

Desenvolvimento Web com Asp.Net Core



Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Monte Everest Participações e Empreendimentos Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."

Desenvolvimento Web com Asp.Net Core

Autoria

Emilio Celso de Souza

Edição nº 3

Dezembro/2019

Sumário

1. .NET Core	7
1.1. <i>Introdução.....</i>	7
1.2. <i>.NET Core e seu ecossistema .NET.....</i>	8
1.3. <i>Comandos do .NET Core.....</i>	9
1.4. <i>Ferramentas de desenvolvimento.....</i>	14
1.5. <i>Aplicação Console</i>	17
1.6. <i>Aplicação App Web</i>	18
1.7. <i>Outros componentes do Visual Studio</i>	20
1.7.1. Microsoft.AspNetCore.Mvc.TagHelpers	21
1.8. <i>Fluxo de execução</i>	22
1.9. <i>Projeto: definição do projeto do curso</i>	25
1.9.1. Iniciando com o Visual Studio 2017	26
1.9.2. Criando o projeto no Visual Studio	27
2. Controllers e Actions.....	33
2.1. <i>Introdução.....</i>	33
2.1.1. Arquitetura MVC (Model – View – Controller).....	33
2.1.2. Principais tipos de retorno de actions.....	36
2.2. <i>Projeto: controllers e actions</i>	37
2.2.1. Definição do controller HomeController e das actions Arquivo e Resultado	37
2.2.2. Adicionando a view Index	39
2.2.3. Executando a aplicação.....	41
2.2.4. Definindo a view Resultado.....	42
3. Elaboração de Formulários (Views).....	43

3.1.	<i>Introdução</i>	43
3.1.1.	O componente ViewData e ViewBag	43
3.1.2.	A sintaxe do Razor.....	44
3.2.	<i>Métodos para Requisição HTTP: GET e POST</i>	45
3.3.	<i>Projeto: Views</i>	47
3.3.1.	Definição de um Model	47
3.3.2.	Inclusão de novos actions	48
3.3.3.	Definição das Views.....	49
4.	Conceitos de Delegates	53
4.1.	<i>Introdução</i>	53
4.2.	<i>Definindo um novo projeto</i>	53
4.2.1.	Definindo os delegates.....	54
4.2.2.	Trabalhando com delegates e expressões lambda	55
5.	Views Fortemente Tipadas	58
5.1.	<i>Introdução</i>	58
5.2.	<i>Projeto: views fortemente tipadas</i>	59
6.	Conceitos do CSS (Cascading Style Sheet)	62
6.1.	<i>Introdução</i>	62
6.2.	<i>Sintaxe</i>	62
6.3.	<i>Conhecendo o Bootstrap</i>	64
6.3.1.	Aplicando o Bootstrap em uma view	65
6.4.	<i>Projeto: CSS e Bootstrap</i>	66
6.4.1.	Projeto: layout do projeto.....	70
6.4.2.	Conhecendo o layout da aplicação	72
6.4.3.	Definindo o controller EventosController	76
6.4.4.	Conhecendo as rotas.....	77
7.	Entity Framework Core	79
7.1.	<i>Introdução</i>	79
7.2.	<i>Projeto: acesso a dados e seus componentes</i>	79
7.2.1.	Definindo o banco de dados	79

7.2.2.	Definindo as entidades.....	80
7.2.3.	Definindo o contexto e o inicializador.....	81
7.2.4.	Configurando a aplicação para a injeção de dependência.....	83
7.2.5.	Preparando o controller para incluir registros	84
7.2.6.	Listando os eventos.....	88
7.2.7.	Alterando e Removendo Registros.....	90
7.2.8.	Alterando registros.....	90
7.2.9.	Removendo registros	93
7.2.10.	Cadastro e Consulta de Participantes	94
7.2.11.	Incluindo Participantes.....	95
7.2.12.	Listando os participantes por evento.....	97
8.	Conceitos do Javascript e Ajax	101
8.1.	<i>Introdução.....</i>	101
8.2.	<i>A estrutura de uma função JQuery</i>	101
8.3.	<i>Aplicação: O controller JScriptController.....</i>	102
8.4.	<i>Aplicação: Definindo funções JQuery</i>	105
9.	Uso do Ajax com views parciais	110
9.1.	<i>Introdução.....</i>	110
9.2.	<i>Projeto: aplicação do Ajax</i>	110
9.2.1.	Definindo a view principal e a view parcial	110
10.	Asp.Net Identity Core	115
10.1.	<i>Introdução.....</i>	115
10.2.	<i>ASP.NET Identity no .NET Core</i>	116
10.3.	<i>Projeto: recursos para o Identity Core</i>	131
10.3.1.	Definindo as entidades.....	131
10.3.2.	Configuração no arquivo Startup.cs.....	132
10.3.3.	Definição dos Models	134
10.3.4.	Criando o controller UsuariosController	135
10.3.5.	Definindo as views.....	138
10.3.6.	Verificando os actions que devem ser controlados por login	141
10.3.7.	Removendo arquivos desnecessários	143

11. WebAPI Core	145
11.1. <i>Introdução.....</i>	145
11.2. <i>Representação de um Webservice REST</i>	145
11.3. <i>Projeto: Criação do projeto WebAPI</i>	146
11.3.1. Definindo o controller para o serviço.....	149
11.3.2. Executando a aplicação.....	151
11.3.3. Consumindo o webservice na aplicação web.....	153

1..NET Core

1.1. Introdução

Até o último instante, a versão que seria a sucessora da versão 4.6 do ASP.NET estava sendo chamada de ASP.NET 5. Poucos dias antes do lançamento oficial, a Microsoft anuncia que a nova versão terá o nome de ASP.NET Core. Essa mudança reflete exatamente o que houve com a nova versão da plataforma: uma reformulação geral, partindo para um modelo completamente novo.

O framework .NET Core da Microsoft é uma plataforma híbrida (*cross-platform*) unificada, elaborada para desenvolvimento em diversas plataformas (dentre elas a plataforma Web), capazes de serem executadas nos sistemas operacionais Windows, Linux e Mac.

Suas principais características são:

1. *Open source*.
2. *Cross platform*.
3. Plataforma escalável de alto desempenho.
4. Suporta containers *Docker* (ambiente de virtualização em nível de sistema) operacional capazes de executar containeres Linux em um único host de controle).

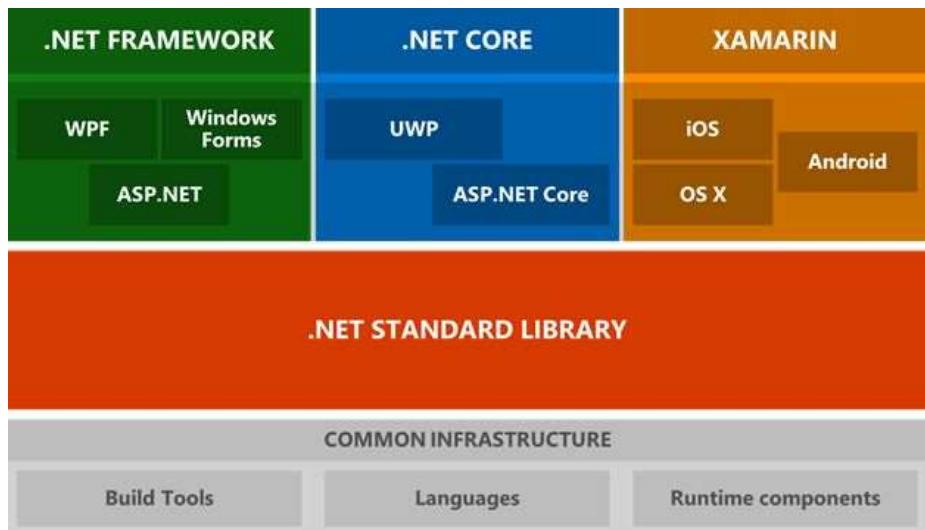
5. Suporta a arquitetura de microserviços.
6. Suporta integração com Github e NuGet.
7. Suporta ferramentas de linhas de comando.
8. Projetado para suportar *deploy* em ambientes na nuvem.

Essa nova direção abre muitas possibilidades de desenvolvimento e integração de sistemas. Até a versão 4.6, os recursos do ASP.NET e do .NET como um todo estavam limitados a servidores ou computadores desktop utilizando sistema operacional Windows. A partir da versão Core 1.0, qualquer sistema operacional, biblioteca, ferramenta de programação ou framework pode ser utilizado para desenvolver, executar ou hospedar aplicativos .NET.

Na prática, isso significa que agora é possível criar um aplicativo para dispositivos Android, iPhone ou para um computador executando Linux. Além disso, é possível distribuir o ASP.NET como parte da aplicação, criando servidores virtuais e serviços de gerenciamento de informações da Internet que são distribuídos junto com o aplicativo.

1.2. .NET Core e seu ecossistema .NET

Vamos analisar a arquitetura da plataforma .NET no seguinte diagrama:



Fonte: <https://medium.com/@renato.groffe/conhecendo-algumas-das-novidades-do-visual-studio-2017-6dc4b8c0e2ed>

Como pode ser visto, o ecossistema .NET possui os três maiores componentes de alto nível:

- .NET Framework.
- .NET Core.
- Xamarin.

No que diz respeito a tipos de projetos, o framework .NET é usado no desenvolvimento de aplicações Windows Forms e WPF, além de aplicações web com MVC e Web Forms.

O .NET Core suporta as bibliotecas UWP e Asp.Net Core. UWP é usado para o desenvolvimento na plataforma Windows 10 e o Asp.Net Core é usado em aplicações web nas plataformas Windows, Linux e Mac.

O propósito do nosso curso é o desenvolvimento web usando o framework .Net Core

1.3. Comandos do .NET Core

A plataforma .NET Core fornece os componentes necessários para executar programas em diversos sistemas operacionais. A página oficial fornece os links

para a instalação do kit de desenvolvimento (SDK). Para os usuários Windows, é preferível utilizar o instalador do Visual Studio, que já faz o download e configura o ambiente de desenvolvimento automaticamente. Se este não foi instalado com o Visual Studio (e mesmo que estejamos em outros sistemas operacionais), a melhor forma é baixá-lo a partir do site: <https://www.microsoft.com/net/core>

Uma vez instalado o SDK, é possível, usando o prompt de comando, criar e executar um aplicativo .NET Core. O exemplo adiante está descrito na página de download e serve de teste das bibliotecas. Neste caso, é o passo a passo para criar um pequeno aplicativo no Windows:

1. Usando o prompt de comando (Console): O principal comando usado é **dotnet**. Um exemplo:

```
>dotnet --info
```

Obtemos um resultado semelhante a este:

```
.NET Command Line Tools (1.0.4)

Product Information:
Version:          1.0.4
Commit SHA-1 hash: af1e6684fd

Runtime Environment:
OS Name:        Windows
OS Version:     10.0.14393
OS Platform:    Windows
RID:            win10-x64
Base Path:      C:\Program Files\dotnet\sdk\1.0.4
```

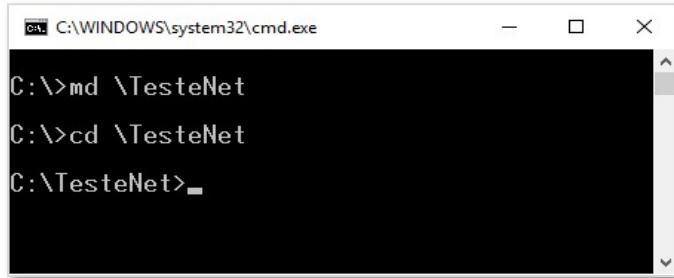
Como exemplo, crie uma pasta para o projeto;

```
>md \TesteNet + ENTER
```



2. Navegue até a pasta criada:

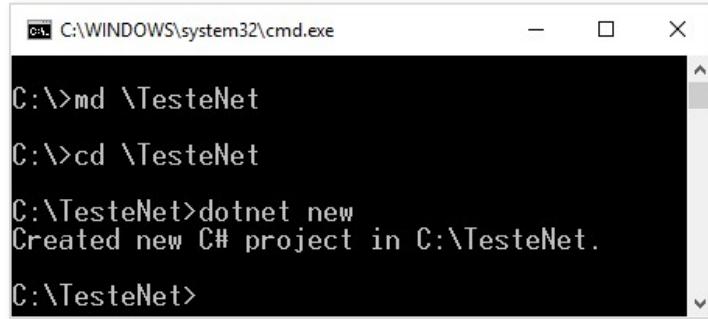
>**cd \TesteNet** + ENTER



```
C:\>md \TesteNet
C:\>cd \TesteNet
C:\TesteNet>
```

3. Crie um projeto de exemplo:

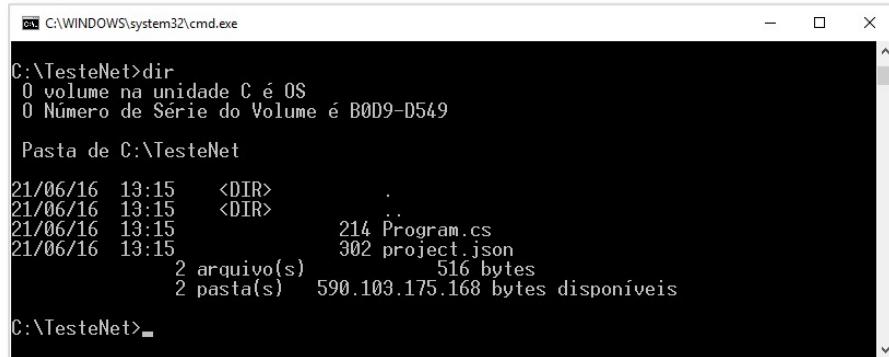
>**dotnet new** + ENTER



```
C:\>md \TesteNet
C:\>cd \TesteNet
C:\TesteNet>dotnet new
Created new C# project in C:\TesteNet.
C:\TesteNet>
```

4. Veja os arquivos criados (**Program.cs** e **project.json**);

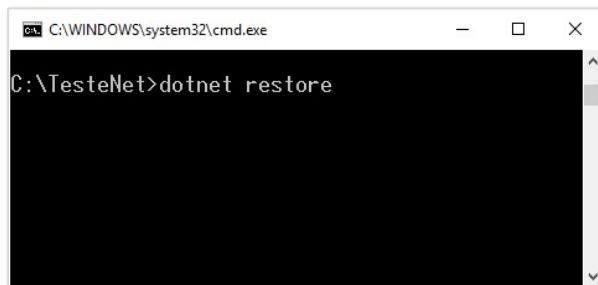
>**dir** + ENTER



```
C:\TesteNet>dir
0 volume na unidade C é OS
0 Número de Série do Volume é B0D9-D549
Pasta de C:\TesteNet
21/06/16 13:15    <DIR>    .
21/06/16 13:15    <DIR>    ..
21/06/16 13:15                214 Program.cs
21/06/16 13:15                302 project.json
2 arquivo(s)          516 bytes
2 pasta(s)      590.103.175.168 bytes disponíveis
C:\TesteNet>
```

5. Restaure os pacotes definidos no arquivo de projeto **project.json**:

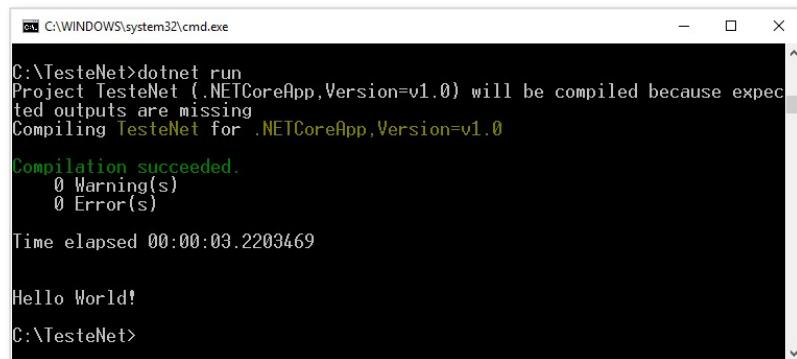
>dotnet restore



A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window shows the command 'dotnet restore' being typed at the prompt 'C:\TesteNet>'. The rest of the window is blank.

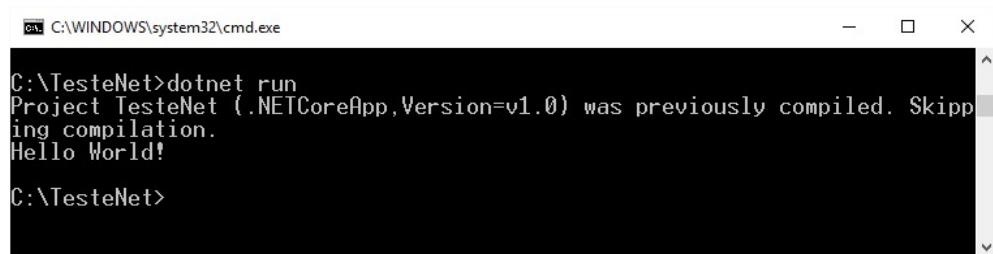
6. Compile e execute o projeto:

>dotnet run



A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window shows the command 'dotnet run' being run. It displays the following output:
C:\TesteNet>dotnet run
Project TesteNet (.NETCoreApp,Version=v1.0) will be compiled because expected outputs are missing
Compiling TesteNet for .NETCoreApp,Version=v1.0
Compilation succeeded.
0 Warning(s)
0 Error(s)
Time elapsed 00:00:03.2203469
Hello World!
C:\TesteNet>

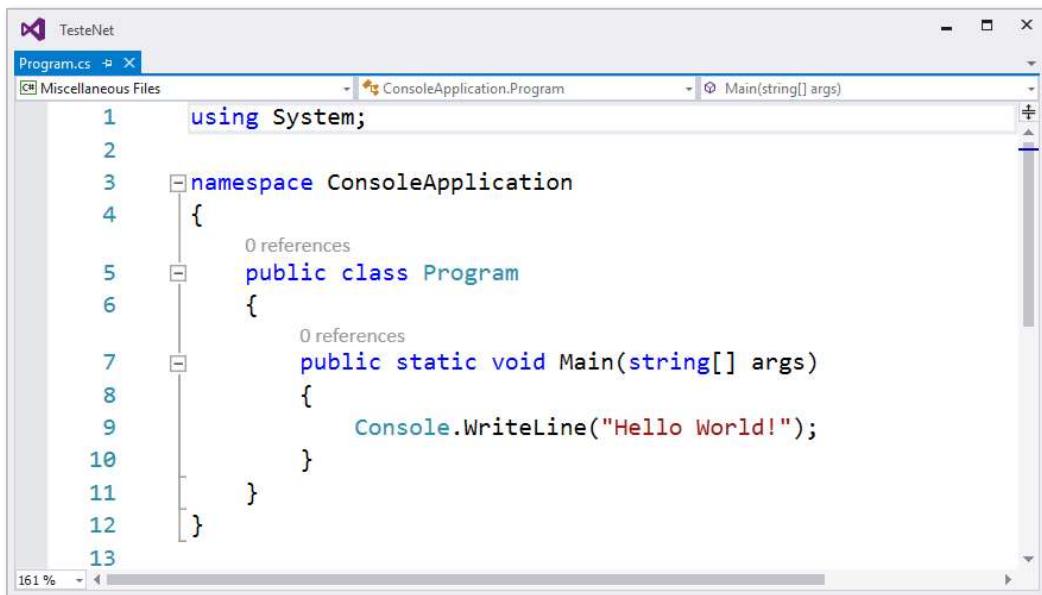
7. Ao pedir para executar novamente, o .NET Core não precisa compilar novamente:



A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window shows the command 'dotnet run' being run again. It displays the following output:
C:\TesteNet>dotnet run
Project TesteNet (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Hello World!
C:\TesteNet>

Os arquivos criados pelo comando **dotnet new** são os seguintes:

- **Program.cs**: Este arquivo não tem nada de novo em relação a uma aplicação Console da versão 4.6 ou inferior do .Net Framework;



```

1  using System;
2
3  namespace ConsoleApplication
4  {
5      public class Program
6      {
7          public static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13

```

Algumas das opções comuns disponíveis para o comando **dotnet** são:

Comando	Descrição
<code>-v --verbose</code>	Habilita output com bastante detalhes (verbose).
<code>--version</code>	Apresenta as informações de versão do .NET CLI.

Alguns comandos comuns são:

Comando	Descrição
<code>new</code>	Inicia um novo projeto .NET básico.
<code>restore</code>	Restaura as dependências especificadas no projeto.

build	Compila um projeto .NET Core
publish	Publica um projeto .NET para <i>deployment</i> (incluindo o <i>runtime</i>)
run	Compila e imediatamente executa um projeto.
test	Executa testes unitários usando o <i>test runner</i> especificado no projeto.
pack	Cria um pacote NuGet

Como exemplo, apresentaremos um processo para criar, buscar dependências, compilar e executar uma aplicação baseada em console (supondo que a pasta do projeto seja **exemplos**):

```
dotnet new console
dotnet restore
dotnet build --output build_app
dotnet build_app/exemplos.dll
```

Este processo cria um projeto baseado em Class Library (DLL) contendo uma classe chamada **Program.cs**.

Esta abordagem foi apresentada como forma de demonstrar que não precisamos obrigatoriamente do Visual Studio para criarmos projetos baseados no .NET Core.

Nas nossas aulas utilizaremos o Visual Studio 2017.

1.4. Ferramentas de desenvolvimento

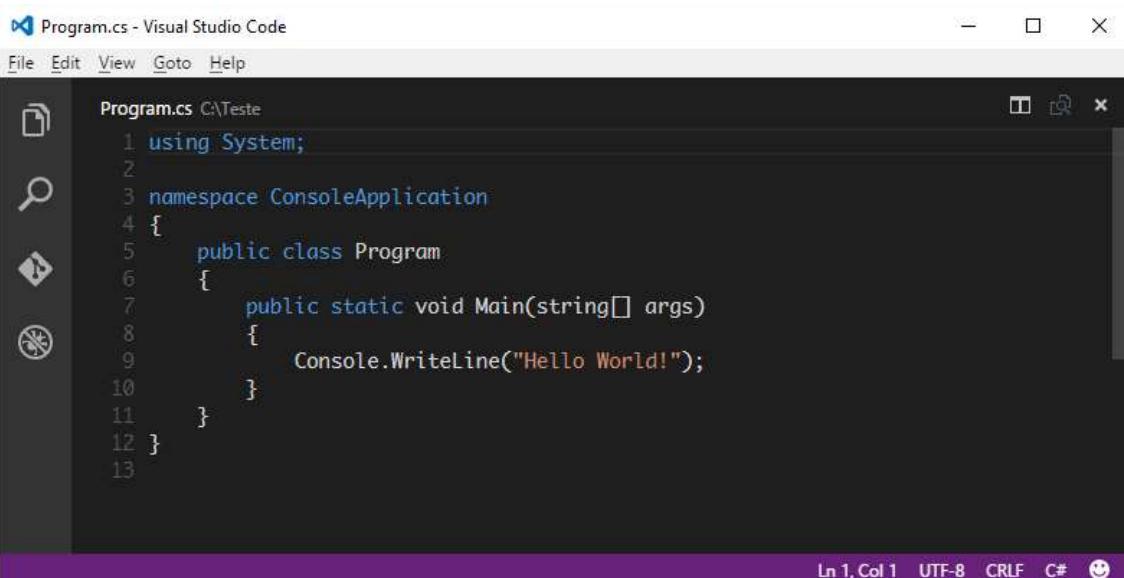
A versão inicial do .NET Core permite criar aplicações Console ou Web. A versão 4.6 e inferiores do .Net Framework permitem criar diversos tipos de aplicações,

como Windows Forms, Console, Web Services, WCF e WPF, entre outros. Existe uma comunidade de programadores, analistas e engenheiros desenvolvendo modelos de programação para o .NET Core e a tendência é termos todos os tipos de modelos usados em todas as plataformas.

Quanto mais complexa é a tecnologia de um aplicativo, mais necessário é ter em mãos uma boa ferramenta de desenvolvimento. É possível desenvolver aplicativos apenas com um editor de texto. O uso, porém, de uma ferramenta de desenvolvimento adequada aumenta em muito a produtividade. As principais IDEs (Integrated Development Environment - ambiente integrado de desenvolvimento) são Visual Studio, Visual Code e Sublime Text.

- **Visual Code**

Ambiente multiplataforma com suporte para diversos tipos de arquivos e linguagens de programação.



The screenshot shows the Visual Studio Code interface with a dark theme. The title bar reads "Program.cs - Visual Studio Code". The menu bar includes File, Edit, View, Goto, Help. The left sidebar has icons for file operations, search, and other tools. The main editor area contains the following C# code:

```
1 using System;
2
3 namespace ConsoleApplication
4 {
5     public class Program
6     {
7         public static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12 }
```

The status bar at the bottom shows "Ln 1, Col 1" and "C#".

- **Sublime Text**

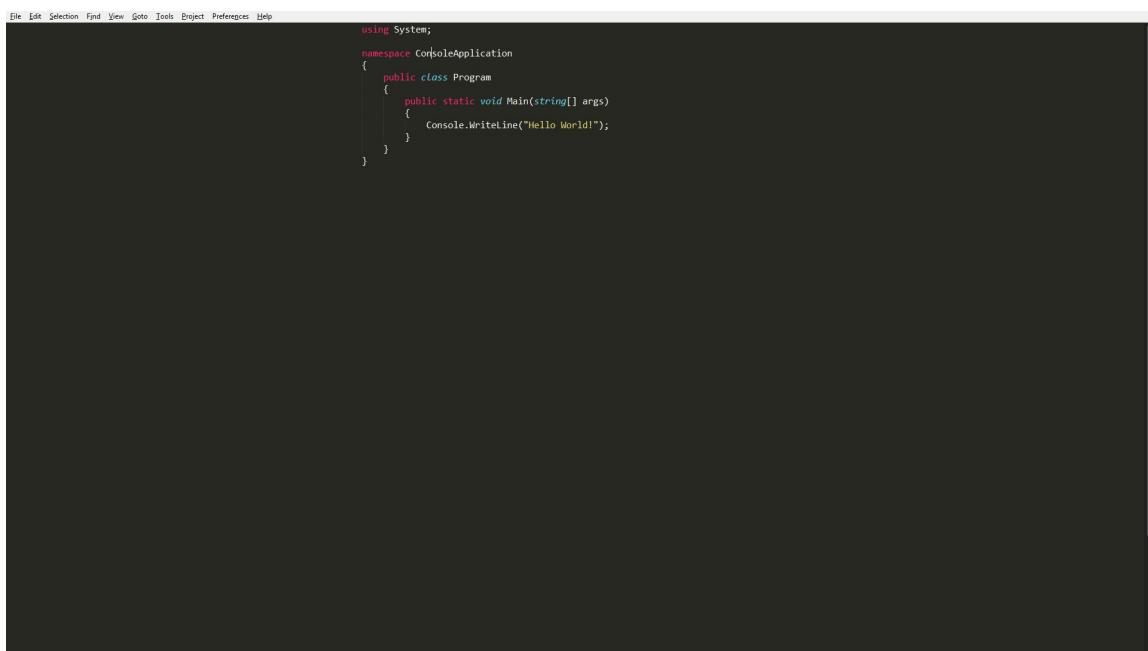
No mesmo estilo do Visual Code, o Sublime Text é uma opção simples, mas que trabalha muito bem diversos tipos de arquivos.

A screenshot of Sublime Text 2 showing a C# program named Program.cs. The code is as follows:

```
1 using System;
2
3 namespace ConsoleApplication
4 {
5     public class Program
6     {
7         public static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12}
13
```

The interface shows the file path "C:\TesteNet\Program.cs - Sublime Text 2 (UNREGISTERED)" at the top. The menu bar includes File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, and Help. The status bar at the bottom indicates "Line 1, Column 1", "Tab Size: 4", and "C#".

