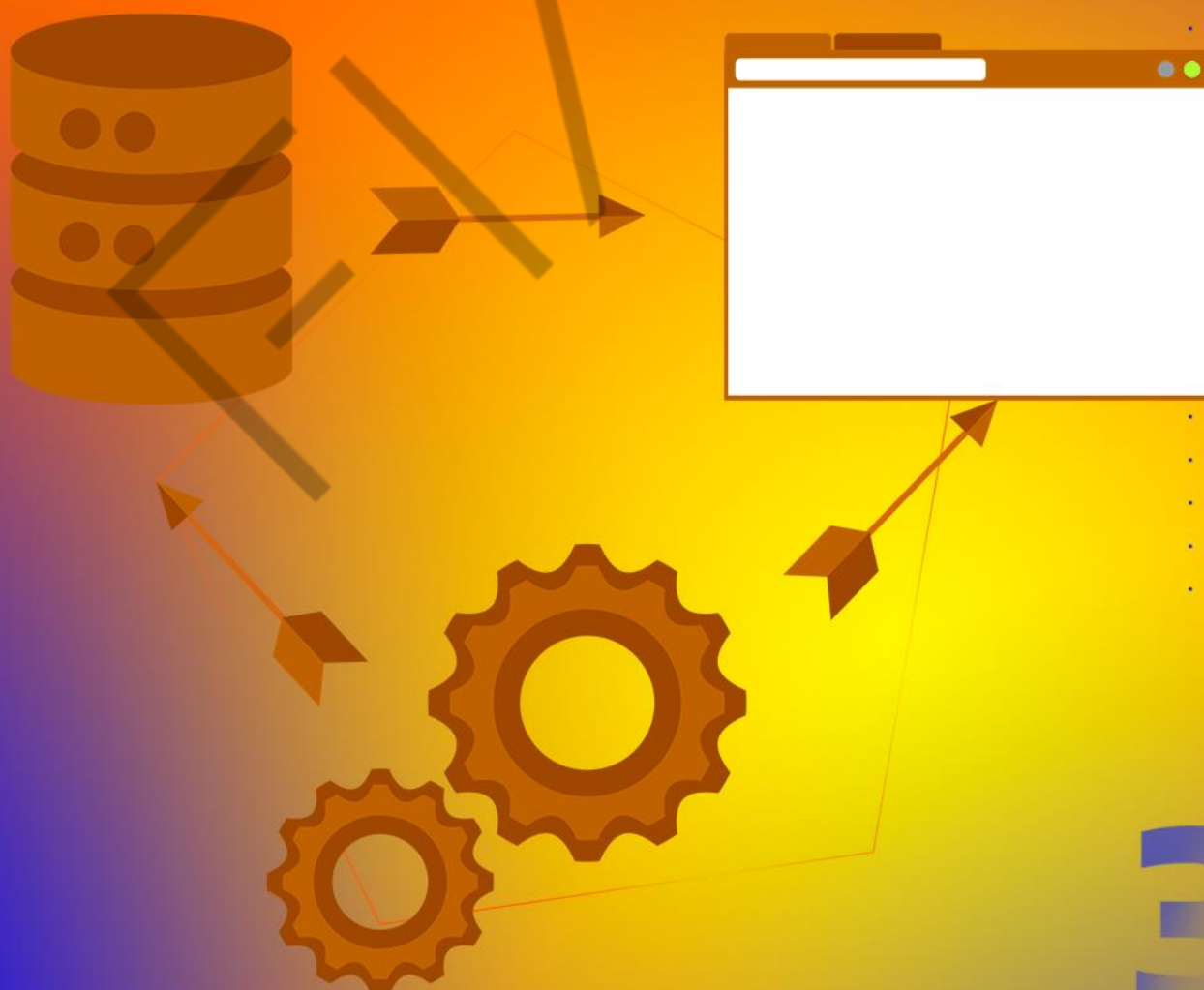


DESENVOLVIMENTO.NET

# ASP.NET MVC

FLAVIO MORENI



3

## LISTA DE FIGURAS

Figura 3.1 – Componentes MVC .....	7
Figura 3.2 – Projeto ASP.NET Core Web Application .....	10
Figura 3.3 – Template MVC .....	11
Figura 3.4 – Estrutura do projeto Asp.NET Core MVC 2 .....	11
Figura 3.5 – Diagrama de Classe – Produto e Categoria .....	12
Figura 3.6 – Tipos de retorno da classe <b>ActionResult</b> .....	16
Figura 3.7 – Adicionando <i>Controller</i> .....	17
Figura 3.8 – Selecionando o <i>Scaffold</i> do <i>Controller</i> .....	17
Figura 3.9 – Detalhes de um <i>Controller</i> .....	18
Figura 3.10 – Testando o <i>Controller</i> .....	19
Figura 3.11 – Criando uma <i>View</i> .....	20
Figura 3.12 – Estrutura da pasta <i>View</i> .....	20
Figura 3.13 – <i>View Index</i> apresentada para usuário .....	21
Figura 3.14 – Sobrecargas do método <i>View()</i> .....	22
Figura 3.15 – Configuração de Rotas .....	24
Figura 3.16 – Estrutura da <i>homepage</i> .....	25

**LISTA DE QUADROS**

Quadro 3.1– Tags Helpers .....	29
Quadro 3.2– <i>Actions</i> de cadastro do <i>Controller</i> TipoProduto .....	34
Quadro 3.3 – <i>Actions</i> de edição do <i>Controller</i> TipoProduto (1) .....	38
Quadro 3.4 – <i>Actions</i> de consulta do <i>Controller</i> TipoProduto .....	41
Quadro 3.5 – <i>Action</i> de excluir do <i>Controller</i> TipoProduto .....	43
Quadro 3.6– <i>Annotation</i> para validação de dados .....	56

EXEMPLO

## LISTA DE CÓDIGOS-FONTE

Código-fonte 3.1 – Modelo TipoProduto.....	13
Código-fonte 3.2 – Modelo TipoProduto.....	13
Código-fonte 3.3 – View Index para tipo de produto .....	21
Código-fonte 3.4 – Controller da homepage.....	25
Código-fonte 3.5 – Script para a criação da tabela Tipo Produto .....	26
Código-fonte 3.6 – Exemplo de bloco de código Razor .....	28
Código-fonte 3.7 – Criando a lista de tipos de produtos no Controller .....	31
Código-fonte 3.8 – Criando a lista de tipos de produtos na View.....	32
Código-fonte 3.9 – Actions de cadastro de tipo de produto.....	35
Código-fonte 3.10 – View de cadastro de tipo de produto.....	36
Código-fonte 3.11 – Actions de edição do tipo de produto .....	39
Código-fonte 3.12 – View de edição do tipo de produto .....	40
Código-fonte 3.13 – Action de consulta do tipo de produto .....	42
Código-fonte 3.14 – View de consulta do tipo de produto .....	42
Código-fonte 3.15 – Action de exclusão do tipo de produto .....	43
Código-fonte 3.16 – Importando CSS com @Url.Content().....	46
Código-fonte 3.17 – Importando JavaScript com @Url.Content() .....	46
Código-fonte 3.18 – View de Layout .....	48
Código-fonte 3.19 – Importando Layout na View .....	50
Código-fonte 3.20 – Validando dados pelo Controller.....	52
Código-fonte 3.21 – Mensagens de erro com asp-validation-summary .....	52
Código-fonte 3.22 – Validações com Data Annotations .....	54
Código-fonte 3.23 – Removendo a validação do Controller .....	55
Código-fonte 3.24 – Anotação para rótulos.....	57
Código-fonte 3.25 – Tag Razor para exibição dos rótulos .....	58
Código-fonte 3.26 – Gravando mensagens na TempData .....	61
Código-fonte 3.27 – Exibindo mensagens na TempData .....	62
Código-fonte 3.28 – Web.Config criando Oracle DataSource .....	66
Código-fonte 3.29 – Web.Config .....	67
Código-fonte 3.30 – ADO.NET ExecuteNonQuery() .....	69
Código-fonte 3.31 – ADO.NET ExecuteScalar().....	69
Código-fonte 3.32 – ADO.NET ExecuteQuery() e DataReader.....	70
Código-fonte 3.33 – Script para criação de tabela Tipo Produto .....	71
Código-fonte 3.34 – Estrutura dos métodos do TipoProdutoRepository .....	73
Código-fonte 3.35 – Código completo do TipoProdutoRepository.....	77
Código-fonte 3.36 – Código completo do TipoProdutoController .....	79
Código-fonte 3.37 – Exemplo de ActionFilters .....	81
Código-fonte 3.38 – Log ActionFilters .....	83
Código-fonte 3.39 – Action Index usando o Filtro de Log.....	83

## SUMÁRIO

3 ASP.NET MVC .....	6
3.1 Introdução .....	6
3.2 Padrão MVC .....	6
3.3 Criando projeto ASP.NET MVC .....	8
3.4 Modelos .....	12
3.5 Implementando Asp.net MVC .....	14
3.5.1 Funcionalidades .....	14
3.5.1.1 Controllers e Actions .....	14
3.5.1.2 Implementando Controllers .....	16
3.5.1.3 Associando uma <i>View</i> e <i>Controller</i> .....	19
3.5.1.4 Método de retorno – <i>View()</i> .....	22
3.5.2 Rotas e navegação .....	22
3.5.2.1 Convenções .....	22
3.5.2.2 Rotas da URL .....	23
3.5.2.3 Views .....	26
3.5.2.4 ASP.NET Razor .....	27
3.5.2.5 HTML <i>Helpers</i> .....	28
3.6 Listando dados na tela ( <i>View</i> ) .....	29
3.6.1 Inserindo dados ( <i>View</i> e <i>Controller</i> ) .....	33
3.6.2 Editando dados ( <i>View</i> e <i>Controller</i> ) .....	37
3.6.3 Consultando dados ( <i>View</i> e <i>Controller</i> ) .....	41
3.6.4 Removendo dados ( <i>View</i> e <i>Controller</i> ) .....	43
3.7 Layout pages e identidade visual .....	43
3.7.1 Instalando Bootstrap .....	44
3.7.2 Criando Layouts .....	45
3.7.3 Validações .....	50
3.7.3.1 Validação pelo Controller .....	51
3.7.3.2 Validação com <i>Data Annotations</i> .....	53
3.7.4 <i>Data Annotations</i> e as Views .....	57
3.7.5 Mensagens de sucesso com <i>TempData</i> .....	59
3.8 Acesso A banco de dados .....	63
3.8.1 ADO.NET .....	63
3.8.2 Configurando acesso .....	64
3.8.3 Componentes ADO.NET .....	67
3.8.4 Refatorando a aplicação .....	71
3.8.5 Implementando ADO.NET .....	72
3.9 Filtros .....	79
3.9.1 Atributos .....	80
3.9.2 Action Filters .....	80
3.9.3 Implementando Action Filters .....	81
3.10 Considerações Finais .....	84
REFERÊNCIAS .....	85

## 3 ASP.NET MVC

### 3.1 Introdução

Neste módulo, serão apresentados o conceito do padrão arquitetural MVC (Model-View-Controller) e o framework ASP.NET Core MVC 2, 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 2019 para implementar aplicações web.

Nos primeiros tópicos deste capítulo, apresentaremos os conceitos básicos do framework ASP.NET MVC, 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.

### 3.2 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:

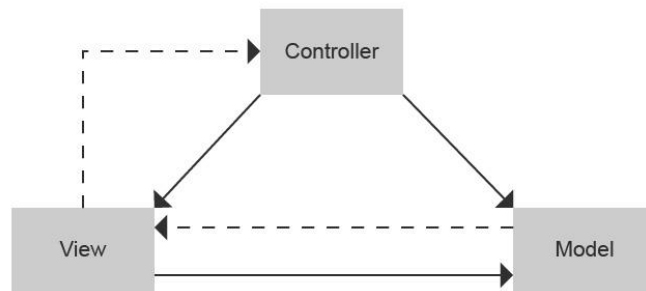


Figura 3.1 – Componentes MVC  
Fonte: Google Imagens (2018)

A lista a seguir traz com detalhes a responsabilidade 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 responsabilidade 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.

O framework .NET também oferece 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 2 do framework .NET, 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 e ASP.NET Core MVC .
- 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.

### 3.3 Criando projeto ASP.NET Core MVC 2

Para iniciar a criação de um novo aplicativo, precisamos entender o modelo de negócio que será implementado, quais são 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 da cidade **Fiap City**. Cidade “virtual” que pretende usar intensivamente a tecnologia para criar melhores condições de sustentabilidade para a população. O objeto deste



capítulo não é a criação completa do aplicativo e, sim, apresentar os conceitos que devem ser aplicados para a sua finalização.

O projeto de Internet da **Fiap City** consiste na divulgação de produtos para a pintura de imóveis que não acumulam resíduos, facilitando a limpeza e reduzindo o uso de água. O portal de Internet a ser construído terá como contexto as apresentações dos produtos (tintas), notícias, captação de moradores interessados no uso das tintas e investidores interessados em patrocinar a cidade.

