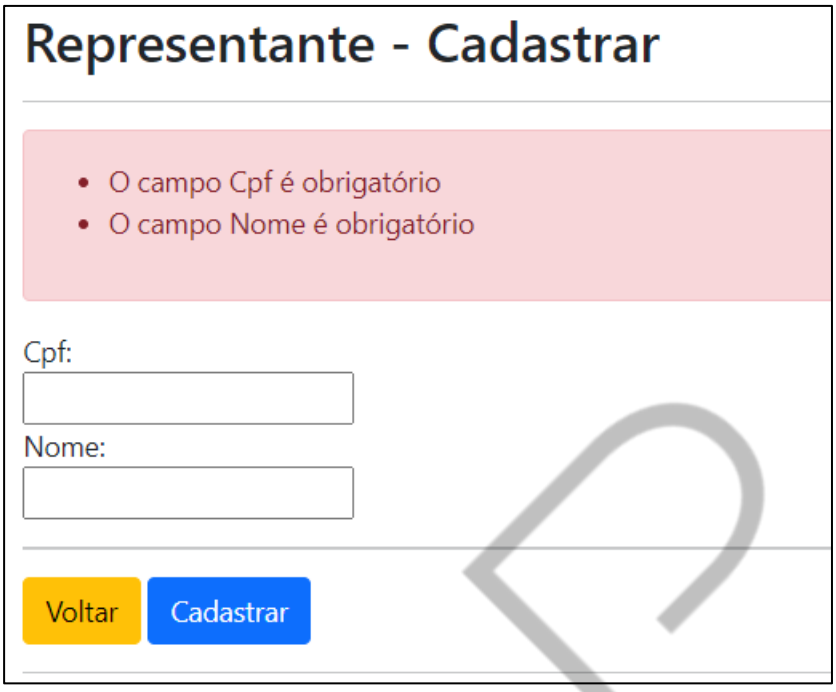


Figura 25 – Mensagem de erro com a tag asp-validation-summary
Fonte: Elaborado pelo autor (2022)

Implementamos nossa primeira validação, porém, cabe uma análise para aplicação futura. Nosso exemplo contou com apenas um atributo sendo validado. Você consegue imaginar um formulário com dez campos para a digitação do usuário? Teríamos que criar dez ou mais condições de verificação, correto?

No próximo bloco, avaliaremos uma alteração para esse nosso problema.

### 3.8.1 Validação com *Data Annotations*

Usar as validações pelo nosso *Controller* é funcional, porém, apresenta alguns pontos negativos, como a digitação de muitas linhas de código que não são reaproveitáveis.

Aqui entram as *Data Annotations*, que têm o mesmo objetivo de validação de dados, porém com algumas vantagens:

- Simplicidade.
- Produtividade.
- Reúso.
- Redução de erros.

As anotações serão utilizadas na nossa camada de modelo, assim, além de validar a entrega de dados nos componentes *View* e *Controller*, podemos usá-las na camada de acesso a dados.

Para o nosso exemplo, vamos inserir duas validações na propriedade descrição do modelo de tipo de projeto. Precisamos importar o *namespace using* **System.ComponentModel.DataAnnotations** e, com as simples {} (chaves) acima da declaração do atributo, escrevemos a validação.

O Código-Fonte "Validações com *Data Annotations*" apresenta a classe *Model* com duas validações no atributo Nome, ambas com o conceito de *Data Annotation,* vamos também colocar uma validação no campo CPF.

```
using System.ComponentModel.DataAnnotations;

namespace Fiap.Web.AspNet.Models
{

   public class RepresentanteModel
   {

      public int RepresentanteId { get; set; }

      [Required(ErrorMessage = "Nome do representante é obrigatório!")]
      [StringLength(80,
         MinimumLength = 2,
         ErrorMessage = "O nome deve ter, no mínimo, 2 e, no máximo, 80 caracteres")]
      public string? NomeRepresentante { get; set; }

      [Required(ErrorMessage = "CPF é obrigatório!")]
      public string? Cpf { get; set; }
```

Código-fonte 23 – Validações com *Data Annotations*
Fonte: Elaborado pelo autor (2022)

Depois de inserir nossas anotações no modelo, vamos remover a validação feita no **Controller.** Veja o Código-Fonte "Removendo a validação do *Controller*":

```
[HttpPost]
      public IActionResult Cadastrar(RepresentanteModel representante)
      {
         // Validando o campo CPF
         //if ( string.IsNullOrEmpty(representante.Cpf) )
         //{
         //    ModelState.AddModelError("Cpf", "O campo Cpf é obrigatório");
         //}

         //// Validando o campo NOME
         //if (string.IsNullOrEmpty(representante.NomeRepresentante))
```

```
        //{
        //    ModelState.AddModelError("NomeRepresentante", "O campo Nome é obrigatório");
        //}

        // Se o ModelState não possuir nenhum erro
        if ( ModelState.IsValid ) {
            // Imprime os valores do modelo
            Console.WriteLine("Descrição: " + representante.Cpf);
            Console.WriteLine("Comercializado: " + representante.NomeRepresentante);

            // Simila que os dados foram gravados.
            Console.WriteLine("Gravando o Representante");

            // Substituímos o return View()
            // pelo método de redirecionamento
            return RedirectToAction("Index", "Representante");

        // Caso o ModelState tenha algum erro
        } else
        {
            // retorna para tela do formulário
            return View(representante);
        }
    }
```

Código-fonte 24 – Removendo a validação do *Controller*
Fonte: Elaborado pelo autor (2022)

Execute a aplicação, refaça o fluxo de cadastro e insira um texto com mais de 50 caracteres no campo de descrição. Clique no botão **Cadastrar** e observe a mensagem de erro, conforme a Figura “Exibindo mensagem de erro com *Data Annotations*” a seguir:

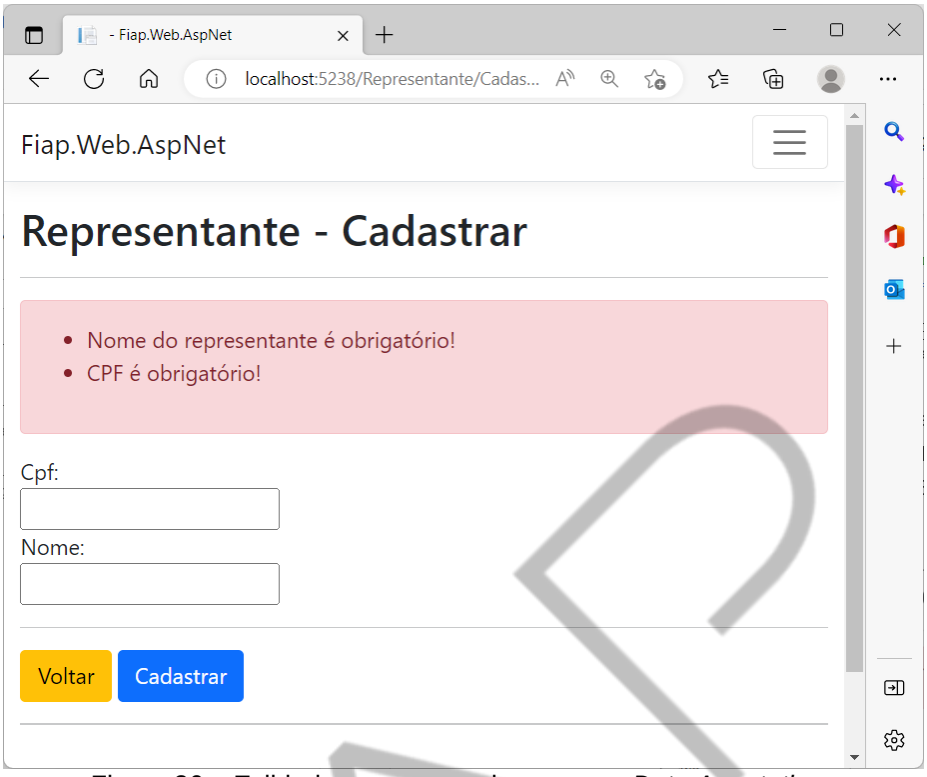


Figura 26 – Exibindo mensagem de erro com *Data Annotations*
Fonte: Elaborado pelo autor (2022)

Além das anotações do exemplo anterior, que foram usadas para validar o conteúdo de um campo e o tamanho máximo de caracteres digitados, está disponível uma série de outras validações, tais como: intervalo de números, validação de e-mail, expressões regulares e tipo de dados.