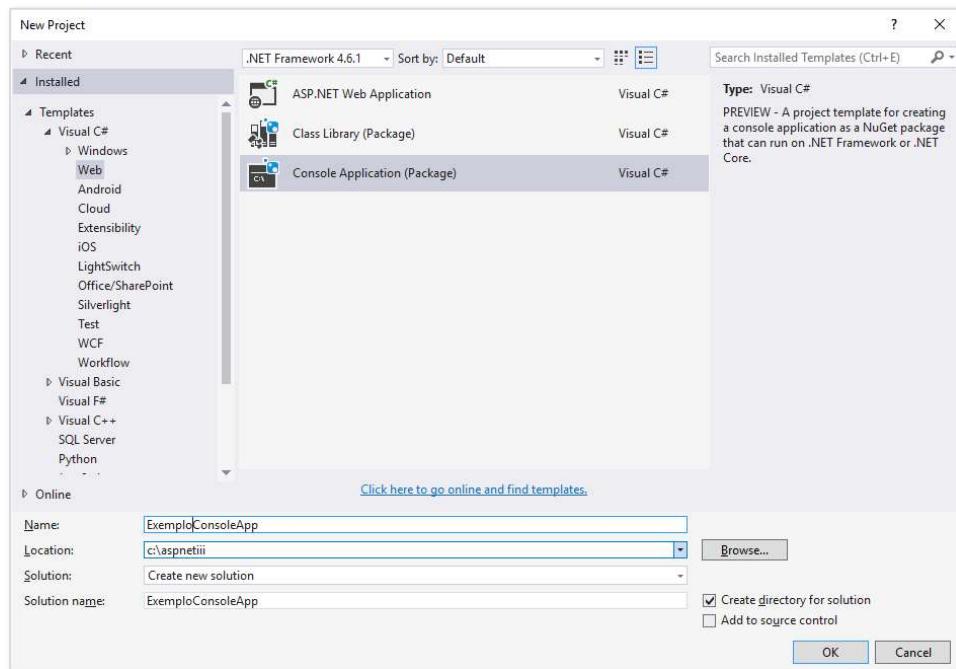
Um dos recursos interessantes do Sublime e que depois foi copiado para outros editores é o modo sem distração (**Distraction Free**): acionado com F11, remove tudo da tela, para o programador se concentrar apenas no código que está escrevendo.



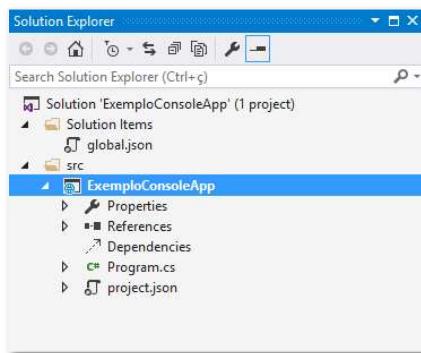
● Visual Studio

A ferramenta oficial da plataforma .NET. Para iniciar um projeto do tipo Console usando o ASP.NET Core, é necessário utilizar o comando de menu **File / New Project / Web** e escolher a opção **Console Application (Package)**.

1.5. Aplicação Console



O projeto cria a seguinte estrutura:



O arquivo **Program.cs** contém o código-fonte do programa principal:

```

namespace ExemploConsoleApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Olá para todos");
        }
    }
}

```

O arquivo **global.json** define o nome dos projetos e a versão do aplicativo:

```

{
    "projects": [ "src", "test" ],

    "sdk": {
        "version": "1.0.0-rc1-update1"
    }
}

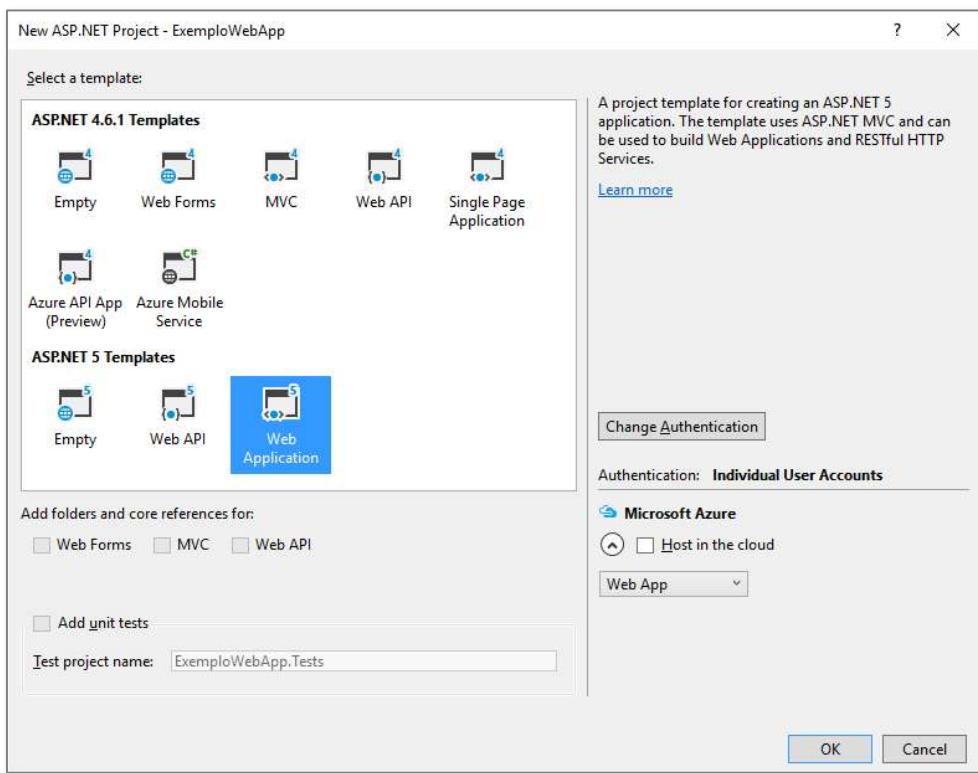
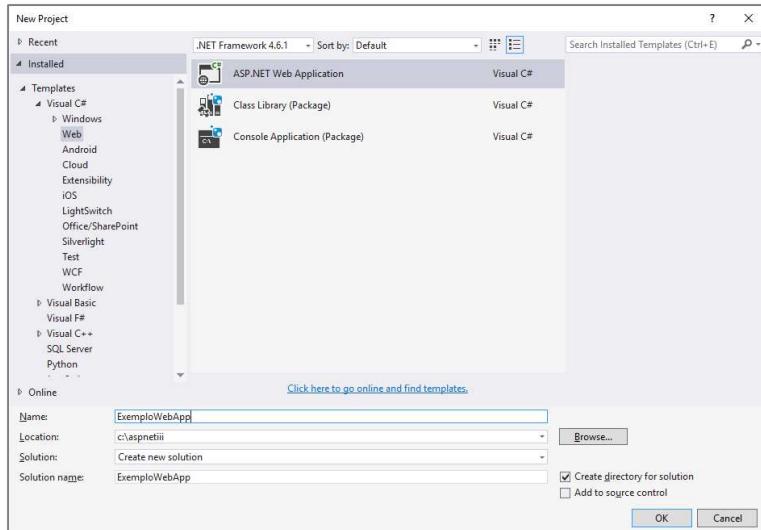
```

Para executar o programa, pressione CTRL + F5.



1.6. Aplicação App Web

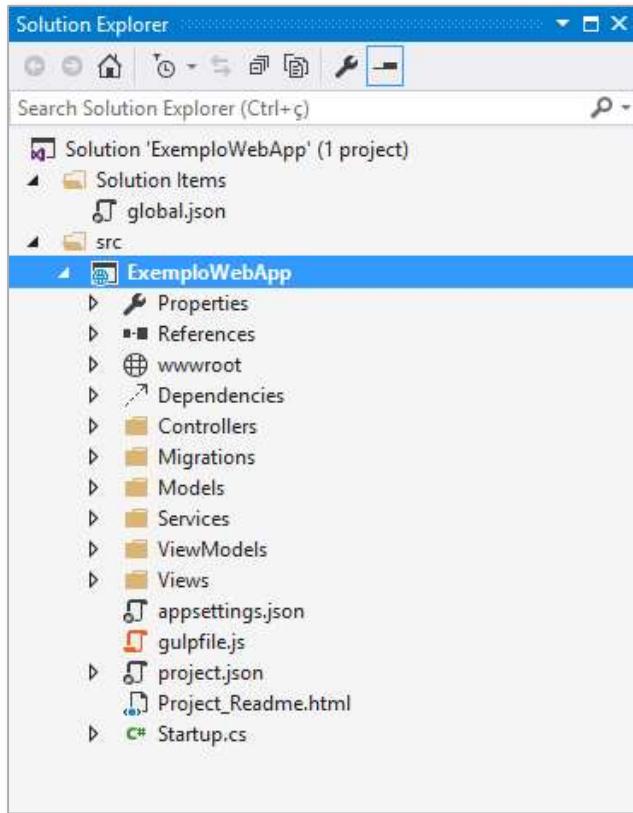
O modelo de Web App para o .NET Core é um bom exemplo de como configurar e implementar os componentes ASP.NET Code. Para incluir uma Web App usando .NET Core, é necessário escolher **File / New Project / Web / ASP.NET Web Application**.



Utilizando a janela Solution Explorer é fácil de observar que a estrutura é bem diferente da utilizada pelo ASP.NET 4.6. Algumas importantes alterações estruturais são as seguintes:

- O formato XML não é utilizado mais para arquivos de sistema. O formato agora é JSON;

- Não existe mais **Web.Config**. O arquivo de configuração agora se chama **appsettings.json**;



1.7. Outros componentes do Visual Studio

Outros componentes que fazem parte do projeto padrão são os mesmos usados na versão 4.6 e inferiores no ASP.NET:

- **Bootstrap**;
- **jQuery**;
- **jQuery-Validation**;
- **Entity Framework**;

- **Microsoft.AspNet.Mvc;**
- **Razor.**

1.7.1. Microsoft.AspNet.Mvc.TagHelpers

Algumas novidades foram introduzidas na geração do código HTML. A classe **TagHelpers** define alguns novos elementos HTML.

- **environment**

Este elemento renderiza ou não um código HTML baseado no valor da propriedade

Microsoft.AspNet.Hosting.IHostingEnvironment.EnvironmentName, que é um valor inserido automaticamente quando a aplicação é iniciada.

No exemplo adiante, os elementos link serão renderizados apenas se o ambiente de execução estiver definido como **Development**.

```
<environment names="Development">
    <link rel="stylesheet" href="css/bootstrap.css" />
    <link rel="stylesheet" href("~/css/site.css" />
</environment>
```

- **asp-controller**

Este atributo define o nome de um controlador. É utilizado para criar uma âncora HTML (hiperlink).

```
<a asp-controller="Home" asp-action="Index">Home</a>
```

- **asp-action**

Este atributo define o nome de um método de um controlador. É utilizado para criar uma âncora HTML (hiperlink).

```
<a asp-controller="Home" asp-action="Index">Home</a>
```

1.8. Fluxo de execução

Quando uma aplicação ASP.NET Core Web App inicia, os comandos seguintes são executados, na sequência adiante:

1. Classe Startup – Método de entrada Main

```
public static void Main(string[] args)
    => WebApplication.Run<Startup>(args);
```

2. Classe Startup – Método construtor

Uma instância da classe **ConfigurationBuilder** é criada e associada à propriedade **Configuration** da classe **Startup**. O método **AddUserSecrets** cria o ambiente necessário para armazenar informações sem compartilhar com ninguém, por meio de um utilitário Console. O método **AddEnvironmentVariables** cria as variáveis do ambiente administrativo. E, finalmente, o método **Build** inicia as classes.

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .AddJsonFile("appsettings.json")
        .AddJsonFile(
            $"appsettings.{env.EnvironmentName}.json",
            optional: true);

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets();
    }

    builder.AddEnvironmentVariables();
```

```
        Configuration = builder.Build();
    }
```

3. Classe Startup – Método ConfigureServices

A classe executa diversos métodos para iniciar os serviços.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<ApplicationContext>(options =>
            options.UseSqlServer(
                Configuration["Data:DefaultConnection:ConnectionString"]));
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();
    services.AddMvc();
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

4. Classe Startup – Método Configure

A classe executa diversos métodos para iniciar os serviços.

```

public void Configure(IApplicationBuilder app,
    IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(
        Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment()) {
        app.UseBrowserLink();
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else {
        app.UseExceptionHandler("/Home/Error");
        try
        { using (var serviceScope =
            app.ApplicationServices
                .GetRequiredService
                <IServiceScopeFactory>())
            .CreateScope() {
                ServiceScope.ServiceProvider
                    .GetService
                    <ApplicationDbContext>()
                    .Database.Migrate();
            }
        }
        catch { }
    }
    app.UseIISPlatformHandler(options =>
        options
            .AuthenticationDescriptions.Clear());
}

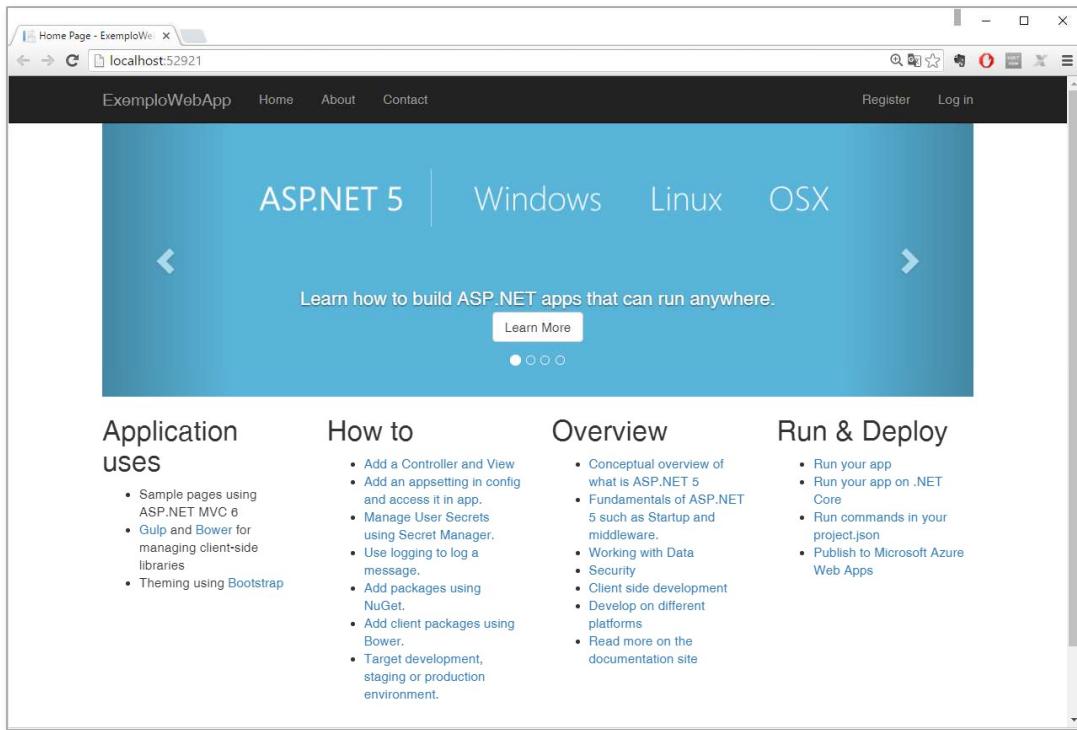
app.UseStaticFiles();

app.UseIdentity();

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

```

5. A View **Home / Index** é exibida:



1.9. Projeto: definição do projeto do curso

Ao longo do curso desenvolveremos um projeto para cadastro de eventos e participantes. Um administrador é responsável pelo cadastro de eventos. Podemos entender como eventos todas as atividades que permitem a presença de participantes, como por exemplo: formatura, festa, um jogo de futebol, etc. Alguns eventos são pagos e outros não, mas no cadastro será contemplado o preço do evento. De posse da lista de eventos, um participante se inscreve em um evento da sua escolha, fornecendo seus dados básicos. A aplicação permite que, a partir de um evento, a lista de participantes seja obtida.

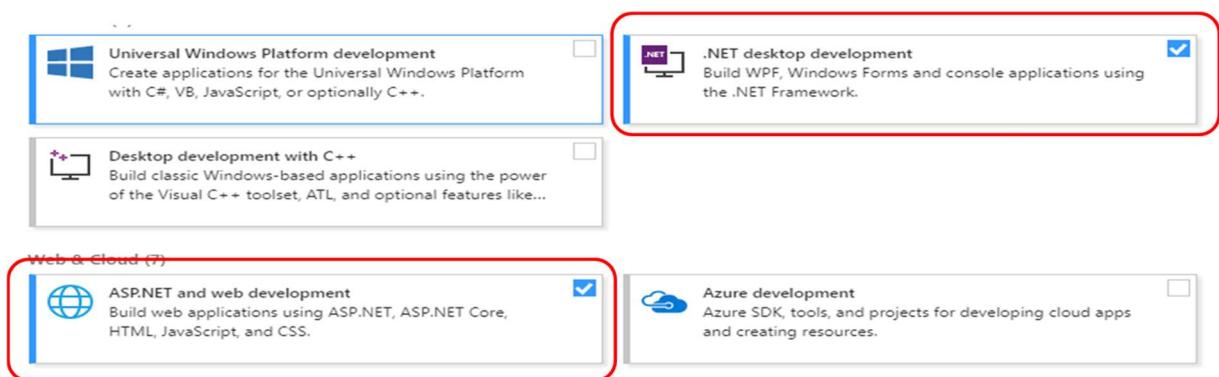
Para que o participante efetue o pagamento, ele poderá usar um cartão de crédito. Uma segunda aplicação simulando uma administradora de cartões de crédito será criada, contemplando a parte que permite interação com aplicações externas, ou seja, a aplicação representará um *webservice*, consumível pela nossa aplicação principal (o que permite a efetivação do pagamento).

1.9.1. Iniciando com o Visual Studio 2017

O Visual Studio possui uma versão chamada de *community*, gratuita, possível de ser usada tanto para fins acadêmicos como comerciais. Esta versão, apesar de mais limitada, permite a criação de projetos robustos, se o desenvolvedor trabalhar individualmente.

Esta versão pode ser obtida no link <https://www.visualstudio.com/pt-br/vs/community/>

No processo de instalação, atentar para marcar os itens:



Uma opção importante é o idioma. Podemos usar o idioma que for mais adequado para nosso propósito, desde que o pacote de idiomas desejado esteja instalado.

Para isso, selecione a aba **Language packs**, e marque os idiomas português e inglês. Esta opção é importante para acompanhar as aulas que apresentaremos:

[Workloads](#) [Individual components](#) [Language packs](#)

Choose your language pack

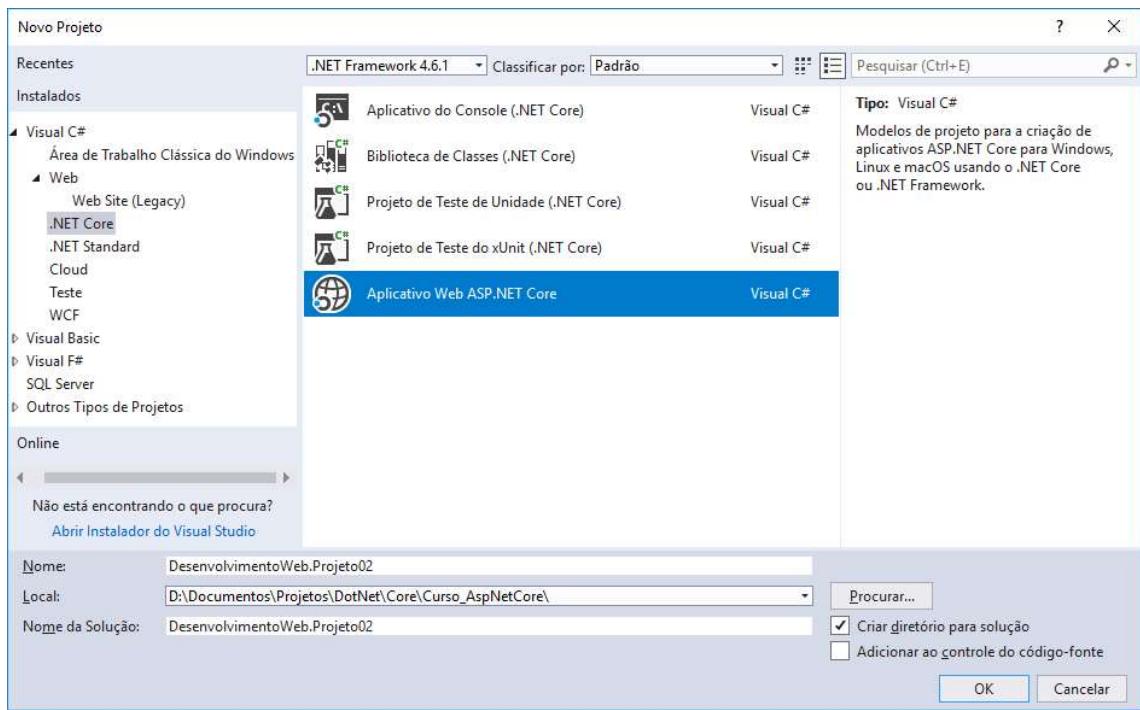
- Chinese (Simplified)
- Chinese (Traditional)
- Czech
- English
- French
- German
- Italian
- Japanese
- Korean
- Polish
- Portuguese (Brazil)
- Russian
- Spanish
- Turkish

Esta etapa pode ser elaborada após a instalação do Visual Studio, bastando para isso repetir o processo de instalação e selecionar a opção **Modify**.

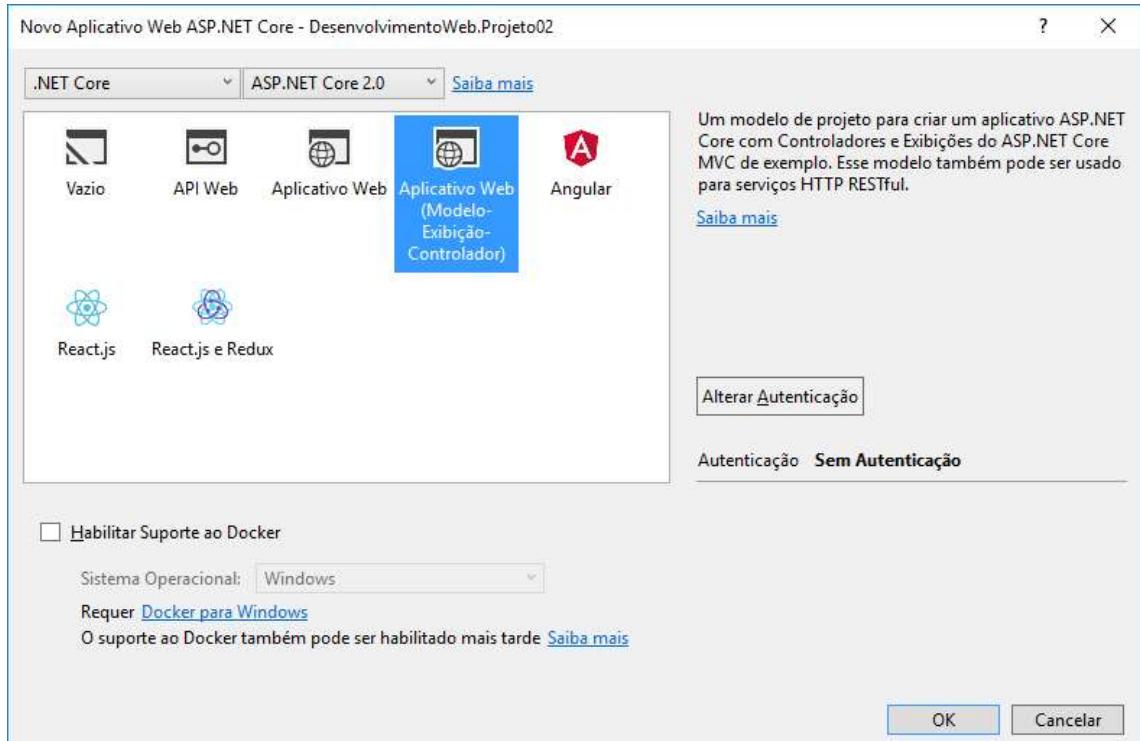
Quando o processo estiver completo, podemos criar nosso projeto web.

1.9.2. Criando o projeto no Visual Studio

No menu **Arquivo**, selecionar **Novo -> Projeto**.

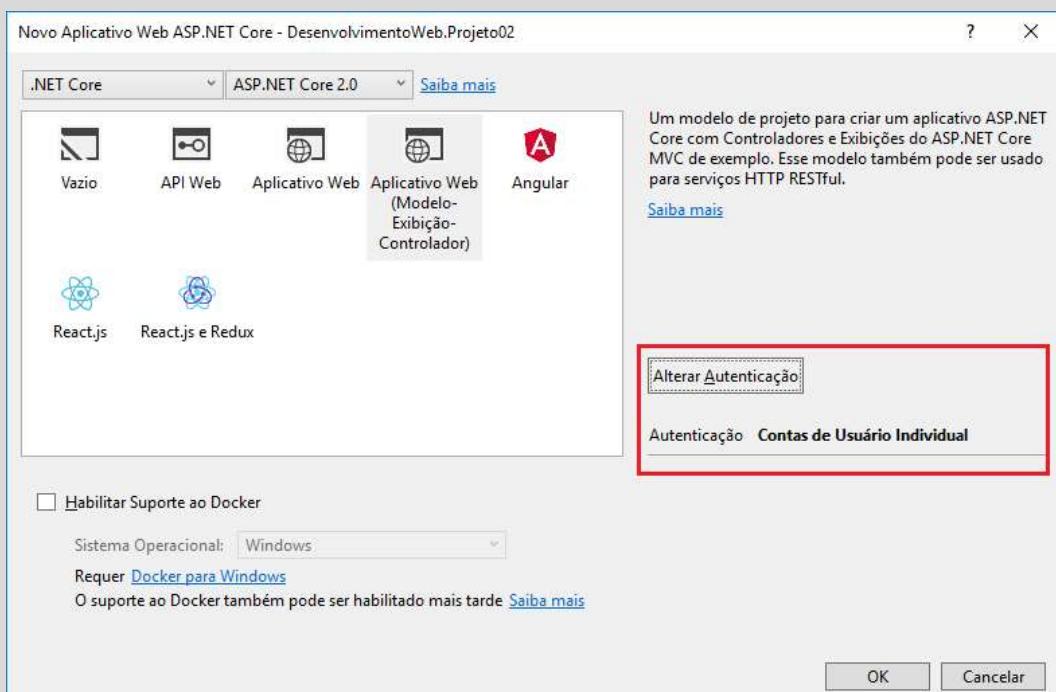


Escolher a opção **Aplicativo Web (Modelo-Exibição-Controlador)**, na aba **ASP.NET Core 2.0** (é possível que haja mais de uma):

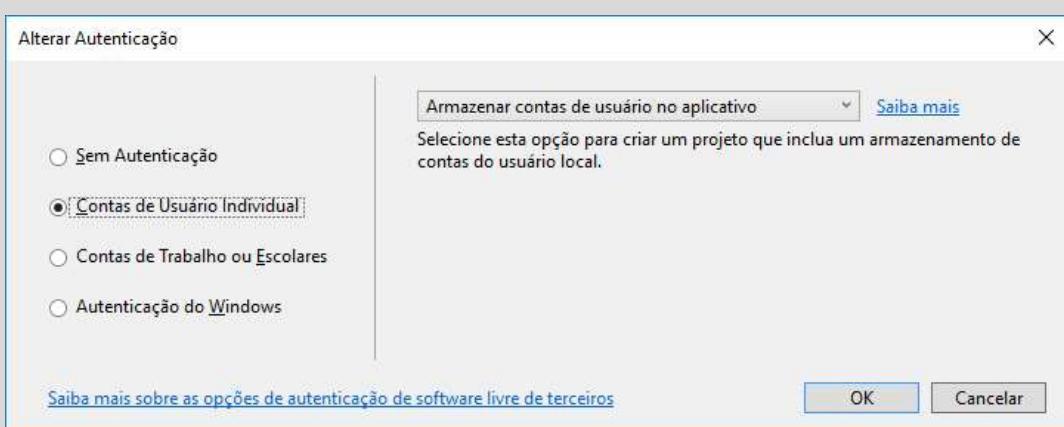


Observação:

Em aulas futuras apresentaremos o mecanismo para criar autenticação de usuários. Serão mostrados comandos para criar o banco de dados de usuários manualmente, mas é possível deixar o projeto pronto para esta tarefa, selecionado a opção marcada abaixo:

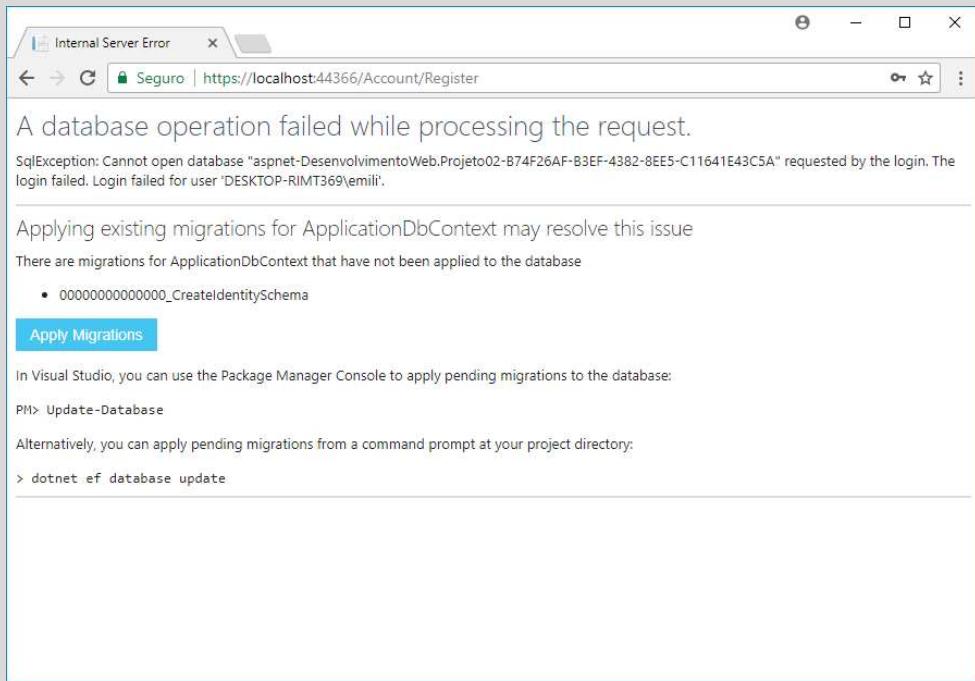


Basta clicar no botão Alterar Autenticação e selecionar Contas de Usuário Individual:



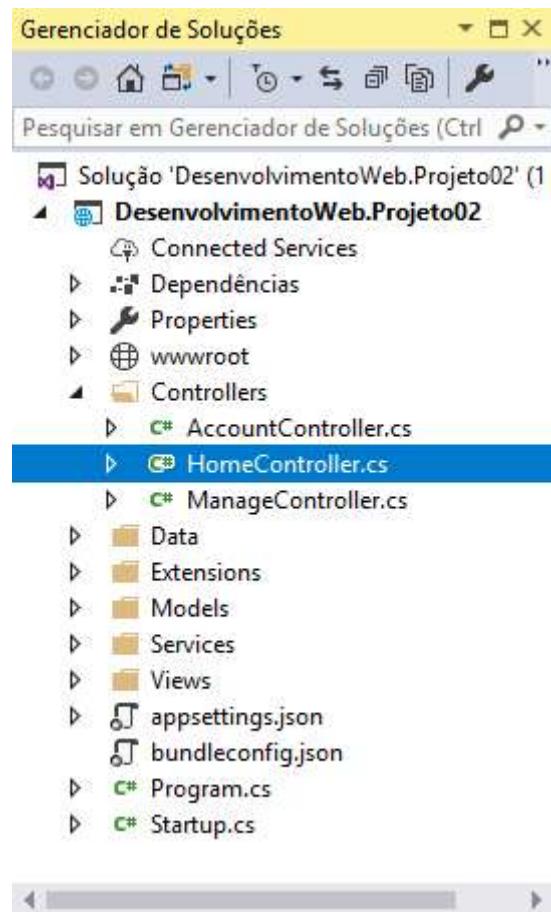
Depois basta refazer e/ou aproveitar as classes criadas no projeto.

Se você marcou a opção “**Conta de Usuário Individual**”, convém executar a aplicação para que o banco de dados e as classes necessárias sejam criadas no projeto. Para tanto, selecionar no menu a opção **Register** e incluir um novo registro. A tela a seguir será apresentada:



Clique em “**Apply Migrations**” que o processo de criação do banco de dados de usuários é criado. Convém atualizar a página para que as alterações sejam efetivadas.

A partir deste ponto estamos prontos para o nosso projeto. A estrutura de pastas do projeto é semelhante a:



Se você marcou a opção “**Conta de Usuário Individual**”, convém executar a aplicação para que o banco de dados e as classes necessárias sejam criados no projeto.

A seguir apresentaremos algumas partes importantes do projeto:

- **Arquivo Program.cs**

Método Main: Este é o ponto de partida da aplicação. Dentre as chamadas contidas neste método, existe uma referência à classe Startup (arquivo **Startup.cs**):