No portal, teremos dois tipos de usuários/atores. O primeiro tipo são os administradores, com o papel de gerenciar a manutenção das informações de produtos e notícias e consultar moradores e investidores. O segundo tipo são os moradores ou empresas, que poderão consultar informações dos produtos e efetuar um cadastro como interessados em uso ou parceria. O foco deste material será a parte administrativa.

Vamos criar nosso projeto?

No Visual Studio 2019, selecione o menu **File > New > Project** (você pode usar a tecla de atalho Ctrl + Shift + N). Selecione a linguagem Visual C# > Web na parte esquerda da janela; no centro, vamos selecionar o tipo de projeto ASP.NET Core *Web Application* (.NET Framework). Na parte inferior, 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 **FiapSmartCity** como nome do projeto e da solução.

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

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

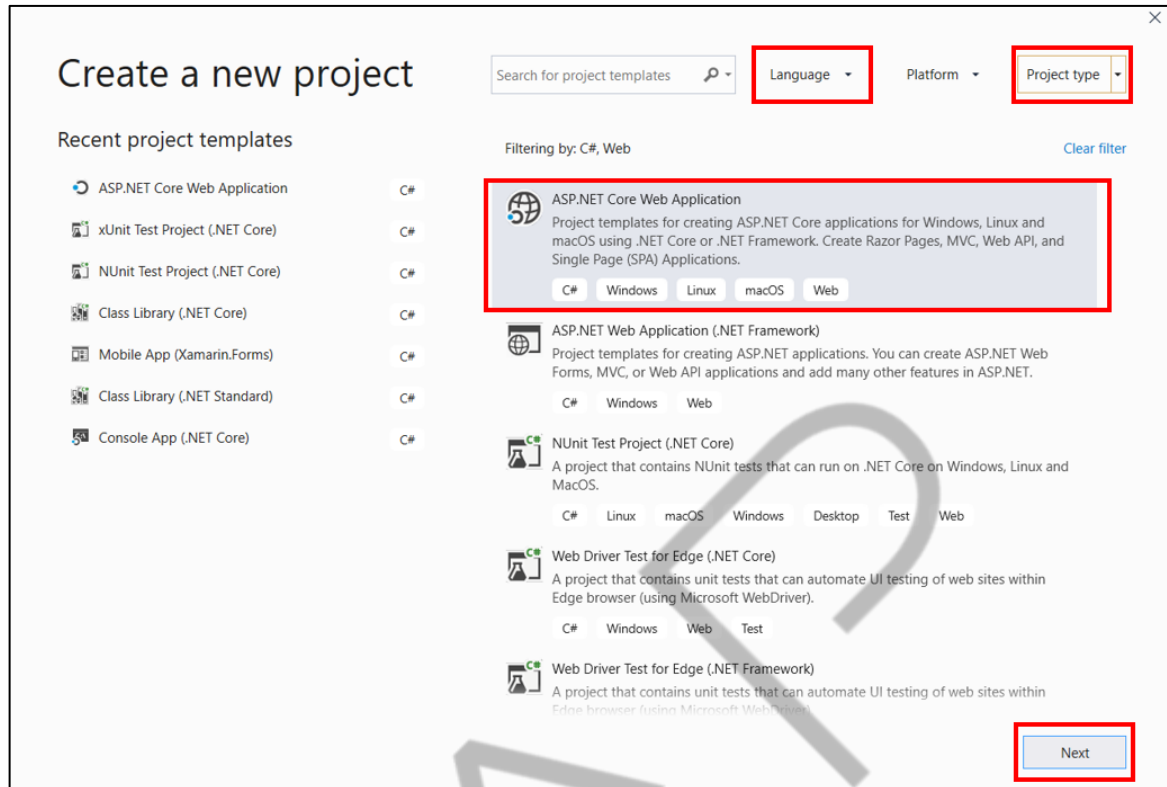


Figura 3.2 – Projeto ASP.NET Core Web Application  
Fonte: Elaborado pelo autor (2020)

Na próxima janela, devemos escolher a versão do framework e o template para nossa aplicação web. Selecione a versão do framework ASP.NET Core 2.2 e a opção *Web Application (Model – View – Controller)*. A Figura Template MVC abaixo apresenta todas as opções que devem ser selecionadas para a criação correta do projeto:

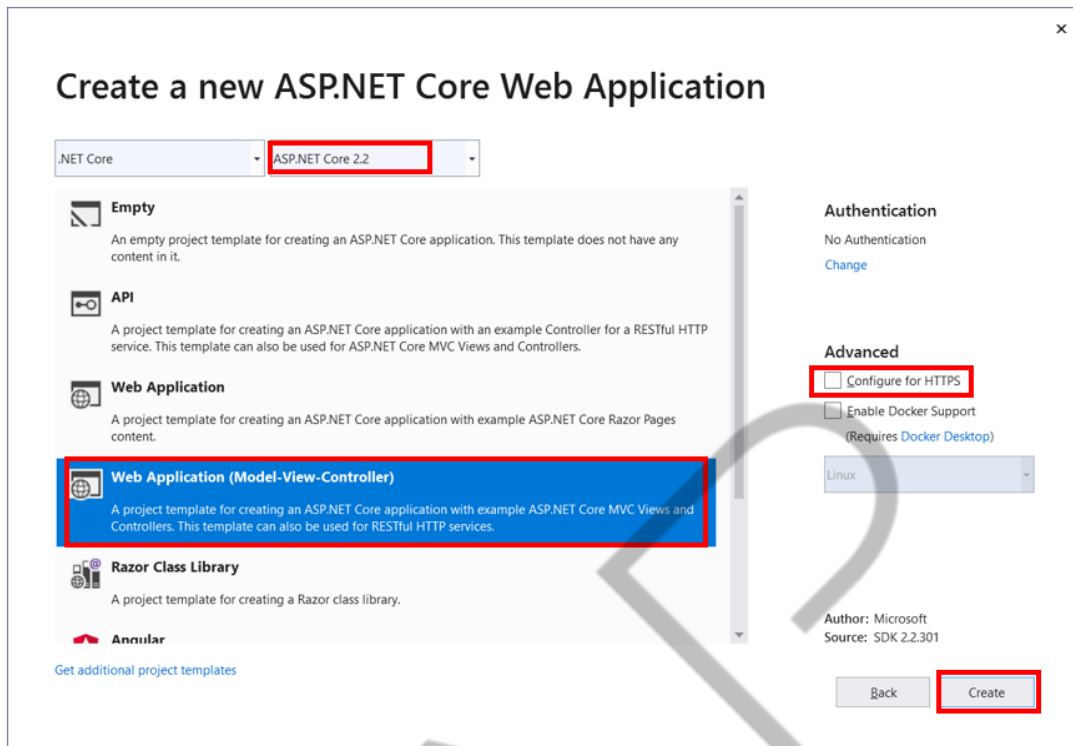


Figura 3.3 – Template MVC  
Fonte: Elaborado pelo autor (2020)

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 MVC 2:

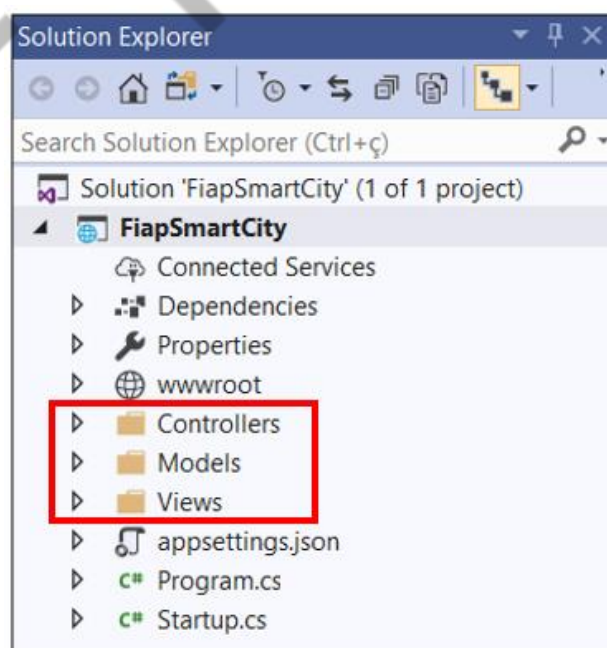


Figura 3.4 – Estrutura do projeto Asp.NET Core MVC 2  
Fonte: Elaborado pelo autor (2020)

### 3.4 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 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 os tipos de produto. Inicialmente, teremos apenas 1 (um) tipo de produto (tinta), mas, no futuro, podemos diversificar para outros itens das cidades inteligentes, como: filtros de água, captadores de energia solar etc. Veja a seguir a representação UML para nossas classes de modelo:

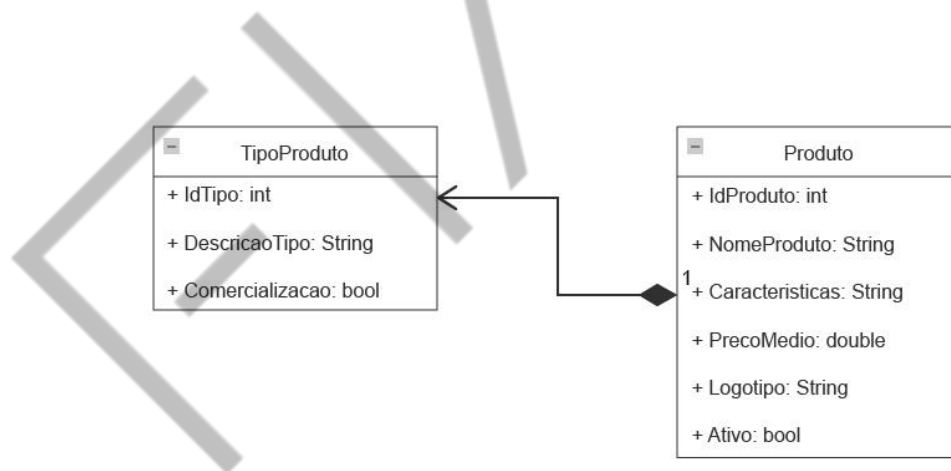


Figura 3.5 – Diagrama de Classe – Produto e Categoria  
Fonte: Elaborado pelo autor (2018)

Os componentes da camada de modelo são simples classes C#, que devem ser adicionadas no *namespace* **Models** do projeto. Para criar o modelo **TipoProduto**, clique com o botão direito na pasta Models e escolha a opção **Add > Class**. Defina o nome como **TipoProduto.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 `TipoProduto`, a seguir, apresenta a implementação do modelo `TipoProduto`.

```
using System;

namespace FiapSmartCity.Models
{
    public class TipoProduto
    {
        public int IdTipo { get; set; }
        public String DescricaoTipo { get; set; }
        public bool Comercializado { get; set; }
    }
}
```

Código-fonte 3.1 – Modelo `TipoProduto`  
Fonte: Elaborado pelo autor (2018)

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

Seguindo os passos anteriores e o diagrama da Figura Diagrama de Classe – Produto e Categoria, vamos criar a classe para o modelo de Produto. O diagrama apresenta uma agregação entre Produto e `TipoProduto`, sendo, assim, na classe de Produto, precisamos ter uma propriedade `TipoProduto`. Veja o exemplo:

```
using System;

namespace FiapSmartCity.Models
{
    public class Produto
    {
        public int IdProduto { get; set; }
        public String NomeProduto { get; set; }
        public String Caracteristicas { get; set; }
        public double PrecoMedio { get; set; }
        public String Logotipo { get; set; }
        public bool Ativo { get; set; }

        // Referência para classe TipoProduto
        public TipoProduto Tipo { get; set; }
    }
}
```

Código-fonte 3.2 – Modelo `TipoProduto`  
Fonte: Elaborado pelo autor (2018)

## 3.5 Implementando ASP.NET Core MVC 2

1.

### 3.5.1 Funcionalidades

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

- Criação de um novo tipo.
- Remoção de um tipo existente.
- Alteração da descrição ou de comercialização.
- Listagem dos tipos já existentes no sistema.

1.

2.

2.1.

#### 3.5.1.1 Controllers e Actions

Em um projeto ASP.NET Core MVC 2, toda solicitação do usuário feita pelo navegador será recebida e gerenciada por um *Controller*, ficando este 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: *CriarTipoProduto*, *ExcluirTipoProduto*, *AlterarTipoProduto* e *ListaTipos*), 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 nosso código. Com o

uso das **Actions**, devemos criar um controlador para cada domínio e ações para cada funcionalidade (por exemplo: *Controller* TipoProduto, *Actions* Criar, Excluir, Alterar e Listar).

Todo *Controller* necessita de uma *Action*, 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** no nome (por exemplo: TipoProdutoControlller).
- 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*.
- 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, ações que serão responsáveis por retornar um arquivo para download. Abaixo, 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 retornar um null o resultado (void).