Veja, abaixo, o Quadro "*Annotation* para validação de dados" com as anotações mais comuns de validação e a sintaxe de uso:

| Nome | Uso | Sintaxe |
|---|---|---|
| Range | Validação de intervalos numéricos | [Range(0, 1000, ErrorMessage = "Mensagem")] |
| Email | Validação do formato de e-mail | [EmailAddress( ErrorMessage = "Erro")] |
| Regex | Validação de uma expressão regular | [RegularExpression(@"^[a-zA-Z''-'\s]{1,40}$", ErrorMessage="Erro")] |
| DataType | Validação do formato ou tipo de dado | [DataType(DataType.CreditCard, ErrorMessage = "Cartão inválido")] |

Quadro 6 – *Annotation* para validação de dados
Fonte: Elaborado pelo autor (2022)

### 3.8.2 Data Annotations e as Views

Conseguimos validar nossos dados usando as *Data Annotations*, mas podemos explorar um pouco mais de recursos e padronizar nossa aplicação.

O objetivo agora é usar as tags Razors para inserir os rótulos em nossos formulários e configurar a descrição do rótulo em nosso modelo. Outro ponto é exibir a mensagem de erro de validação em cada um dos campos.

O primeiro passo é inserir a anotação Display nos atributos do nosso *Model*. Veja o Código-Fonte “Anotação para rótulos” abaixo:

```
[Required(ErrorMessage = "Nome do representante é obrigatório!")]
    [StringLength(80,
        MinimumLength = 2,
        ErrorMessage = "O nome deve ter, no mínimo, 2 e, no máximo, 80 caracteres")]
    [Display(Name = "Nome do Representante")]
    public string? NomeRepresentante { get; set; }

    [Required(ErrorMessage = "CPF é obrigatório!")]
    [Display(Name = "CPF")]
```

Código-fonte 25 – Anotação para rótulos
Fonte: Elaborado pelo autor (2022)

O segundo passo é inserir no rótulo descritivo do campo a propriedade **asp-for** e incluir um elemento span abaixo da caixa de texto com a propriedade **asp-validation-for** para exibir a mensagem de erro do campo específico. Segue exemplo:

```
@model Fiap.Web.AspNet.Models.RepresentanteModel

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Representante - Cadastrar</title>
</head>
<body>
  <h1>Representante - Cadastrar</h1>

  <!-- formulário HTML com Tag Helpers-->
  <form asp-action="Cadastrar" asp-controller="Representante" method="post">
    <div class="form-horizontal">
      <hr />
      <!-- Trecho de validação para se existe mensagem a ser exibida -->
      @if (!Html.ViewData.ModelState.IsValid)
      {
        <!-- Tag para exibição da lista de erros -->
        <div asp-validation-summary="All" class="alert alert-danger"></div>
```

```
        }


        <div class="form-group">
            <label asp-for="Cpf" class="control-label"></label>
            <div class="col-md-10">
                <!-- Caixa de Texto -->
                <input asp-for="Cpf" />
                <!-- Bloco para exibir mensagem de erro -->
                <span asp-validation-for="Cpf" class="text-danger"></span>
            </div>
        </div>

        <div class="form-group">
            <label asp-for="NomeRepresentante" class="control-label"></label>
            <div class="col-md-10">
                <!-- Caixa de Texto -->
                <input asp-for="NomeRepresentante"  />
                <!-- Bloco para exibir mensagem de erro -->
                <span asp-validation-for="NomeRepresentante" class="text-danger"></span>
            </div>
        </div>
        <hr />
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <a asp-controller="Representante" asp-action="Index" class="btn btn-warning">Voltar</a>
                <input type="submit" value="Cadastrar" class="btn btn-primary" />
            </div>
        </div>
        <hr />
    </div>
</form>

</body>
</html>
```

Código-fonte 26 – Tag Razor para exibição dos rótulos
Fonte: Elaborado pelo autor (2022)

Você pode remover o bloco da tag **asp-validation-summary** para evitar a duplicidade das mensagens de erro na tela.

Observe como a mensagem é apresentada nesse novo formato:

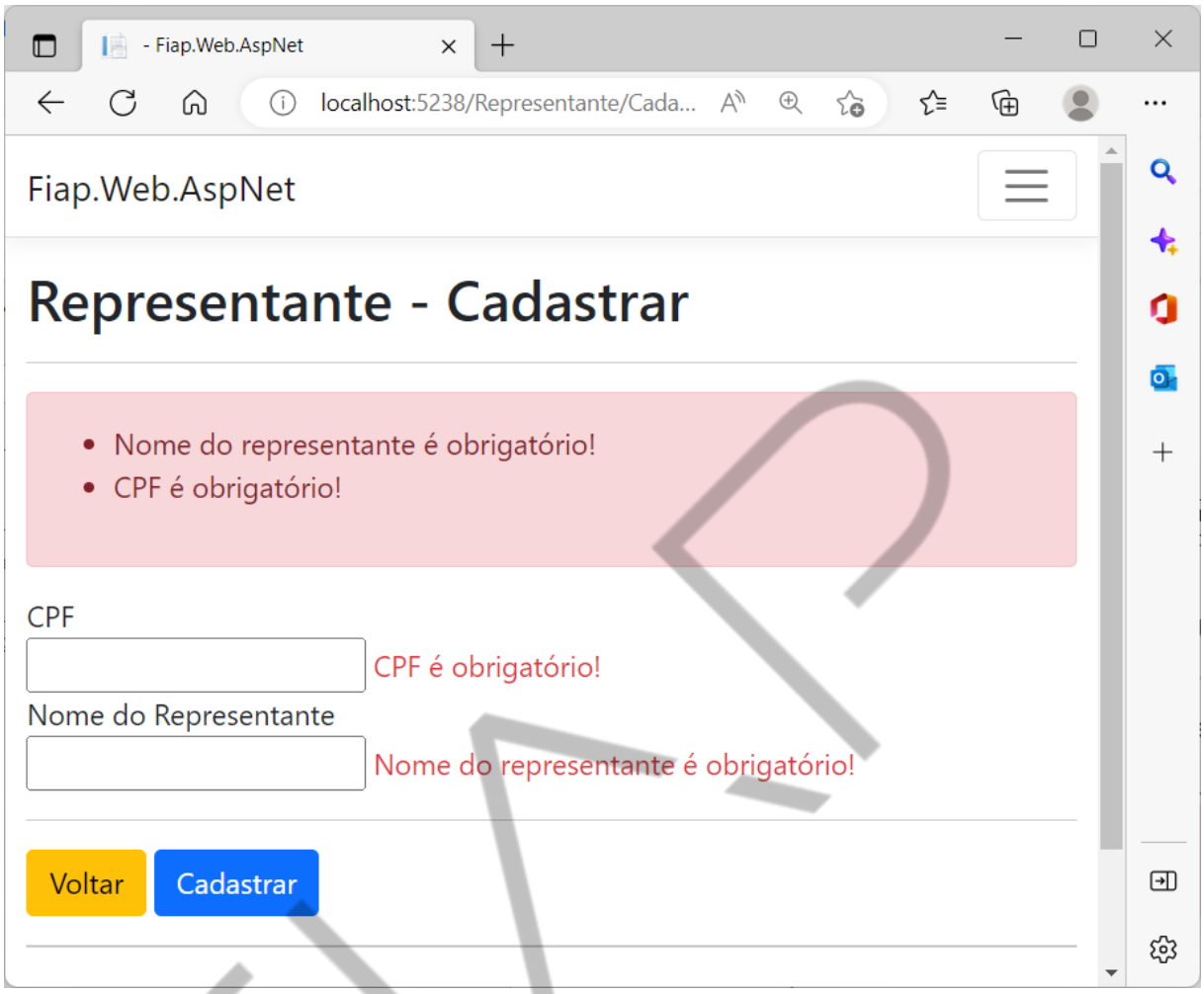


Figura 27 – Exibindo mensagem de erro com *Data Annotations*
Fonte: Elaborado pelo autor (2022)

### 3.8.3 Mensagens de sucesso com TempData

Chegou a hora de mostrar ao usuário as mensagens informando que as operações foram efetuadas com sucesso, pois, até aqui, apresentamos apenas mensagem de erro.