```
namespace DesenvolvimentoWeb.Projeto02
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }
    }
}
```

```
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
    }

}
```

- **Arquivo Startup.cs**

Método **ConfigureServices**: Neste método são configuradas as dependências da aplicação, tanto as que são utilizadas pelo Framework como aquelas específicas da aplicação.

Método **Configure**: Este método é usado para definir as configurações referentes às requisições HTTP.

Ambos os métodos são executados durante a execução, uma vez que esta classe é executada assim que a aplicação entra em execução.

Nos próximos tópicos apresentaremos mais detalhes da aplicação, e como configurar dados no arquivo **Startup.cs**.

2.Controllers e Actions

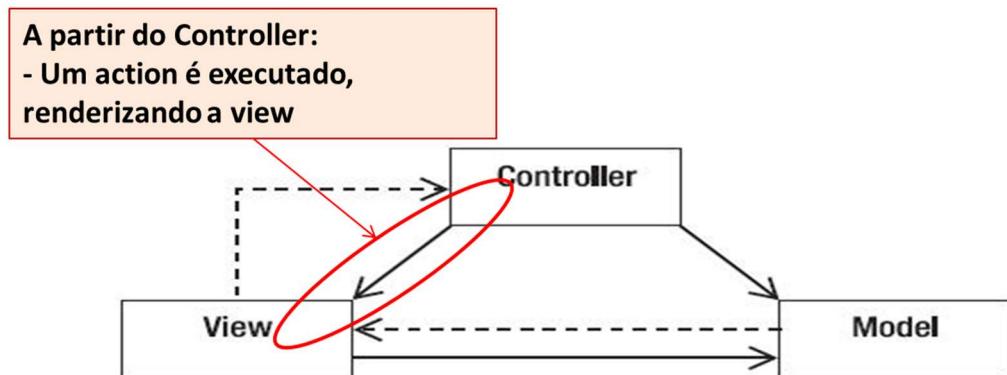
2.1. Introdução

Os controllers compõem a parte fundamental de uma aplicação MVC, principalmente por conta da arquitetura de mesmo nome.

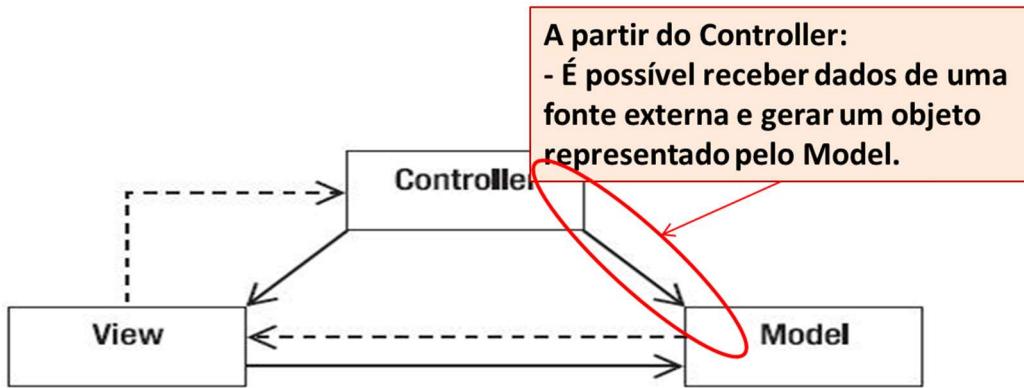
Um **controller** é na verdade uma classe contendo métodos capazes de gerar a camada de interação com o usuário. Estes métodos são chamados de **actions**.

2.1.1. Arquitetura MVC (Model – View – Controller)

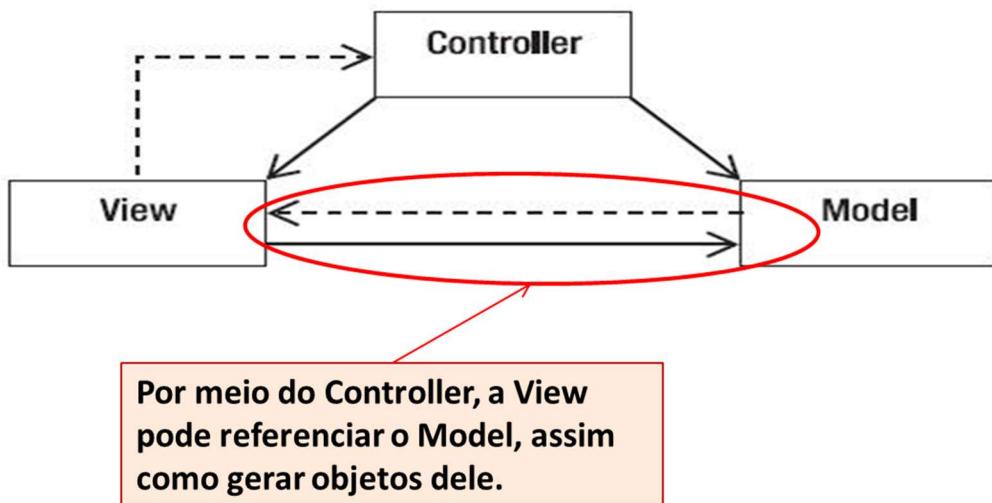
Esta arquitetura pode ser representada pelo esquema:



Quando as informações de uma classe (um objeto, na verdade) são necessárias, o controller o utiliza. Esta classe é o **Model**:



De forma semelhante, a camada de visualização (a view) pode referenciar o Model, de forma a criar o conteúdo da página contendo vínculos a ele:



Na aplicação, o tipo de retorno do método representando o action define o tipo de view que será criada. Podemos ter um resultado com conteúdo HTML, um conjunto de dados no formato JSON, o conteúdo de um PDF, e assim por diante:

Controller (uma classe)

Action (um método):

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

Index.cshtml

View
(um arquivo .cshtml)
Conteúdo HTML +
Razor

Se as informações de um objeto forem exibidas na view, este é passado como parâmetro para o action:

Controller (uma classe)

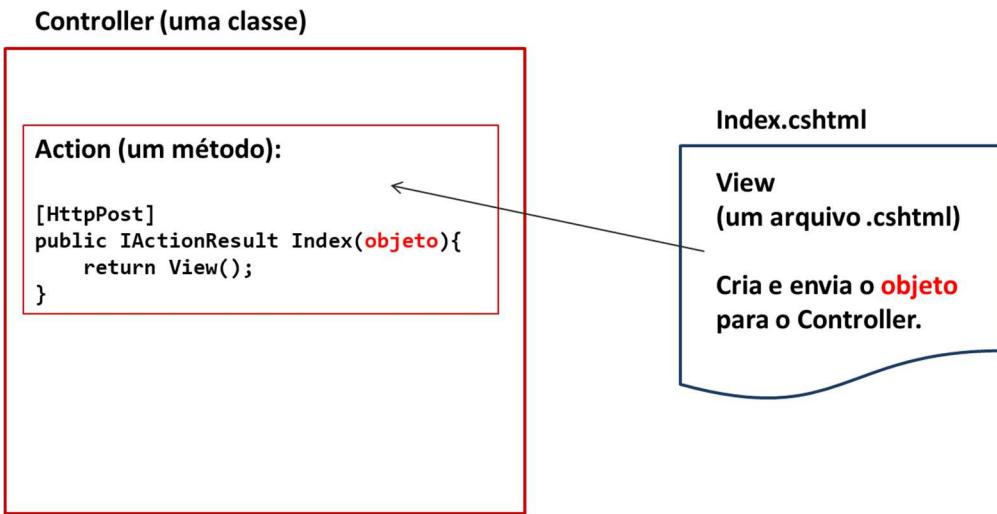
Action (um método):

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

Index.cshtml

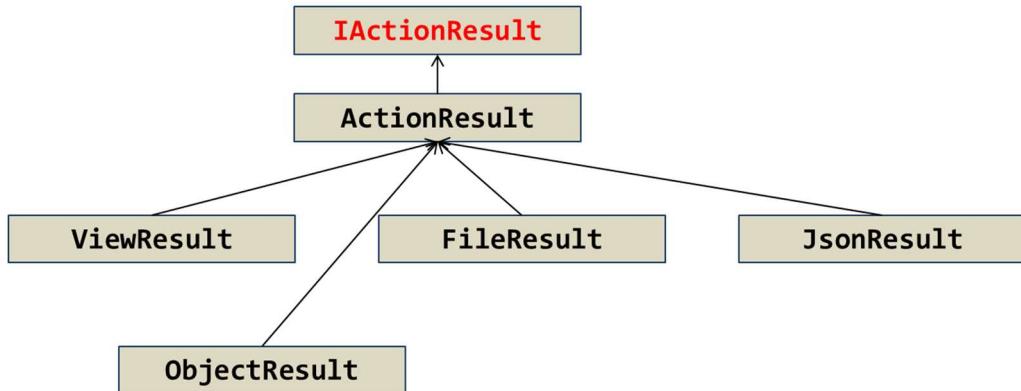
View
(um arquivo .cshtml)
Exibe os dados do
objeto informado no
Controller.

Quando uma página HTML precisar enviar os dados de um formulário para que o controller possa processá-los, fazemos o procedimento inverso: enviamos o objeto da view para o action:



2.1.2. Principais tipos de retorno de actions

O conteúdo a ser renderizado para o usuário depende do retorno que aplicamos aos nossos actions. O diagrama a seguir ilustra algumas das principais classes usadas nestes retornos:



Neste diagrama, temos:

IActionResult: Interface implementada por todas as classes. Representa todo tipo de retorno, assim como a classe **ActionResult**.

ViewResult: Tipo usado para renderizar necessariamente um conteúdo HTML.

FileResult: Adequado para apresentar o conteúdo de arquivos. Como exemplo, o usamos para retornar conteúdo de arquivo HTML no browser.

JsonResult: Retorna informações no formato Json. Adequado para webservices.

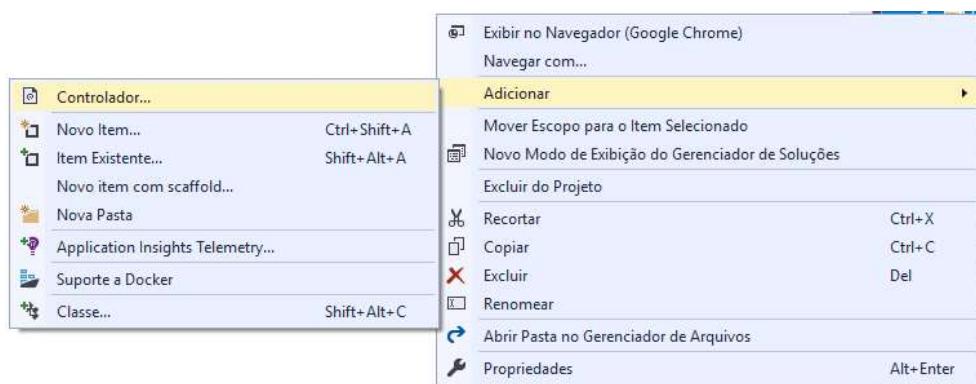
ObjectResult: Apresenta o conteúdo das propriedades e/ou retorno de métodos provenientes de instâncias criadas no controller.

2.2. Projeto: controllers e actions

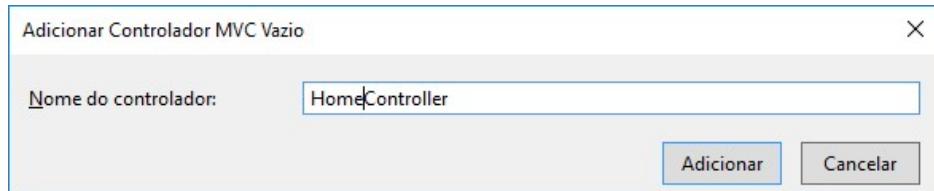
Vamos dar continuidade no projeto iniciado no Capítulo 1.

2.2.1. Definição do controller HomeController e das actions Arquivo e Resultado

Na pasta **Controllers**, clicar com o direito do mouse e adicionar um novo controlador, através da opção:



Se a classe **HomeController**, não tiver sido criada, cria-la. Na caixa de diálogos, selecionar a opção **Controlador MVC Vazio** e, em seguida, digitar apenas a palavra **Home**, mantendo o sufixo **Controller**:



Se você escolheu a opção “**Contas de Usuário Individual**” na criação do projeto, certamente esta classe estará presente. Em todo caso, limpar o conteúdo desta classe e manter o conteúdo a seguir:

```
namespace DesenvolvimentoWeb.Projeto02.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

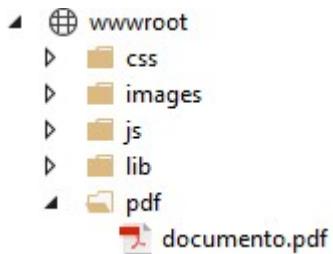
        public FileResult Arquivo()
        {
            return File("~/pdf/documento.pdf", "application/pdf");
        }

        public IActionResult Resultado()
        {
            ViewData["valor1"] = "Primeiro Resultado";
            ViewBag.Mensagem = "Mensagem gerada em " + DateTime.Now;

            return View();
        }
    }
}
```

Observe os retornos dos métodos.

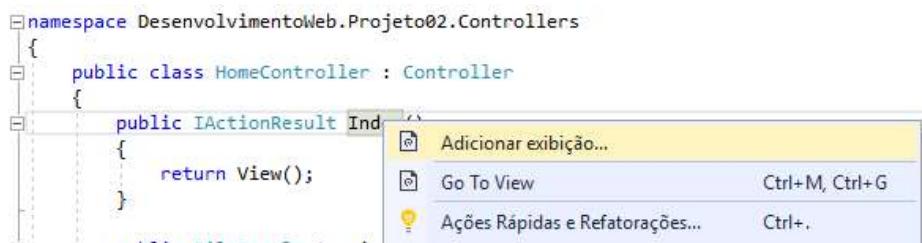
Para que possamos dar continuidade com esta estrutura, defina um arquivo PDF de sua escolha, nomeie-o como **documento.pdf** e coloque na pasta indicada a seguir:



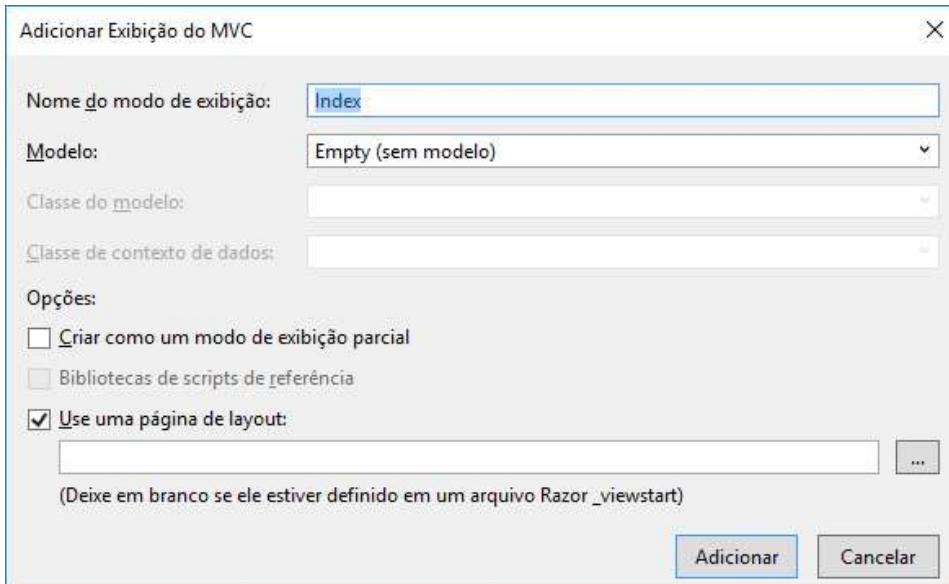
A pasta **wwwroot** representa o local de todo conteúdo estático, como arquivos javascript, css, e no nosso caso, de arquivos pdf. Observe que a pasta **pdf** foi criada.

2.2.2. Adicionando a view Index

Vamos agora definir as views para estes actions. Uma forma direta de fazer isso é clicando com o direito do mouse sobre o nome do action, e fornecer os dados necessários na interface. Para o action **Index**, definiremos o que será nossa página inicial da aplicação:



Deixe as demais opções como está mostrado:



Será criado o arquivo **Index.cshtml** na pasta **Views/Home**.

Seu conteúdo inicial será o mostrado a seguir:

```

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

<h1>Exemplos de Action</h1>

<ul>

    <li><a asp-controller="Home" asp-action="Arquivo">
        Visualizar PDF</a></li>
    <li><a asp-controller="Home" asp-action="Resultado">
        Visualizar Resultados ViewData/ViewBag</a></li>

</ul>

```

Aqui estamos usando as **Tag Helpers**, novo conceito disponível no Asp.Net Core.

Opcionalmente podemos usar os **Html Helpers** existentes desde as versões anteriores do Asp.Net MVC. Veja o exemplo a seguir:

```

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

```

```
<h1>Exemplos de Action</h1>

<ul>
    <li>@Html.ActionLink("Visualizar PDF", "Arquivo", "Home") </li>
    <li>@Html.ActionLink("Visualizar Resultados ViewData/ViewBag",
        "Resultado") </li>
</ul>
```

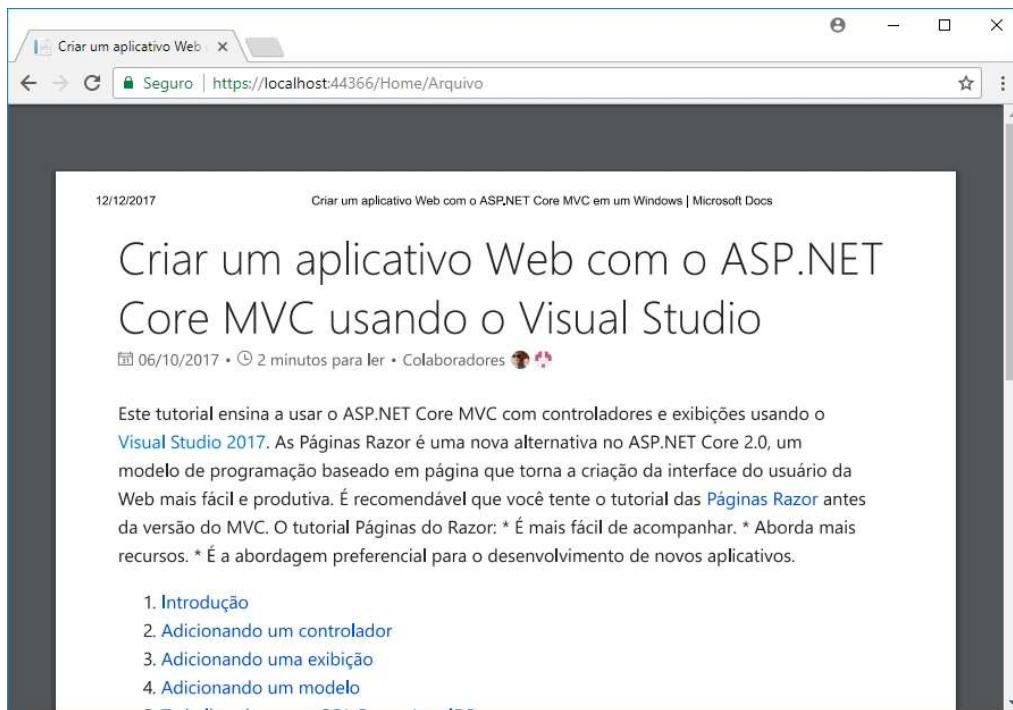
2.2.3. Executando a aplicação

No caso do action Arquivo, não temos uma view a ser definida, pois o conteúdo será o próprio PDF exibido no browser.

Ao executar a aplicação, temos os links:

Visualizar PDF
Visualizar Resultados ViewData/ViewBag

Clicando no primeiro link, teremos um resultado semelhante ao mostrado a seguir (não esqueça que o conteúdo do arquivo poderá mudar de projeto para projeto):



2.2.4. Definindo a view Resultado

Vamos criar a view para o action **Resultado**. Repita o procedimento de criação de views, clicando com o direito do mouse no método, mantendo as opções padrão.

Escreva o resultado a seguir:

```
@{  
    ViewData["Title"] = "Resultado";  
}  
  
<h2>Informações do Action</h2>  
  
<p>  
    @ViewData["valor1"]  
</p>  
<p>  
    @ViewBag.Mensagem  
</p>
```

Execute a aplicação, selecione o segundo link do Index, e veja o resultado.

3. Elaboração de Formulários (Views)

3.1. Introdução

Formulários são os elementos mais importantes de uma aplicação, especialmente em uma aplicação web. É através dos formulários que a aplicação interage com o usuário. Quando for necessário realizar um cadastro, uma consulta, ou uma simples conferência, é através de formulários que estas tarefas são executadas.

Neste capítulo estudaremos os critérios usados no Asp.Net Core para processar formulários.

3.1.1. O componente ViewData e ViewBag

Estes componentes são usados para transferir informações do controller para a view, sem a utilização de uma classe Model.

A diferença entre eles está na forma de utilização. Considere o action **Resultado**, desenvolvido no Capítulo 02:

```
public IActionResult Resultado()
{
    ViewData["valor1"] = "Primeiro Resultado";
    ViewBag.Mensagem = "Mensagem gerada em " + DateTime.Now;

    return View();
}
```

Neste exemplo usamos o componente **ViewData**. Ele usa uma chave como identificador de elementos (neste caso a chave é **valor1**).

O **ViewBag** usa propriedades dinâmicas ao invés de chaves.

Estes mesmos elementos devem ser referenciados na View, como forma de utilização:

```
@{  
    ViewData["Title"] = "Resultado";  
}  
  
<h2>Informações do Action</h2>  
  
<p>  
    @ViewData["valor1"]  
</p>  
<p>  
    @ViewBag.Mensagem  
</p>
```

3.1.2. A sintaxe do Razor

O Razor é um *engine* usado na elaboração de páginas. Existem diferentes *engines* (aspx, por exemplo) mas o Razor é considerado o de código mais limpo, mais intuitivo e fácil de usar. Por esta razão o Asp.Net o utiliza por padrão.

Todo elemento usando a sintaxe Razor inicia com “@”. Exemplos:

```
@{  
    ViewBag.Title = "About";  
}  
  
<hgroup class="title">  
    <h1>@ViewBag.Title.</h1>  
    <h2>@ViewBag.Message</h2>  
</hgroup>
```

Seguindo esta sintaxe, podemos definir também:

Links

```
<ul>  
    <li>@Html.ActionLink("Home", "Index", "Home")</li>  
    <li>@Html.ActionLink("About", "About", "Home")</li>  
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
```

```
</ul>
```

Iteração

```
@foreach (var item in Model) {  
    <tr>  
        <td>@Html.DisplayFor(modelItem => item.UserName)</td>  
        <td>@Html.DisplayFor(modelItem => item.Password)</td>  
        <td>@Html.DisplayFor(modelItem => item.RememberMe)</td>  
    </tr>  
}
```

Variáveis

```
@{ var total = 7; }  
{@{ var mensagem = "Bem vindo"; }  
  
<p>O valor da variável total é: @total </p>  
<p>O valor da mensagem é: @mensagem</p>  
  
{@  
    var titulo = "Asp.Net Core";  
    var data = DateTime.Now;  
    var resposta = titulo + " inicia em : " + data;  
}
```

Blocos de código