Figura 3.6 – Tipos de retorno da classe **ActionResult**  
 Fonte: Microsoft MSDN (2018)

### 3.5.1.2 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*.



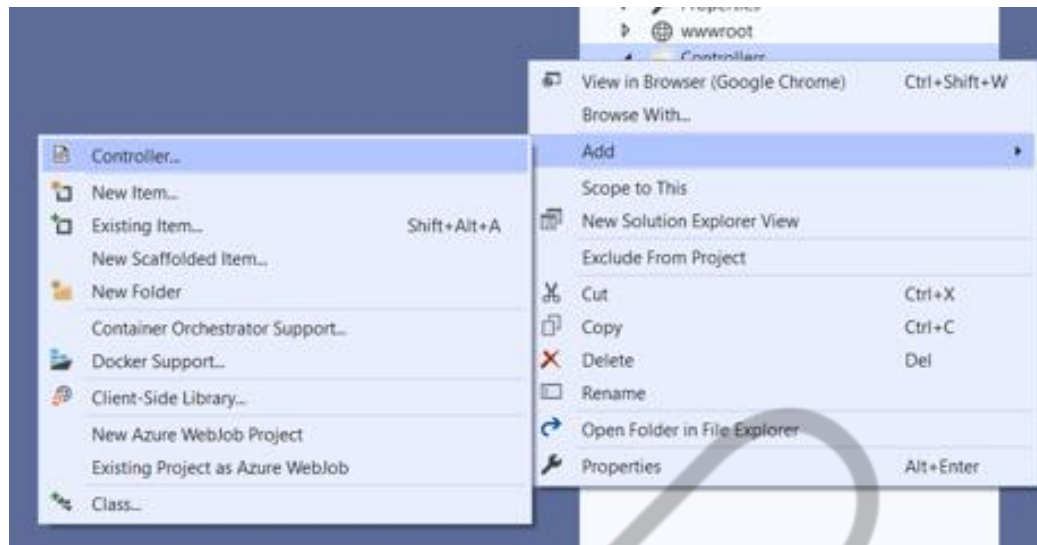


Figura 3.7 – Adicionando *Controller*  
Fonte: Elaborado pelo autor (2020)

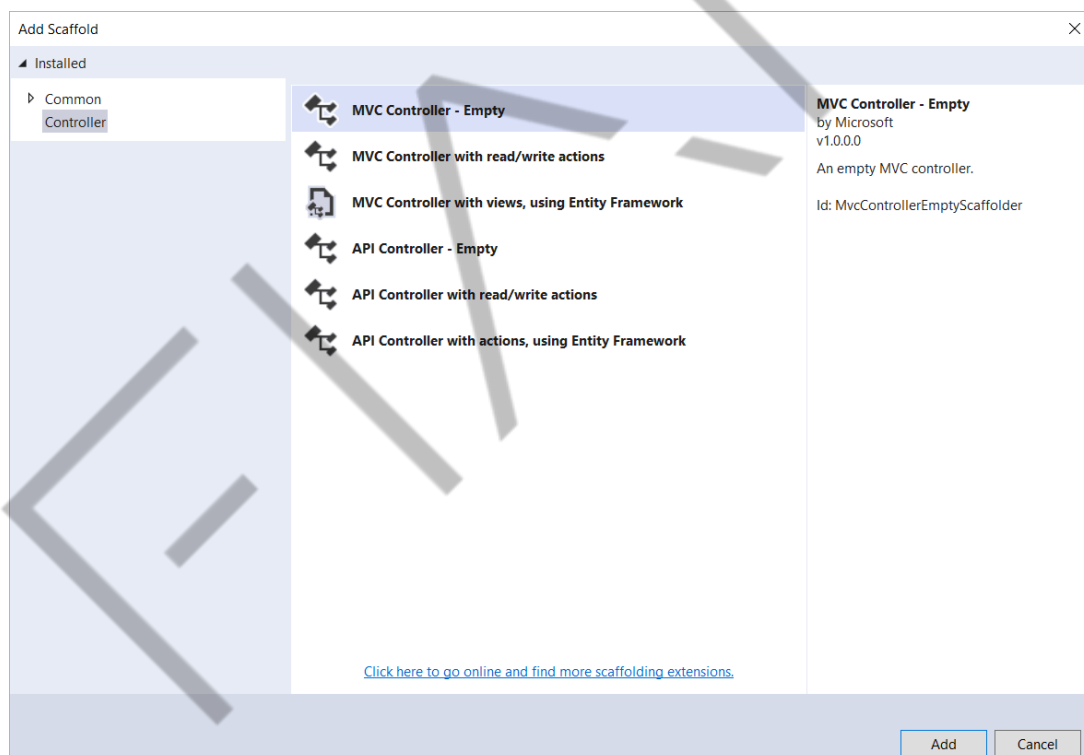


Figura 3.8 – Selecionando o *Scaffold* do *Controller*  
Fonte: Elaborado pelo autor (2020)

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

Pronto! Primeiro controlador criado no projeto. 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 **ActionResult**. A Figura Detalhes de um *Controller* traz todos os detalhes do *Controller*.

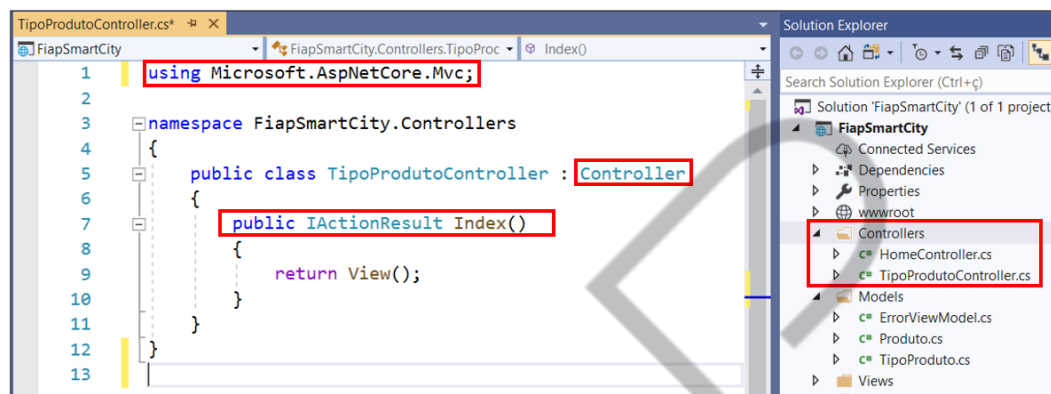


Figura 3.9 – Detalhes de um *Controller*  
Fonte: Elaborado pelo autor (2020)

*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 **/TipoProduto** e pressione enter. O navegador irá 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*.

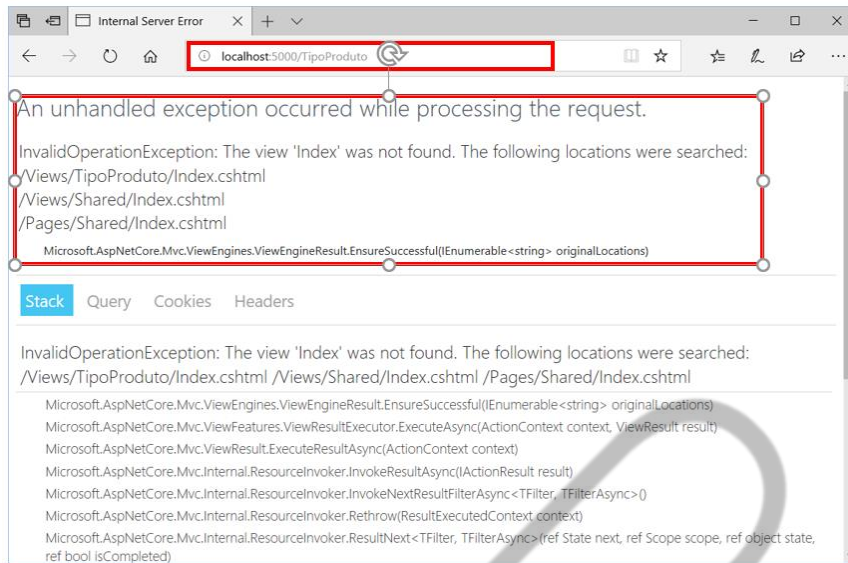


Figura 3.10 – Testando o *Controller*  
Fonte: Elaborado pelo autor (2020)

**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 6588. Na execução em outro computador, podemos ter um novo número de porta. Atente-se para o número da porta gerada na execução do projeto.

### 3.5.1.3 Associando uma *View* e *Controller*

Anteriormente, criamos e testamos nosso *Controller*, porém a validação da execução foi feita por meio da tela de erro, informando que não existe uma *View* para ser exibida. Então, vamos 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* **TipoProdutoController** aberto na janela de edição, clique com o botão direito sobre o nome da *Action* Index e selecione a opção “**Add View**” (uma janela com detalhes da *view* será apresentada). Mantenha o nome de “**Index**” e o *template* como “**Empty**”. No rodapé da janela, remova a opção “*Use a layout page*” e clique no botão **Add**. A Figura Criando uma *View* abaixo mostra as opções para a criação da *view*:

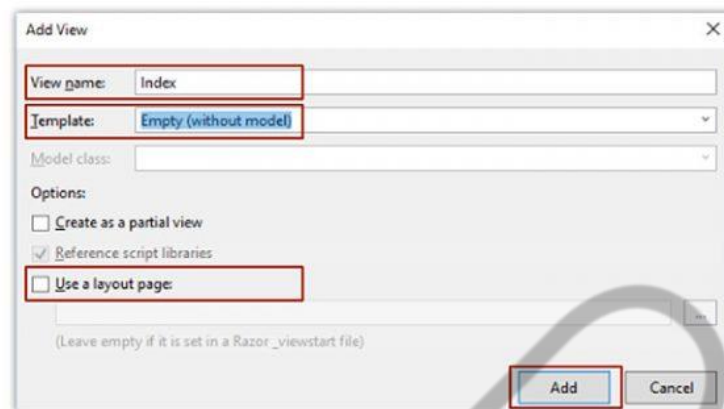


Figura 3.11 – Criando uma View  
Fonte: Elaborado pelo autor (2018)

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

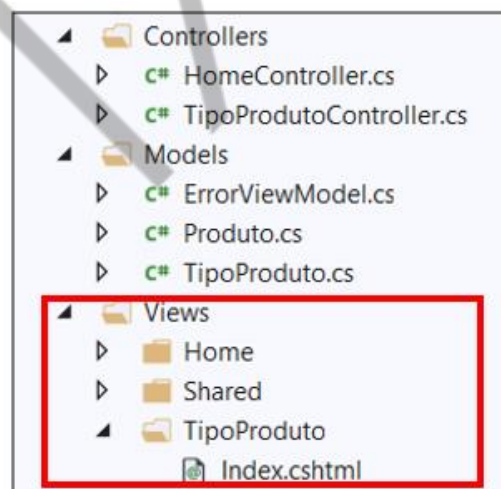


Figura 3.12 – Estrutura da pasta View  
Fonte: Elaborado pelo autor (2020)

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 abaixo:

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

Código-fonte 3.3 – View Index para tipo de produto  
Fonte: Elaborado pelo autor (2018)

Arquivo editado, voltamos a testar nosso *Controller*. Pressione F5, aguarde o navegador ser carregado, informe o caminho `/TipoProduto/Index` e tecele 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 3.13 – View Index apresentada para usuário  
Fonte: Elaborado pelo autor (2018)

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

### 3.5.1.4 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 informar 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
	<i>View()</i>	Cria um <i>ViewResult</i> objeto que processa um modo de exibição para a resposta.
	<i>View(Object)</i>	Cria um <i>ViewResult</i> o objeto usando o modelo que processa um modo de exibição para a resposta.
	<i>View(String)</i>	Cria um <i>ViewResult</i> o objeto usando o nome de exibição que processa um modo de exibição.
	<i>View(IView)</i>	Cria um <i>ViewResult</i> que processa especificado do objeto <i>IView</i> objeto.
	<i>View(String, Object)</i>	Cria um <i>ViewResult</i> o objeto usando o nome de exibição e o modelo que processa um modo de exibição para a resposta.
	<i>View(String, String)</i>	Cria um <i>ViewResult</i> objeto usando o nome e o nome da página mestra que processa um modo de exibição para a resposta.
	<i>View(IView, Object)</i>	Cria um <i>ViewResult</i> que processa especificado do objeto <i>IView</i> objeto.
	<i>View(String, String, Object)</i>	Cria um <i>ViewResult</i> objeto usando o nome de exibição, o nome da página mestra e o modelo que processa um modo de exibição.

Figura 3.14 – Sobrecargas do método *View()*  
Fonte: Microsoft MSDN (2018)

## 3.5.2 Rotas e navegação

### 3.5.2.1 Convenções

O framework ASP.NET Core MVC 2 usa uma simples convenção para associar *Actions* dos *Controllers* às *Views*. Para o nosso exemplo do *Controller*

TipoProdutoController, foi criada uma subpasta com o nome “TipoProduto” 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 com o *framework*.

Com isso, ficam claras a facilidade e a simplicidade de se seguir as recomendações de uso das nomenclaturas e estruturas criadas e sugeridas pelo ASP.NET Core MVC 2.