O recurso usado dessa vez é o **TempData**, cuja função é armazenar um valor de objeto em uma curta sessão de tempo entre requisições. É acessado pelo conjunto chave-valor, pode ser criado e acessado pelas *Views* e *Controller* e tem o seu conteúdo mantido até o momento em que algum componente recupere-o.

Vamos aplicar o conceito nas camadas de *Controller* e *Views*, respectivamente. Na *Action* Cadastrar do **RepresentanteController**, é necessário adicionar uma linha de comando que grave uma mensagem de sucesso na TempData. Essa mensagem

será adicionada ao fluxo de sucesso do cadastro. Veja o Código-Fonte “Gravando mensagens na TempData”:

```
// Anotação de uso do Verb HTTP Post
[HttpPost]
public IActionResult Cadastrar(RepresentanteModel representante)
{
   // Se o ModelState não possuir nenhum erro
   if ( ModelState.IsValid ) {
      // Imprime os valores do modelo
      Console.WriteLine("Descrição: " + representante.Cpf);
      Console.WriteLine("Comercializado: " + representante.NomeRepresentante);

      // Simila que os dados foram gravados.
      Console.WriteLine("Gravando o Representante");


      TempData["mensagem"] = "Representante cadastrado com sucesso";

      return RedirectToAction("Index", "Representante");

   // Caso o ModelState tenha algum erro
   } else
   {
      // retorna para tela do formulário
      return View(representante);
   }
}
```

Código-fonte 27 – Gravando mensagens na TempData
Fonte: Elaborado pelo autor (2022)

Mensagem de sucesso inserida na TempData, agora, precisamos exebí-la para o usuário. Lembre-se, quando o usuário finaliza um cadastro com sucesso, ele é direcionado para a *View* de lista de tipos, assim, a exibição do valor da **TempData** precisa ser inserida na **Representante\Index.cshtml**. Segue exemplo no Código-Fonte “Exibindo mensagens na TempData”:

```
@model IEnumerable<Fiap.Web.AspNet.Models.RepresentanteModel>

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Representantes</title>
</head>
<body>
  <h1>Representantes</h1>
  <p>
    <!-- uso de TagHelpers para definir o Controller e a Action -->
    <a asp-controller="Representante" asp-action="Cadastrar">Novo Representante</a>
```

```
    </p>

    <!-- Verifica se a chave "Mensagem" existe no TempData -->
    @if (@TempData["Mensagem"] != null)
    {
        <div class="alert alert-success" role="alert">
            <!-- Imprime para o usuário a mensagem -->
            @TempData["Mensagem"]
        </div>
    }


    <table class="table" border="1">
        <tr>
            <th>Id</th>
            <th>CPF</th>
            <th>Nome</th>
            <th></th>
        </tr>

        @foreach (var item in Model)
        {
            <tr>
                <td>
                    <label>@item.RepresentanteId</label>
                </td>
                <td>
                    <label>@item.Cpf</label>
                </td>
                <td>
                    <label>@item.NomeRepresentante</label>
                </td>
                <td>

                    <!-- asp-route-id é usado para informar o Id do Item selecionado. -->
                    <a asp-controller="Representante"
                       asp-action="Editar"
                       asp-route-id="@item.RepresentanteId">Editar</a>

                    <a asp-controller="Representante"
                       asp-action="Consultar"
                       asp-route-id="@item.RepresentanteId">Consultar</a>

                    <a asp-controller="Representante"
                       asp-action="Excluir"
                       asp-route-id="@item.RepresentanteId">Excluir</a>
                </td>
            </tr>
        }
    </table>
</body>
</html>
```

Código-fonte 28 – Exibindo mensagens na TempData
Fonte: Elaborado pelo autor (2022)

Execute a aplicação, faça o fluxo de cadastro de um novo tipo e verifique a mensagem de sucesso ao final do fluxo. Veja, na Figura “Exibindo mensagem de sucesso com TempData”, a tela de lista de tipos com a mensagem de sucesso exibida para o usuário:

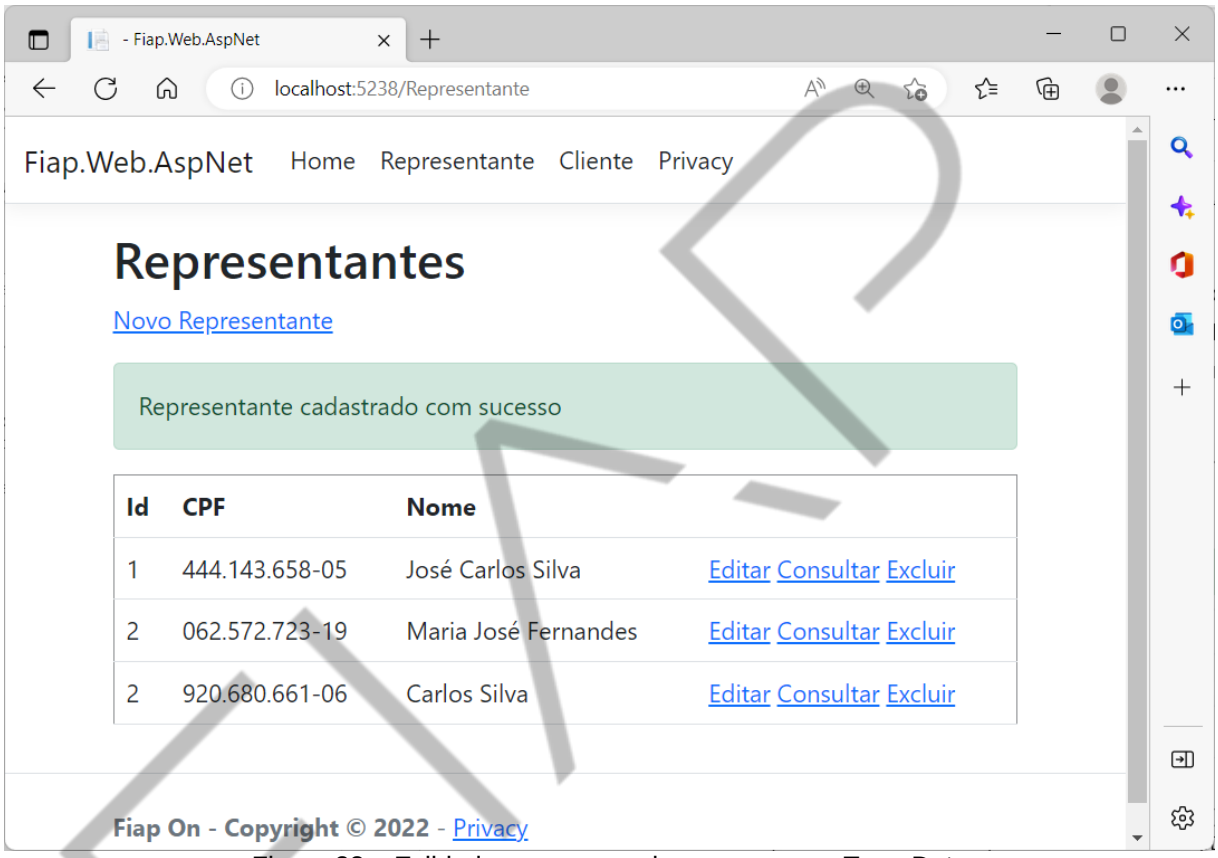


Figura 28 – Exibindo mensagem de sucesso com TempData
Fonte: Elaborado pelo autor (2020)

# 4 ACESSO A BANCO DE DADOS

## 4.1 ADO.NET

Chegamos ao momento de conectar nosso projeto ao nosso banco de dados e remover nossos códigos simulados (*mock*). Para isso, o .NET Framework disponibiliza um conjunto de classes e interfaces responsáveis por prover acesso e mecanismos de manipulação de dados.