```
@{  
    var result = "";  
    if(IsPost)  
    {  
        result = "This page was posted using the Submit button.";  
    }  
    else  
    {  
        result = "This was the first request for this page.";  
    }  
}
```

3.2. Métodos para Requisição HTTP: GET e POST

Toda requisição realizada a um servidor de aplicações ocorre através de um dos diversos métodos HTTP. Os mais importantes são os métodos GET e POST:

- **Método GET:** Obtém informações do servidor – usualmente um documento – especificando qual documento e eventuais parâmetros através de um URI (*Universal Resource Identifier*).
- **Método POST:** Envia informações para o servidor processar ou armazenar.

Cabeçalhos de requisição

Para que o servidor tenha informações da requisição, o processo consiste no envio de informações da requisição através de cabeçalhos (que normalmente não são disponibilizados para visualização no desenvolvimento da aplicação). Vamos conhecer os cabeçalhos.

HTTP GET

Header	<pre> GET /hello/Hello.cshtml?userid=Joao HTTP/1.1 Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, application/x-shockwave-flash, /*/ Accept-Language: en-us Accept-Encoding: gzip, deflate User-Agent: Mozilla/57.0 (compatible; MSIE 10.0; Windows 10; SV1) Host: 127.0.0.1 Connection: Keep-Alive </pre>
	
Linha em branco	

Passagem de parâmetros:

<http://localhost:1234/hello/Hello.cshtml?userid=10&nome=Joao>

- Cada parâmetro tem um identificador, seguido pelo sinal de igual e valor.
- Cada par parâmetro/valor é separado pelo caractere &
- Todos os parâmetros são separados da URL através do caractere ? (somente quando utilizado GET)
- Toda URL é *case sensitive*

HTTP POST

```

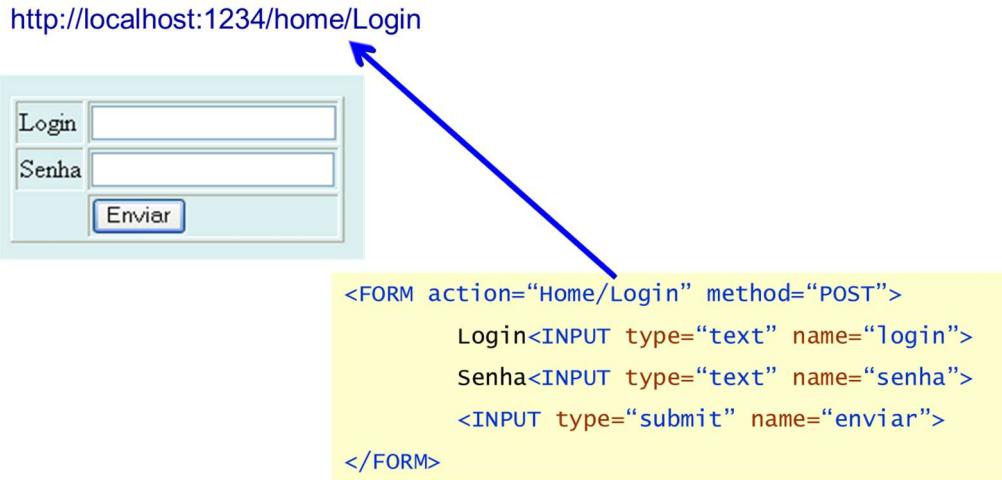
POST /hello/Hello.cshtml HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-shockwave-flash, /*
Referer: http://localhost:1234/hello/hello.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/57.0 (compatible; MSIE 10.0; Windows 10; SV1)
Host: 127.0.0.1
Content-Length: 12
Connection: Keep-Alive
Cache-Control: no-cache

Header { }

Dados [userId=Joao]

```

Utilização



3.3. Projeto: Views

3.3.1. Definição de um Model

Na pasta **Models**, criar a classe **Produto** (a ser usada no action **ExibirProduto()**):

```

namespace DesenvolvimentoWeb.Projeto02.Models
{
    public class Produto
    {
        [Display(Name = "Código")]
        public int Código { get; set; }

        [Display(Name = "Descrição")]

```

```

        public string Descricao { get; set; }

        [Display(Name = "Preço")]
        public double Preco { get; set; }

        [DataType(DataType.Date)]
        [Display(Name = "Data Fabricação")]
        public DateTime DataFabricacao { get; set; }
    }
}

```

Observe os atributos de validação na classe.

3.3.2. Inclusão de novos actions

No projeto, vamos incluir mais alguns métodos no controller **HomeController**.

Adicionar os actions a seguir na classe:

```

[HttpGet]
public IActionResult Calculadora(double? n1, double? n2)
{
    ViewBag.n1 = n1;
    ViewBag.n2 = n2;
    return View();
}

[HttpPost]
public IActionResult Calculadora(IFormCollection form)
{
    double v1 = double.Parse(form["valor1"]);
    double v2 = double.Parse(form["valor2"]);

    double resultado = v1 + v2;
    ViewBag.Resultado = "Resultado da soma: " + (v1 + v2);
    return View();
}

//Actions usados com a classe Produto
[HttpGet]
public IActionResult IncluirProduto01()
{
    return View();
}

[HttpPost]
public IActionResult IncluirProduto01(Produto produto)

```

```

{
    return View();
}

public IActionResult ExibirProduto()
{
    Produto produto = new Produto()
    {
        Codigo = 25,
        Descricao = "Bicicleta",
        Preco = 500,
        DataFabricacao = Convert.ToDateTime("20/10/2016")
    };

    return View(produto);
}

```

Observe os atributos **[HttpGet]** e **[HttpPost]** em alguns actions.

É importante usar os recursos do Visual Studio para importar os namespaces que forem necessários. Uma forma comum é colocar o cursor na classe e pressionar **[Ctrl + .]** (Ctrl + ponto).

3.3.3. Definição das Views

Vamos definir a View **Calculadora**, sem modelo (empty) como nas views que fizemos até agora:

```

@{
    ViewData["Title"] = "Calculadora";
}

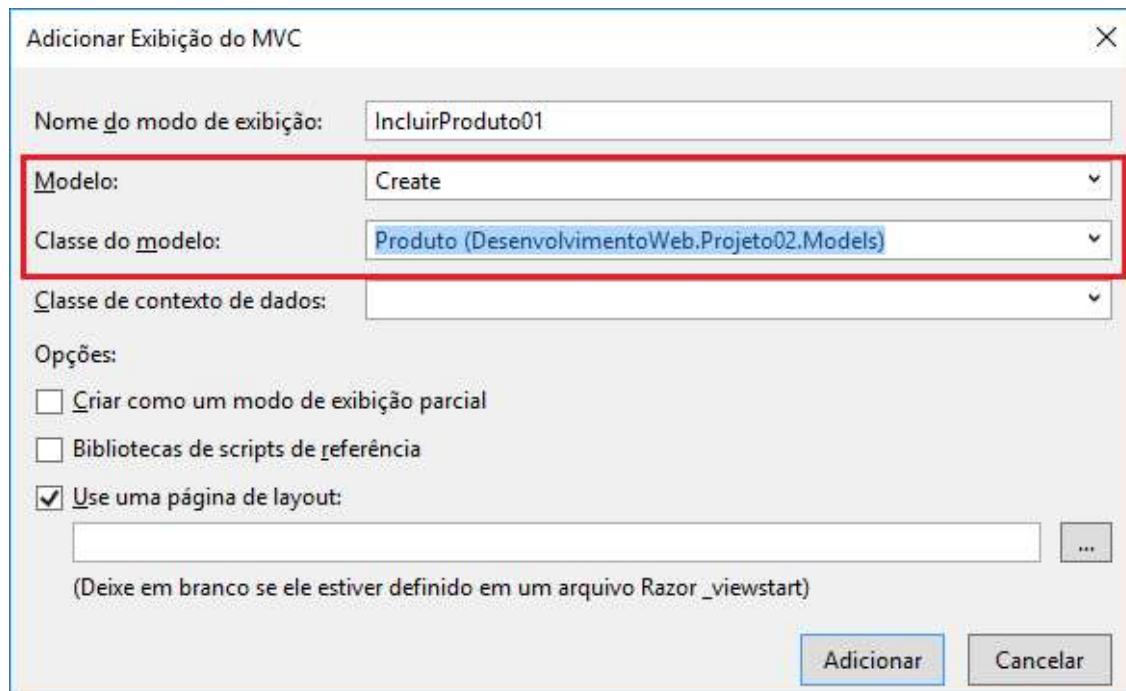
<h2>Calculadora</h2>

@using (Html.BeginForm())
{
    @Html.Label("Primeiro Valor:")
    @Html.TextBox("valor1", (double?)ViewBag.n1)
    @Html.Label("Segundo Valor:")
    @Html.TextBox("valor2", (double?)ViewBag.n2)

    <input type="submit" value="Calcular" />
    <br />
    @ViewBag.Resultado
}

```

Para criar a View para os actions **IncluirProduto01** e **ExibirProduto**, vamos adicionar o model produto que definimos neste projeto. O procedimento de inclusão de View muda um pouco:



O procedimento deverá ser o mesmo para **ExibirProduto**.

Não se preocupar com o conteúdo gerado no momento. Manter o conteúdo de **IncluirProduto01.cshtml** e **ExibirProduto.cshtml** como no exemplo abaixo:

IncluirProduto01.cshtml

```
@model DesenvolvimentoWeb.Projeto02.Models.Produto

@{
    ViewData["Title"] = "IncluirProduto01";
}



## Incluir Produto - Parte 01


@using (Html.BeginForm())
{
    @Html.LabelFor(m => m.Codigo)
    @Html.EditorFor(m => m.Codigo)<br />

    @Html.LabelFor(m => m.Descricao)
```

```

@Html.EditorFor(m => m.Descricao)<br />
@Html.LabelFor(m => m.Preco)
@Html.EditorFor(m => m.Preco)<br />
@Html.LabelFor(m => m.DataFabricacao)
@Html.EditorFor(m => m.DataFabricacao)<br />
<input type="submit" value="Enviar" />
}

```

ExibirProduto.cshtml

```

@model DesenvolvimentoWeb.Projeto02.Models.Produto
 @{
     ViewData["Title"] = "ExibirProduto";
 }

<h2>Exibir Produto</h2>
@Html.LabelFor(m => m.Codigo)
@Html.DisplayFor(m => m.Codigo) <br />

@Html.LabelFor(m => m.Descricao)
@Html.DisplayFor(m => m.Descricao) <br />

@Html.LabelFor(m => m.Preco)
@Html.DisplayFor(m => m.Preco) <br />

@Html.LabelFor(m => m.DataFabricacao)
@Html.DisplayFor(m => m.DataFabricacao) <br />

```

No arquivo **Index.cshtml** deste controller, incluir links para estes novos actions:

```

<li><a asp-controller="Home" asp-action="IncluirProduto01">
Formulário de Produtos</a></li>
<li><a asp-controller="Home" asp-action="Calculadora">
    Calculadora</a></li>
<li><a asp-controller="Home" asp-action="ExibirProduto">
    Exibir Produto</a></li>

```

Executar a aplicação e testar estes novos links.

Ainda não temos uma aplicação completa, mas serve para ilustrar a utilização de formulários, especialmente no exemplo do action **Calculadora**.

4.Conceitos de Delegates

4.1. Introdução

Da mesma forma que temos as classes que geram referencias a objetos, temos os delegates que produzem referencias a funções, ou a operações.

Suponha que alguém pergunte: Se desejarmos escrever um método que recebe como parâmetros dois inteiros, e retorna um inteiro, quantas possibilidades temos? Observe que não estamos preocupados com uma aplicação em específico, e sim com a sintaxe. Existe uma infinidade de métodos que atendem a este requisito funcional, mas sabemos que se tornaria inviável escrevermos todos os métodos para atender a todas as possibilidades.

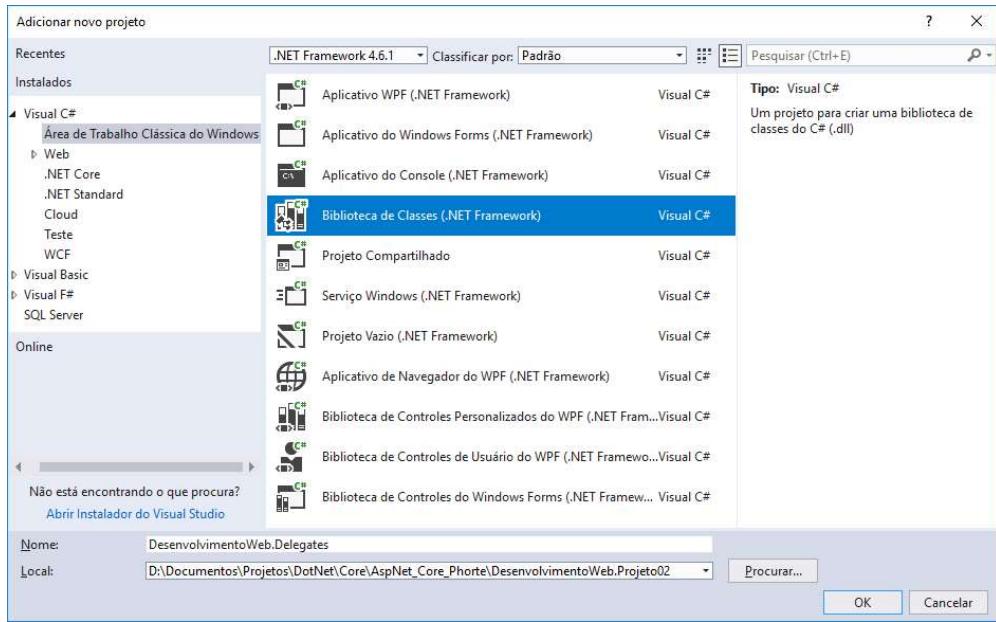
Muitas vezes, a operação é decidida pelo programador de acordo com a necessidade.

É sobre isso que falaremos neste capítulo. Os delegates nos atendem nesta questão.

4.2. Definindo um novo projeto

Vamos dar uma pausa no nosso projeto Asp.Net Core, e vamos definir um novo projeto para explicar os conceitos dos delegates. Nosso novo projeto nem será do tipo web. Uma aplicação console será suficiente para esta tarefa.

Clicando com o direito do mouse sobre o solution (solução), selecionar a opção **Adicionar > Novo Projeto**. Escolher a opção:



O nome do projeto deverá ser **DesenvolvimentoWeb.Delegates**.

4.2.1. Definindo os delegates

No projeto temos apenas a classe **Program.cs**. Além do método **Main()** já disponível na classe, vamos adicionar os dois métodos a seguir:

```
static void ExibirMensagem01()
{
    Console.WriteLine("Primeira Mensagem");
}

static void ExibirMensagem02()
{
    Console.WriteLine("Segunda Mensagem");
}
```

Trata-se de dois métodos sem parâmetros e sem retorno. Não vamos executar estes métodos diretamente. Vamos definir um delegate para executá-los.

Fora da classe, mas dentro do namespace, incluir a instrução:

```
public delegate void Mensagem();
```

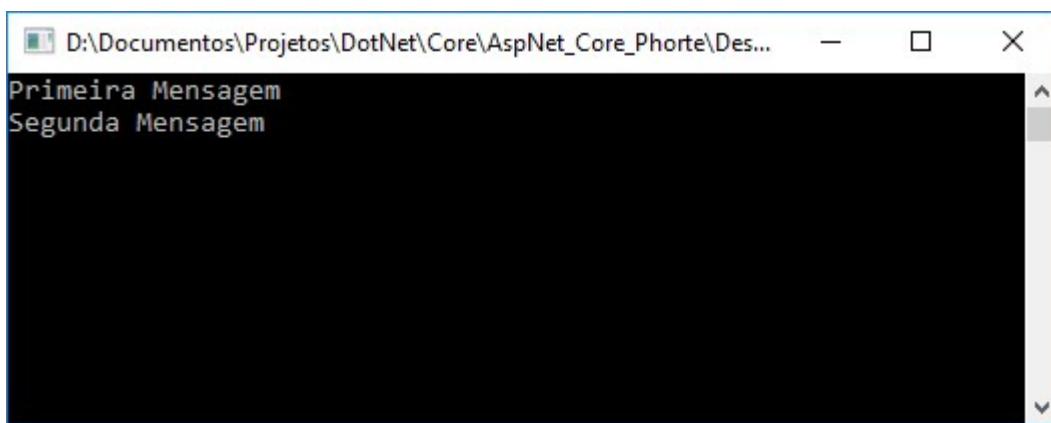
Desconsiderando a palavra reservada **delegate**, verifique que o resto se parece com um método, contendo a mesma definição dos dois métodos anteriores.

O termo **delegate** define um novo tipo de dado, e como tal, serve para declararmos variáveis.

No método main(), definir a referência aos métodos por meio do delegate:

```
Mensagem msg = new Mensagem(ExibirMensagem01);
msg += ExibirMensagem02;
msg();
```

Executando esta aplicação, vemos o resultado (você deve tornar este projeto padrão, já que nossa solução agora possui dois projetos):



Executamos a referência ao delegate, como se ela fosse o método. Como referenciamos o primeiro método, e depois anexamos o segundo, os endereços dos dois métodos foram acessados.

4.2.2. Trabalhando com delegates e expressões lambda

Vamos entender os delegates: operações definidas dinamicamente.

Adicione o seguinte delegate no namespace, logo abaixo de **Mensagem**:

```
public delegate double Calcular(double x, double y);
```

Desenvolveremos a expressão lambda a partir de um método. Se fosse um método tradicional, teríamos o seguinte exemplo:

```
static double Calcular(double x, double y)
{
    return x + y;
}
```

Observe que consideramos como operação a soma dos dois parâmetros, mas poderia ser qualquer outra.

Como a proposta do delegate é referenciar apenas a operação, vamos fazer alguns ajustes.

- Não importa o nome do método. Então ele pode ser omitido:

```
static double (double x, double y)
{
    return x + y;
}
```

- Sabemos que o retorno é double, e que os parâmetros necessariamente são do tipo double. Eles também podem ser omitidos:

```
( x, y)
{
    return x + y;
}
```

- Como existe apenas uma instrução, o comando return pode ser omitido:

```
( x, y) { x + y; }
```

- Havendo apenas uma instrução, as chaves podem ser omitidas:

```
( x, y) x + y;
```

- Para justificar o resultado anterior, usamos o operador => entre os dois termos:

```
( x, y ) => x + y;
```

Esta é nossa expressão lambda!

Para usá-la, podemos escrever no método Main():

```
Calcular calc = (x, y) => x + y; //expressão lambda
double resultado = calc(12, 15);
Console.WriteLine("Resultado: " + resultado);
```

De forma geral, uma expressão lambda possui a sintaxe:

(parâmetros) => resultado;

cuja leitura pode ser feita:

“Uma lista de parâmetros PRODUZ o resultado desejado.”

5.Views Fortemente Tipadas

5.1. Introdução

Já tivemos a oportunidade de usar uma classe chamada **Produto** para ilustrar o conceito de formulários e de passagem de objeto para uma view.

Vamos relembrar o que vimos:

Classe HomeController

```
[HttpGet]
public IActionResult IncluirProduto01()
{
    return View();
}
```

View

```
@model DesenvolvimentoWeb.Projeto02.Models.Produto

@{
    ViewData["Title"] = "IncluirProduto01";
}

<h2>Incluir Produto - Parte 01</h2>
@using (Html.BeginForm())
{
    @Html.LabelFor(m => m.Codigo)
    @Html.EditorFor(m => m.Codigo)<br />

    @Html.LabelFor(m => m.Descricao)
    @Html.EditorFor(m => m.Descricao)<br />

    @Html.LabelFor(m => m.Preco)
    @Html.EditorFor(m => m.Preco)<br />

    @Html.LabelFor(m => m.DataFabricacao)
    @Html.EditorFor(m => m.DataFabricacao)<br />
```

```
<input type="submit" value="Enviar" />  
}
```

O Asp.net Core, apresentam as **Tag Helpers** e os **Html Helpers**.

Se forem utilizados Html Helpers (como no exemplo), a maioria dos seus métodos utiliza expressões lambda como parâmetros.

No exemplo apresentado, criamos um formulário com elementos definidos em Html Helpers, e quando o submetemos, o controller o recebe e o processa.

Vamos escrever o código para o action anotado com **[HttpPost]**:

5.2. Projeto: views fortemente tipadas

Localizar o método

```
[HttpPost]  
public IActionResult IncluirProduto01(Produto produto)  
{  
    return View();  
}
```

Verifique que ele recebe um parâmetro do tipo Produto.

A View definiu os valores das propriedades deste objeto por meio das expressões lambda. O objeto criado foi enviado para ao método **IncluirProduto01(Produto)**.

Vamos apresentar uma solução para este problema que havia ficado pendente.

O método ficará assim:

```
[HttpPost]  
public IActionResult IncluirProduto01(Produto produto)  
{  
    StreamWriter writer = new  
    StreamWriter("wwwroot/pdf/documento.txt", false,  
    Encoding.UTF8);
```

```

writer.WriteLine("Codigo: " + produto.Codigo);
writer.WriteLine("Descrição: " + produto.Descricao);
writer.WriteLine("Preço: " + produto.Preco.ToString("c"));
writer.WriteLine("Data Fabricação: " + produto.DataFabricacao);
writer.Close();

//return View();
return File("~/pdf/documento.txt", "text/plain");

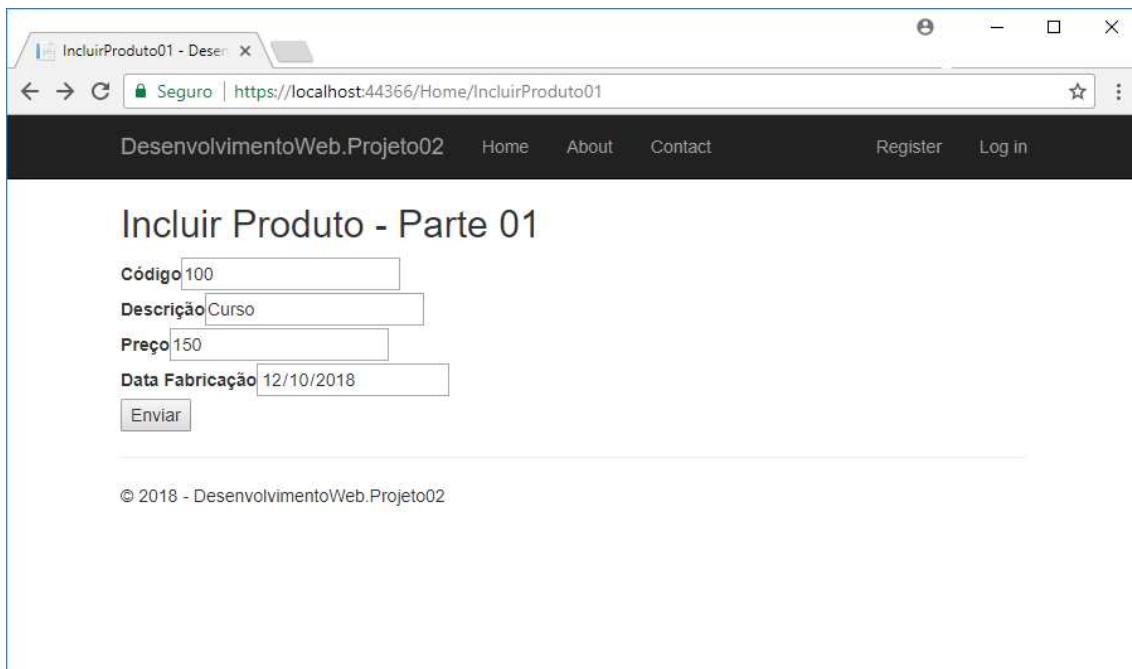
}

```

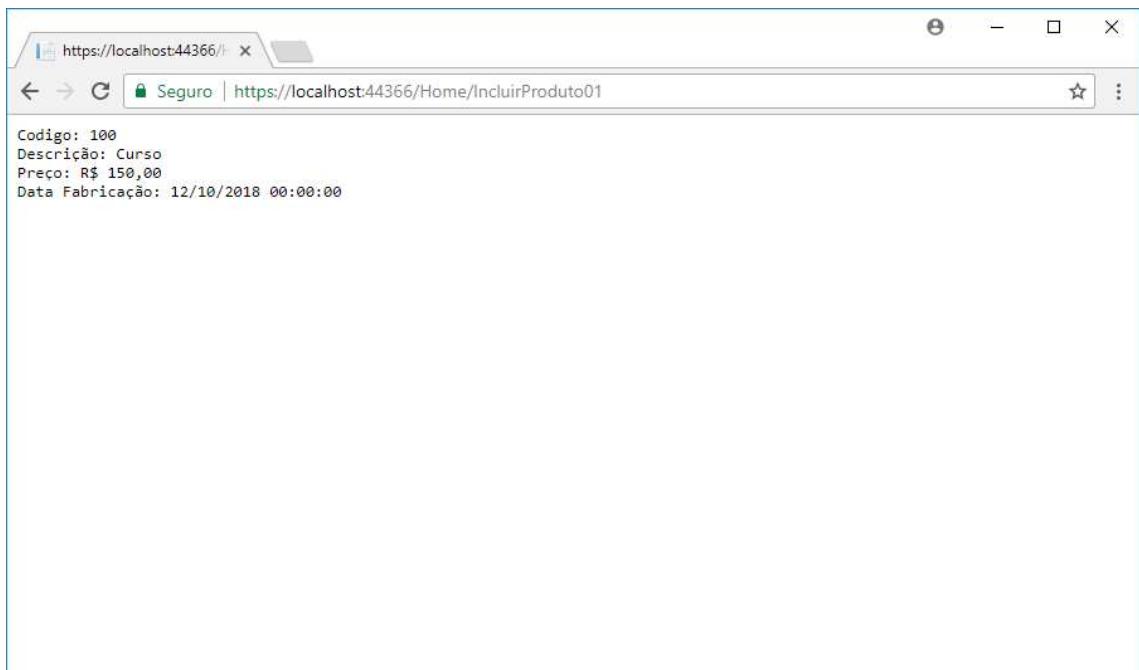
Este código cria um arquivo com as informações do produto, e o exibe no browser.

Observe a peculiaridade dos caminhos usados tanto na criação do arquivo como na sua leitura.

Um exemplo de execução da aplicação é mostrado a seguir:



Resultado após submeter o formulário:



Se você buscar o arquivo na pasta **pdf**, ele estará lá!

6. Conceitos do CSS (Cascading Style Sheet)

6.1. Introdução

Quando visualizamos o HTML no browser, cada conteúdo marcado com uma tag é exibido de uma forma diferente.

Podemos alterar este estilo padrão como os navegadores apresentam cada **tag**, para isso usamos o CSS.

Existem algumas tags HTML que foram criadas com este mesmo objetivo do CSS. Alguns exemplos são: **** e **<i>**.

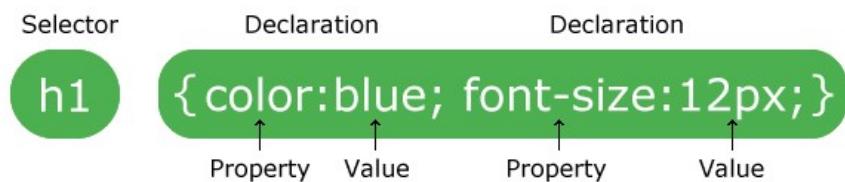
O CSS se popularizou por permitir a formatação de estilos no HTML permitindo uma melhor experiência com o usuário.

6.2. Sintaxe

A forma geral do CSS está ilustrada a seguir:

```
seletor {  
    propriedade1: valor1;  
    propriedade2: valor2;  
    ...  
    propriedade_n: valor_n;  
}
```

No link https://www.w3schools.com/css/css_syntax.asp temos uma ilustração interessante sobre o CSS:



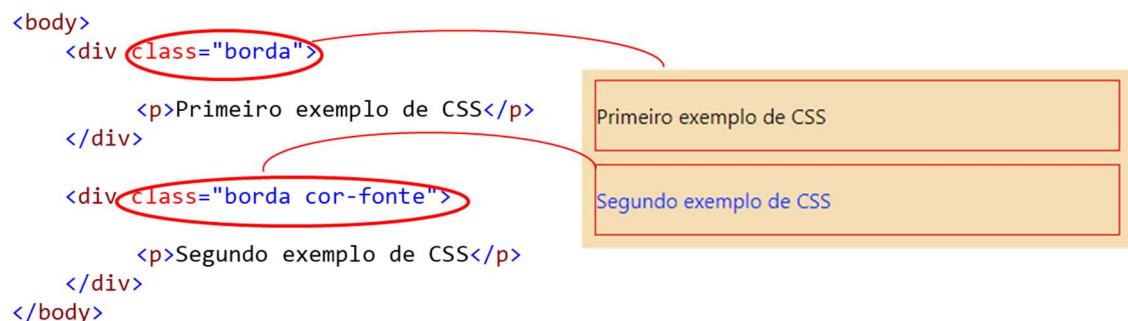
Neste mesmo link podemos obter uma rica documentação sobre CSS. Vale a pena explorá-lo.

Alguns exemplos de utilização do CSS é mostrada a seguir:

Conteúdo CSS