### 3.5.2.2 Rotas da URL

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

Analisamos a URL da aplicação <<http://localhost:6588/TipoProduto/Index>>, o primeiro bloco apresenta o protocolo, nome do servidor e a porta de comunicação; o segundo bloco representa:

TipoProduto – *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 2 possui uma classe C# responsável por essa configuração. Na janela da *Solution Explorer*, navegue até a classe “**Startup.cs**” e abra o código do método “**Configure**”. A Figura “Configuração de Rotas”, a seguir, exhibe o conteúdo da classe **Startup.cs**:



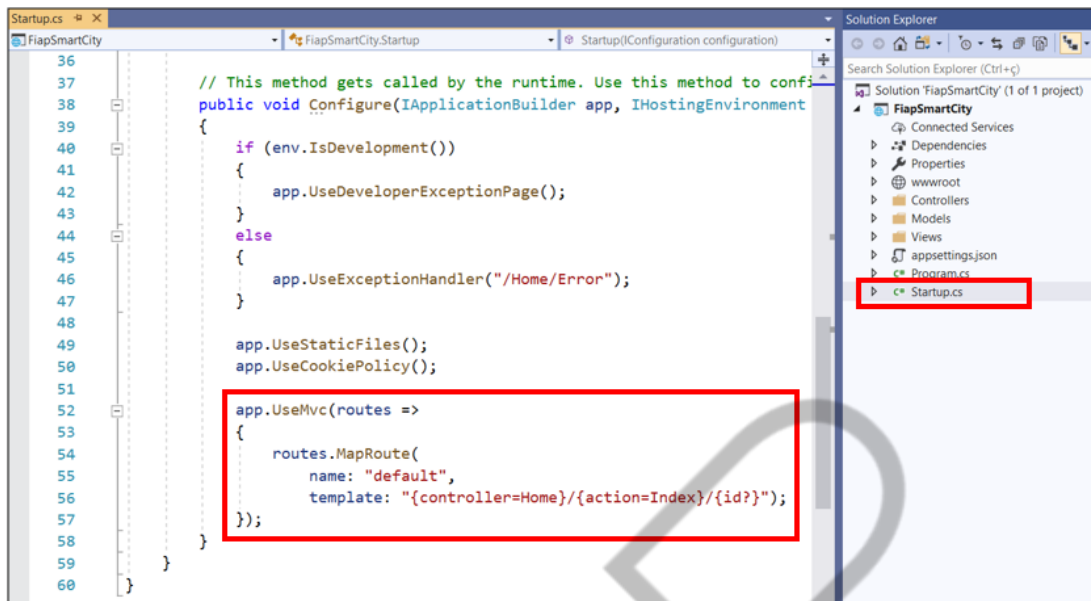


Figura 3.15 – Configuração de Rotas  
Fonte: Elaborado pelo autor (2020)

O bloco de código do método **Configure** é o responsável por interceptar todas as chamadas do aplicativo, analisar o caminho da URL requisitada e mapear para o *Controller* e a *Action* correspondentes. Verifique o código da implementação **routes.MapRoute()** 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 **MapRoute**, temos uma definição “**default**”, que define quais *Controller* e *Action* deverão ser executados; caso nenhuma informação seja informada na url, o padrão é o *Controller* chamado Home e a *Action* Index.

Podemos alterar o controlador-padrão de **Home** para **TipoProduto** e executar o aplicativo novamente, assim será possível acessar nossa funcionalidade usando apenas a url <<http://localhost:6588/>>.

Embora essas configurações possam ser alteradas, é recomendado manter o padrão. Assim, vamos manter o *Controller* **Home** como padrão.

Para não deixar nossa aplicação sem uma apresentação inicial, vamos executar os passos dos capítulos anteriores e criar um novo *Controller* (**HomeController**) e uma *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:

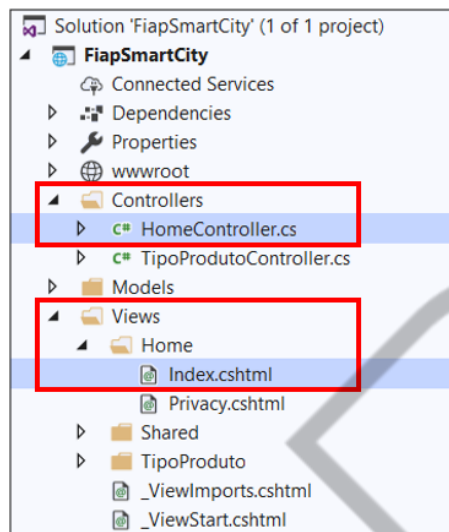


Figura 3.16 – Estrutura da *homepage*  
Fonte: Elaborado pelo autor (2020)

```
using FiapSmartCity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;

namespace FiapSmartCity.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

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

        [ResponseCache(Duration = 0, Location =
ResponseCacheLocation.None, NoStore = true)]
        public ActionResult Error()
        {
            return View(new ErrorViewModel { RequestId =
Activity.Current?.Id ?? HttpContext.TraceIdentifier });
        }
    }
}
```

Código-fonte 3.4 – *Controller da homepage*  
Fonte: Elaborado pelo autor (2018)

```
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        <h1>Nossa home-page.</h1>
    </div>
</body>
</html>
```

Código-fonte 3.5 – Script para a criação da tabela Tipo Produto  
Fonte: Elaborado pelo autor (2018)

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

- <http://localhost:6588/>
- <http://localhost:6588/Home>
- <http://localhost:6588/Home/Index>

**IMPORTANTE:** Verifique a porta que está sendo usada pela aplicação. Se for diferente de **6588**, altere os endereços acima adicionando o número da nova porta.

### 3.5.2.3 Views

Até este ponto do nosso curso, vimos que as *Views* no framework ASP.NET Core MVC 2 são arquivos .cshtml com base em HTML e que, por convenção, são salvas na pasta *Views* e na subpasta com o nome do *Controller* associado.

Apenas com o uso 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 2 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*.

### 3.5.2.4 ASP.NET Razor

O Razor é um dos mecanismos do ASP.NET Core MVC 2 responsáveis por construir nossas *Views* dinâmicas; antes de seu lançamento, o mecanismo-padrão era o ASPX, que usava como base *scriptlets* ASP.NET puro. 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 @. Abaixo, o Código-fonte 3.6 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 3.6 – Exemplo de bloco de código Razor  
Fonte: Elaborado pelo autor (2018)

### 3.5.2.5 Tags *Helpers*

O framework ASP.NET Core MVC 2 disponibiliza os componentes auxiliares para o desenvolvimento dos componentes *Views*.

Os Auxiliares 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 auxiliares para todos os elementos HTML comuns (por exemplo: formulários, links, imagens, botões e outros).

Veja abaixo o Quadro “Tags Helpers”, com os Tag Helpers disponíveis no ASP.NET Core MVC 2:

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 3.1– Tags Helpers  
Fonte: Elaborado pelo autor (2018)

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

```

<!-- Código RAZOR -->
@Html.TextBoxFor(m => m.DescricaoTipo)

<!-- Código HTML (renderizado) -->
<input id="DescricaoTipo" name="DescricaoTipo" type="text" value="" />

```

Figura 3.17 – Exemplo de uso de TagHelpers  
Fonte: Elaborado pelo autor (2018)

### 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* TipoProduto 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 **FiapSmartCity**. Para cada informação listada, será necessário criar uma ação que será implementada posteriormente para consultar, editar, excluir, e uma opção para criar um novo tipo. Vamos usar a **Action** e a **View** já criadas e adicionar nosso código.

Na **Action Index** do TipoProdutoController, vamos criar um atributo do tipo lista e adicionar três objetos do modelo TipoProduto. Nesse momento, vamos simular que temos os seguintes produtos: Tinta, Filtro de água e Captador de energia. No método de retorno, vamos passar como parâmetro o atributo lista. O Código-fonte Criando a lista de tipos de produtos no *Controller*, a seguir, mostra a criação do atributo lista e o retorno do método **View()**:

```
using FiapSmartCity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace FiapSmartCity.Controllers
{
    public class TipoProdutoController : Controller
    {
        public IActionResult Index()
        {
            // Criando o atributo da lista
            IList<Models.TipoProduto> listaTipo = new
            List<Models.TipoProduto>();

            // Adicionando na lista o TipoProduto da Tinta
            listaTipo.Add(new TipoProduto()
            {
                IdTipo = 1,
                DescricaoTipo = "Tinta",
                Comercializado = true
            });

            listaTipo.Add(new TipoProduto()
            {
```

```
        IdTipo = 2,
        DescricaoTipo = "Filtro de água",
        Comercializado = true
    });

    listaTipo.Add(new TipoProduto()
    {
        IdTipo = 3,
        DescricaoTipo = "Captador de energia",
        Comercializado = false
    });

    // Retornando para View a lista de Tipos
    return View(listaTipo);
}
}
```

Código-fonte 3.7 – Criando a lista de tipos de produtos no *Controller*  
Fonte: Elaborado pelo autor (2020)

Com a lista de tipos de produtos 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 apresenta 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 tipos de produtos na *View* mostra o resultado do nosso componente *View*, com alguns comentários explicativos do uso dos métodos *HtmlHelper*, veja:

```
@model IEnumerable<FiapSmartCity.Models.TipoProduto>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Tipo de Produto</title>
```

```
</head>
<body>
  <h1>Tipo de Produto</h1>
  <p>
    <!-- uso de TagHelpers para definir o Controller e a
    Action -->
    <a asp-controller="TipoProduto" asp-
    action="Cadastrar">Novo Tipo</a>
  </p>
  <table class="table" border="1">
    <tr>
      <th>Id</th>
      <th>Descrição</th>
      <th></th>
    </tr>

    @foreach (var item in Model)
    {
      <tr>
        <td>
          <label>@item.IdTipo</label>
        </td>
        <td>
          <label>@item.DescricaoTipo</label>
        </td>
        <td>
          <!-- asp-route-id é usado para informar o
          Id do Item selecionado. -->
          <a asp-controller="TipoProduto"
            asp-action="Editar"
            asp-route-id="@item.IdTipo">Editar</a>

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

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

Código-fonte 3.8 – Criando a lista de tipos de produtos na View  
Fonte: Elaborado pelo autor (2020)



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



Figura 3.18 – Resultado da tela de Tipo de produtos  
Fonte: Elaborado pelo autor (2018)

**IMPORTANTE:** Ao clicar em qualquer link da tela de tipo de produtos, 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 criar os elementos do framework MVC que permitem ao usuário preencher os dados de tipo de produto e simular a gravação na base de dados.

Ainda no mesmo *Controller*, vamos adicionar dois novos métodos (*Actions*). Os dois métodos vão receber o nome “**Cadastrar**”. Pode parecer estranho, mas vamos adotar 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* irá aceitar em execução. A segunda forma é por meio de um parâmetro, um dos métodos receberá como *model* TipoProduto. O Quadro *Actions* de cadastro do *Controller* TipoProduto, detalha os dois métodos a ser 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 <i>Controller</i> deverá passar para a <i>View</i> uma instância do objeto <i>model</i> 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 3.2– *Actions* de cadastro do *Controller* TipoProduto  
Fonte: Elaborado pelo autor (2018)

Para usar as anotações que indicam qual o 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 **Debug.Print()** do *namespace* **System.Diagnostics**.

Veja no Código-fonte *Actions* de cadastro de tipo de produto, como ficou a implementação das *Actions* Cadastrar:

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

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

// Anotação de uso do Verb HTTP Post
[HttpPost]
public IActionResult Cadastrar(Models.TipoProduto
```

```
tipoProduto)
{
    // Imprime os valores do modelo
    System.Diagnostics.Debug.Print("Descrição: " +
    tipoProduto.DescricaoTipo);
    System.Diagnostics.Debug.Print("Comercializado: " +
    tipoProduto.Comercializado);

    // Simila que os dados foram gravados.
    System.Diagnostics.Debug.Print("Gravando o Tipo de
    Produto");

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

Código-fonte 3.9 – *Actions* de cadastro de tipo de produto  
Fonte: Elaborado pelo autor (2020)

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

Seguindo as convenções do framework, a *View* terá o mesmo nome da *Action* “**Cadastrar**”. Deverá fazer uso dos *tag helpers* **asp-controller** e **asp-action** para a criação do formulário, além dos elementos HTML puros para posicionamento e formatação da tela.

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

Veja o resultado da *View* Cadastrar no Código-fonte *View* de cadastro de tipo de produto:

```
@model FiapSmartCity.Models.TipoProduto

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Tipo de Produto - Cadastrar</title>
</head>
```