Esses conjuntos de classes, ou essa biblioteca, são chamados ADO.NET (ActiveX Data Objects). Para aqueles que são familiarizados com a linguagem Java, podemos comparar o ADO.NET com as bibliotecas Java JDBC. Suas classes são acessadas pelo *namespace* **System.Data**.

A Figura “Classes ADO.NET” apresenta o conceito das bibliotecas ADO.NET.

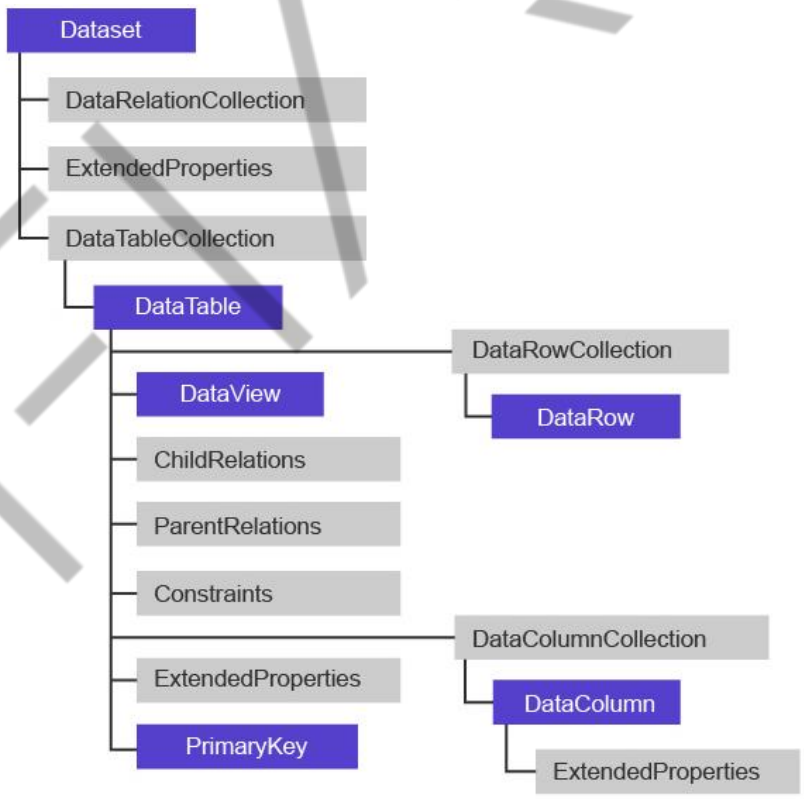


Figura 29 – Classes ADO.NET
Fonte: Elaborado pelo autor (2020)

## 4.2 Configurando acesso

Um dos primeiros passos para o trabalho com banco de dados é a configuração inicial, que consiste em baixar as bibliotecas necessárias e configurar usuário, senha, endereço do banco de dados, porta eentre outros requisitos.

Em nosso exemplo, usaremos o banco de dados Oracle hospedado na infraestrutura da FIAP. Assim, é necessário baixar via **Nuget Package Manager** a biblioteca para o cliente de acesso ao Oracle.

Acesse o menu Project > Nuget Package Manager (Projeto > Gerenciar pacotes do Nuget).

Faça uma busca pela palavra Oracle, selecione a opção **Oracle.ManagedDataAccess.Core** e solicite a instalação. A Figura “Cliente Oracle no Nuget” apresenta a escolha da biblioteca no Nuget Package Manager, veja:

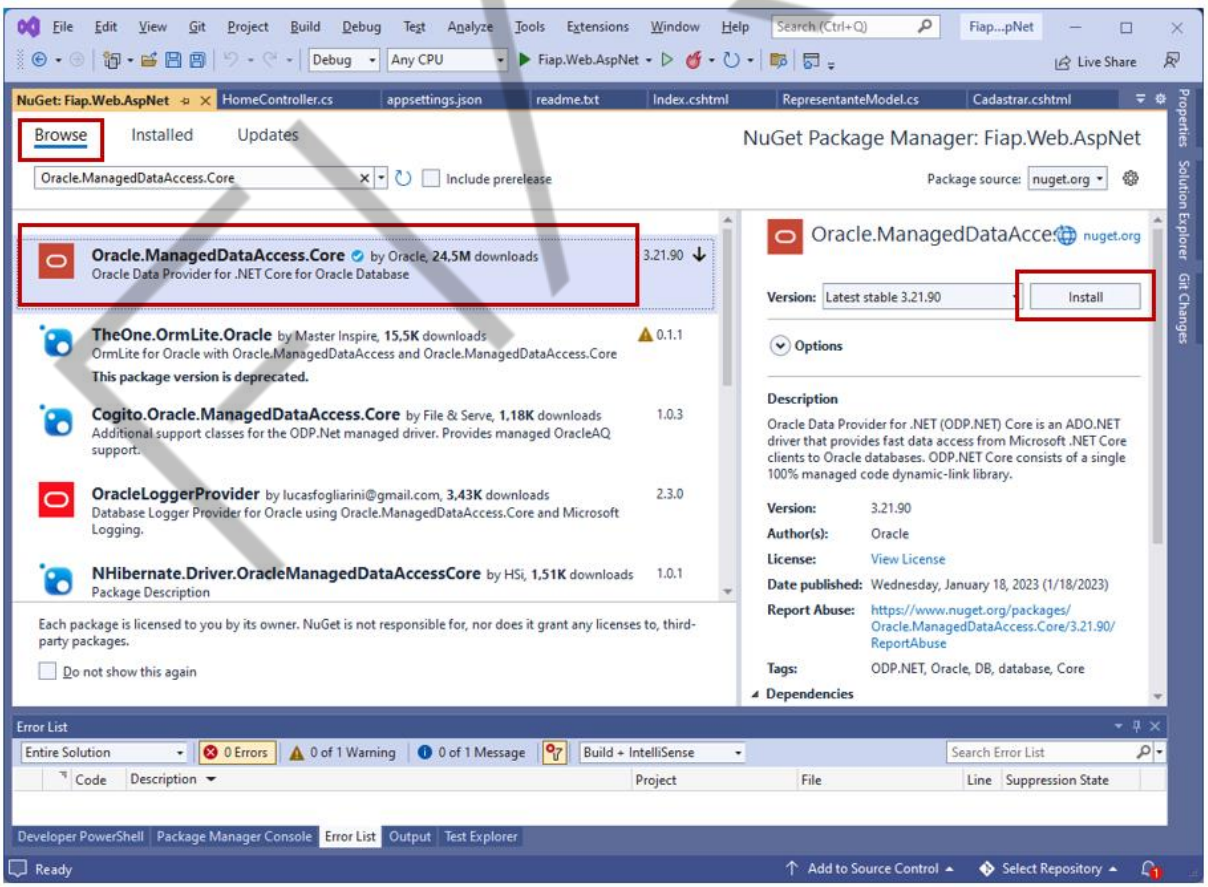


Figura 30 – Cliente Oracle no Nuget
Fonte: Elaborado pelo autor (2022)

Após a instalação do **Oracle.ManagedDataAcess.Core**, verifique como ficou a estrutura de bibliotecas do projeto **Fiap.Web.AspNet**. Veja o exemplo na Figura “Biblioteca Oracle no projeto”:

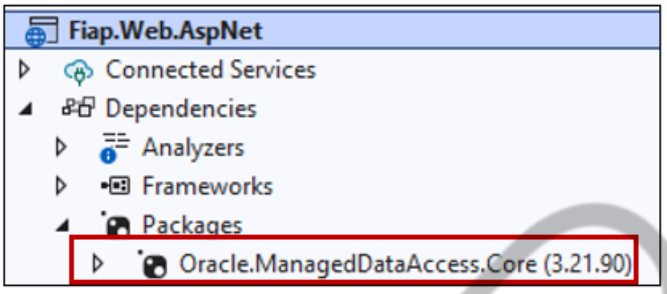


Figura 31 – Biblioteca Oracle no projeto
Fonte: Elaborado pelo autor (2022)

Com a biblioteca para acesso ao banco instalada, agora, precisamos configurar o caminho do banco de dados, usuário, senha e os demais requisitos. Para não ficar repetindo as configurações em todas as classes de acesso ao banco de dados, adicionaremos essas informações no arquivo de configuração do projeto **appsettings.json** uma única vez, assim, qualquer alteração será facilitada por estar em um único ponto.