```
body {  
    background-color: #F5DEB3;  
    font-family:'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;  
}  
.borda {  
    margin: 10px;  
    border: solid 1px #ff0000;  
}  
.cor-fonte {  
    color: #0026ff;  
}
```

Página HTML utilizando as classes CSS acima, com seu resultado



6.3. Conhecendo o Bootstrap

O Bootstrap é um **framework front-end** de código aberto, desenvolvido pela equipe do Twitter.

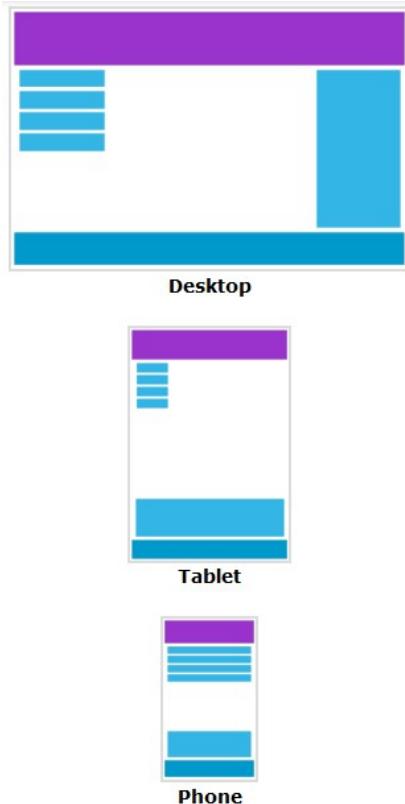
É totalmente compatível com HTML5 e CSS3. Ele possibilita a criação de layouts responsivos através do uso de grids e a intervenção de código Javascript, permitindo que seu conteúdo seja organizado em até 12 colunas e que comporte-se de maneira diferente para cada resolução.

Como principal vantagem podemos destacar sua versatilidade na aplicação em páginas web, sem a necessidade de escrever longas instruções CSS.

A desvantagem é que se torna necessário aprender sobre os elementos CSS adequados e sua utilização. Mas temos a documentação oficial do Bootstrap que nos fornece uma rica quantidade de exemplos. Basta saber como usar CSS.

É importante conhecer e entender suas funcionalidades para saber os momentos certos de utilizá-lo.

O bootstrap permite a criação de design responsivo com facilidade. Veja a imagem a seguir:



Esta imagem pode ser encontrada no link
https://www.w3schools.com/css/css_rwd_intro.asp

A documentação com exemplos do bootstrap está disponível no site:
<http://getbootstrap.com/docs/4.0/getting-started/introduction/>

6.3.1. Aplicando o Bootstrap em uma view

Vamos ver um exemplo de aplicação do bootstrap. Considere um trecho de código HTML5:

```

<div class="container">
    <h2>Exemplo bootstrap</h2>

    <div class="form-horizontal col-md-6">
        <div class="form-group">
            <label class="control-label col-md-2">Num. Documento</label>
            <div class="col-md-10">
                <input type="text" class="form-control" />
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Enviar" class="btn btn-default" />
            </div>
        </div>
    </div>
</div>

```

Este formulário produzirá o resultado:

Exemplo bootstrap

A screenshot of a web page titled "Exemplo bootstrap". It contains a form with a single text input field. The input field has a placeholder "Num. Documento" and a blue border. Below the input field is a blue rectangular button with the text "Enviar" in white.

É necessário referenciar o arquivo do bootstrap, ou mesmo os arquivos definidos por nós, para que as páginas possam reconhecer seus elementos.

No projeto teremos a oportunidade de ver a aplicação do Bootstrap tanto no layout como nas views, e como podemos referenciar os arquivos.

6.4. Projeto: CSS e Bootstrap

Vamos continuar nosso projeto. Nesta fase olharemos mais de perto o layout gerado pelo Visual Studio, e os elementos onde podemos interferir.

Abra o **HomeController** e adicione os dois actions:

```

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

```

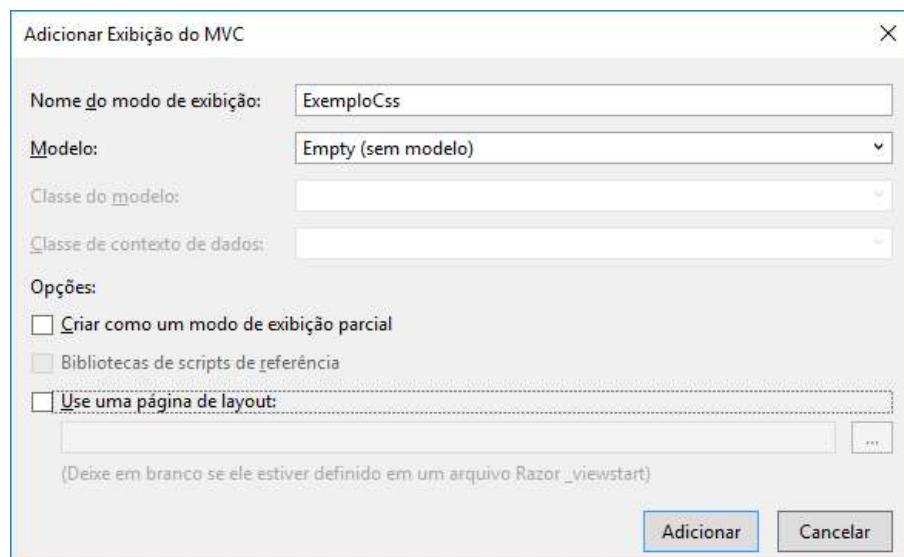
```

}

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

```

Para gerar a view **ExemploCss.cshtml**, não escolhemos nenhum layout e nenhum model. Use esta opção:



Na pasta **wwwroot**, crie a pasta **css** (se não existir) e acrescente o arquivo **estilos.css** com o conteúdo:

```

body {
    background-color: #F5DEB3;
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}

.borda {
    margin: 10px;
    border: solid 1px #ff0000;
}

.cor-fonte {
    color: #0026ff;
}

```

Na view escreva o código:

```

@{
    Layout = null;
}

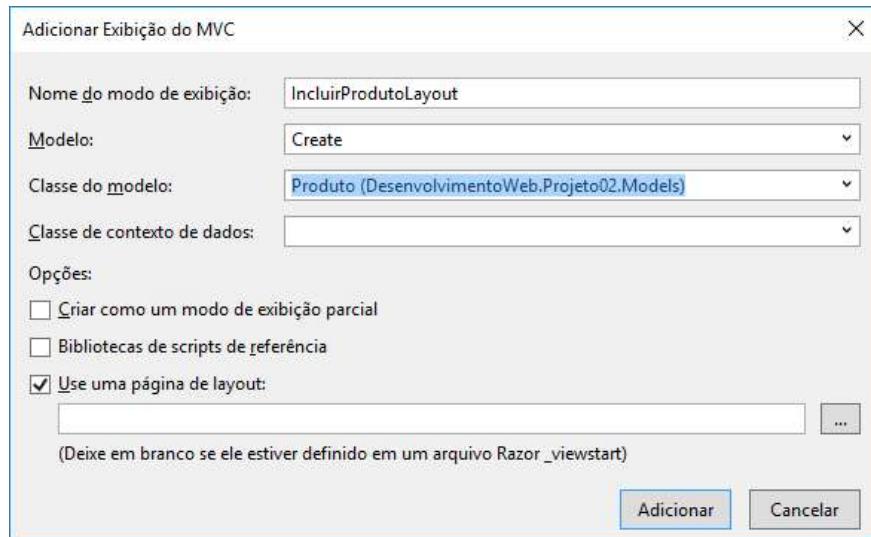
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href("~/css/estilos.css" rel="stylesheet" />
    <title>ExemploCss</title>
</head>
<body>
    <div class="borda">
        <p>Primeiro exemplo de CSS</p>
    </div>

    <div class="borda cor-fonte">
        <p>Segundo exemplo de CSS</p>
    </div>
</body>
</html>

```

Agora vamos criar a view **IncluirProdutoLayout.cshtml**. Neste caso manteremos o layout e o model **Produto**:



Analise o conteúdo gerado:

```
@model DesenvolvimentoWeb.Projeto02.Models.Produto
```

```
@{
```

```

        ViewData["Title"] = "IncluirProdutoLayout";
    }

<h2>IncluirProdutoLayout</h2>

<h4>Produto</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="IncluirProdutoLayout">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Codigo" class="control-label"></label>
                <input asp-for="Codigo" class="form-control" />
                <span asp-validation-for="Codigo" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Descricao" class="control-label"></label>
                <input asp-for="Descricao" class="form-control" />
                <span asp-validation-for="Descricao" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Preco" class="control-label"></label>
                <input asp-for="Preco" class="form-control" />
                <span asp-validation-for="Preco" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="DataFabricacao" class="control-label"></label>
                <input asp-for="DataFabricacao" class="form-control" />
                <span asp-validation-for="DataFabricacao" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

```

O destaque é nas classes CSS geradas a partir do bootstrap.

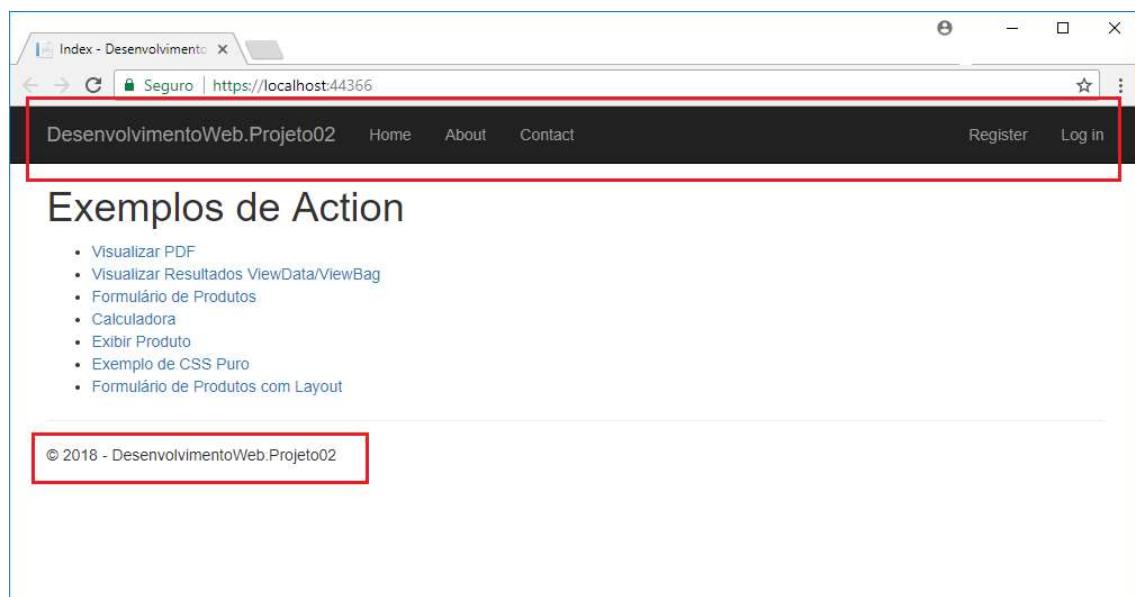
No arquivo **Index.cshtml** adicionar dois links para estes novos recursos:

```
<li><a asp-controller="Home" asp-action="Arquivo">Visualizar  
PDF</a></li>  
<li><a asp-controller="Home" asp-action="Resultado">  
    Visualizar Resultados ViewData/ViewBag</a></li>  
<li><a asp-controller="Home" asp-action="IncluirProduto01">  
    Formulário de Produtos</a></li>  
<li><a asp-controller="Home" asp-  
action="Calculadora">Calculadora</a></li>  
<li><a asp-controller="Home" asp-action="ExibirProduto">Exibir  
Produto</a></li>  
<li><a asp-controller="Home" asp-action="ExemploCss">Exemplo de CSS  
Puro</a></li>  
<li><a asp-controller="Home" asp-action="IncluirProdutoLayout">  
    Formulário de Produtos com Layout</a></li>
```

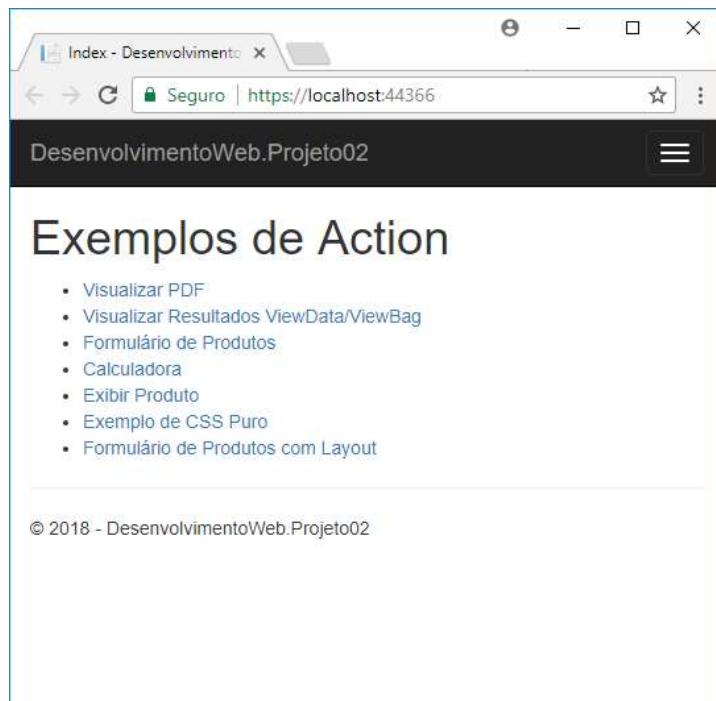
Ao executar a aplicação podemos ver as diferenças entre as views com e sem conteúdo css.

6.4.1. Projeto: layout do projeto

Em praticamente todos os exemplos que vimos até agora, pudemos notar a presença de um menu de opções e um rodapé na aplicação:



Se reduzíssemos a largura, ou se visualizássemos no formato de celular, o menu se altera:



Este comportamento caracteriza a **responsividade**, ou seja, o layout se adaptando de acordo com o dispositivo. Em celulares o menu muda para o formato suspenso.

Este menu se encontrou presente nas nossas views porque ele tá definido no layout.

6.4.2. Conhecendo o layout da aplicação

O Asp.Net MVC e o Asp.Net Core permitem a utilização de layout na aplicação de forma bastante simples. Na pasta **Views/Shared** temos um arquivo chamado **_Layout.cshtml** e na pasta Views, um arquivo chamado **_ViewStart.cshtml**.

Quando executamos um action, a view é renderizada, mas antes deste processo, o conteúdo do arquivo **_ViewStart.cshtml** é chamado. Seu conteúdo é:

```
@{  
    Layout = "_Layout";  
}
```

Ou seja, ele chama o layout antes de todas as views.

Se analisarmos o conteúdo do arquivo **_Layout.cahtml**, vemos uma grande quantidade de instruções HTML com referencias a arquivos CSS e Javascript.

Abrir este arquivo estudar seu conteúdo e deixa-lo com a configuração indicada a seguir. Vamos aproveitar e destacar alguns pontos importantes:

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <meta name="viewport" content="width=device-width, initial-  
scale=1.0" />  
    <title>@ ViewData["Title"] - DesenvolvimentoWeb.Projeto02</title>  
  
    <environment include="Development">  
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />  
        <link rel="stylesheet" href="~/css/site.css" />  
    </environment>  
    <environment exclude="Development">
```

```

        <link rel="stylesheet"
      href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.mi
n.css"
          asp-fallback-
      href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only"
          asp-fallback-test-property="position"
          asp-fallback-test-value="absolute" />
      <link rel="stylesheet" href="~/css/site.min.css"
          asp-append-version="true" />
    </environment>
  </head>
<body>
  <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle"
          data-toggle="collapse" data-target=".navbar-
collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-area="" asp-controller="Home"
          asp-action="Index" class="navbar-brand">
          Aplicação Web - Eventos</a>
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a asp-area="" asp-controller="Home"
            asp-action="Index">Home</a></li>
          <li><a asp-area="" asp-controller="Eventos"
            asp-action="Index">Gestão de Eventos</a></li>
        </ul>
      </div>
    </div>
  </nav>
  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; @DateTime.Now.Year - Desenvolvimento Web</p>
    </footer>
  </div>

  <environment include="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
  </environment>
  <environment exclude="Development">

```

```

<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-
2.2.0.min.js"
       asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
       asp-fallback-test="window.jQuery"
       crossorigin="anonymous"
       integrity="sha384-
K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfhII9k8z8l9ggpc8X+Ytst4yBo/hH+8FK">
</script>
<script
src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
       asp-fallback-
src="~/lib/bootstrap/dist/js/bootstrap.min.js"
       asp-fallback-test="window.jQuery && window.jQuery.fn
&& window.jQuery.fn.modal"
       crossorigin="anonymous"
       integrity="sha384-
Tc5IQib027qvyjSMfHjOMaLkfUWxZxUPnCJA7l2mCWNIPG9mGCD8wGNICPD7Txz">
</script>
<script src("~/js/site.min.js" asp-append-
version="true"></script>
</environment>

[@RenderSection("Scripts", required: false)
</body>
</html>

```

Alguns pontos que merecem destaque:

1. Configuração do padrão de caracteres para aceitar acentuações.

```
<meta charset="utf-8" />
```

2. Configuração da proporção dos caracteres em telas grandes em relação à telas de celulares.

```
<meta name="viewport" content="width=device-width, initial-
scale=1.0" />
```

3. inclusão dos arquivos **css** disponíveis nas pastas estáticas localizadas em **wwwroot**. Observe a inclusão da biblioteca bootstrap (já incluída na criação do projeto):

```

<environment include="Development">
    <link rel="stylesheet"
href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href "~/css/site.css" />
</environment>

```

4. Configuração do menu principal. O controller **Eventos** ainda não foi criado, mas podemos indicá-lo:

```
<a asp-area="" asp-controller="Home" asp-action="Index"
class="navbar-brand">DesenvolvimentoWeb.Projeto02</a>

<ul class="nav navbar-nav">
    <li><a asp-area="" asp-controller="Home" asp-action="Index">
Home</a></li>
    <li><a asp-area="" asp-controller="Eventos" asp-action="Index">
Gestão de Eventos</a></li>
</ul>
```

5. Local onde o conteúdo da view real é inserido na execução da aplicação.

```
@RenderBody()
```

Como o nome sugere, ao executar um action, este renderiza uma view. O conteúdo desta view é mesclado no local onde este método é chamado.

6. Definição do rodapé (ele ficará abaixo do conteúdo da view):

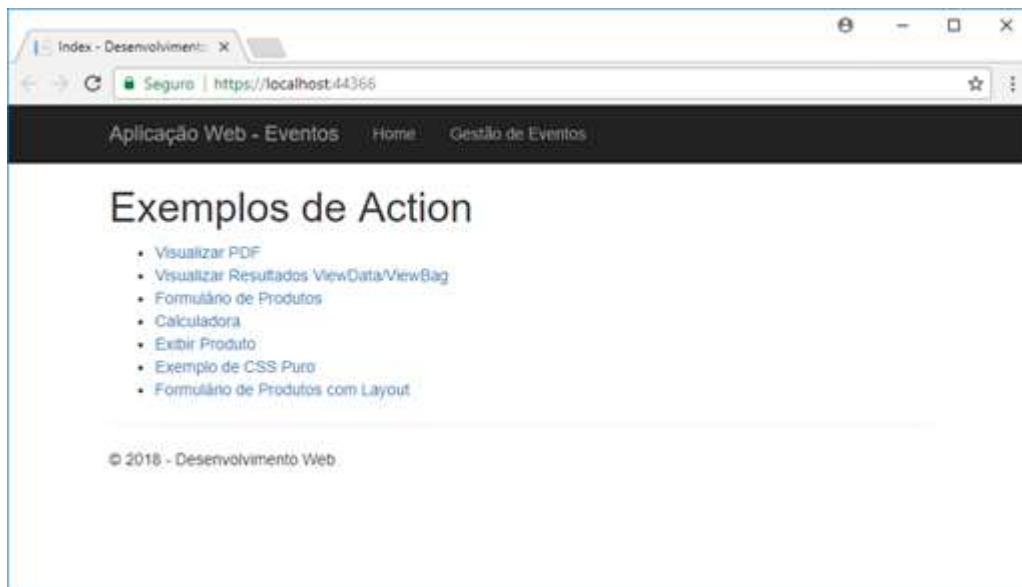
```
<footer>
    <p>&copy; @DateTime.Now.Year - Desenvolvimento Web</p>
</footer>
```

7. Inclusão de bibliotecas Javascript, especificamente do JQuery. É uma boa prática os arquivos Javascript serem incluídos como última instrução da página, seja ela um layout ou uma página sem layout.

```
<environment include="Development">
    <script src "~/lib/jquery/dist/jquery.js"></script>
    <script src "~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src "~/js/site.js" asp-append-version="true"></script>
</environment>
```

Ao longo do curso faremos outros ajustes neste layout, mas por hora temos ferramentas suficientes para trabalhar.

Executando a aplicação com esta nova configuração, teremos:



6.4.3. Definindo o controller EventosController

Neste momento vamos iniciar a parte fundamental da nossa aplicação: a elaboração de um novo controller.

Na pasta **Controllers**, inserir um novo controller vazio, **modelo MVC**, chamado **Eventos** (classe **EventosController**).

Este controller deverá ter apenas o action **Index()**.

```
namespace DesenvolvimentoWeb.Projeto02.Controllers
{
    public class EventosController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

A partir do action **Index**, criar uma view sem modelo, com layout. Deixar com o conteúdo a seguir:

```

@{
    ViewData["Title"] = "Gestão de Eventos";
}

<h2>Gestão de Eventos</h2>
<ul>
    <li><a asp-controller="Eventos" asp-action="IncluirEvento">
        Incluir Novo Evento</a> </li>
    <li><a asp-controller="Eventos" asp-action="ListarEventos">
        Listar Eventos</a> </li>
</ul>

```

Os actions indicados **IncluirEvento** e **ListarEventos** serão criados no próximo capítulo, quando falaremos sobre acesso a dados.

Já é possível executar a aplicação, e no menu principal (aquele definido no layout) acessar **Gestão de Eventos**. Observe que este item de menu acessa justamente o action **Index** do controller **Eventos**.

6.4.4. Conhecendo as rotas

Tivemos a oportunidade de definir até o momento dois controllers: **HomeController** e **EventosController**.

Se a aplicação for executada através da url padrão, o action **Index** do controller **Home** é acionado por default. Isso porque, no método **Configure** do arquivo **Startup.cs** temos a configuração da rota que indica esse processo. Vamos dar uma olhada:

```

public void Configure(IApplicationBuilder app,
    IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
}

```

```
app.UseStaticFiles();

    app.UseAuthentication();

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

O método de extensão **UseMvc()** especifica uma rota com o template:

```
template: "{controller=Home}/{action=Index}/{id?}")
```

significando que se nada for informado, o controller padrão é **Home**; o action padrão é **Index**, e no action **Index()** podemos opcionalmente ter um parâmetro chamado **id**.

Este parâmetro normalmente indica um valor informado na **url** para a aplicação.

Teremos oportunidade em aulas futuras de explorar o uso deste parâmetro.

7.Entity Framework Core

7.1. Introdução

O Entity Framework é parte integrante do framework, responsável pelo acesso a banco de dados. Não é o único, mas certamente é o mais simples e rápido de usar, além de ter uma estrutura otimizada que permite desde a criação até operações fundamentais sobre um banco de dados.

Nós precisamos definir a string de conexão, as entidades e o contexto para acesso ao banco de dados, além da estrutura de criação do banco.

O .Net Core utiliza o conceito de **injeção de dependência** para fornecer a instância de um contexto no controller. Neste capítulo vamos criar o banco de dados por meio da aplicação, considerando a tarefa de cadastro de eventos e participantes, como já apresentado anteriormente.

7.2. Projeto: acesso a dados e seus componentes.

7.2.1. Definindo o banco de dados

A estrutura do banco de dados que utilizaremos é mostrada a seguir:



Temos um banco de dados chamado **DBEventos**, com duas tabelas: **TBEventos** e **TBPesquisadores**. O banco de dados será descrito na string de conexão e as tabelas serão geradas a partir das entidades.

7.2.2. Definindo as entidades

Na pasta **Models** (se não existir, cria-la) incluir as duas classes mostradas abaixo. Estas classes são as entidades a serem manipuladas na aplicação:

```

namespace DesenvolvimentoWeb.Projeto02.Models
{
    public class Evento
    {
        public int Id { get; set; }
        public string Descricao { get; set; }
        public DateTime Data { get; set; }
        public string Local { get; set; }
        public double Preco { get; set; }

        public ICollection<Participante> Participantes { get; set; }
    }
}

namespace DesenvolvimentoWeb.Projeto02.Models
{
    public class Participante
    {
        public int Id { get; set; }

```

```

        public int IdEvento { get; set; }
        public string Nome { get; set; }
        public string Email { get; set; }
        public string Cpf { get; set; }
        public DateTime DataNascimento { get; set; }

        public Evento EventoInfo { get; set; }

    }
}

```

7.2.3. Definindo o contexto e o inicializador

No projeto criar uma pasta chamada **Dados**. Nesta pasta, definir o contexto do banco de dados: **EventosContext**:

```

namespace DesenvolvimentoWeb.Projeto02.Dados
{
    public class EventosContext : DbContext
    {
        public EventosContext(DbContextOptions<EventosContext> opcoes)
        :
            base(opcoes)
        { }

        public DbSet<Evento> Eventos { get; set; }
        public DbSet<Participante> Participantes { get; set; }

        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            modelBuilder.Entity<Evento>().ToTable("TBEventos");

            modelBuilder.Entity<Participante>().ToTable("TBParticipantes");
        }
    }
}

```

Importar os namespaces necessários.

A entidade **Participantes** poderia ter sido omitida, uma vez que existe um relacionamento entre elas.

As tabelas serão criadas com o mesmo nome das entidades, porém em muitos casos é conveniente alterá-las, como fizemos na classe.

Vamos agora registrar a injecão de dependência. Abrir o arquivo **Startup.cs**. Neste arquivo, importar o namespace **Microsoft.EntityFrameworkCore**. Incluir a instrução abaixo no método **ConfigureServices**:

```
services.AddDbContext<EventosContext>(options => options.UseSqlServer(  
    Configuration.GetConnectionString("DefaultConnection")));
```

Observação importante: Se você tiver incluído o recurso **Contas de Usuário Individual** quando criou o projeto, mude o nome da string de conexão. Uma sugestão será:

```
services.AddDbContext<EventosContext>(options => options.UseSqlServer(  
    Configuration.GetConnectionString("EventosConnection")));
```

Vamos agora abrir o arquivo **appsettings.json**. Configurar a string de conexão como indicado a seguir (é possível alterá-la para refletir um banco de dados real, ao invés de criar um arquivo local na aplicação):

```
{  
  "ConnectionStrings": {  
    "EventosConnection":  
      "Server=(localdb)\\mssqllocaldb;Database=DBEventos;ConnectRetryCount=0  
      ;Trusted_Connection=True;MultipleActiveResultSets=true",  
    "DefaultConnection":  
      "Server=(localdb)\\mssqllocaldb;Database=aspnet-  
      DesenvolvimentoWeb.Projeto02-B74F26AF-B3EF-4382-8EE5-  
      C11641E43C5A;Trusted_Connection=True;MultipleActiveResultSets=true"  
  },  
  "Logging": {  
    "IncludeScopes": false,  
    "LogLevel": {  
      "Default": "Warning"  
    }  
  }  
}
```

Na pasta **Dados**, incluir uma classe estática chamada **DbInitializer**:

```

namespace DesenvolvimentoWeb.Projeto02.Dados
{
    public static class DbInitializer
    {
        public static void Initialize(EventosContext context)
        {
            context.Database.EnsureCreated();
        }
    }
}

```

A instrução

```
context.Database.EnsureCreated();
```

cria o banco de dados, se este não existir.