```
<body>
  <h1>Tipo de Produto - Cadastrar</h1>

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

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

</body>
</html>
```

Código-fonte 3.10 – View de cadastro de tipo de produto  
Fonte: Elaborado pelo autor (2020)

Podemos usar duas estratégias para validar na implementação. Uma delas é adicionando *breakpoints* nos trechos de código do *Controller* e, com a tela F10, percorrer linha a linha para acompanhar a execução. E a outra forma é observar pela janela Output do Visual Studio as mensagens que são impressas pelo comando `System.Diagnostics.Debug.Print()`. Veja na Figura Janela *Output* do Visual Studio a janela Output e algumas mensagens da execução do fluxo de cadastrar:

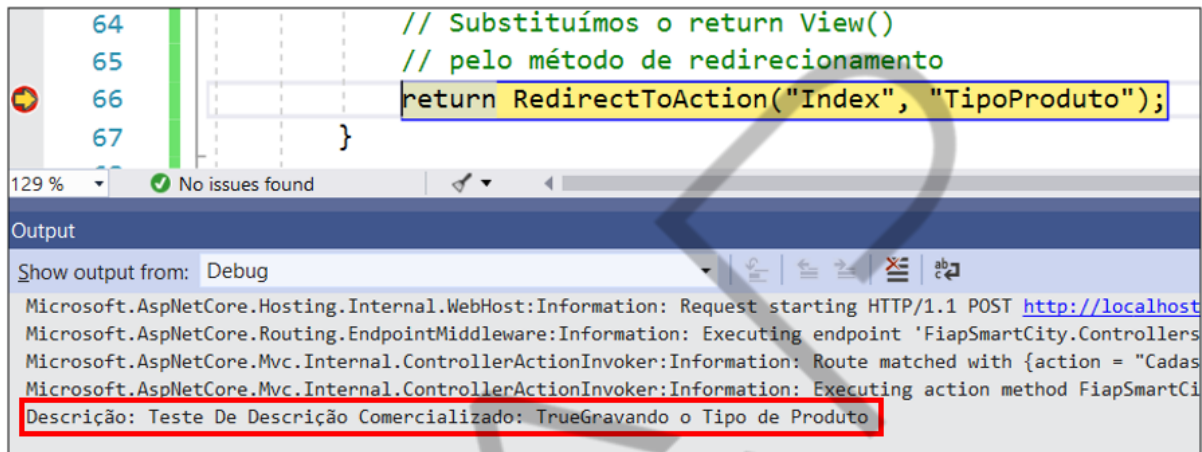


Figura 3.19 – Janela *Output* do Visual Studio  
Fonte: Elaborado pelo autor (2020)

Execute a aplicação e acesse a lista de tipos. No link “Novo Tipo”, simule um cadastro de tipo, use *breakpoints* ou a janela **Output** 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 tipo de produto.

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 <i>Controller</i> deverá passar para a <i>View</i> uma instância do objeto <i>model</i> 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 3.3 – *Actions* de edição do *Controller* TipoProduto (1)  
Fonte: Elaborado pelo autor (2018)

Veja a implementação no Código-fonte *Actions* de edição do tipo de produto:

```
[HttpGet]
public IActionResult Editar(int Id)
{
    // Imprime a mensagem de execução
    System.Diagnostics.Debug.Print("Consultando o Tipo com
    Id = " + Id);

    // Cria o modelo que SIMULA a consulta no banco de
    dados
    TipoProduto tipoProduto = new TipoProduto()
    {
        IdTipo = Id,
        DescricaoTipo = "Tinta",
        Comercializado = true
    };

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

[HttpPost]
public IActionResult Editar(models.TipoProduto tipoProduto)
{
    // Imprime os valores do modelo
    System.Diagnostics.Debug.Print("Descrição: " +
    tipoProduto.DescricaoTipo);
    System.Diagnostics.Debug.Print("Comercializado: " +
    tipoProduto.Comercializado);

    // Simula que os dados foram gravados.
    System.Diagnostics.Debug.Print("Gravando o Tipo
    Editado");
}
```

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

Código-fonte 3.11 – *Actions* de edição do tipo de produto  
Fonte: Elaborado pelo autor (2020)

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 irá armazenar o Id do tipo.

É preciso armazenar o Id do tipo de produto, 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 tipo de produto os detalhes da *View* **Editar**:

```
@model FiapSmartCity.Models.TipoProduto

@{
    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="TipoProduto"
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="TipoProduto" asp-
action="Index">Voltar</a>
</div>

</body>
</html>
```

Código-fonte 3.12 – View de edição do tipo de produto  
Fonte: Elaborado pelo autor (2020)

Execute a aplicação e acesse a lista de tipos. No link “Novo Tipo”, simule um cadastro de tipo, use *breakpoints* ou a janela Output para acompanhar os dados digitados. Lembre-se, como estamos usando trechos de código para simulação, 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	<p>Objetivo: abrir um formulário com os dados do tipo selecionado na lista.</p> <p>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.</p> <p>O <i>Controller</i> deverá passar para a <i>View</i> uma instância do objeto <i>model</i> preenchido com o resultado da pesquisa no banco de dados.</p>

Quadro 3.4 – *Actions* de consulta do *Controller* TipoProduto  
Fonte: Elaborado pelo autor (2018)

Veja a implementação no Código-fonte *Action* de consulta do tipo de produto abaixo:

```
[HttpGet]
public IActionResult Consultar(int Id)
{
    // Imprime a mensagem de execução
    System.Diagnostics.Debug.Print("Consultando o Tipo com
Id = " + Id);
    // Cria o modelo que SIMULA a consulta no banco de
dados
    TipoProduto tipoProduto = new TipoProduto()
    {
        IdTipo = Id,
        DescricaoTipo = "Tinta",
        Comercializado = true
    };
    // Retorna para a View o objeto modelo
    // com as propriedades preenchidas com dados do banco de
dados
    return View(tipoProduto);
}
```

Código-fonte 3.13 – *Action* de consulta do tipo de produto  
Fonte: Elaborado pelo autor (2020)

Criando a *View* **Consultar**, reaproveitando 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 FiapSmartCity.Models.TipoProduto

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Tipo de Produto - Consultar</title>
</head>
<body>
    <h1>Tipo de Produto - Consultar</h1>
    <div class="form-horizontal">
        <hr />

        <div class="form-group">
            <label><b>Descrição:</b></label>
            <div class="col-md-10">
                <span>@Model.DescricaoTipo</span>
            </div>
        </div>

        <div class="form-group">
            <label><b>Comercializado:</b></label>
            <div class="checkbox">
                <span>@Model.Comercializado</span>
            </div>
        </div>
        <hr />
    </div>
    <div>
        <a asp-controller="TipoProduto" asp-
action="Index">Voltar</a>
    </div>
</body>
</html>
```

Código-fonte 3.14 – *View* de consulta do tipo de produto  
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 utilizaremos *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 3.5 – *Action* de excluir do *Controller* TipoProduto  
Fonte: Elaborado pelo autor (2018)

Veja a implementação no Código-fonte *Action* de exclusão do tipo de produto abaixo:

```
[HttpGet]
public IActionResult Excluir(int Id)
{
    // Imprime a mensagem de execução
    System.Diagnostics.Debug.Print("Excluir o Tipo com Id = " + Id);

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

Código-fonte 3.15 – *Action* de exclusão do tipo de produto  
Fonte: Elaborado pelo autor (2020)

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

## 3.7 Layout pages e identidade visual

Nos capítulos anteriores, criamos nosso aplicativo com o objetivo de testar o 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 2 para facilitar a

evolução do aplicativo. Para isso, vamos usar a biblioteca **Bootstrap**, pois, além de ser uma biblioteca bem difundida, foi utilizada em módulos anteriores.

### 3.7.1 Instalando Bootstrap

A primeira alteração em nosso projeto neste capítulo será a instalação da biblioteca **Bootstrap**, utilizando a ferramenta **Nuget** disponível no Visual Studio.

O **Nuget** é um gerenciador de pacotes da tecnologia .NET com o qual possível usar pacotes de bibliotecas externas ou construir bibliotecas para ser 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, assim 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 estrutura do projeto na Figura Estrutura do projeto com Bootstrap abaixo:

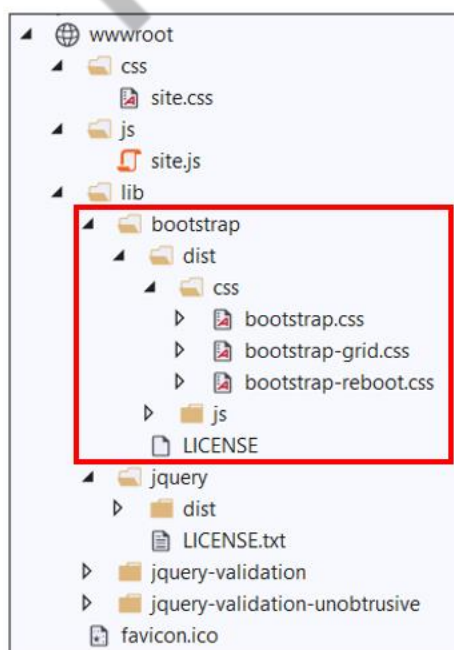


Figura 3.21 – Estrutura do projeto com Bootstrap  
Fonte: Elaborado pelo autor (2020)

### 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, vamos centralizar 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 por 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**, limpar 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:

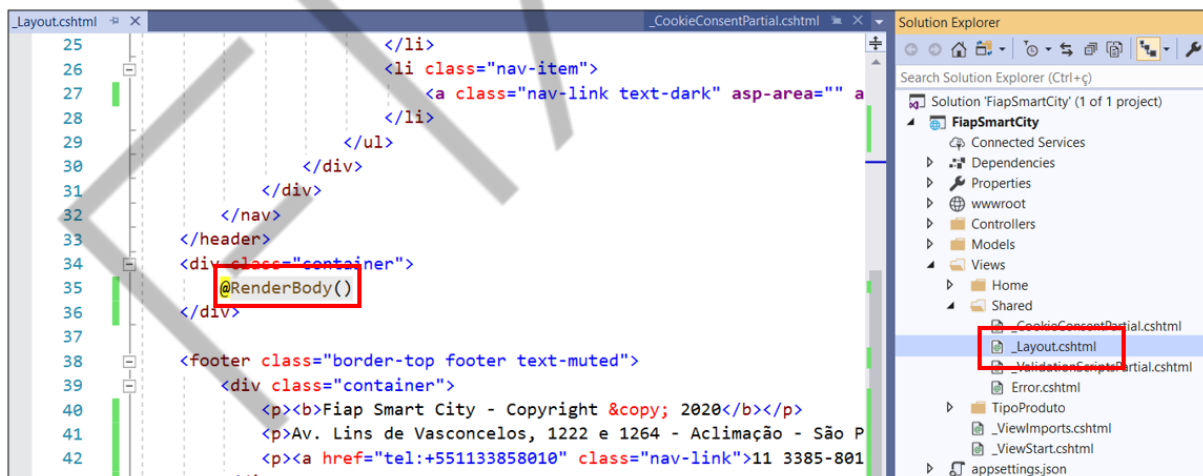


Figura 3.22 – Arquivo de Layout  
Fonte: Elaborado pelo autor (2020)

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

Iremos falar sobre as tags `@ViewBag` e `@RenderBody` logo mais, antes precisamos inserir as tags HTML para o uso do Bootstrap. No corpo da tag `<head>`,

é necessário incluir a tag <link> com referência ao arquivo .css do Bootstrap. É possível fazer isso usando a tag HTML pura, mas, para explorar os recursos do framework, vamos usar o recurso do símbolo “~”, que permite que você transforme caminhos de arquivos relativos para caminhos semiabsolutos, ou seja, não importa o endereço em que sua 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>FiapSmartCity</title>

  <!-- importando bootstrap e o css do nosso site-->
  <link rel="stylesheet"
href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</head>
```

Código-fonte 3.16 – Importando CSS com @Url.Content()

Fonte: Elaborado pelo autor (2020)

Finalizada a importação do arquivo de estilo, é preciso importar os arquivos de *script*, para isso é adotada a composição da tag HTML <script> e novamente a o recurso do símbolo “~”. Abaixo é apresentada a nova versão do bloco de importação das bibliotecas em **JavaScript**.

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

Código-fonte 3.17 – Importando JavaScript com @Url.Content()

Fonte: Elaborado pelo autor (2020)

Para incrementar um pouco mais nosso aplicativo, vamos adicionar uma seção de cabeçalho e rodapé. O cabeçalho será composto por um menu de opção com links para as funcionalidades Tipo de Produto e Produto (implementado futuramente) e o rodapé terá uma seção de contato da **FiapSmartCity**.

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>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
  <title>FiapSmartCity</title>

  <!-- importando bootstrap e o css do nosso site-->
  <link rel="stylesheet"
href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</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">
        <a class="navbar-brand" asp-area="" asp-
controller="Home" asp-action="Index">FiapSmartCity</a>
        <button class="navbar-toggler" type="button"
data-toggle="collapse" data-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 flex-sm-row-reverse">
          <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>
            <li class="nav-item">
              <a class="nav-link text-dark"
asp-area="" asp-controller="TipoProduto" asp-
action="Index">Tipo Produtos</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  </header>
  <div class="container">
    @RenderBody ()
  </div>
</body>
</html>
```

```
</div>

<footer class="border-top footer text-muted">
  <div class="container">
    <p><b>Fiap Smart City - Copyright &copy;
2020</b></p>
    <p>Av. Lins de Vasconcelos, 1222 e 1264 -
Aclimação - São Paulo/SP</p>
    <p><a href="tel:+551133858010" class="nav-
link">11 3385-8010</a></p>
  </div>
</footer>

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

</body>
</html>
```

Código-fonte 3.18 – View de Layout  
Fonte: Elaborado pelo autor (2020)

Entre nosso cabeçalho e 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 especificar para nossas Views o nome do arquivo de layout, que é feito pelo bloco `@{ Layout }` do arquivo `.cshtml`. Edite no arquivo **Views\TipoProduto\Index.cshtml** a declaração do layout logo após a tag `@model`. É recomendado remover todo o conteúdo HTML duplicado entre View e Layout (por exemplo: `<head>`, parte do `<body>`, `</body>`), para não gerar nenhuma quebra ou incompatibilidade no HTML final. O Código-fonte Importando Layout na View aponta o código ideal para a tela de listagem de tipos.