O **appsettings.json** é um arquivo no formato JSON que contém as configurações do projeto. Agora, precisamos inserir a configuração do nosso banco de dados.

Abra o arquivo **appsettings.json** (raiz do projeto) e acrescente a configuração da String de conexão para acesso ao banco de dados Oracle da Fiap. O nome da string de conexão será *DatabaseConnection*.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DatabaseConnection": "Data Source=(DESCRIPTION = (ADDRESS_LIST = (ADDRESS = (PROTOCOL = TCP)(HOST = oracle.fiap.com.br)(PORT = 1521)))(CONNECT_DATA = (SID = orcl)));Persist Security Info=True;User ID=rm99999;Password=senha;Pooling=True;Connection Timeout=60;"
  }

}
```

Código-fonte 29 – appsettings.json criando String de conexão Oracle
Fonte: Elaborado pelo autor (2022)

**IMPORTANTE:** É preciso trocar os dados de conexão de acordo com o local do seu banco de dados Oracle, usuário e senha. Para o arquivo **appsettings.json**, é obrigatório seguir algumas estruturas para a declaração das *strings* de conexão e da configuração da fonte de dados. Por isso, evite trocar o posicionamento dessas seções.

Com a configuração pronta e disponível para acesso da nossa aplicação ao Oracle, já é possível efetuar a conexão e executar comandos em nossa base. O Código-Fonte “ADO.NET exemplo de comandos” mostra uma forma genérica de executar um comando SELECT da base de dados:

```
// STRING DE CONEXAO
var connectionString = new ConfigurationBuilder()

     .SetBasePath(Directory.GetCurrentDirectory())

     .AddJsonFile("appsettings.json")

     .Build().GetConnectionString("DatabaseConnection");

// CONEXAO COM O BANCO DE DADOS
OracleConnection Connection = new OracleConnection(connectionString);
Connection.Open();

// EXECUTANDO A QUERY
OracleCommand Command = new OracleCommand("SELECT * FROM NOME_DA_TABELA", Connection);
OracleDataReader Reader = Command.ExecuteReader();
```

```
while (Reader.Read())
{
     System.Diagnostics.Debug.Print(Reader[1].ToString());
}

// Fechando as Conexões
Reader.Close();
```

Código-fonte 30 – ADO.NET exemplo de comandos
Fonte: Elaborado pelo autor (2022)

### 4.3 Componentes ADO.NET

Para executar comandos no banco de dados com ADO.NET, precisamos de um conjunto de classes para executar os processos de abrir uma conexão, executar um comando e receber o resultado. Segue a lista das principais classes e suas características:

- **Connection** – O objeto tem a função de gerar conexão com uma base de dados. É necessário informar a *string* de conexão.
- **Command** – É o responsável pela execução de comando no banco de dados. Possui três métodos para a execução dos comandos. **ExecuteReader**, utilizado para recuperação de dados (por exemplo: select); **ExecuteNonReader**, usado para comandos que não retornam dados (por exemplo: Insert); e **ExecuteScalar**, utilizado para comandos que retornam apenas uma informação (por exemplo: Max. Count).
- **DataReader** – É o responsável por ler os dados retornados dos objetos Command, permitindo percorrer a lista de registros retornados.

**DICA:** Os objetos acima são da especificação ADO.NET. Em nosso projeto, utilizaremos objetos da especificação Oracle e, por esse motivo, o nome das classes possui o prefixo Oracle (por exemplo: OracleCommand).

Abaixo, segue o Código-Fonte “ADO.NET ExecuteNonQuery()” com exemplo de uso dos objetos Connection, Command – com as três formas de execução – e DataReader.

Veja que o Código-Fonte “ADO.NET ExecuteNonQuery()”, para abrir a conexão, usa o objeto Command, passando parâmetros para o comando SQL e o método ExecuteNonQuery:

```
// Recuperando a String de conexão
var connectionString = new ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json")

.Build().GetConnectionString("Fiap.Web.AspNetConnection");

            // Criando o Objeto de Conexão
            using (OracleConnection connection = new
OracleConnection(connectionString))
            {
                // Comando SQL
                // Simbolo : significa que são parâmetros
informados para o comando
                String query =
                    "INSERT INTO TABELA VALUES (:nome,
:descricao, :preco) ";

                // Criando o objeto que executar o comando
                OracleCommand command = new
OracleCommand(query, connection);

                // Adicionando o valor ao comando
                command.Parameters.Add(new
OracleParameter("nome", "Produto 1"));
                command.Parameters.Add(new
OracleParameter("descricao", "Descrição do produto 1"));
                command.Parameters.Add(new
OracleParameter("preco", 2098.98));


                // Abrindo a conexão com  o Banco
                connection.Open();

                // Executa o comando de Insert
                command.ExecuteNonQuery();

                // Fecha a conexão
                connection.Close();

            } // Finaliza o objeto connection
```

Código-fonte 31 – ADO.NET ExecuteNonQuery()
Fonte: Elaborado pelo autor (2022)

Veja, no Código-Fonte “ADO.NET ExecuteScalar()”, como usar o objeto Command e o método ExecuteScalar:

```
// Recuperando a String de conexao
var connectionString = new ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json")

.Build().GetConnectionString("DatabaseConnection");

// Criando o Objeto de Conexão
using (OracleConnection connection = new
OracleConnection(connectionString))
{
     // Simbolo :significa que são parâmetros informados para
o comando
     String query =
          "SELECT MAX(COLUNA) FROM TABELA WHERE PRECO > :preco
";

     // Criando o objeto que executar o comando
     OracleCommand command = new OracleCommand(query,
connection);

     // Adicionando o valor ao comando
     command.Parameters.Add(new OracleParameter("preco",
2098.98));

     // Abrindo a conexão com  o Banco
     connection.Open();

     // Executa o comando e recupera o valor da função SQL MAX
     int maximo = (int) command.ExecuteScalar();

     // Fecha a conexão
     connection.Close();

} // Finaliza o objeto connection
```

Código-fonte 32 – ADO.NET ExecuteScalar()
Fonte: Elaborado pelo autor (2022)

Veja, nos Códigos-Fonte “ADO.NET ExecuteQuery() e DataReader”, como usar os objetos Command e DataReader para recuperar uma lista de dados no resultado:

```
using (OracleConnection connection = new
OracleConnection(connectionString))
{
     // Simbolo significa que são parâmetros informados para o
comando
     String query =
          "SELECT ID, NOME FROM TABELA WHERE PRECO > :preco ";

     // Criando o objeto que executar o comando
     OracleCommand command = new OracleCommand(query,
connection);

     // Adicionando o valor ao comando
     command.Parameters.Add(new OracleParameter("preco",
2098.98));

     // Abrindo a conexão com  o Banco
     connection.Open();

     // Criando o objeto DataReader com o retorno do comando
SELECT
     OracleDataReader dataReader = command.ExecuteReader();

     // Percorre para lista retornada
     while (dataReader.Read())
     {
          // Recupera o valor pela posição da coluna
          var id = (int) dataReader[1];

          // Recupera o valor pelo nome da coluna
          var nome = dataReader["NOME"];
     }

     // Fecha a conexão
     connection.Close();

} // Finaliza o objeto connection
```

Código-fonte 33 – ADO.NET ExecuteQuery() e DataReader
Fonte: Elaborado pelo autor (2022)

**DICA:** O bloco *using* provê ao desenvolvedor a habilidade de se criar um bloco de código isolado dentro de um determinado programa. Veja mais detalhes em: https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/using-statement.

### 4.4 Refatorando a aplicação

Agora que temos uma conexão configurada com o nosso banco de dados Oracle e exemplos ADO.NET de como acessar o banco e executar os comandos SQL mais comuns em aplicações comerciais, chegou a hora de remover os códigos simulados e conectar nosso aplicativo MVC em nossa base de dados.

O primeiro passo é criar nossa tabela de Representante e, para isso, use o script SQL abaixo:

```
CREATE TABLE REPRESENTANTE (
        REPRESENTANTEID NUMBER PRIMARY KEY,
        NOMEREPRESENTANTE VARCHAR2(80) NOT NULL,
        CPF VARCHAR2(14) NOT NULL
);

CREATE SEQUENCE REPRESENTANTE_ID_SEQ;

CREATE OR REPLACE TRIGGER TR_SEQ_REPRESENTANTE BEFORE INSERT ON REPRESENTANTE FOR EACH
ROW
BEGIN
        SELECT REPRESENTANTE_ID_SEQ.NEXTVAL INTO :new.REPRESENTANTEID FROM dual;
END;

--DROP TRIGGER TR_SEQ_REPRESENTANTE;
--DROP SEQUENCE REPRESENTANTE_ID_SEQ;
--DROP TABLE REPRESENTANTE;
```

Código-fonte 34 – *Script* para criação de tabela Representante
Fonte: Elaborado pelo autor (2022)

Tabela criada, podemos seguir para o segundo passo: a criação de um *namespace* chamado **Repository,** que funcionará como nossa Data Access Layer. O *namespace* **Repository** será o responsável pelas classes que vão acessar o banco de dados e executar os comandos. Em seguida, já podemos executar o terceiro passo: criar uma classe com o nome **RepresentanteRepository** dentro da pasta **Repository**. A Figura “Camada Repository”, a seguir, apresenta a estrutura do projeto com o *namespace* Repository:

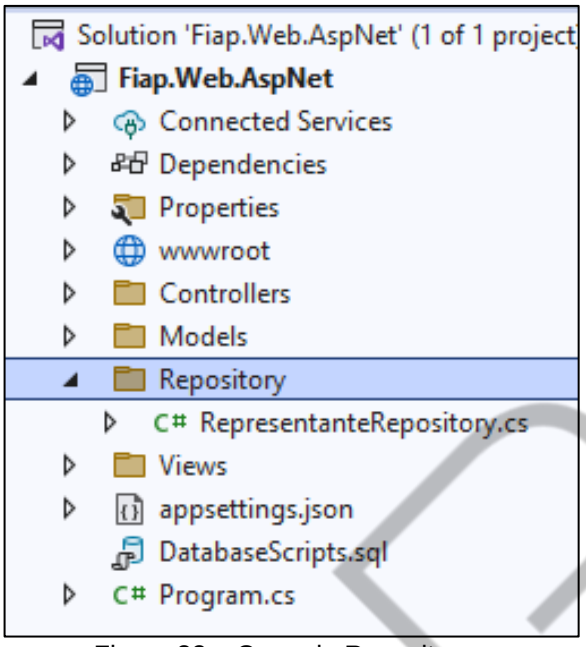


Figura 32 – Camada Repository
Fonte: Elaborado pelo autor (2022)

## 4.5 Implementando ADO.NET

Chegamos ao quarto passo e agora é a hora de definir quais operações precisamos ter com o nosso banco de dados.

Navegando pela funcionalidade de representante, é possível notar que precisamos listar os representantes da nossa base, inserir novos, excluir, alterar os existentes e consultar detalhes de um único representante. No total, precisamos de cinco operações, então vamos precisar de cinco métodos na classe **RepresentanteRepository**.

O Código-Fonte “Estrutura dos métodos do RepresentanteRepository” mostra a declaração dos métodos para a classe Repository. Observe os parâmetros de entrada e o tipo de retorno de cada um deles:

```
using Fiap.Web.AspNet.Models;

namespace Fiap.Web.AspNet.Repository
{
   public class RepresentanteRepository
   {

      public IList<RepresentanteModel> Listar()
      {
         // Codificar e corrigir o retorno
         return null;
      }

      public RepresentanteModel Consultar(int id)
      {
         // Codificar e corrigir o retorno
         return null;
      }

      public void Inserir(RepresentanteModel Representante)
      {
         // Codificar
      }

      public void Alterar(RepresentanteModel Representante)
      {
         // Codificar
      }

      public void Excluir(int id)
      {
         // Codificar
      }

   }
}
```

Código-fonte 35 – Estrutura dos métodos do RepresentanteRepository
Fonte: Elaborado pelo autor (2022)

A estrutura dos métodos está criada. É preciso escrever os comandos SQL e, utilizando os recursos do ADO.NET, também os códigos para a execução dos comandos. Para evitar redundância de código vamos usar propriedades/variáveis no escopo da classe e vamos usar o construtor para iniciá-las. Abaixo, segue o resultado da classe **RepresentanteRepository**:

```
using Fiap.Web.AspNet.Models;
using Oracle.ManagedDataAccess.Client;

namespace Fiap.Web.AspNet.Repository
```

```
{
    public class RepresentanteRepository
    {

        private string? connectionString;
        private OracleConnection? connection;

        public RepresentanteRepository()
        {
            connectionString = new ConfigurationBuilder()
                                .SetBasePath(Directory.GetCurrentDirectory())
                                .AddJsonFile("appsettings.json")
                                .Build().GetConnectionString("DatabaseConnection");

            connection = new OracleConnection(connectionString);
        }



        public IList<RepresentanteModel> Listar()
        {
            var lista = new List<RepresentanteModel>();

            using (connection)
            {
                var query = "SELECT REPRESENTANTEID, NOMEREPRESENTANTE, CPF FROM REPRESENTANTE";

                connection.Open();

                OracleCommand command = new OracleCommand(query, connection);
                OracleDataReader dataReader = command.ExecuteReader();

                while (dataReader.Read())
                {
                    // Recupera os dados
                    RepresentanteModel representante = new RepresentanteModel();
                    representante.RepresentanteId = Convert.ToInt32(dataReader["REPRESENTANTEID"]);
                    representante.NomeRepresentante = dataReader["NOMEREPRESENTANTE"].ToString();
                    representante.Cpf = dataReader["CPF"].ToString();

                    // Adiciona o modelo da lista
                    lista.Add(representante);
                }

                connection.Close();

            } // Finaliza o objeto connection

            // Retorna a lista
            return lista;

        }

        public RepresentanteModel Consultar(int id)
        {
```

```
        var representante = new RepresentanteModel();

        using (connection)
        {
            var query = "SELECT REPRESENTANTEID, NOMEREPRESENTANTE, CPF FROM REPRESENTANTE
WHERE REPRESENTANTEID = :ID ";

            connection.Open();

            OracleCommand command = new OracleCommand(query, connection);
            command.Parameters.Add("ID", id);

            OracleDataReader dataReader = command.ExecuteReader();

            if (dataReader.Read())
            {
                // Recupera os dados
                representante.RepresentanteId = Convert.ToInt32(dataReader["REPRESENTANTEID"]);
                representante.NomeRepresentante = dataReader["NOMEREPRESENTANTE"].ToString();
                representante.Cpf = dataReader["CPF"].ToString();
            }

            connection.Close();

        } // Finaliza o objeto connection

        // Retorna o objeto representante

        return representante;
    }

    public void Inserir(RepresentanteModel representante)
    {
        using (connection)
        {
            String query = "INSERT INTO REPRESENTANTE ( NOMEREPRESENTANTE, CPF ) VALUES ( :nome,
:cpf ) ";

            connection.Open();

            OracleCommand command = new OracleCommand(query, connection);

            // Adicionando o valor ao comando
            command.Parameters.Add("nome", representante.NomeRepresentante);
            command.Parameters.Add("cpf", representante.Cpf);

            command.ExecuteNonQuery();
            connection.Close();

        }

    }

    public void Alterar(RepresentanteModel representante)
    {
```

```
        using (connection)
        {
            String query = "UPDATE REPRESENTANTE SET NOMEREPRESENTANTE = :nome, CPF = :cpf
WHERE REPRESENTANTEID = :id ";

            connection.Open();

            OracleCommand command = new OracleCommand(query, connection);