7.2.4. Configurando a aplicação para a injeção de dependência

Sabemos que a classe **Program.cs** contém o método **Main()**, representando o ponto de partida de uma aplicação. Vamos alterar o código desta classe.

Usaremos um método de extensão disponível no namespace **Microsoft.Extensions.DependencyInjection** que deve ser importado. Em seguida, deixe o método **Main()** como no modelo abaixo:

```

public static void Main(string[] args)
{
    //BuildWebHost(args).Run();
    var host = BuildWebHost(args);

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services
                .GetRequiredService<EventosContext>();
        }
    }
}

```

```

        DbInitializer.Initialize(context);
    }
    catch (Exception ex)
    {
        var logger = services
            .GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, ex.Message);
        throw;
    }
}
host.Run();
}

```

Testar a aplicação até este ponto. Se tudo estiver correto, teremos nosso banco de dados criado, e a aplicação pronta para funcionar com este banco.

7.2.5. Preparando o controller para incluir registros

Vamos alterar o controller `EventosController` para receber o contexto e adicionar novos eventos.

Abra o controller e faça as alterações:

```

namespace DesenvolvimentoWeb.Projeto02.Controllers
{
    public class EventosController : Controller
    {
        private EventosContext Context { get; set; }

        public EventosController(EventosContext context)
        {
            this.Context = context;
        }
        public IActionResult Index()
        {
            return View();
        }

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

        [HttpPost]
        public IActionResult IncluirEvento(Evento evento)
        {

```

```

        if (ModelState.IsValid)
    {
        Context.Add<Evento>(evento);
        Context.SaveChanges();
        return RedirectToAction("Index");
    }
    else
    {
        return View();
    }
}
}

```

Para que o modelo seja validado, e que a instrução **ModelState.IsValid** retorne **true**, vamos alterar as entidades para incluir os validadores:

```

namespace DesenvolvimentoWeb.Projeto02.Models
{
    public class Evento
    {
        public int Id { get; set; }
        [Required]
        [Display(Name = "Descrição")]
        public string Descricao { get; set; }

        [Required]
        [Display(Name = "Data do evento")]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}",
            ApplyFormatInEditMode = true)]
        public DateTime Data { get; set; }

        [Required]
        public string Local { get; set; }

        [Required]
        public double Preco { get; set; }

        public ICollection<Participante> Participantes { get; set; }
    }
}

namespace DesenvolvimentoWeb.Projeto02.Models
{
    public class Participante
    {
        public int Id { get; set; }
        public int IdEvento { get; set; }
    }
}

```

```

[Required]
public string Nome { get; set; }

[Required]
[EmailAddress(ErrorMessage = "O Email está inválido")]
public string Email { get; set; }

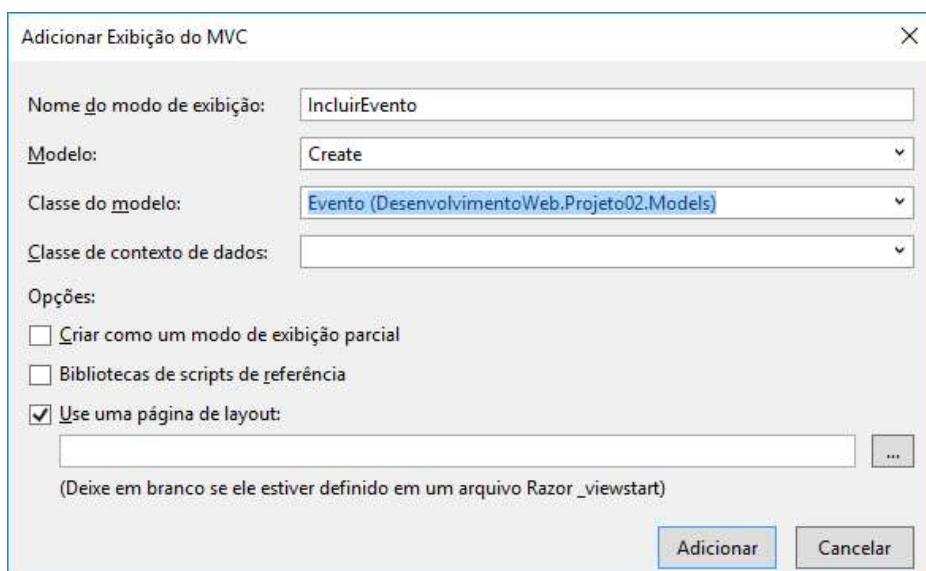
public string Cpf { get; set; }

[DataType(DataType.Date)]
[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}",
    ApplyFormatInEditMode = true)]
public DateTime DataNascimento { get; set; }

public Evento EventoInfo { get; set; }
}
}

```

No action **IncluirEvento**, definir a view com o layout e com o modelo **Evento**:



Faça as alterações necessárias, deixando a view como no código a seguir:

```

@model DesenvolvimentoWeb.Projeto02.Models.Evento

@if
    ViewData["Title"] = "IncluirEvento";
}

<h2>Incluir Evento</h2>

```

```

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="IncluirEvento">
            <div asp-validation-summary="ModelOnly" class="text-
danger"></div>
                @*<div class="form-group">
                    <label asp-for="Id" class="control-label"></label>
                    <input asp-for="Id" class="form-control" />
                    <span asp-validation-for="Id" class="text-
danger"></span>
                </div>*@
                <div class="form-group">
                    <label asp-for="Descricao" class="control-
label"></label>
                    <input asp-for="Descricao" class="form-control" />
                    <span asp-validation-for="Descricao" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Data" class="control-label"></label>
                    <input asp-for="Data" class="form-control" />
                    <span asp-validation-for="Data" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Local" class="control-label"></label>
                    <input asp-for="Local" class="form-control" />
                    <span asp-validation-for="Local" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Preco" class="control-label"></label>
                    <input asp-for="Preco" class="form-control" />
                    <span asp-validation-for="Preco" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <input type="submit" value="Incluir" class="btn btn-
default" />
                </div>
            </form>
        </div>
    </div>

```

Observe que comentamos o **Id**, pois o Entity Framework considera o campo Id como sendo chave primária, auto incremento.

Na view Index do controller Eventos já colocamos um link para este action. Podemos executar a aplicação e constatar o resultado.

The screenshot shows a web browser window with the title 'IncluirEvento - Desenvolvimento'. The address bar indicates a secure connection to 'https://localhost:44366/Eventos/IncluirEvento'. The page content is titled 'Incluir Evento'. It contains four input fields: 'Descrição' with the value 'Aula de ASP.NET', 'Data do evento' with the value '20/10/2018', 'Local' with the value 'Phorte', and 'Preco' with the value '150'. Below these fields is a single button labeled 'Incluir'.

Veja o exemplo de registro que coloquei. Inclua quantos registros você desejar.

7.2.6. Listando os eventos

Para fechar este capítulo vamos apresentar a listagem dos eventos cadastrados.

Incluir o action abaixo no controller Eventos:

```
public IActionResult ListarEventos()
{
    List<Evento> lista = Context.Eventos.ToList<Evento>();
    return View(lista);
}
```

A view deverá ter o template List:

```
@model IEnumerable<DesenvolvimentoWeb.Projeto02.Models.Evento>

@{
    ViewData["Title"] = "ListarEventos";
}
```

```

<h2>ListarEventos</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Id)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Descricao)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Data)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Local)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Preco)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Id)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Descricao)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Data)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Local)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Preco)
                </td>
                <td>
                    @Html.ActionLink("Editar", "Editar", new { id =
item.Id })
                </td>
            </tr>
        }
    </tbody>
</table>

```

Observe a alteração que fizemos no final do arquivo. Observe também o model no início do arquivo.

Ao executarmos a aplicação, temos o resultado:

ID	Descrição	Data do evento	Local	Preço	
1	Aula de ASP.NET	20/10/2018	Ponte	150	Edit
2	Formatura do Curso	20/12/2018	Auditorio	0	Edit

7.2.7. Alterando e Removendo Registros

O processo de alteração e remoção de registros é semelhante ao de incluir. As diferenças estão na forma como codificamos o Entity Framework e como lidamos com a view.

Inicaremos com a alteração do registro e, em seguida, com a remoção.

7.2.8. Alterando registros

Na listagem de registros, deixamos um link para **Editar** um registro. É este action que definiremos agora.

Verifique que incluímos o parâmetro opcional id no link:

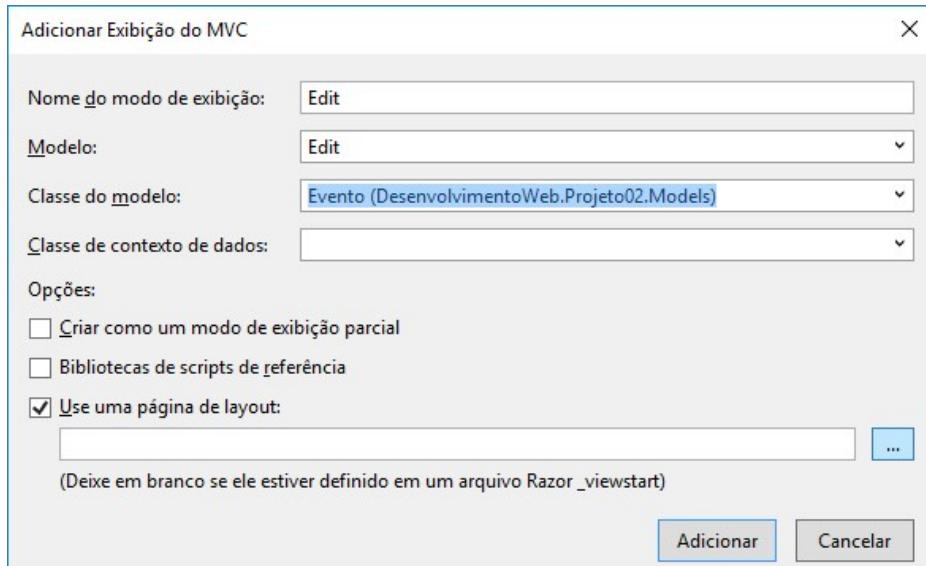
```
@Html.ActionLink("Editar", "Editar", new { id = item.Id })
```

O action Editar deve ter um parâmetro chamado Id, que receberá o Id do evento, de acordo com a instrução produzida. No controller **Eventos** definir os actions:

```
public IActionResult Editar(int id)
{
    Evento evento = Context.Eventos.FirstOrDefault(p => p.Id == id);
    return View(evento);
}

[HttpPost]
public IActionResult Editar(int id, Evento evento)
{
    evento.Id = id;
    Context.Entry<Evento>(evento).State =
        Microsoft.EntityFrameworkCore.EntityState.Modified;
    Context.SaveChanges();
    return RedirectToAction("Index");
}
```

Ao definir a view, o template usado é o Edit:



```
@model DesenvolvimentoWeb.Projeto02.Models.Evento
```

```
@{
    ViewData["Title"] = "Editar";
}
```

```
<h4>Evento</h4>
```

```

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Editar">
            <div asp-validation-summary="ModelOnly" class="text-
danger"></div>
                @*<div class="form-group">
                    <label asp-for="Id" class="control-label"></label>
                    <input asp-for="Id" class="form-control" />
                    <span asp-validation-for="Id" class="text-
danger"></span>
                </div>*@
                <div class="form-group">
                    <label asp-for="Descricao" class="control-
label"></label>
                    <input asp-for="Descricao" class="form-control" />
                    <span asp-validation-for="Descricao" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Data" class="control-label"></label>
                    <input asp-for="Data" class="form-control" />
                    <span asp-validation-for="Data" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Local" class="control-label"></label>
                    <input asp-for="Local" class="form-control" />
                    <span asp-validation-for="Local" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Preco" class="control-label"></label>
                    <input asp-for="Preco" class="form-control" />
                    <span asp-validation-for="Preco" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <input type="submit" value="Alterar" class="btn btn-
default" />
                </div>
            </form>
        </div>
    </div>

```

Quando selecionarmos o link Editar na listagem, chegaremos nesta view. Basta alterar um evento à sua escolha, e testar.

7.2.9. Removendo registros

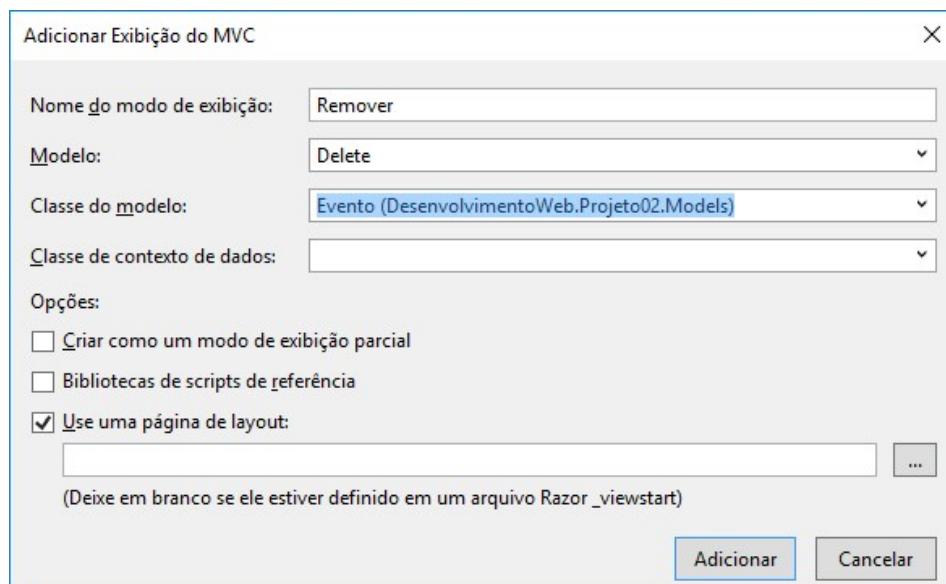
Vamos remover um registro, mas lembrando que para remover, é importante solicitarmos uma conformação do usuário.

No controller, incluir os actions:

```
public IActionResult Remover(int id)
{
    Evento evento = Context.Eventos.FirstOrDefault(p => p.Id == id);
    return View(evento);
}

[HttpPost]
public IActionResult Remover(int id, Evento evento)
{
    evento.Id = id;
    Context.Entry<Evento>(evento).State =
        Microsoft.EntityFrameworkCore.EntityState.Deleted;
    Context.SaveChanges();
    return RedirectToAction("Index");
}
```

Criar a view com o template Delete:



Deixe a view com o conteúdo a seguir:

```
@model DesenvolvimentoWeb.Projeto02.Models.Evento
```

```

@{
    ViewData["Title"] = "Remover";
}

<h3>Tem certeza que deseja remover o evento?</h3>
<div>
    <h4>Evento</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Descricao)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Descricao)
        </dd>
    </dl>

    <form asp-action="Remover">
        <input type="submit" value="Sim, quero remover" class="btn btn-danger" /> |
        <a asp-action="Index">Não, quero voltar para Index</a>
    </form>
</div>

```

Para testar a remoção, vamos alterar a listagem de forma a incluir uma opção de remoção no link:

```

@Html.ActionLink("Editar", "Editar", new { id = item.Id }) |
@Html.ActionLink("Remover", "Remover", new { id = item.Id })

```

Testar a alteração e a remoção.

7.2.10. Cadastro e Consulta de Participantes

Da mesma forma que pudemos incluir e listar eventos, vamos fazer o mesmo com os participantes. Resolvemos colocar este tópico separadamente porque temos duas características distintas:

1. Para cadastrar um participante, temos que fornecer também a lista de eventos para a view.

- Para listar os participantes, temos que considerar a lista de participantes por evento, que será selecionado em um componente **DropDownList**.

Vamos iniciar este processo.

7.2.11. Incluindo Participantes

No controller Eventos, vamos definir o action para gerar o formulário de inclusão de participantes, e o que receberá o objeto Participante. Fique atendo às alterações que faremos especialmente na view. Criar os actions:

```
public IActionResult IncluirParticipante()
{
    List<Evento> lista = Context.Eventos.ToList<Evento>();
    ViewBag.Eventos = new SelectList(lista, "Id", "Descricao");
    return View();
}

[HttpPost]
public IActionResult IncluirParticipante(Participante participante)
{
    if (ModelState.IsValid)
    {
        Context.Add<Participante>(participante);
        Context.SaveChanges();
        return RedirectToAction("Index");
    }
    else
    {
        List<Evento> lista = Context.Eventos.ToList<Evento>();
        ViewBag.Eventos = new SelectList(lista, "Id",
        "Descricao");
        return View();
    }
}
```

Vamos gerar a view, com o template Create e o model Participante:

```
@model DesenvolvimentoWeb.Projeto02.Models.Participante

@{
    ViewData["Title"] = "IncluirParticipante";
}
```

```

<h2>Incluir Participante</h2>

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="IncluirParticipante">
            <div asp-validation-summary="ModelOnly" class="text-
danger"></div>
                @*<div class="form-group">
                    <label asp-for="Id" class="control-label"></label>
                    <input asp-for="Id" class="form-control" />
                    <span asp-validation-for="Id" class="text-danger"></span>
                </div>*@
                @*<div class="form-group">
                    <label asp-for="IdEvento" class="control-label"></label>
                    <input asp-for="IdEvento" class="form-control" />
                    <span asp-validation-for="IdEvento" class="text-
danger"></span>
                </div>*@
                <div class="form-group">
                    <label asp-for="IdEvento" class="control-
label"></label>
                    <select asp-for="IdEvento"
                            asp-items="(SelectList)ViewBag.Eventos"
                            class="form-control"></select>
                </div>
                <div class="form-group">
                    <label asp-for="Nome" class="control-label"></label>
                    <input asp-for="Nome" class="form-control" />
                    <span asp-validation-for="Nome" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Email" class="control-label"></label>
                    <input asp-for="Email" class="form-control" />
                    <span asp-validation-for="Email" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Cpf" class="control-label"></label>
                    <input asp-for="Cpf" class="form-control" />
                    <span asp-validation-for="Cpf" class="text-
danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="DataNascimento" class="control-
label"></label>
                    <input asp-for="DataNascimento" class="form-control"
/>
                    <span asp-validation-for="DataNascimento" class="text-
danger"></span>
                </div>
                <div class="form-group">

```

```

        <input type="submit" value="Incluir" class="btn btn-
default" />
    </div>
</form>
</div>
</div>

```

Verifique que a listagem de eventos passada para a view através de um componente ViewBag foi recuperada em um elemento select.

Vamos incluir um link para cadastro e outro para listagem de participantes na view Index.cshtml:

```

<ul>
    <li><a asp-controller="Eventos" asp-action="IncluirEvento">
        Incluir Novo Evento</a> </li>
    <li><a asp-controller="Eventos" asp-action="ListarEventos">
        Listar Eventos</a> </li>
    <li><a asp-controller="Eventos" asp-action="IncluirParticipante">
        Incluir Novo Participante</a> </li>
    <li><a asp-controller="Eventos" asp-action="ListarParticipantes">
        Listar Participantes</a> </li>
</ul>

```

Já estamos prontos para testar o cadastro de participantes. Basta selecionar um evento e inscrever um participante para este evento. Procure incluir participantes para todos os eventos. É importante ter mais de um evento para testarmos a listagem de participantes.

7.2.12. Listando os participantes por evento

Nossa tarefa agora é apresentar a lista de participantes. O action que mostrará esta lista terá um parâmetro opcional: o código do evento pode ou não ser fornecido. Se for, mostramos os participantes daquele evento. Se não for, mostramos todos os participantes.

Vamos definir o action:

```

public IActionResult ListarParticipantes(int idEvento)
{

```

```

        List<Participante> lista;
        ViewBag.Eventos = new
            SelectList(
                Context.Eventos.ToList<Evento>(), "Id", "Descricao");
        if (idEvento == 0)
        {
            lista = Context.Participantes.ToList<Participante>();
        }
        else
        {
            lista = Context.Participantes
                .Where(p => p.IdEvento == idEvento)
                .ToList<Participante>();
        }
        return View(lista);
    }
}

```

Geramos a view com o template **List**, e o model **Participante**:

```

@model IEnumerable<DesenvolvimentoWeb.Projeto02.Models.Participante>

 @{
    ViewData["Title"] = "ListarParticipantes";
}



## Listar Participantes



<form asp-action="ListarParticipantes" asp-controller="Eventos"
method="get">

    <label class="control-label col-sm-1">ID Evento</label>
    <div class="col-sm-5">
        <select id="id" name="idEvento" asp-
items="(SelectList)ViewBag.Eventos"
            class="form-control col-sm-6"></select>
    </div>
    <div class="form-group">
        <input type="submit" value="Buscar" class="btn btn-default" />
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Nome)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Email)
            </th>
        
```

```

        <th>
            @Html.DisplayNameFor(model => model.Cpf)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.DataNascimento)
        </th>
        <th></th>
    </tr>
</thead>
<tbody>
    @foreach (var item in Model)
    {
        <tr>

            <td>
                @Html.DisplayFor(modelItem => item.Nome)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Email)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Cpf)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.DataNascimento)
            </td>
            <td></td>
        </tr>
    }
</tbody>
</table>

```

Verifique que incluímos um formulário para gerar a lista de participantes de acordo com o evento selecionado. Testar a aplicação, selecionado um evento e clicando no botão Buscar. Veja o que ocorre na Url.

Para fechar, vamos personalizar nosso Model Participante, da mesma forma que fizemos com os eventos:

```

public class Participante
{
    public int Id { get; set; }
    [Display(Name = "Evento")]
    public int IdEvento { get; set; }

    [Required]
    public string Nome { get; set; }

    [Required]

```

```
[EmailAddress(ErrorMessage = "O Email está inválido")]
public string Email { get; set; }

public string Cpf { get; set; }

[Display(Name = "Data Nascimento")]
[DataType(DataType.Date)]
[DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}",
    ApplyFormatInEditMode = true)]
public DateTime DataNascimento { get; set; }

public Evento EventoInfo { get; set; }
}
```

8. Conceitos do Javascript e Ajax

8.1. Introdução

Javascript é uma linguagem interpretada pelo browser, e possui diversas utilidades. Dentre elas destacamos a execução de funções de forma assíncrona, também conhecida como **ajax**.

Muitas validações podem ser realizadas antes que os dados de um formulário, por exemplo, sejam enviados para o servidor.

A partir do Javascript, muitas bibliotecas foram geradas com o propósito de melhorar a interação com o usuário, na camada de visualização. As bibliotecas de destaque são: JQuery, Angular, React.js, Node.js, entre outras.

Não temos intenção de fazer uma abordagem completa sobre Javascript, até mesmo porque justificaria um curso inteiro só pra ele. Nós vamos apresentar alguns conceitos importantes para darmos prosseguimento ao nosso projeto.

8.2. A estrutura de uma função JQuery

No nosso curso utilizaremos funções JQuery. Para tanto, apresentaremos a forma geral e, em seguida, exemplos de utilização.

Forma Geral

```
$(seletor).funcao();  
  
seletor: semelhante ao seletor CSS.  
funcao(): um evento ou comando JQuery.
```

Parece simples, mas este é o formato geral de uma função JQuery. Para utilizá-la, vamos a um exemplo.

Exemplo HTML:

```
<input type="button" id="botao" value="OK" />  
<div id="saida"></div>
```

JQuery:

```
$("#botao").click(function(){ });  
  
$("#botao").click(function(){  
    //funcionalidade do evento click  
});  
  
$("#botao").click(function(){  
    $("#saida").html("conteúdo da div");  
});
```

Mostramos uma evolução de uma instrução JQuery para melhorar seu entendimento.

Vamos acrescentar alguns exemplos JQuery no projeto, para melhor compreensão.

8.3. Aplicação: O controller JScriptController

Vamos adicionar um novo controller ao projeto chamado **JScriptController**. Seu conteúdo deve ser:

```

namespace DesenvolvimentoWeb.Projeto02.Controllers
{
    public class JScriptController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

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

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

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

```

Vamos agora definir uma view para cada um dos actions, sem considerar nenhum template, mas todos com layout:

Index.cshtml

```

@{
    ViewData["Title"] = "Index";
}

<h2>Exemplos Javascript</h2>

<ul>
    <li><a asp-controller="JScript" asp-action="Calculadora">
        Calculadora Javascript</a></li>
    <li><a asp-controller="JScript" asp-action="Formulario">
        Exemplo de Formulário com JQuery</a></li>
    <li><a asp-controller="JScript" asp-action="CidadesEstados">
        Cidades e Estados</a></li>
</ul>

```

Calculadora.cshtml

```

@{
    ViewData["Title"] = "Calculadora";
}

<h2>Calculadora - Javascript</h2>
<label for="valor1">Valor 1:</label><br />
<input type="text" id="valor1" /><br />

<label for="valor2">Valor 2:</label><br />
<input type="text" id="valor2" /><br />

<input type="button" id="calculo" value="Calcular Soma" />
<br />
<div id="resultado"></div>

```

Formulário.cshtml

```

@{
    ViewData["Title"] = "Formulario";
}

<h2>Simulação de Cadastro</h2>

<form>
    Nome:<br />
    <input type="text" id="nome" /><br />

    Idade:<br />
    <input type="text" id="idade" /><br />

    Telefone:<br />
    <input type="text" id="telefone" /><br />

    <input type="button" id="enviar" value="Enviar" /><br />
    <div id="resultado"></div>
</form>

```

CidadesEstados.cshtml

```

@{
    ViewData["Title"] = "CidadesEstados";
}

<h2>Cidades e Estados</h2>
<div>
    <span>Estado:</span>

```

```

<select id="estado">
    <option>Selecione o estado</option>
</select>

<span>Cidade:</span>
<select id="cidade"></select>
</div>

```

8.4. Aplicação: Definindo funções JQuery

Na pasta **wwwroot**, temos uma pasta **js**. Nesta pasta tem um arquivo chamado **site.js**. Abrir o arquivo e incluir este código:

```

var calcular = document.getElementById("calculo");

calcular.addEventListener("click", function () {
    //obtendo os valores das caixas de texto
    var v1 = document.getElementById("valor1").value;
    var v2 = document.getElementById("valor2").value;
    var resposta;
    var resultado = document.getElementById("resultado");

    if (v1 == "" || v2 == "") {
        resposta = "<div class='alert alert-danger' role='alert'>" +
                   "Por favor fornecer os operandos!!" + "</div>";
    }
    else {
        v1 = parseFloat(v1);
        v2 = parseFloat(v2);
        var soma = v1 + v2;

        resposta = "<div class='alert alert-success' role='alert'>" +
                   "Valor da soma: " + soma + "</div>";
    }
    resultado.innerHTML = resposta;
});

```

Este código será usado para o botão calcular da view **Calculadora**.

Na mesma pasta, criar um arquivo chamado **funcoes.js**. O propósito é mostrar como inserir um novo javascript ao projeto, especialmente no arquivo **_Layout.cshtml**.