```
@model IEnumerable<FiapSmartCity.Models.TipoProduto>
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<!DOCTYPE html>
<html>
```



```
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Tipo de Produto</title>
</head>
<body>
    <h1>Tipo de Produto</h1>
    <p>
        <!-- uso de TagHelpers para definir o Controller e a
Action -->
        <a          asp-controller="TipoProduto"          asp-
action="Cadastrar">Novo Tipo</a>
    </p>
    <table class="table" border="1">
        <tr>
            <th>Id</th>
            <th>Descrição</th>
            <th></th>
        </tr>

        @foreach (var item in Model)
        {
            <tr>
                <td>
                    <label>@item.IdTipo</label>
                </td>
                <td>
                    <label>@item.DescricaoTipo</label>
                </td>
                <td>

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

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

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

Código-fonte 3.19 – Importando Layout na View  
Fonte: Elaborado pelo autor (2020)

Execute o projeto e navegue para a tela de listagem de tipos (Index.cshtml), o resultado será semelhante ao da Figura Resultado da aplicação de Layout abaixo:

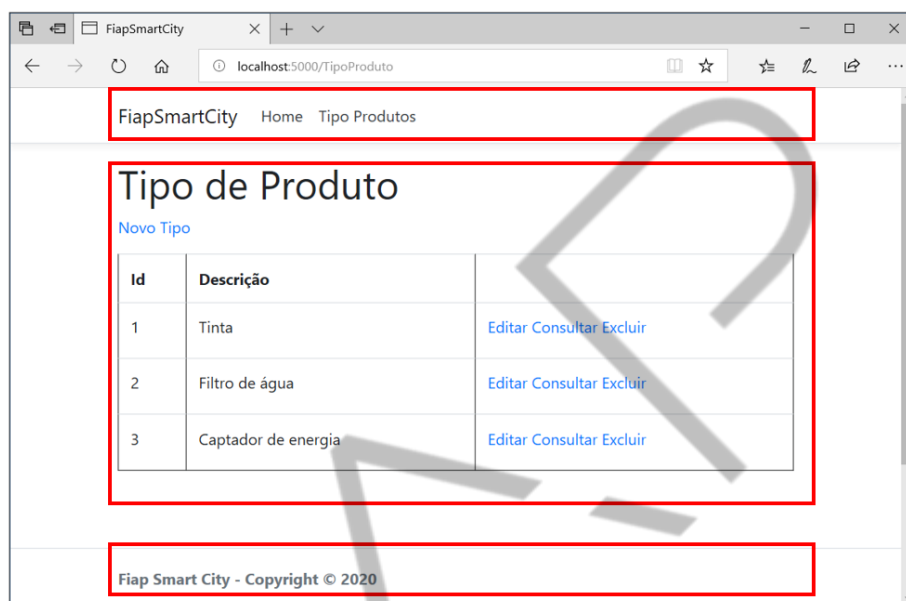


Figura 3.23 – Resultado da aplicação de Layout  
Fonte: Elaborado pelo autor (2020)

Com o layout aplicado na tela de listagem, podemos passar para as demais Views e fazer uso do layout, utilizando a tag `@{ Layout }`. E com a remoção das partes comuns, aplique em todas as Views da funcionalidade de tipo de produto.

**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.

### 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 2, 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, para isso, serão apresentadas algumas técnicas com o uso de recursos do framework para a implementação de validações.

### 3.7.3.1 Validação pelo Controller

Para as validações no *Controller*, tomaremos como base a *Action* Cadastrar() do *TipoProdutoController*. Nela, será adicionada a validação que não permitirá o cadastro de um tipo sem que a descrição seja digitada.

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 nosso exemplo deverá ser implementada com os seguintes passos:

- Validar o conteúdo da descrição digitada.
- Adicionar uma mensagem de erro ao ModelState.
- Validar se existe algum erro no ModelState.
- Encontrou erro no ModelState – manter o usuário na tela do formulário e exibir a mensagem de erro.
- Não encontrou erro 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 (Models.TipoProduto tipoProduto)
{
    // Validando o Campo Descricao
    if ( string.IsNullOrEmpty (tipoProduto.DescricaoTipo) )
    {
        // Adicionando a mensagem de Erro para descrição em
        branco
        ModelState.AddModelError ("Descricao",      "Descrição
        obrigatória!");
    }
}
```

```
}

// Se o ModelState não tem nenhum erro
if (ModelState.IsValid)
{
    // Simula que os dados foram gravados.
    System.Diagnostics.Debug.Print("Descrição: " +
tipoProduto.DescricaoTipo);
    System.Diagnostics.Debug.Print("Comercializado: " +
tipoProduto.Comercializado);
    System.Diagnostics.Debug.Print("Gravando o Tipo de
Produto");

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

// Encontrou um erro no preenchimento do campo descrição
} else
{
    // retorna para tela do formulário
    return View(tipoProduto);
}
}
```

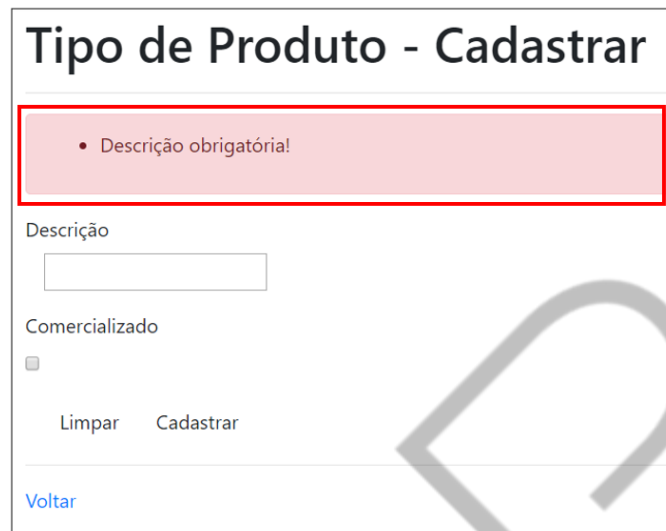
Código-fonte 3.20 – Validando dados pelo *Controller*  
Fonte: Elaborado pelo autor (2020)

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**, vai ajudar 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 abaixo o trecho HTML e o exemplo de uso da tag:

```
<!-- formulário HTML com Tag Helpers-->
<form asp-action="Cadastrar" asp-controller="TipoProduto"
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>
</form>
```

Código-fonte 3.21 – Mensagens de erro com *asp-validation-summary*  
Fonte: Elaborado pelo autor (2020)

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



**Tipo de Produto - Cadastrar**

• Descrição obrigatória!

Descrição

Comercializado

Limpar Cadastrar

[Voltar](#)

Figura 3.24 – Mensagem de erro com a tag asp-validation-summary  
Fonte: Elaborado pelo autor (2020)

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, vamos avaliar uma alteração para esse nosso problema.

### 3.7.3.2 Validação com *Data Annotations*

Usando anteriormente as validações pelo nosso *Controller*, elas são funcionais, porém apresentam 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 Descrição, ambas com o conceito da *Data Annotation*.

```
using System;
using System.ComponentModel.DataAnnotations;

namespace FiapSmartCity.Models
{
    public class TipoProduto
    {
        public int IdTipo { get; set; }

        [Required(ErrorMessage= "Descrição obrigatória!")]
        [StringLength(50,
            MinimumLength = 3,
            ErrorMessage = "A descrição deve ter, no mínimo, 3 e, no máximo, 50 caracteres")]
        public String DescricaoTipo { get; set; }
        public bool Comercializado { get; set; }
    }
}
```

Código-fonte 3.22 – Validações com *Data Annotations*  
Fonte: Elaborado pelo autor (2018)

Depois de inserir nossas anotações no modelo, vamos remover a validação feita no **Controller**, veja o Código-fonte Removendo a validação do *Controller*.

```
// Anotação de uso do Verb HTTP Post
[HttpPost]
public ActionResult Cadastrar(Models.TipoProduto tipoProduto)
{
    // Validando o Campo Descricao
    //if ( string.IsNullOrEmpty (tipoProduto.DescricaoTipo)
    )
}
```

```
//{
//    // Adicionando a mensagem de Erro para descrição
//    em branco
//    ModelState.AddModelError("Descricao", "Descrição
//    obrigatória!");
//}

// Se o ModelState não tem nenhum erro
if (ModelState.IsValid)
{
    // Simila que os dados foram gravados.
    System.Diagnostics.Debug.Print("Descrição: " +
    tipoProduto.DescricaoTipo);
    System.Diagnostics.Debug.Print("Comercializado: " +
    tipoProduto.Comercializado);
    System.Diagnostics.Debug.Print("Gravando o Tipo de
    Produto");

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

// Encontrou um erro no preenchimento do campo descrição
} else
{
    // retorna para tela do formulário
    return View(tipoProduto);
}
}
```

Código-fonte 3.23 – Removendo a validação do *Controller*  
Fonte: Elaborado pelo autor (2018)

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 3.25 – Exibindo mensagem de erro com *Data Annotations*  
 Fonte: Elaborado pelo autor (2018)

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, como: intervalo de números, validação de e-mail, expressões regulares e tipo de dados.

Abaixo, segue o quadro 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 3.6– *Annotation* para validação de dados  
 Fonte: Elaborado pelo autor (2018)



### 3.7.4 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= "Descrição obrigatória!")]
[StringLength(50, ErrorMessage = "A descrição deve ter no
máximo 50 caracteres")]
[Display(Name="Descrição:")]
public String DescricaoTipo { get; set; }
```

Código-fonte 3.24 – Anotação para rótulos  
Fonte: Elaborado pelo autor (2018)

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 FiapSmartCity.Models.TipoProduto

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

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

    <!-- formulário HTML com Tag Helpers-->
    <form asp-action="Cadastrar" asp-controller="TipoProduto"
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
class="alert alert-danger"></div>
        }

        <div class="form-group">
            <label asp-for="DescricaoTipo" class="control-
label"></label>
            <input asp-for="DescricaoTipo" class="form-
control col-md-4" />
            <span asp-validation-for="DescricaoTipo"
class="text-danger"></span>
        </div>

        <div class="form-group">
            <label asp-for="Comercializado" class="control-
label"></label>
            <input asp-for="Comercializado" />
        </div>

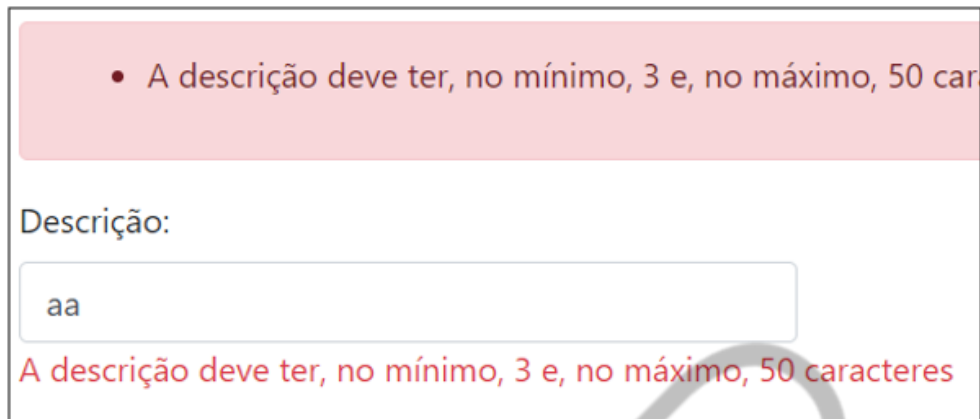
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="reset" value="Limpar"
class="btn btn-default" />
                <!-- HTML Simple para envio dos dados do
formulário -->
                <input type="submit" value="Cadastrar"
class="btn btn-default" />
            </div>
        </div>
        <hr />
    </div>
</form>
<div>
    <a asp-controller="TipoProduto" asp-
action="Index">Voltar</a>
</div>
</body>
</html>

```

Código-fonte 3.25 – Tag Razor para exibição dos rótulos  
Fonte: Elaborado pelo autor (2020)

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

Note como a mensagem é apresentada nesse novo formato:



• A descrição deve ter, no mínimo, 3 e, no máximo, 50 car

Descrição:

aa

A descrição deve ter, no mínimo, 3 e, no máximo, 50 caracteres

Figura 3.26 – Exibindo mensagem de erro com *Data Annotations*  
Fonte: Elaborado pelo autor (2020)

### 3.7.5 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**, que tem a função de 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 que algum componente o recupere.

Vamos aplicar o conceito nas camadas de *Controller* e *Views*, respectivamente. Porém, antes de aplicar as mensagens em nossos fluxos é necessário efetuar uma pequena configuração na classe **Startup.cs** no método **Configure**. É necessário mudar o posicionamento da linha **app.UseCookiePolicy()** para a última linha de configuração do projeto. Observe o Código-fonte “Configuração do projeto para uso do TempData” abaixo:

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {

```

```
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();
    //app.UseCookiePolicy();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template:
                "{controller=Home}/{action=Index}/{id?}");
    });
    app.UseCookiePolicy();
}
```