            // Adicionando o valor ao comando
            command.Parameters.Add("nome", representante.NomeRepresentante);
            command.Parameters.Add("cpf", representante.Cpf);
            command.Parameters.Add("id", representante.RepresentanteId);


            command.ExecuteNonQuery();
            connection.Close();

        }
    }

    public void Excluir(int id)
    {
        using (connection)
        {
            String query = "DELETE REPRESENTANTE WHERE REPRESENTANTEID = :id ";

            connection.Open();

            OracleCommand command = new OracleCommand(query, connection);

            // Adicionando o valor ao comando
            command.Parameters.Add("id", id);


            command.ExecuteNonQuery();
            connection.Close();

        }
    }


  }
}
```

Código-fonte 36 – Código completo do RepresentanteRepository
Fonte: Elaborado pelo autor (2022)

Agora podemos remover os códigos simulados em nosso componente *Controller*. Precisamos passar em todos os métodos para remover o código simulado

e adicionar uma instância de **RepresentanteRepository** e a chamada do método correspondente para a operação desejada.

Veja como ficou o código-fonte da versão final do nosso *Controller*:

```
using Fiap.Web.AspNet.Models;
using Fiap.Web.AspNet.Repository;
using Microsoft.AspNetCore.Mvc;

namespace Fiap.Web.AspNet.Controllers
{
   public class RepresentanteController : Controller
   {

      private RepresentanteRepository representanteRepository;

      public RepresentanteController()
      {

         representanteRepository = new RepresentanteRepository();
      }

      public IActionResult Index()
      {
         // Retornando para View a lista de Representantes
         var lista = representanteRepository.Listar();

         return View(lista);
      }

      // Anotação de uso do Verb HTTP Get
      [HttpGet]
      public IActionResult Cadastrar()
      {
         // Retorna para a View Cadastrar um
         // objeto modelo com as propriedades em branco
         return View(new RepresentanteModel());
      }

      // Anotação de uso do Verb HTTP Post
      [HttpPost]
      public IActionResult Cadastrar(RepresentanteModel representante)
      {
         if ( ModelState.IsValid ) {

            representanteRepository.Inserir(representante);

            TempData["mensagem"] = "Representante cadastrado com sucesso";
            return RedirectToAction("Index", "Representante");
         } else
```

```
        {
            return View(representante);
        }
    }


    [HttpGet]
    public IActionResult Editar([FromRoute] int id)
    {
        var representante = representanteRepository.Consultar(id);
        return View(representante);
    }

    [HttpPost]
    public IActionResult Editar(RepresentanteModel representante)
    {
        if (ModelState.IsValid)
        {
            representanteRepository.Alterar(representante);

            TempData["mensagem"] = "Representante alterado com sucesso";
            return RedirectToAction("Index", "Representante");
        }
        else
        {
            return View(representante);
        }

    }


    [HttpGet]
    public IActionResult Consultar(int id)
    {
        var representante = representanteRepository.Consultar(id);
        return View(representante);
    }

    [HttpGet]
    public IActionResult Excluir(int id)
    {
        representanteRepository.Excluir(id);

        TempData["mensagem"] = "Representante excluído com sucesso";
        return RedirectToAction("Index", "Representante");
    }


  }
}
```

Código-fonte 37 – Código completo do RepresentanteController
Fonte: Elaborado pelo autor (2022)

## 4.6 Filtros

Filtros são uma técnica de acrescentar regras em nossos *Controller* e *Actions*. O framework MVC disponibiliza os atributos **Action Filters** para facilitar a implementação da técnica de filtros.

### 4.6.1 Atributos

Quando abordamos filtros no framework MVC, precisamos falar de atributos, que, nesse contexto, é a nomenclatura para definir uma classe que possui um comportamento diferente do normal.

Um atributo é uma informação a mais que damos para uma classe, um método ou um parâmetro, sua forma de inserção é feita por anotações. Na sequência deste capítulo, usaremos muito a inserção de anotações, assim o conceito de atributo ficará mais claro.

### 4.6.2 Action Filters

O ***Action Filter*** é um conceito de biblioteca que permite executar lógicas antes ou depois da execução de uma ação do *Controller*. Dessa forma, é possível injetar comportamento em determinadas ações ou partes da aplicação, o que ajudará a resolver problemas com apenas uma linha de código.

O framework ASP.NET Core MVC (Web App) disponibiliza quatro tipos de filtros, são eles:

- **Authorization Filter** – usado para a implementação de autenticação e segurança.
- **Action Filter** – possibilita inserir comportamento na execução de um ActionMethod.

- **Result Filter** – permite inserir comportamento na execução de um ActionResult.
- **Exception Filter** – facilita o tratamento de exceções no sistema em um ponto centralizado.

O Código-Fonte “Exemplo de ActionFilters” mostra um exemplo de uso de um Action Filter. Nele, é possível ver que o *Controller* possui uma anotação [Authorize] que, possivelmente, é um Action Filter customizado para validar se o usuário efetuou login, e a outra anotação é para o filtro [AllowAnonymous], que significa uma exceção ao filtro [Authorize].

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login(string returnUrl)
    {
        ViewBag.ReturnUrl = returnUrl;
        return View();
    }
}
```

Código-fonte 38 – Exemplo de ActionFilters
Fonte: Elaborado pelo autor (2022)

Para entender melhor o uso e fixar esse conceito, implementaremos um filtro em nosso projeto.

### 4.6.3 Implementando *Action Filters*

Usaremos um exemplo simples para implementarmos um filtro em nosso aplicativo. Imagine que, por alguma necessidade do negócio, precisamos gerar um log de acesso em algumas ações e *Controllers* da nossa aplicação. Dessa forma, implementaremos um ***Action Filter*** que salvará as informações de execução em uma base de dados para consulta futura.

Os códigos implementados neste capítulo não farão a gravação no banco de dados. No lugar, será impressa uma mensagem de execução na janela Output do Visual Studio.

Para iniciarmos, de fato, a codificação é preciso criar uma pasta com o nome Filtros em nossa pasta *Controller* e, em seguida, criar uma simples classe C# com o nome de **LogFilter**. Veja na Figura "*Namespace* Controller\Filters e classe LogFilter", a estrutura do projeto após a criação:

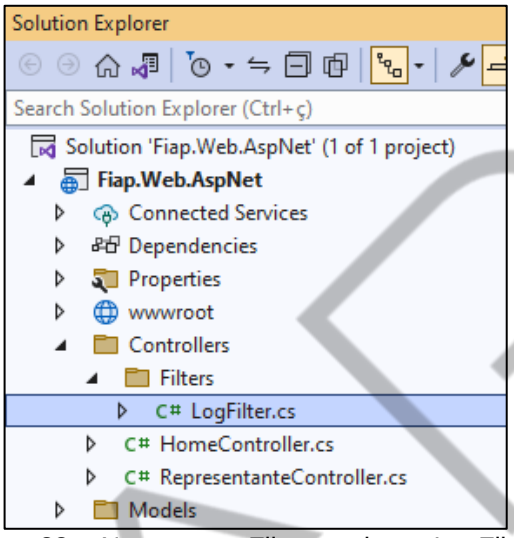


Figura 33 – *Namespace* Filtros e classe LogFilter
Fonte: Elaborado pelo autor (2022)

Edite a classe LogFilter e, em sua declaração, estenda a superclasse **ActionFilterAttribute**. O próximo ponto é sobrescrever os método **OnActionExecuting, OnActionExecuted, OnResultExecuting e OnResultExecuted** que vamos inserir no nosso código para logar as informações necessárias.