No arquivo criado, incluir o código:

```
$(document).ready(function () {
    $("#enviar").click(function () {
        //obter o conteúdo dos campos de texto
        var nome = $("#nome").val();
        var idade = $("#idade").val();
        var telefone = $("#telefone").val();

        var resumo = "Dados informados: " +
            "<br/>Nome: " + nome +
            "<br/>Idade: " + idade +
            "<br/>Telefone: " + telefone;

        if (idade < 18) {
            resumo += "<br/>Você é menor de idade";
        } else {
            resumo += "<br/>Você é maior de idade";
        }

        $("#resultado").html(resumo);
    });

    //cidades e estados
    //Array de Estados (JSON - Javascript Object Notation)
    var estados = [
        { "id": 1, "estado": "SP" },
        { "id": 2, "estado": "RJ" },
        { "id": 3, "estado": "MG" },
        { "id": 4, "estado": "BA" }
    ];

    var cidades = [
        { "id": 1, "idestado": 1, "cidade": "CAMPINAS" },
        { "id": 2, "idestado": 1, "cidade": "SOROCABA" },
        { "id": 3, "idestado": 2, "cidade": "NITEROI" },
        { "id": 4, "idestado": 2, "cidade": "CABO FRIO" },
        { "id": 5, "idestado": 2, "cidade": "ANGRA" },
        { "id": 6, "idestado": 3, "cidade": "BELO HORIZONTE" },
        { "id": 7, "idestado": 3, "cidade": "BETIM" },
        { "id": 8, "idestado": 3, "cidade": "EXTREMA" },
        { "id": 9, "idestado": 4, "cidade": "SALVADOR" },
        { "id": 10, "idestado": 4, "cidade": "PORTO SEGURO" }
    ];

    $.each(estados, function (i, item) {
        $("#estado").append($('<option>', {
            value: item.id,
            text: item.estado
        }));
    });

    $("#estado").change(function () {


```

```

//lendo o id do estado, e guardando em uma variavel
var idestado = $(this).val();

//gerando uma sublista a partir da lista de cidades
var cidadesFiltradas = $.grep(cidades, function (e) {
    return e.idestado == idestado;
});

$("#cidade").html("<option>Selecione</option>");
$.each(cidadesFiltradas, function (i, item) {
    $("#cidade").append($('<option>', {
        value: item.id,
        text: item.cidade
    }));
});
});
});

```

Abrir o arquivo `_Layout.cshtml`, e fazer as inclusões indicadas:

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>@ViewData["Title"] - Desenvolvimento Web - Projeto</title>

    <environment include="Development">
        <link rel="stylesheet"
        href("~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href "~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
        href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.mi
n.css"
            asp-fallback-
        href "~/lib/bootstrap/dist/css/bootstrap.min.css"
            asp-fallback-test-class="sr-only"
            asp-fallback-test-property="position"
            asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href "~/css/site.min.css"
            asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle"
                data-toggle="collapse" data-target=".navbar-collapse">

```

```

        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
    </button>
    <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">Aplicação Web - Eventos</a>
</div>
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
        <li><a asp-area="" asp-controller="Eventos" asp-action="Index">Gestão de Eventos</a></li>
        <li><a asp-area="" asp-controller="JScript" asp-action="Index">Exemplos Javascript</a></li>
    </ul>
</div>
</div>
</nav>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - Desenvolvimento Web</p>
    </footer>
</div>

<environment include="Development">
    <script src "~/lib/jquery/dist/jquery.js"></script>
    <script src "~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src "~/js/site.js" asp-append-version="true"></script>
    <script src "~/js/funcoes.js" asp-append-version="true"></script>
</environment>
<environment exclude="Development">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
           asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
           asp-fallback-test="window.jQuery"
           crossorigin="anonymous"
           integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
    </script>
    <script
        src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
        asp-fallback-
        src="~/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn
&& window.jQuery.fn.modal"
        crossorigin="anonymous"
        integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuhVxZxUPnCJA7l2mCWNIpG9mGCD8wGNICPD7Txa">

```

```
</script>
<script src("~/js/site.min.js" asp-append-
version="true"></script>
</environment>

@RenderSection("Scripts", required: false)
</body>
</html>
```

Já estamos prontos para testar as funcionalidades Javascript que criamos. Executar a aplicação e testar as funcionalidades implementadas. Dê uma atenção especial aos efeitos que incluímos nos resultados.

9.Uso do Ajax com views parciais

9.1. Introdução

Nesta fase do projeto implementaremos o conceito de **Partial View**, ou View Parcial.

Este tipo de view é um pouco diferente da view tradicional por ser uma parte da View, devidamente inserida na view principal. Sua principal vantagem é a reutilização, e além disso, é adequada para processar conteúdo assíncrono.

Vamos listar os convidados por evento, desta vez usando Partial View. Usaremos função JQuery para atender a este requisito.

9.2. Projeto: aplicação do Ajax

9.2.1. Definindo a view principal e a view parcial

No controller vamos acrescentar o action a seguir. Observe o retorno de uma view parcial:

```
public IActionResult ListarParticipantesAjax(int idEvento)
{
    List<Participante> lista;
    ViewBag.Eventos = new SelectList(
        Context.Eventos.ToList<Evento>(), "Id", "Descricao");
    if (idEvento == 0)
    {
        //lista = Context.Participantes.ToList<Participante>();
        return View();
    }
    else
    {
```

```

        lista = Context.Participantes.Where(p => p.IdEvento == idEvento).ToList<Participante>();
        return PartialView("_ListarParticipantes", lista);
    }
}

```

Para criar a view para este action, vamos defini-la vazia. O conteúdo deverá ser o mostrado abaixo:

```

@{
    ViewData["Title"] = "ListarParticipantes";
}



## ListarParticipantes



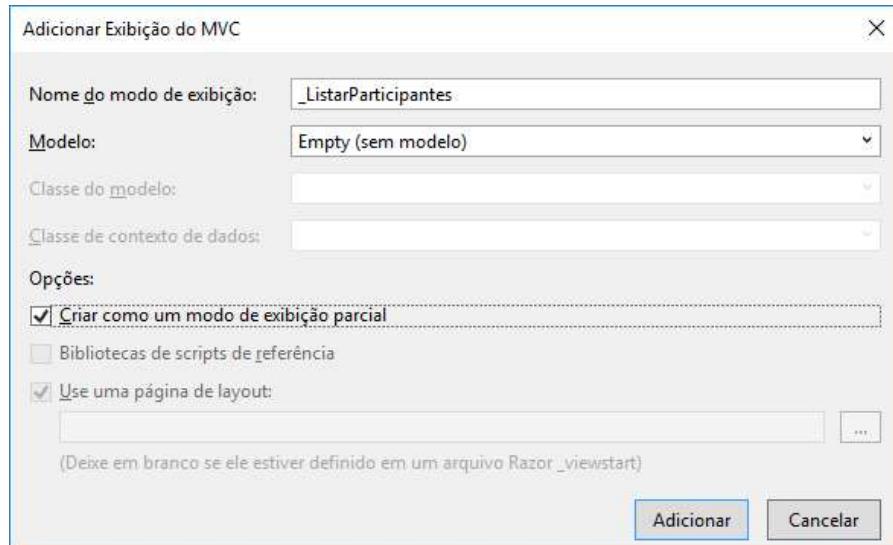
Create New



<form asp-action="ListarParticipantesAjax" asp-controller="Eventos" method="get">
    <label class="control-label">ID Evento</label>
    <select id="idEvento" name="idEvento" asp-items="(SelectList)ViewBag.Eventos">
        <option value="">SELECCIONE</option>
    </select>
    <input type="submit" value="Buscar" class="btn btn-default" />
</form>
<div id="resultado"></div>

```

Para criar a view parcial, vamos marcar esta opção. Na pasta **Views/Shared** incluir o arquivo **_ListarParticipantes.cshtml**:



Mantenha o conteúdo:

```

@model IEnumerable<DesenvolvimentoWeb.Projeto02.Models.Participante>
<div class="alert alert-success">
    <table class="table">
        <thead>
            <tr>

                <th>
                    @Html.DisplayNameFor(model => model.Nome)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Email)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Cpf)
                </th>
                <th>
                    @Html.DisplayNameFor(model =>
model.DataNascimento)
                </th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (var item in Model)
            {
                <tr>

                    <td>
                        @Html.DisplayFor(modelItem => item.Nome)
                    </td>
                    <td>
                        @Html.DisplayFor(modelItem => item.Email)
                    </td>

```

```

        <td>
            @Html.DisplayFor(modelItem => item.Cpf)
        </td>
        <td>
            @Html.DisplayFor(modelItem =>
item.DataNascimento)
        </td>

    </tr>
}
</tbody>
</table>
</div>

```

No arquivo **ListarParticipantesAjax.cshtml**, incluir o section a seguir:

```

@section Scripts {
    <script type="text/javascript">
        $(document).ready(function () {
            $("#idEvento").change(function () {
                var selecao = $("#idEvento").val();
                if (selecao == "") {
                    var erro = "<div class='alert alert-danger'>
Nenhum evento selecionado</div>";
                    $("#resultado").html(erro);
                }
                else {
                    $("#resultado")
                        .load("/Eventos/ListarParticipantesAjax?idEvento=" +
                            selecao);
                }
            });
        });
    </script>
}

```

Analizar a instrução

```
@RenderSection("Scripts", required: false)
```

no layout da aplicação. Neste ponto será inserido o conteúdo da section com o nome **Scripts**. Podemos definir na view tantos sections quanto desejarmos, e no layout especificar um local adequado para inseri-lo. Este processo é semelhante ao `RenderBody()`, só que nomeado, e com a opção de obrigatoriedade ou não.

No **Index.cshtml** do controller **Eventos**, incluir a opção para listar os participantes via ajax.

```
<ul>
    <li><a asp-controller="Eventos" asp-action="IncluirEvento">
        Incluir Novo Evento</a> </li>
    <li><a asp-controller="Eventos" asp-action="ListarEventos">
        Listar Eventos</a> </li>
    <li><a asp-controller="Eventos" asp-action="IncluirParticipante">
        Incluir Novo Participante</a> </li>
    <li><a asp-controller="Eventos" asp-action="ListarParticipantes">
        Listar Participantes</a> </li>
    <li><a asp-controller="Eventos"
            asp-action="ListarParticipantesAjax">
        Listar Participantes - Ajax</a> </li>
    </ul>
```

Veja o resultado.

10. Asp.Net Identity Core

10.1. Introdução

Existem dois processos básicos envolvidos na segurança dos dados disponibilizados por uma aplicação: **autenticação** e **autorização**. A **autenticação** é o processo de identificar a pessoa, software ou serviço que realizará alguma tarefa no programa. Já a **autorização** é o processo de permitir ou proibir o acesso às funcionalidades do sistema pelo usuário identificado.

É comum criarmos grupos de usuários e atribuirmos permissões a esses grupos, incluindo sempre o usuário identificado em um ou mais grupos definidos no sistema. Por exemplo, um sistema de controle de uma escola teria grupos como **Professores**, **Alunos** e **Funcionários**. Já um sistema de comércio eletrônico poderia ter grupos como **Clientes**, **Fornecedores**, **Administradores** ou **Visitantes**.

Esses grupos teriam permissões diferentes. Por exemplo, um fornecedor poderia incluir um novo produto, um administrador poderia liberar o produto para venda, um cliente poderia visualizar o seu (e apenas o seu) carrinho de compras, um fornecedor não poderia visualizar informações de venda de produtos de outros fornecedores, e assim por diante.

Repare que o processo de autorização é sempre uma operação de **permitir** ou **proibir** determinadas ações.

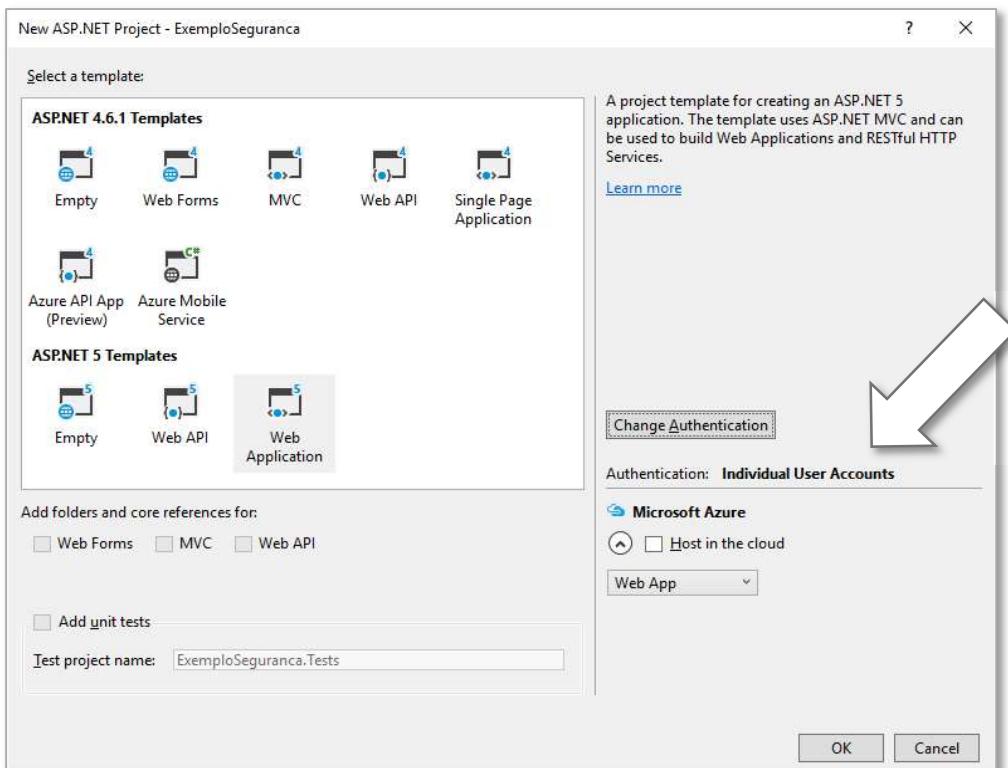
O **Asp.Net Core Identity**, ou simplesmente **Identity**, é um sistema integrado que permite incluir funcionalidade de logon ao aplicativo.

Os usuários podem criar uma conta contendo um nome de usuário e uma senha, que seguem determinados padrões. O Identity é poderoso o suficiente para permitir logon com elementos externos, como Facebook, Google, Microsoft Account, Twitter ou outros.

Neste capítulo veremos como incluir recursos do Identity na nossa aplicação. Se o projeto foi criado com recurso de contas de usuário individual, o mecanismo já foi incluído. Basta personalizar o que for necessário.

10.2. ASP.NET Identity no .NET Core

O modelo de projeto padrão do ASP.NET Core inclui a autenticação por meio de contas individuais armazenadas em um banco de dados SQL Server.



O processo de identificação começa na classe **Startup**, no método **ConfigureServices**, que é chamado automaticamente em tempo de execução. Esta é a hora e o local para adicionar os serviços que fazem parte da aplicação.

```
class Startup
{
    ...
    public void ConfigureServices(IServiceCollection services){
        ...
        services.AddIdentity< ApplicationUser, IdentityRole>()
        ...
    }
}
```

O método **AddIdentity** é um método de extensão presente na biblioteca **Microsoft.AspNetCore.Identity**, na classe **Microsoft.Extensions.DependencyInjection.IdentityServiceCollectionExtensions**.

Esse método (**AddEntity**) espera receber dois parâmetros de tipos: **User** e **Role** (usuário e papel). A declaração desse método é a seguinte:

```
public static IdentityBuilder AddIdentity< TUser, TRole >(
    this IServiceCollection services)
    where TUser : class
    where TRole : class;
```

Repare que os tipos **TUser** e **TRole** devem ser uma classe qualquer, nem mesmo uma interface está definida. A implementação é completamente livre. Na hora certa, o sistema vai instanciar essa classe. Esse processo se chama **injeção de dependência** e por isso está no namespace **xxx.DependencyInjection**.

Para o tipo **TUser** é passado o tipo **ApplicationUser**, e para o tipo **TRole**, o tipo **IdentityRole**.

A classe **ApplicationUser** está definida na aplicação, e a classe **IdentityRole**, no Entity Framework.

Adiante, o método completo `ConfigureServices`. Depois de o método `AddIdentity` ser executado, o método `AddEntityFrameworkStores` é chamado e, depois dele, o método `AddDefaultTokenProviders`.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<ApplicationContext>(
            options =>options.UseSqlServer(
                Configuration[
                    "Data:DefaultConnection:ConnectionString"]));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();
}

// Add application services.
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

`AddEntityFrameworkStores` é um método de extensão da classe `IdentityBuilder`, que é o tipo retornado quando uma identidade é adicionada.

Esse método espera uma parâmetro do tipo ou derivado de `DbContext`. Veja a declaração desse método de extensão:

```
namespace Microsoft.Extensions.DependencyInjection
{
    public static class IdentityEntityFrameworkBuilderExtensions
    {
        public static IdentityBuilder
            AddEntityFrameworkStores<TContext>(
                this IdentityBuilder builder)
            where TContext : DbContext;

        ....
    }
}
```

```
}
```

E, finalmente, o método **AddDefaultTokenProviders** é invocado na saída do método **AddEntityFrameworkStores**:

```
services.AddIdentity< ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores< ApplicationDbContext >()
    .AddDefaultTokenProviders();
```

AddDefaultTokenProviders é um método definido na classe **IdentityBuilder** e está no namespace **Microsoft.AspNetCore.Identity**. Token é uma informação usada para validar algumas ações, como trocar a senha ou alterar o e-mail.

O processo de identificação continua na classe **Startup**, no método **Configure**, que também é chamado automaticamente em tempo de execução. Esse método recebe uma instância da classe que implementa a interface **IApplicationBuilder**, e o método **UseIdentity()** é chamado.

```
class Startup
{
    public void Configure( IApplicationBuilder app,
                          IHostingEnvironment env
                          ILoggerFactory loggerFactory)
    {
        ...
        app.UseIdentity();
        ...
    }
}
```

O método **UseIdentity** é um método de extensão da classe **BuilderExtensions**, que está no assembly **Microsoft.AspNetCore.Identity**.

```
namespace Microsoft.AspNetCore.Builder
{
    public static class BuilderExtensions
    {
```

```
        public static IApplicationBuilder UseIdentity(this  
IApplicationBuilder app);  
    }  
}
```

As seguintes classes foram usadas até o momento:

```
namespace MeuApp  
    class Startup
```

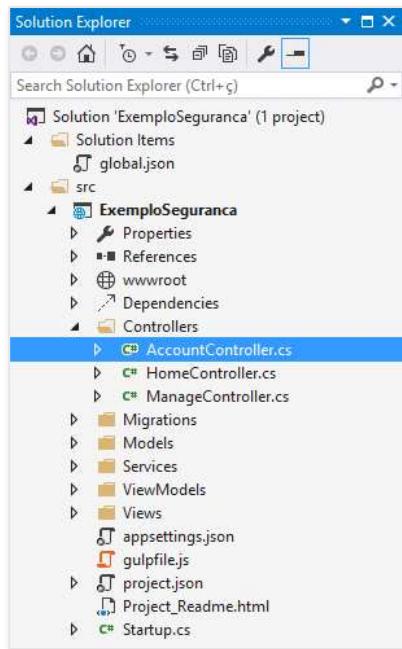
```
namespace MeuApp.Models  
    class ApplicationUser  
    class ApplicationDbContext
```

```
namespace Microsoft.AspNet.Identity.EntityFramework  
    class IdentityDbContext<TUser>  
    class IdentityRole
```

```
namespace Microsoft.Data.Entity  
    class DbContext
```

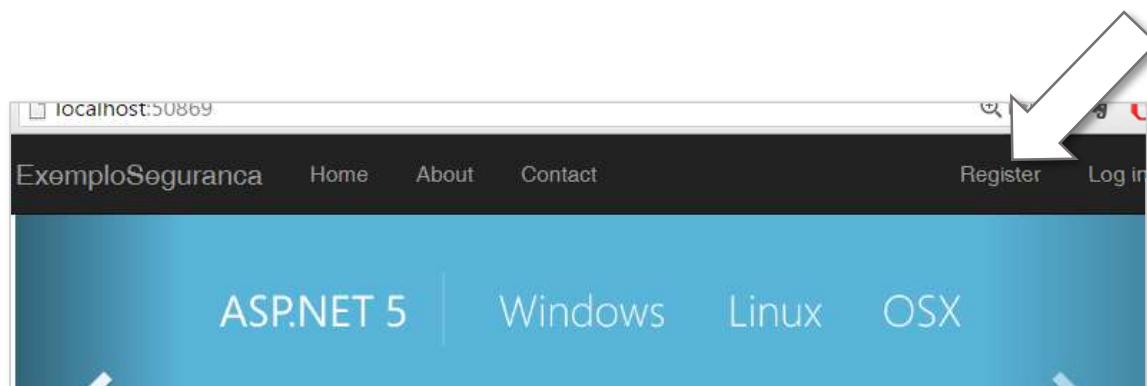
```
namespace Microsoft.AspNet.Identity  
    class IdentityBuilder  
    class BuilderExtensions
```

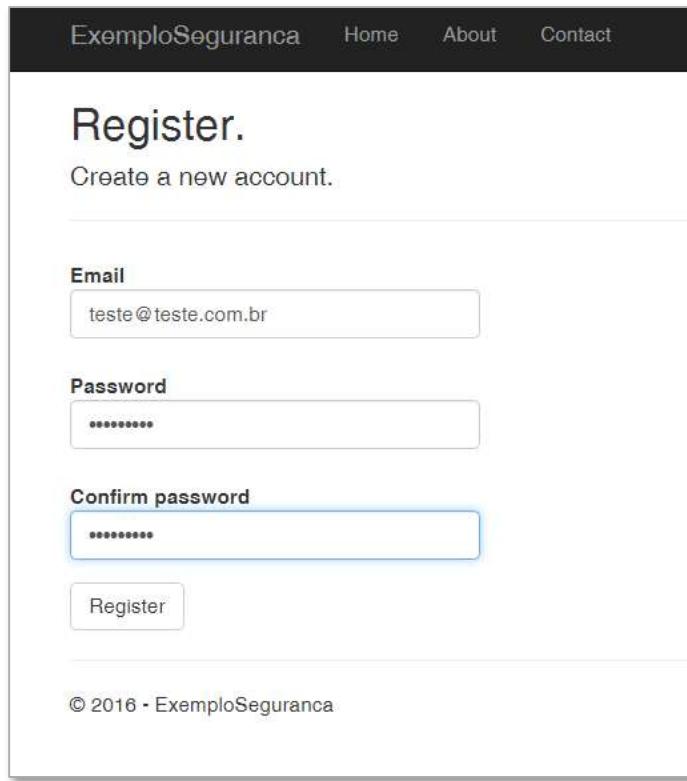
Para o registro de um usuário (**ApplicationUser**) é usado o método **Register** da classe **AccountController**:



```
public class AccountController : Controller
{
    ...
    public IActionResult Register()
    {
        return View();
    }
    ...
}
```

Quando o programa é executado, o item de menu **Register** direciona para a URL **Account/Register**, sendo executado o método correspondente e acionada a View:





Ao clicar no botão **Register**, o formulário HTML envia os dados usando o verbo POST, fazendo o MVC acionar o método correspondente:

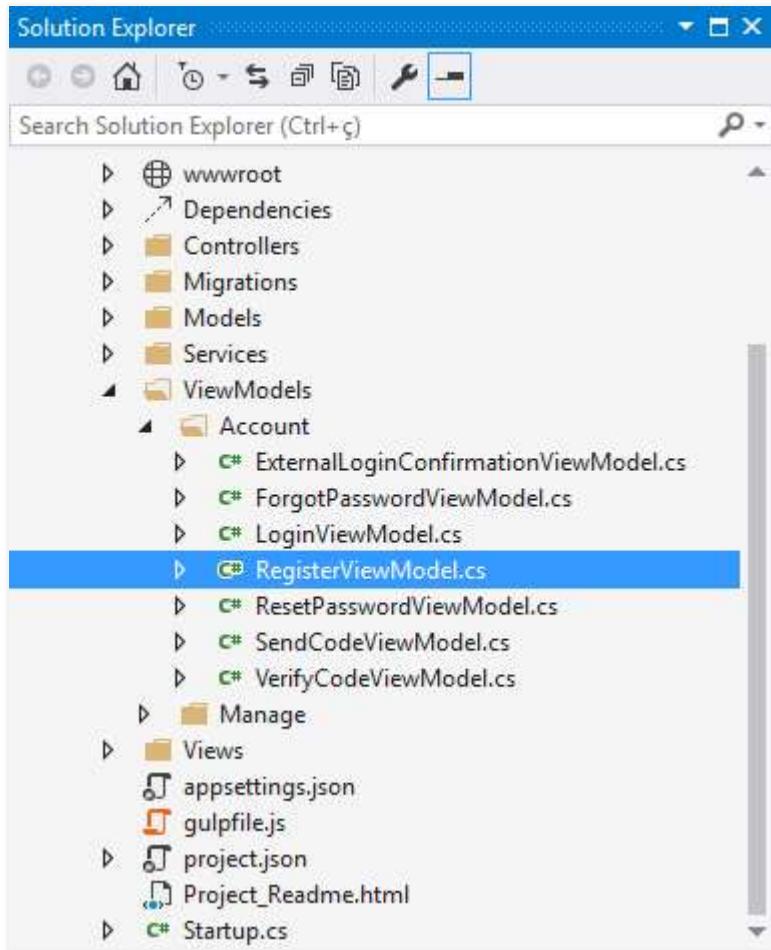
```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email,
                                         Email = model.Email };

        var result = await _userManager.CreateAsync(user,
                                                 model.Password);

        if (result.Succeeded) { ... (código omitido) }

        return View(model);
    }
}
```

O método **Register** recebe uma instância da classe **RegisterViewModel**, definida na pasta **ViewModels**:



A classe **RegisterViewModel** contém apenas o necessário ao cadastro de usuário: **Email**, **Senha** e **Confirmação da Senha**. As anotações foram omitidas na listagem adiante para maior clareza:

```
public class RegisterViewModel
{
    public string Email { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }
}
```

Dentro do método **Register**, uma instância de **ApplicationUser** é criada usando os dados fornecidos pelo usuário:

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    ...
    var user = new ApplicationUser
```

```
{ UserName = model.Email, Email = model.Email };  
...  
}
```

A linha seguinte grava os dados do usuário no banco de dados:

```
var result = await _userManager.CreateAsync(user, model.Password);
```

`_userManager` é uma variável local criada no construtor da classe **AccountController**. A classe **AccountController** recebe como parâmetro várias instâncias de classes que serão usadas para gravar o usuário, realizar a autenticação, enviar e-mail e registrar um log de atividades.

De onde vêm essas instâncias? É aí que entra a injeção de dependência. A classe **AccountController** não sabe quais classes concretas vai utilizar, mas conhece as interfaces. O mecanismo do ASP.NET é responsável por instanciar essas classes de acordo com as configurações feitas no início da aplicação, na classe **Startup**.

A classe **AccountController** não é responsável por autenticar, gravar ou enviar e-mails. Ela recebe instâncias dos responsáveis e apenas executa os dados. Esse mecanismo (injeção de dependência) é muito poderoso, pois permite interligar componentes com segurança e é a base de todo o .NET Core.

Para visualizar o processo de injeção de dependência, vamos analisar o código relativo ao envio de e-mails. Adiante, está a listagem da classe **AccountController** apenas com o construtor. Veja a declaração da interface **IEmailSender**.

```
public class AccountController : Controller  
{  
    private readonly UserManager< ApplicationUser > _userManager;  
    private readonly SignInManager< ApplicationUser >  
    _signInManager;  
    private readonly IEmailSender _emailSender;  
    private readonly ISmsSender _smsSender;  
    private readonly ILogger _logger;  
  
    public AccountController(  
        UserManager< ApplicationUser > userManager,
```

```

        SignInManager<ApplicationUser> signInManager,
        IEmailSender emailSender,
        ISmsSender smsSender,
        ILoggerFactory loggerFactory)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _emailSender = emailSender;
        _smsSender = smsSender;
        _logger = loggerFactory.CreateLogger<AccountController>();
    }
}

```

O que faz uma instância que implementa a interface **IEmailSender** cair dentro do construtor da classe **Account** é este método da classe **Startup**:

```

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddTransient<IEmailSender, AuthMessageSender>();
        ...
    }
}

```

O método **ConfigureServices** recebe como parâmetro uma classe que implementa a interface **IServiceCollection**. O objetivo dessa interface é gerenciar os componentes da aplicação e permitir que o sistema crie instâncias de classes que serão passadas para o construtor de um **Controller** se esse construtor esperar um parâmetro do tipo de algum serviço definido nessa lista.

O método **AddTransient<T1, T2>()** adiciona um serviço à coleção de serviços. É necessário passar como parâmetro a interface (**IEmailSender**) e a classe que implementa essa interface (**AuthMessageSender**).