Configuração do projeto para uso do TempData  
Fonte: Elaborado pelo autor (2020)

Na *Action* Cadastrar do **TipoProdutoController**, é necessário adicionar uma linha de comando que grava uma mensagem de sucesso na TempData. Essa mensagem será adicionada ao fluxo de sucesso do cadastro, veja o Código-fonte Gravando mensagens na TempData:

```
[HttpPost]
public IActionResult Cadastrar (Models.TipoProduto
tipoProduto)
{
    // Se o ModelState não tem nenhum erro
    if (ModelState.IsValid)
    {
        // Simula que os dados foram gravados.
        System.Diagnostics.Debug.Print("Descrição: " +
            tipoProduto.DescricaoTipo);
        System.Diagnostics.Debug.Print("Comercializado: " +
            tipoProduto.Comercializado);
        System.Diagnostics.Debug.Print("Gravando o Tipo de
            Produto");

        // Gravação efetuada com sucesso.
        // Gravando mensagem de sucesso na TempData
        TempData["mensagem"] = "Tipo cadastrado com
            sucesso!";

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

        // Encontrou um erro no preenchimento do campo descrição
    } else
```

```
{  
    // retorna para tela do formulário  
    return View(tipoProduto);  
}  
}
```

Código-fonte 3.26 – Gravando mensagens na TempData  
Fonte: Elaborado pelo autor (2020)

Mensagem de sucesso inserida na TempData, agora precisamos exibir 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 **TipoProduto\Index.cshtml**. Segue exemplo no Código-fonte Exibindo mensagens na TempData:

```
<!-- Views\TipoProduto\Index.cshtml -->  
  
@model IEnumerable<FiapSmartCity.Models.TipoProduto>  
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Tipo de Produto</title>  
</head>  
<body>  
    <h1>Tipo de Produto</h1>  
    <p>  
        <!-- uso de TagHelpers para definir o Controller e a  
Action -->  
        <a asp-controller="TipoProduto" asp-  
action="Cadastrar">Novo Tipo</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>Descrição</th>
<th></th>
</tr>

@foreach (var item in Model)
{
    <tr>
        <td>
            <label>@item.IdTipo</label>
        </td>
        <td>
            <label>@item.DescricaoTipo</label>
        </td>
        <td>

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

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

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

Código-fonte 3.27 – Exibindo mensagens na TempData

Fonte: Elaborado pelo autor (2020)

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:

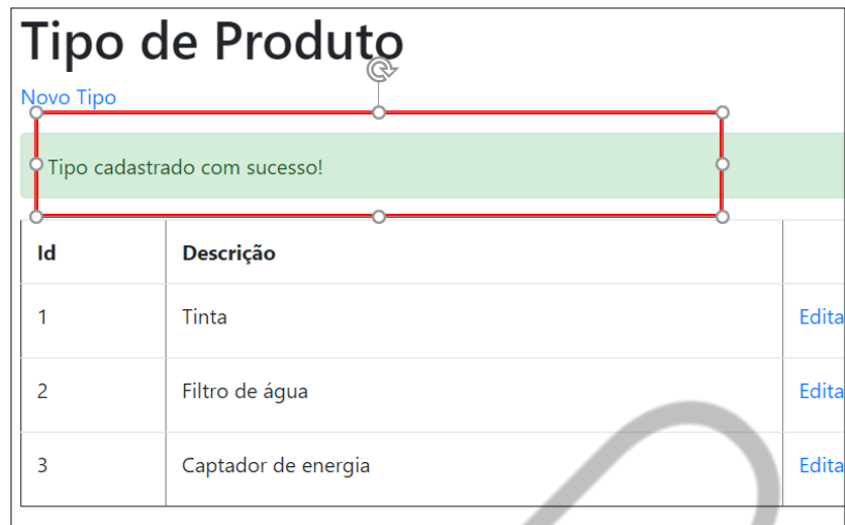


Figura 3.27 – Exibindo mensagem de sucesso com TempData  
Fonte: Elaborado pelo autor (2020)

### 3.8 Acesso A banco de dados

#### 3.8.1 ADO.NET

Chegamos ao momento de conectar nosso projeto ao nosso banco de dados e remover nossos códigos simulados (*mock*). Para isso, o framework .NET 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.

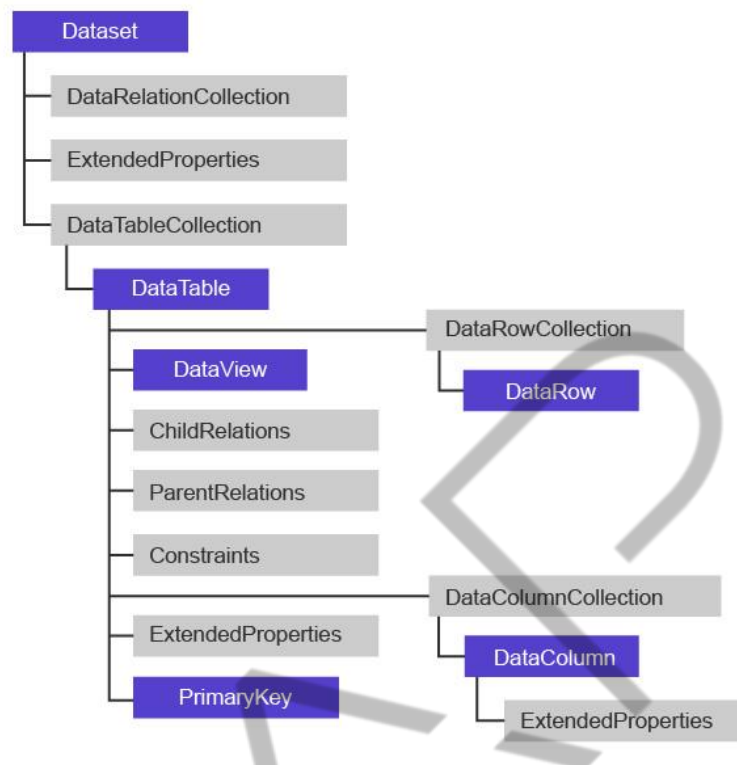


Figura 3.28 – Classes ADO.NET  
Fonte: Elaborado pelo autor (2018)

### 3.8.2 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 e outros requisitos.

Em nosso exemplo, vamos usar 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.

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:



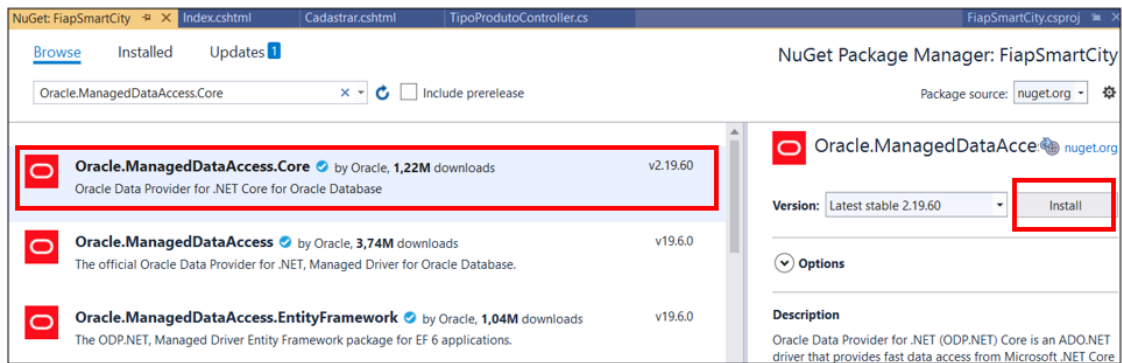


Figura 3.29 – Cliente Oracle no Nuget  
Fonte: Elaborado pelo autor (2020)

Após a instalação do **Oracle.ManagedDataAccess.Core**, verifique como ficou a estrutura de bibliotecas do projeto **FiapSmartCity**. Veja exemplo na Figura Biblioteca Oracle no projeto:

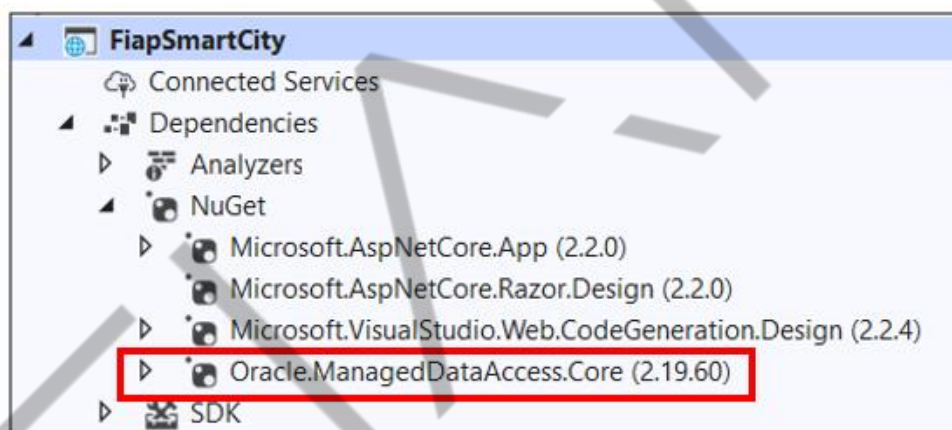


Figura 3.30 – Biblioteca Oracle no projeto  
Fonte: Elaborado pelo autor (2020)

Biblioteca para acesso do 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, vamos adicionar essas informações no arquivo de configuração do projeto **appsettings.json** uma única vez, assim, qualquer alteração será facilitada por estar 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 **FiapSmartCityConnection**.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "FiapSmartCityConnection": "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=XXXXXXX;Pooling=True;Connection
Timeout=60;"
  }
}
```

Código-fonte 3.28 – Web.Config criando Oracle DataSource  
Fonte: Elaborado pelo autor (2020)

**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("FiapSmartCityConnection");

// 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 3.29 – Web.Config  
Fonte: Elaborado pelo autor (2020)

### 3.8.3 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); **ExecuteNonQuery**, 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, vamos utilizar objetos da especificação Oracle, sendo, assim, 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("FiapSmartCityConnection");

    // Criando o Objeto de Conexão
    using (OracleConnection connection = new
OracleConnection(connectionString))
    {
        // Comando SQL
        // Símbolo : 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 3.30 – ADO.NET ExecuteNonQuery()

Fonte: Elaborado pelo autor (2020)

Veja 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("FiapSmartCityConnection");

// 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 3.31 – ADO.NET ExecuteScalar()

Fonte: Elaborado pelo autor (2020)

Veja 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 3.32 – ADO.NET ExecuteQuery() e DataReader

Fonte: Elaborado pelo autor (2020)

**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>.

### 3.8.4 Refatorando a aplicação

Agora que temos uma conexão configurada com 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 TipoProduto, para isso use o script SQL abaixo:

```
CREATE TABLE TIPOPRODUTO (
  IDTIPO      NUMBER          PRIMARY KEY,
  DESCRICAOTIPO VARCHAR2(250) NOT NULL,
  COMERCIALIZADO CHAR(1)
);

CREATE SEQUENCE TIPOPRODUTO_IDTIPO_SEQ;

CREATE OR REPLACE TRIGGER TR_SEQ_TIPOPRODUTO BEFORE INSERT ON
TIPOPRODUTO FOR EACH ROW
BEGIN
  SELECT TIPOPRODUTO_IDTIPO_SEQ.NEXTVAL
  INTO :new.IDTIPO
  FROM dual;
END;

--DROP TRIGGER TR_SEQ_TIPOPRODUTO;
--DROP SEQUENCE TIPOPRODUTO_IDTIPO_SEQ;
--DROP TABLE TIPOPRODUTO;
```

Código-fonte 3.33 – *Script* para criação de tabela Tipo Produto  
Fonte: Elaborado pelo autor (2018)

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

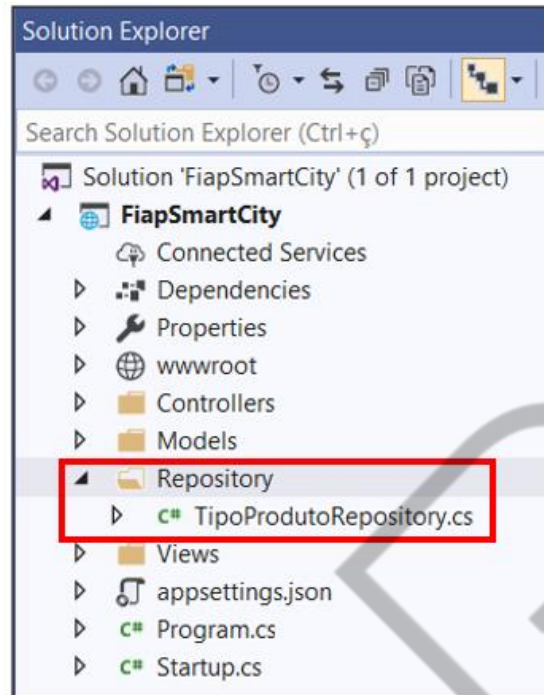


Figura 3.31 – Camada Repository  
Fonte: Elaborado pelo autor (2020)

### 3.8.5 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 tipo de produto, é possível notar que precisamos listar os tipos da nossa base, inserir novos, excluir, alterar os existentes e consultar detalhes de um único tipo. No total, precisamos de cinco operações, sendo, assim, vamos precisar de cinco métodos na classe **TipoProdutoRepository**.