Neste exemplo, será usada a classe Console para imprimir a execução e o ação executada. Veja, no Código-Fonte "Log ActionFilters", a classe filtro, atente para a extensão da classe **ActionFilterAttribute** e seus métodos:

```
using Microsoft.AspNetCore.Mvc.Filters;

namespace Fiap.Web.AspNet.Controllers.Filters
{
    public class LogFilter : ActionFilterAttribute
    {
        public override void OnActionExecuting(ActionExecutingContext context)
        {
            Console.WriteLine("FiapLogFilter - OnActionExecuting");
            base.OnActionExecuting(context);
        }

        public override void OnActionExecuted(ActionExecutedContext context)
        {
```

```
            Console.WriteLine("FiapLogFilter - OnActionExecuted");
            base.OnActionExecuted(context);
        }

        public override void OnResultExecuting(ResultExecutingContext context)
        {
            Console.WriteLine("FiapLogFilter - OnResultExecuting");
            base.OnResultExecuting(context);
        }

        public override void OnResultExecuted(ResultExecutedContext context)
        {
            Console.WriteLine("FiapLogFilter - OnResultExecuted");
            base.OnResultExecuted(context);
        }
    }
}
```

Código-fonte 39 – Log ActionFilters
Fonte: Elaborado pelo autor (2022)

Criado o primeiro filtro, agora, é só usá-lo. Anote a *Action* Index do **RepresentanteController** com a anotação **[Filtros.LogFilter]** ou **[LogFilter]**, se você tiver importado o *namespace* Filtros. Veja, no Código-Fonte "*Action* Index usando o Filtro de Log", a forma de uso da anotação:

```
        [LogFilter]
        public IActionResult Index()
        {
            // Retornando para View a lista de Representantes
            var lista = representanteRepository.Listar();

            return View(lista);
        }
```

Código-fonte 40 – *Action* Index usando o Filtro de Log
Fonte: Elaborado pelo autor (2022)

Execute a aplicação e abra a tela de lista de tipos. Após a abertura, observe a janela ***Console*** do projeto, será possível ver a mensagem de log. Verifique a janela a cada interação com o site, valide se apenas a *Action* Index tiver mensagem de log no ***Console***.

Depois de testado o nosso filtro de log, remova da Index, anote toda a classe RepresentanteController e repita a navegação pelo site. Dessa forma, todas as ações desse *Controller* serão gravadas no log. Veja a janela de Output na Figura "Mensagens geradas pelo LogFilter":

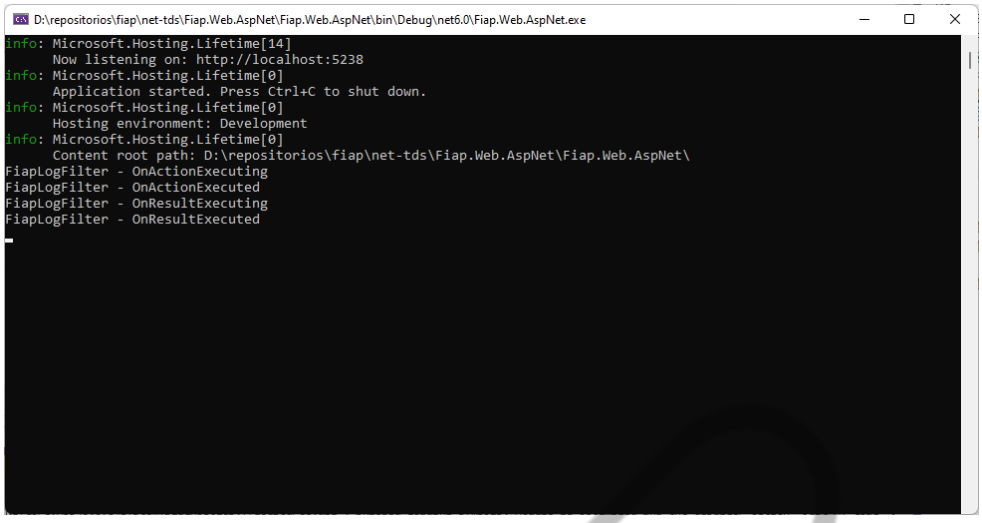


Figura 34 – Mensagens geradas pelo LogFilter
Fonte: Elaborado pelo autor (2022)

O filtro está implementado e em funcionamento, caso queira, pode anotar outros *Controllers* com essa anotação e deixar o filtro acionado. Outro ponto que deve ser alterado é a mensagem na janela Console. Sinta-se livre para fazer com que essa informação seja armazenada em uma base de dados.

**DICA:** Os filtros são formas fáceis e produtivas de levar comportamento para os Controllers sem muita codificação. No simples exemplo anterior fizemos uma forma de Log bem rudimentar, mas é comum em aplicações profissionais ter uma trilha de auditoria, que grava em algum repositório do sistema todas as operações feitas pelo usuário, assim com o uso de ***Actions Filters*** seria muito fácil essa implementação em todo um grande sistema.

Seguem os links para o download da solução Fiap.Web.AspNet com toda a funcionalidade de representante implementada:

Git: https://github.com/FIAPON/Fiap.Web.AspNethttps://github.com/FIAP/smartcities-orm/tree/08-socket

Zip: https://github.com/FIAPON/Fiap.Web.AspNet/archive/refs/heads/master.zip

Vale também baixar a versão abaixo e verificar como ficaria a funcionalidade de clientes. Não se esqueça de criar a tabela de clientes no banco de dados, o script para criação encontra-se no projeto no arquivo **DatabaseScripts.sql**.

Git: https://github.com/FIAPON/Fiap.Web.AspNet/tree/cliente

Zip: https://github.com/FIAPON/Fiap.Web.AspNet/archive/refs/heads/cliente.zip

https://github.com/FIAP/smartcities-orm/archive/refs/heads/08-socket.zip

## Considerações finais

Neste capítulo, apresentamos um exemplo de funcionalidade implementada com uso do framework ASP.NET Core MVC (Web App). Demonstramos o fluxo da aplicação e o fluxo de criação dos componentes *Model*, *View* e *Controller,* usando os assistentes do framework e do Visual Studio.

Mostramos também o conjunto de bibliotecas ADO.NET, junto com o MVC, para efetuar a conexão com o banco de dados e realizar um fluxo completo com componentes MVC e banco de dados.

Para finalizar, abordamos o conteúdo de filtros para *Controller* MVC, permitindo, com anotações, inserir regras em ações executadas pelos controladores.

# REFERÊNCIAS

**ADO.NET**. Disponível em: https://msdn.microsoft.com/pt-br/library/e80y5yhx(v=vs.110).aspx. Acesso em: 13 jan. 2021.

ARAÚJO, E. C. **ASP.NET MVC5** – Crie aplicações web na plataforma Microsoft. São Paulo: Casa do Código, 2017.

ARAÚJO, E. C. **ASP.NET Core MVC** – Aplicações modernas em conjunto com o Entity Framework. São Paulo: Casa do Código, 2018.

ARAÚJO, E. C. **C# e Visual Studio** – Desenvolvimento de aplicações desktop. São Paulo: Casa do Código, 2015.

ARAÚJO, E. C. **Orientação a Objetos em C#** – Conceitos e implementações em .NET. São Paulo: Casa do Código, 2017.

**Classe HtmlHelper**. Disponível em: https://msdn.microsoft.com/pt-br/library/system.web.mvc.htmlhelper(v=vs.118).aspx. Acesso em: 13 jan. 2021.

DYKSTRA, T.; ANDERSON, R. Getting started with entity Framework 6 code first using MVC5. Microsoft, 2014.

SANCHEZ, F.; ALTHMANN, M. F. **Desenvolvimento web com ASP.NET CORE MVC**. São Paulo: Casa do Código, 2013.

MICROSOFT. **Entity Framework Model First.** Disponível em: https://msdn.microsoft.com/en-us/library/jj205424(v=vs.113).aspx. Acesso em: 13 jan. 2021.

MICROSOFT. **Implementando a funcionalidade básica CRUD com o Entity Framework no aplicativo ASP.NET CORE MVC.** Disponível em: https://docs.microsoft.com/pt-br/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application. Acesso em: 13 jan. 2021.

MICROSOFT. **O ASP.NET CORE MVC exibições visão geral (C#)**. Disponível em: https://docs.microsoft.com/pt-br/aspnet/mvc/overview/older-versions-1/views/asp-net-mvc-views-overview-cs. Acesso em: 13 jan. 2021.

ROTH, D.; ANDERSON, R.; LUTTIN, S. **Introdução ao ASP.NET Core**. Disponível em: https://docs.microsoft.com/pt-br/aspnet/core/?view=aspnetcore-3.1. Acesso em: 13 jan. 2021.