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

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

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



```
}

public void Inserir(TipoProduto tipoProduto)
{
    // Codificar
}

public void Alterar(TipoProduto tipoProduto)
{
    // Codificar
}

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

Código-fonte 3.34 – Estrutura dos métodos do TipoProdutoRepository  
Fonte: Elaborado pelo autor (2020)

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. Abaixo, segue o resultado da classe **TipoProdutoRepository**:

```
public IList<TipoProduto> Listar()
{
    IList<TipoProduto> lista = new
List<TipoProduto>();

    var connectionString = new ConfigurationBuilder()
.SetBasePath(Directory.GetCurrentDirectory())
.AddJsonFile("appsettings.json")
.Build().GetConnectionString("FiapSmartCityConnection");

    using (OracleConnection connection = new
OracleConnection(connectionString))
    {
        String query =
            "SELECT IDTIPO, DESCRICAOTIPO,
COMERCIALIZADO FROM TIPOPRODUTO ";

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

```
        while (dataReader.Read())
        {
            // Recupera os dados
            TipoProduto tipoProd = new TipoProduto();
            tipoProd.IdTipo =
Convert.ToInt32(dataReader["IDTIPO"]);
            tipoProd.DescricaoTipo =
dataReader["DESCRICAOTIPO"].ToString();
            tipoProd.Comercializado =
dataReader["COMERCIALIZADO"].Equals("1");

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

        connection.Close();

    } // Finaliza o objeto connection

    // Retorna a lista
    return lista;
}

public TipoProduto Consultar(int id)
{
    TipoProduto tipoProd = new TipoProduto();

    var connectionString = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json")
        .Build().GetConnectionString("FiapSmartCityConnection");

    using (OracleConnection connection = new
OracleConnection(connectionString))
    {
        String query =
            "SELECT IDTIPO, DESCRICAOTIPO,
COMERCIALIZADO FROM TIPOPRODUTO WHERE IDTIPO = :IDTIPO ";

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

        OracleDataReader dataReader =
command.ExecuteReader();
    }
}
```

```
        while (dataReader.Read())
        {
            // Recupera os dados
            tipoProd.IdTipo =
Convert.ToInt32(dataReader["IDTIPO"]);
            tipoProd.DescricaoTipo =
dataReader["DESCRICAOTIPO"].ToString();
            tipoProd.Comercializado =
dataReader["COMERCIALIZADO"].Equals("1");
        }

        connection.Close();

    } // Finaliza o objeto connection

    // Retorna a lista
    return tipoProd;
}

public void Inserir(TipoProduto tipoProduto)
{
    var connectionString = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json")
        .Build().GetConnectionString("FiapSmartCityConnection");

    using (OracleConnection connection = new
OracleConnection(connectionString))
    {
        String query =
            "INSERT INTO TIPOPRODUTO ( DESCRICAOTIPO,
COMERCIALIZADO ) VALUES ( :descr, :comerc ) ";

        OracleCommand command = new
OracleCommand(query, connection);

        // Adicionando o valor ao comando
        command.Parameters.Add("descr",
tipoProduto.DescricaoTipo);
        command.Parameters.Add("comerc",
Convert.ToInt32(tipoProduto.Comercializado));

        // Abrindo a conexão com o Banco
        connection.Open();
        command.ExecuteNonQuery();
        connection.Close();
    }
}
```

```
}

public void Alterar(TipoProduto tipoProduto)
{
    var connectionString = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json")
        .Build().GetConnectionString("FiapSmartCityConnection");

    using (OracleConnection connection = new
OracleConnection(connectionString))
    {
        String query =
            "UPDATE TIPOPRODUTO SET DESCRICAOTIPO =
:descr , COMERCIALIZADO = :comerc WHERE IDTIPO = :id ";

        OracleCommand command = new
OracleCommand(query, connection);

        // Adicionando o valor ao comando
        command.Parameters.Add("descr",
tipoProduto.DescricaoTipo);
        command.Parameters.Add("comerc",
Convert.ToInt32(tipoProduto.Comercializado));
        command.Parameters.Add("id",
tipoProduto.IdTipo);

        // Abrindo a conexão com o Banco
        connection.Open();
        command.ExecuteNonQuery();
        connection.Close();
    }
}

public void Excluir(int id)
{
    var connectionString = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json")
        .Build().GetConnectionString("FiapSmartCityConnection");

    using (OracleConnection connection = new
OracleConnection(connectionString))
    {
```

```
        String query =  
            "DELETE TIPOPRODUTO WHERE IDTIPO = :id ";  
  
        OracleCommand command = new  
OracleCommand(query, connection);  
  
        // Adicionando o valor ao comando  
        command.Parameters.Add("id", id);  
  
        // Abrindo a conexão com o Banco  
        connection.Open();  
        command.ExecuteNonQuery();  
        connection.Close();  
    }  
}
```

Código-fonte 3.35 – Código completo do TipoProdutoRepository  
Fonte: Elaborado pelo autor (2020)

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 *TipoProdutoRepository* e a chamada do método correspondente para a operação desejada.

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

```
using FiapSmartCity.Models;  
using FiapSmartCity.Repository;  
using Microsoft.AspNetCore.Mvc;  
  
namespace FiapSmartCity.Controllers  
{  
    public class TipoProdutoController : Controller  
    {  
  
        private readonly TipoProdutoRepository  
tipoProdutoRepository;  
  
        public TipoProdutoController()  
        {  
            tipoProdutoRepository = new  
TipoProdutoRepository();  
        }  
  
        [HttpGet]  
        public IActionResult Index()  
        {  
            var listaTipo = tipoProdutoRepository.Listar();  
            return View(listaTipo);  
        }  
    }  
}
```

```
}

// Anotação de uso do Verb HTTP Get
[HttpGet]
public ActionResult Cadastrar()
{
    return View(new TipoProduto());
}

// Anotação de uso do Verb HTTP Post
[HttpPost]
public ActionResult Cadastrar(models.TipoProduto
tipoProduto)
{
    if (ModelState.IsValid)
    {
        tipoProdutoRepository.Inserir(tipoProduto);

        TempData["mensagem"] = "Tipo cadastrado com
sucesso!";
        return RedirectToAction("Index",
"TipoProduto");
    }
    else
    {
        return View(tipoProduto);
    }
}

[HttpGet]
public ActionResult Editar(int Id)
{
    var tipoProduto =
tipoProdutoRepository.Consultar(Id);
    return View(tipoProduto);
}

[HttpPost]
public ActionResult Editar(models.TipoProduto
tipoProduto)
{
    if (ModelState.IsValid)
    {
        tipoProdutoRepository.Alterar(tipoProduto);

        TempData["mensagem"] = "Tipo alterado com
sucesso!";
        return RedirectToAction("Index",
"TipoProduto");
    }
}
```

```
        }
        else
        {
            return View(tipoProduto);
        }
    }

    [HttpGet]
    public ActionResult Consultar(int Id)
    {
        var tipoProduto =
tipoProdutoRepository.Consultar(Id);
        return View(tipoProduto);
    }

    [HttpGet]
    public ActionResult Excluir(int Id)
    {
tipoProdutoRepository.Excluir(Id);

        TempData["mensagem"] = "Tipo removido com
sucesso!";

        return RedirectToAction("Index", "TipoProduto");
    }
}
}
```

Código-fonte 3.36 – Código completo do TipoProdutoController  
Fonte: Elaborado pelo autor (2020)

### 3.9 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.

### 3.9.1 Atributos

Quando vamos falar de 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, método ou parâmetro, sua forma de inserção é feita por anotações. Na sequência deste capítulo, vamos usar muito a inserção de anotações, assim, ficará mais claro o conceito de atributo.

### 3.9.2 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 2 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 3.37 – Exemplo de ActionFilters  
Fonte: Elaborado pelo autor (2018)

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

### 3.9.3 Implementando Action Filters

Vamos usar 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, vamos implementar um **Action Filter** que irá salvar 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 dentro na nossa pasta *Controller* e, em seguida, criar uma simples classe C# com o nome de **LogFilter**. Veja na Figura *Namespace Filtros* e classe LogFilter a estrutura do projeto após a criação:

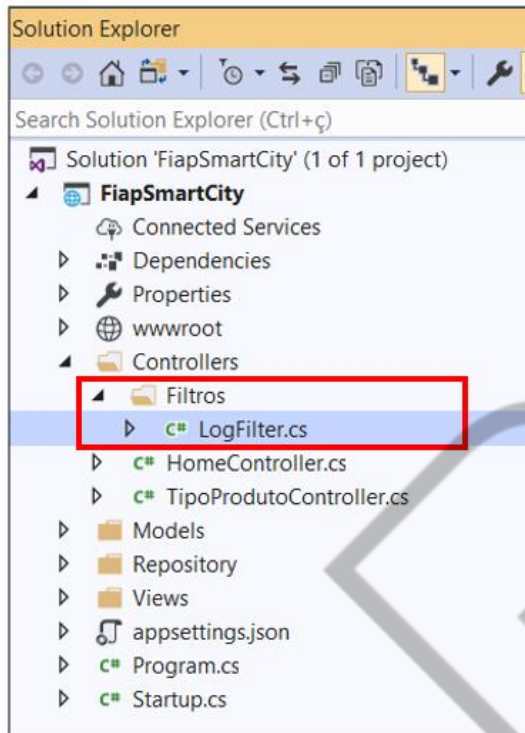


Figura 3.32 – *Namespace* Filtros e classe LogFilter  
Fonte: Elaborado pelo autor (2020)

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

Neste exemplo, será usada a classe Debug para imprimir a data e a hora da execução e o nome do *Controller* executado. Veja no Código-fonte Log ActionFilters a classe filtro, atente para a extensão da classe **ActionFilterAttribute** e o método **OnActionExecuting**:

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

namespace FiapSmartCity.Controllers.Filtros
{
    public sealed class LogFilter : ActionFilterAttribute
    {
        public override void
        OnActionExecuting(ActionExecutingContext context)
        {
            base.OnActionExecuting(context);

            System.Diagnostics.Debug.WriteLine("=====
```

```

=====");
        System.Diagnostics.Debug.WriteLine("== Iniciando a
gravação da mensagem de log");
        System.Diagnostics.Debug.WriteLine("Controller : "
+ context.RouteData.Values["Controller"] + " executado");
        System.Diagnostics.Debug.WriteLine("Action : " +
context.RouteData.Values["Action"] + " executado");
        System.Diagnostics.Debug.WriteLine("Data e Hora :
" + DateTime.Now);

System.Diagnostics.Debug.WriteLine("=====
=====");
    }

}
}

```

Código-fonte 3.38 – Log ActionFilters  
 Fonte: Elaborado pelo autor (2020)

Criado o primeiro filtro, agora é só usá-lo. Anote a *Action* Index do *TipoProdutoController* 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:

```

[Filtros.LogFilter]
[HttpGet]
public IActionResult Index()
{
    var listaTipo = tipoProdutoRepository.Listar();
    return View(listaTipo);
}

```

Código-fonte 3.39 – *Action* Index usando o Filtro de Log  
 Fonte: Elaborado pelo autor (2020)

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

Depois de testado o nosso filtro de log, remova da Index, anote toda a classe *TipoProdutoController* e repita novamente 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:

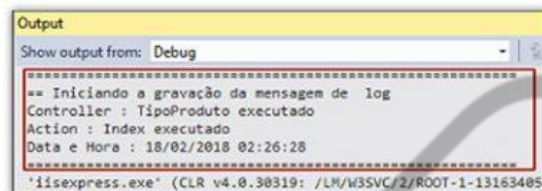


Figura 3.33 – Mensagens geradas pelo LogFilter  
Fonte: Elaborado pelo autor (2018)

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 Output. Sinta-se livre para fazer com que essa informação seja armazenada em uma base de dados.

### 3.10 Considerações Finais

O presente capítulo apresentou um exemplo de funcionalidade implementada com uso do framework ASP.NET MVC 5. Foram demonstrados o fluxo da aplicação e o fluxo de criação dos componentes *Model*, *View* e *Controller*, usando os assistentes do framework e do Visual Studio.

Foi apresentado 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, foi introduzido 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](https://msdn.microsoft.com/pt-br/library/e80y5yhx(v=vs.110).aspx)>. Acesso em: 2 fev. 2018.

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

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

\_\_\_\_\_. **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](https://msdn.microsoft.com/pt-br/library/system.web.mvc.htmlhelper(v=vs.118).aspx)>. Acesso em: 2 fev. 2018.

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

MICROSOFT. **Entity Framework Model First.** Disponível em: <[https://msdn.microsoft.com/en-us/library/jj205424\(v=vs.113\).aspx](https://msdn.microsoft.com/en-us/library/jj205424(v=vs.113).aspx)>. Acesso em: 15 fev. 2018.

MICROSOFT. **Implementando a funcionalidade básica CRUD com o Entity Framework no aplicativo ASP.NET 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: 15 fev. 2018.

MICROSOFT. **O ASP.NET 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: 2 fev. 2018.

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