Isso é tudo. Automaticamente, se algum **Controller** tiver um construtor que espere um parâmetro do tipo **IEmailSender**, o mecanismo do MVC vai criar uma instância da classe **EmailSender** e passar essa instância como parâmetro quando criar a classe **Controller**. É o que acontece na classe **AccountController**:

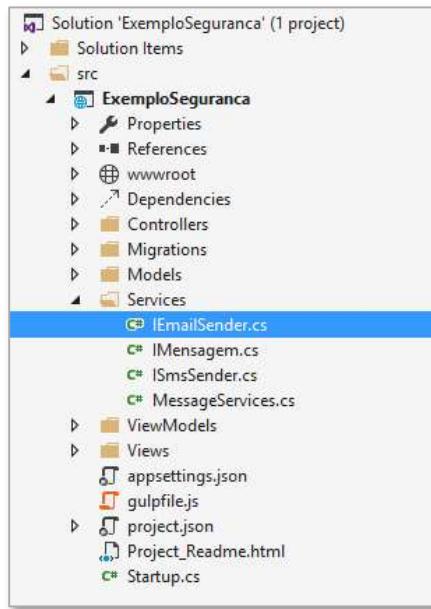
```
public class AccountController : Controller
```

```

{
...
public AccountController(
    UserManager< ApplicationUser > userManager,
    SignInManager< ApplicationUser > signInManager,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    ...
}

```

A interface **IEmailSender** e a classe que a implementa (que na verdade não tem nenhuma implementação) estão definidas na pasta **Services**:



A interface e a classe são apenas declarações, sem nenhum implementação real:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ExemploSeguranca.Services
{
    public interface IEmailSender
    {
        Task SendEmailAsync(string email, string subject, string
message);
    }
}

```

```

        }

    }

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ExemploSegurança.Services
{
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public Task SendEmailAsync(string email, string subject,
string message)
        {
            return Task.FromResult(0);
        }

        public Task SendSmsAsync(string number, string message)
        {
            return Task.FromResult(0);
        }
    }
}

```

Apenas para relembrar como está definido na classe **Startup**:

```

public class Startup
{

    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddTransient<IEmailSender, AuthMessageSender>();
        ...
    }
}

```

● Autenticação

Uma vez identificado o usuário, é necessário autenticá-lo. É muito comum usar cookies para a autenticação. Logo após o registro do usuário no banco de dados, ele é autenticado no sistema e um cookie de autenticação é gerado.

```

public async Task<ActionResult> Register(RegisterViewModel model)
{

```

```

    if (ModelState.IsValid)
    {
        var user = new ApplicationUser ....
        var result = await _userManager.CreateAsync...

        if (result.Succeeded)
        {

            await _signInManager.SignInAsync(user, isPersistent: false);

            _logger.LogInformation(3, "Usuário criado.");
        }
        else
            AddErrors(result);
    }
    return View(model);
}

```

Uma instância de **SignInManager** é criada pelo ASP.NET Core.

● Autorização

A autorização consiste em proibir ou permitir acesso a recursos do sistema. Nesta versão, a autorização é definida usando **DataAnnotations** nas classes de controle. O atributo **Authorize** define que apenas usuários autenticados podem acessar:

```

[Authorize]
public class AccountController : Controller

```

Para liberar um método dessa exigência, usa-se o atributo **AllowAnonymous**:

```

[AllowAnonymous]
public IActionResult Login(string returnUrl = null)

```

● Teste de login e autorização

Para testar o processo de autorização, é necessário tentar entrar em uma página proibida para usuários anônimos. Depois, fazer o login e tentar novamente. Dentro da classe **AccountManager** foi criado este método:

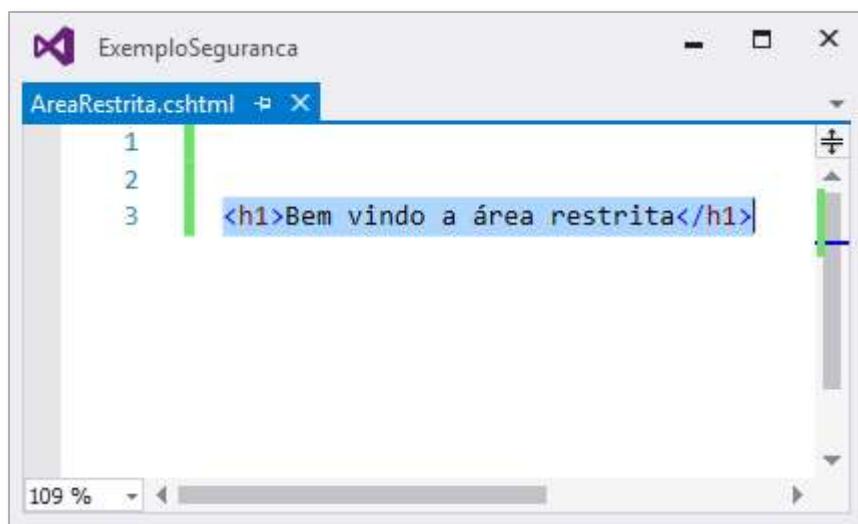
```

public ViewResult AreaRestrita()

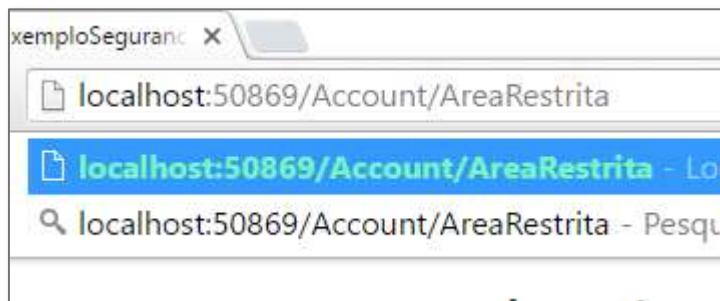
```

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

E dentro da pasta **Views**, um documento razor chamado **AreaRestrita.cshtml**:



- Tentando visualizar (não é possível, sempre cai no login):



- Fazendo o login:

ExemploSegurança Home About Contact

Log in.

Use a local account to log in.

Email

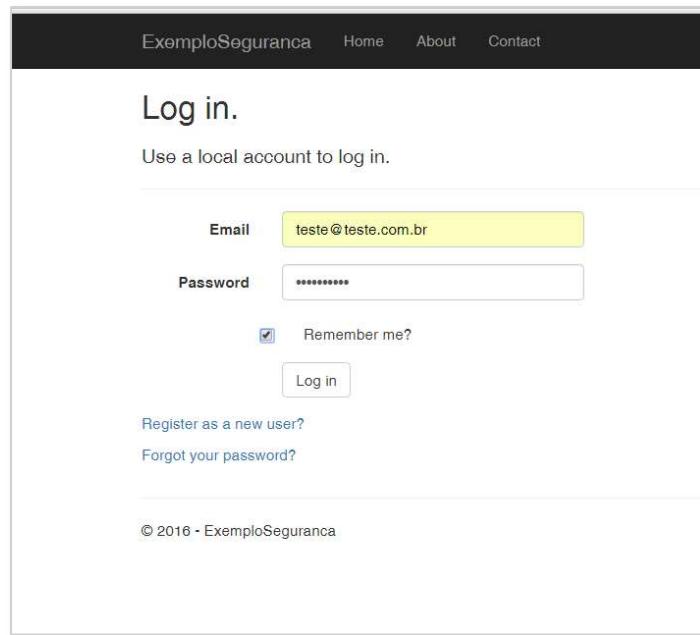
Password

Remember me?

[Register as a new user?](#)

[Forgot your password?](#)

© 2016 - ExemploSegurança



- E agora está disponível:

ExemploSegurança Home About Contact

Bem vindo a área restrita

© 2016 - ExemploSegurança



10.3. Projeto: recursos para o Identity Core

10.3.1. Definindo as entidades

Nosso projeto foi criado considerando o recurso de contas de usuário individuais, mas apresentaremos também as opções de inclusão.

Vamos criar a classe **Usuario** na pasta **Models**, responsável por representar o usuário (observe a superclasse):

```
public class Usuario : IdentityUser
{
}
```

É sempre uma boa prática criar uma subclasse de **IdentityUser**, pois este recurso permite adicionar propriedades relevantes para a aplicação.

Na pasta **Dados**, definir a classe que representa o acesso a dados, o **DbContext**. Verifique a sua superclasse (o parâmetro **options** será passado via injeção de dependência):

```
namespace DesenvolvimentoWeb.Projeto02.Dados
{
    public class UsuariosDbContext : IdentityDbContext<Usuario>
    {
        public UsuariosDbContext(DbContextOptions<UsuariosDbContext>
options) :
            base(options)
        {

        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
        }
    }
}
```

10.3.2. Configuração no arquivo Startup.cs

Abrir o arquivo **Startup.cs**. Inclua a configuração no método **ConfigureServices**, alterando a configuração original, considerando que o recurso **Contas de Usuário Individual** tenha sido selecionada no início do projeto:

```
public void ConfigureServices(IServiceCollection services)
{
    //services.AddDbContext<ApplicationDbContext>(options =>
    //    options.UseSqlServer(Configuration
    //    .GetConnectionString("DefaultConnection")));

    services.AddDbContext<UsuariosDbContext>(
        options => options.UseSqlServer(Configuration
            .GetConnectionString("DefaultConnection")));
}

services.AddDbContext<EventosContext>(
    options => options.UseSqlServer(Configuration
        .GetConnectionString("EventosConnection")));

//services.AddIdentity< ApplicationUser, IdentityRole>()
//    .AddEntityFrameworkStores<ApplicationDbContext>()
//    .AddDefaultTokenProviders();

services.AddIdentity< Usuario, IdentityRole>()
    .AddEntityFrameworkStores< UsuariosDbContext>()
    .AddDefaultTokenProviders();

services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/ Usuarios/Login";
    options.LogoutPath = "/ Usuarios/Logout";
});

// Add application services.
services.AddTransient< IEmailSender, EmailSender>();

services.AddMvc();
}
```

Ainda no arquivo **Startup.cs**, acesse o método **Configure()**, e inclua o código a seguir para adicionar o recurso de autenticação na aplicação (se não tiver sido incluído):

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env)
```

```

{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseAuthentication();

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

```

Estra opção será necessária se o projeto foi criado do zero!
Migrations (se necessário)

Através do NuGet vamos instalar os pacotes, anotando as versões:

```

Install-packages Microsoft.EntityFrameworkCore.Tools (2.0.1)
Install-packages Microsoft.EntityFrameworkCore.Tools.DotNet (2.0.3)

```

Clicar com o direito do mouse no projeto e selecionar o arquivo **.csproj**.

Adicionar os comandos:

```

<Project Sdk="Microsoft.NET.Sdk.Web">

    <PropertyGroup>
        <TargetFramework>netcoreapp2.0</TargetFramework>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
            Version="2.0.1" />
        <PackageReference Include="Microsoft.jQuery.Unobtrusive.Ajax"
            Version="3.2.3" />
    </ItemGroup>

```

```

<PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design"
    Version="2.0.1" />
</ItemGroup>

<ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools"
        Version="2.0.1" />
    <DotNetCliToolReference
    Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.1" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
        Version="2.0.1" />
</ItemGroup>

</Project>

```

No prompt de comandos, executar as instruções:

```

dotnet ef migrations add Initial --context UsuariosDbContext
dotnet ef database update --context UsuariosDbContext

```

Verifique se o banco de dados foi criado com sucesso!

10.3.3. Definição dos Models

Vamos agora trabalhar nas páginas de registro e validação de usuários. Na pasta **Models**, incluir as classes:

```

namespace DesenvolvimentoWeb.Projeto02.Models
{
    public class UsuarioViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        [StringLength(100, ErrorMessage = "Mínimo de 6 caracteres.",
            MinimumLength = 6)]
        public string Senha { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirma Senha")]
        public string ConfirmaSenha { get; set; }
    }
}

```

```

        }

    }

namespace DesenvolvimentoWeb.Projeto02.Models
{
    public class LogonViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Senha { get; set; }

    }
}

```

10.3.4. Criando o controller UsuariosController

Neste controller já apresentaremos os métodos **Registrar** e **Login**. Analise seu código:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using DesenvolvimentoWeb.Projeto02.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Authentication;

namespace DesenvolvimentoWeb.Projeto02.Controllers
{
    public class UsuariosController : Controller
    {
        private readonly UserManager<Usuario> _userManager;
        private readonly SignInManager<Usuario> _signInManager;

        public UsuariosController(UserManager<Usuario> userManager,
            SignInManager<Usuario> signInManager)
        {
            this._userManager = userManager;
            this._signInManager = signInManager;
        }
    }
}

```

```

    }

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

    //Métodos Auxiliares
    private void AddErrors(IdentityResult result)
    {
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty,
error.Description);
        }
    }

    private IActionResult RedirectToLocal(string returnUrl)
    {
        if (Url.IsLocalUrl(returnUrl))
        {
            return Redirect(returnUrl);
        }
        else
        {
            return RedirectToAction(nameof(HomeController.Index),
"Home");
        }
    }

    //Métodos de Registro
    [HttpGet]
    [AllowAnonymous]
    public IActionResult Registrar(string returnUrl = null)
    {
        ViewBag.ReturnUrl = returnUrl;
        return View();
    }

    [HttpPost]
    [AllowAnonymous]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Registrar(UsuarioViewModel
model,
        string returnUrl = null)
    {
        ViewBag.ReturnUrl = returnUrl;
        if (ModelState.IsValid)
        {
            var user = new Usuario { UserName = model.Email,
Email = model.Email };
            var result = await _userManager.CreateAsync(user,
model.Senha);
        }
    }
}

```

```

        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user,
isPersistent: false);
            return RedirectToLocal(returnUrl);
        }
        AddErrors(result);
    }
    return View(model);
}

//Login
[HttpGet]
[AllowAnonymous]
public async Task<IActionResult> Login(string returnUrl =
null)
{
    await
HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);

    ViewBag.ReturnUrl = returnUrl;
    return View();
}

[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LogonViewModel model,
    string returnUrl)
{
    ViewBag.ReturnUrl = returnUrl;

    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(
model.Email, model.Senha, false, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            return RedirectToLocal(ReturnUrl);
        }
        if (result.IsLockedOut)
        {
            return RedirectToAction(nameof(Lockout));
        }
        else
        {
            ModelState.AddModelError(string.Empty,
"Usuário ou senha inválidos.");
            return View(model);
        }
    }
}

return View(model);
}

```

```

[HttpGet]
[AllowAnonymous]
public IActionResult Lockout()
{
    return View();
}

//Logout
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return RedirectToAction(nameof(HomeController.Index),
"Home");
}
}

```

10.3.5. Definindo as views

Um arquivo que permite apresentar todos os recursos a serem importados pelas classes é o **_ViewImports.cshtml**. Verifique seu conteúdo, se possuir os itens assinalados. Se não possuir, incluí-los.

```

@using Microsoft.AspNetCore.Identity
@using DesenvolvimentoWeb.Projeto02
@using DesenvolvimentoWeb.Projeto02.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

```

Precisamos agora gerar as views **Registrar** e **Login**.

Registrar.cshtml

```

@model DesenvolvimentoWeb.Projeto02.Models.UsuarioViewModel

 @{
    ViewData["Title"] = "Registrar";
}

<h2>Registrar</h2>

```

```

<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Registrar">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
                <div class="form-group">
                    <label asp-for="Email" class="control-label"></label>
                    <input asp-for="Email" class="form-control" />
                    <span asp-validation-for="Email" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Senha" class="control-label"></label>
                    <input asp-for="Senha" class="form-control" />
                    <span asp-validation-for="Senha" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="ConfirmaSenha" class="control-label"></label>
                    <input asp-for="ConfirmaSenha" class="form-control" />
                    <span asp-validation-for="ConfirmaSenha" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <input type="submit" value="Create" class="btn btn-default" />
                </div>
            </form>
        </div>
    </div>

```

Login.cshtml (observe as mudanças realizadas)

```

@model DesenvolvimentoWeb.Projeto02.Models.LogonViewModel
@inject SignInManager<Usuario> SignInManager
#{@
    ViewData["Title"] = "Login";
}

<h2>Login</h2>

<h4>LogonViewModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-route-returnurl="@ViewBag.ReturnUrl">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
                <div class="form-group">
                    <label asp-for="Email" class="control-label"></label>

```

```

        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-
danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Senha" class="control-label"></label>
        <input asp-for="Senha" class="form-control" />
        <span asp-validation-for="Senha" class="text-
danger"></span>
    </div>
    <div class="form-group">
        <input type="submit" value="Create" class="btn btn-
default" />
    </div>
</form>
</div>
</div>

```

Criar a view parcial na pasta **views/shared** chamada **_LoginPartial.cshtml** (se existir, alterar seu conteúdo):

```

@using Microsoft.AspNetCore.Identity
@using DesenvolvimentoWeb.Projeto02.Models

@inject SignInManager<Usuario> SignInManager
@inject UserManager<Usuario> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="" asp-controller="Usuarios" asp-action="Logout"
method="post" id="logoutForm" class="navbar-right">
    <ul class="nav navbar-nav navbar-right">
        <li>
            <a asp-area="" asp-controller="Usuarios" asp-
action="Verificar"
                title="Verificar">Olá@
UserManager.GetUserName(User)!</a>
        </li>
        <li>
            <button type="submit" class="btn btn-link navbar-btn
navbar-link">Logout</button>
        </li>
    </ul>
</form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="" asp-controller="Usuarios" asp-
action="Registrar">

```

```

        Novo usuário</a></li>
    <li><a asp-area="" asp-controller="Usuarios" asp-
action="Login">
        Log in</a></li>
</ul>
}

```

Alterar o arquivo `_Layout.cshtml` de forma a incluir esta view parcial (na execução, observe no meu à direita):

```

<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-area="" asp-controller="Home" asp-
action="Index">Home</a></li>
        <li><a asp-area="" asp-controller="Eventos" asp-action="Index">
            Gestão de Eventos</a></li>
        <li><a asp-area="" asp-controller="JScript" asp-action="Index">
            Exemplos Javascript</a></li>
    </ul>
    @await Html.PartialAsync("_LoginPartial")
</div>

```

10.3.6. Verificando os actions que devem ser controlados por login

Uma vez definidos os actions para **registro** e **login**, vamos agora controlar o que deve ser executado por um usuário autenticado.

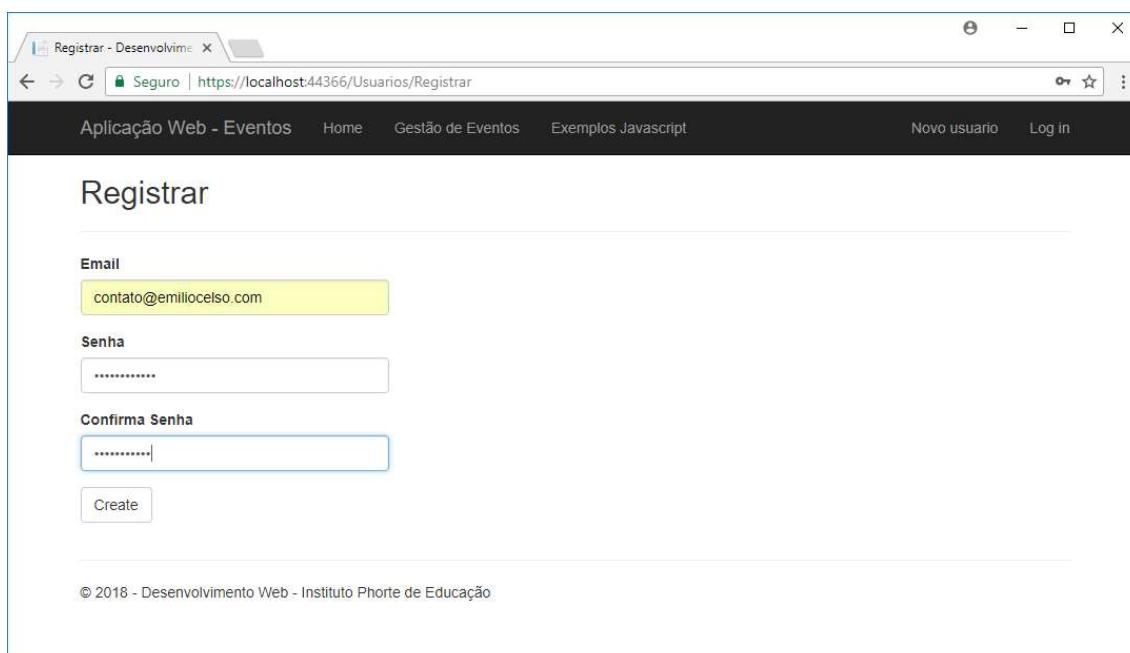
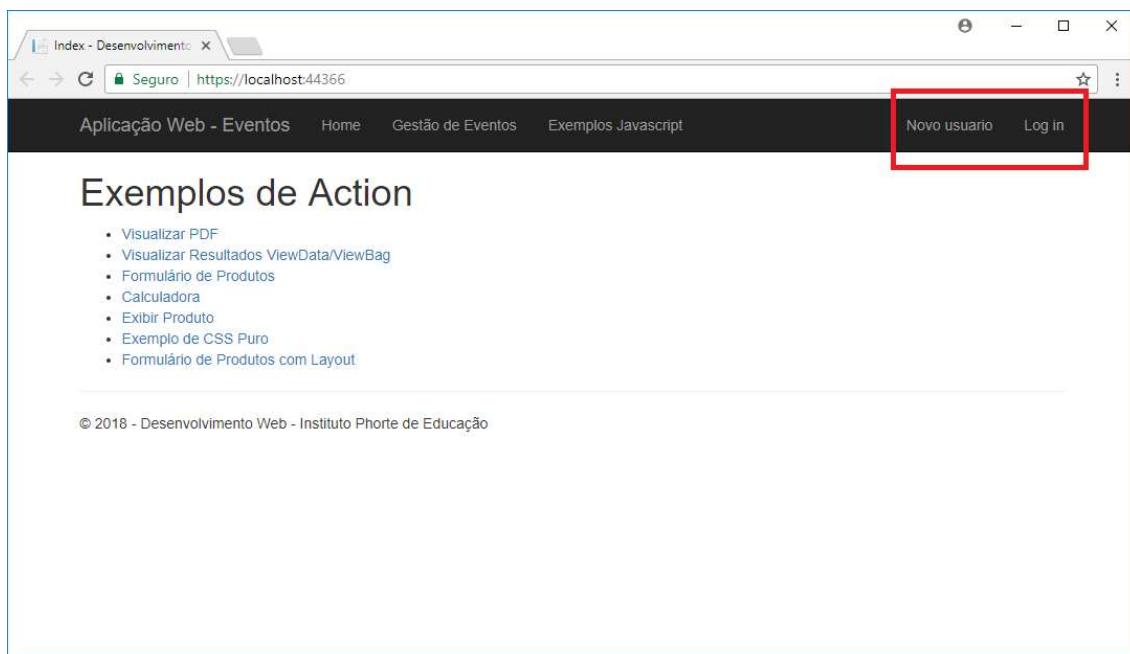
Considere, por exemplo, que para realizar um cadastro de Eventos, o usuário deva ser, por exemplo, um administrador devidamente logado. No action **IncluirEvento** devemos incluir o atributo:

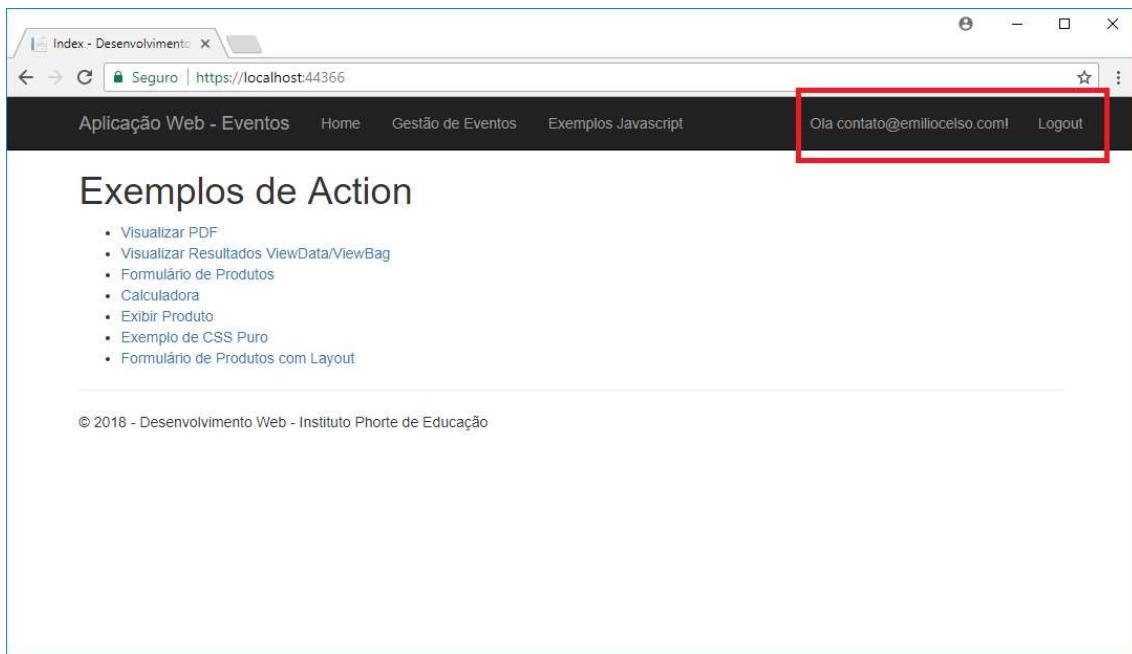
```

[Authorize]
public IActionResult IncluirEvento()
{
    return View();
}

```

Vamos agora executar a aplicação. Primeiro, vamos criar um novo usuário:





Faça logoff. Tente agora incluir um novo evento. Observe que você é redirecionado para a tela de login. Ao fazer o login, você é direcionado para sua requisição original, ou seja para a inclusão de um novo evento.

10.3.7. Removendo arquivos desnecessários

Se o projeto NÃO foi criado do zero (se foi incluído o recurso de contas de usuário) temos certamente vários arquivos desnecessários. É importante fazer uma limpeza destes arquivos e suas configurações, para que não tenhamos um projeto poluído demais.

Podemos remover os arquivos abaixo:

Pasta Controllers:

AccountController,
ManageController

Pasta Models:

Pasta AccountViewModels,
Pasta ManageViewModels,
Arquivo ErrorViewModel,

Pasta Views:

Pasta Account,

Pasta Manage

Pasta Views/Home:

Arquivos About, Contact

Deixar o arquivo **_ViewImports.cshtml** com a configuração abaixo:

```
@using Microsoft.AspNetCore.Identity  
@using DesenvolvimentoWeb.Projeto02  
@using DesenvolvimentoWeb.Projeto02.Models  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Observações:

- Recompilar o projeto e remover as referencias que estiverem causando erros, se for o caso.
- Se estiver inseguro, mantenha como está. O importante é o conceito do curso. Com o tempo e a experiência, você saberá facilmente o que deve ou não permanecer.

11. WebAPI Core

11.1. Introdução

WebAPI é a tecnologia da Microsoft para criar e consumir serviços baseados em REST.

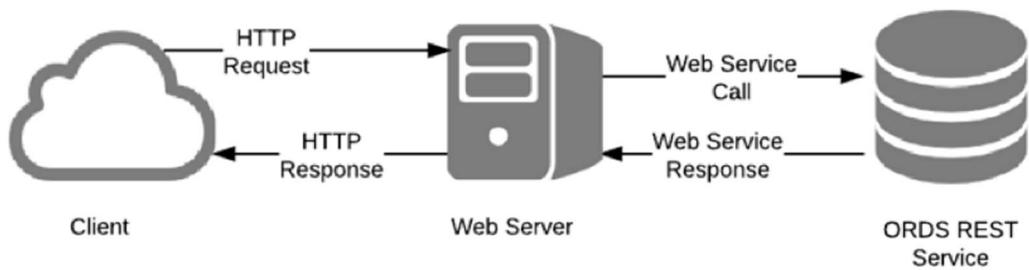
REST significa **Representation State Transfer**. Um recurso no sentido RESTful é qualquer coisa que tenha uma URI representando seus recursos.

Então, Web Services RESTful exigem não apenas recursos para representar, mas também operações chamadas pelo cliente destes recursos. No centro da abordagem RESTful está a percepção de que HTTP é uma API e não simplesmente um protocolo de transporte. HTTP tem seus verbos bem conhecidos, oficialmente conhecidos como métodos.

Uma diretriz importante do estilo RESTful é respeitar os significados originais dos verbos HTTP.

A abordagem REST não implica que os recursos ou processamento necessários para gerar representações adequadas deles sejam simples. A abordagem RESTful mantém a complexidade referente ao mecanismo de transferência fora do nível de transporte, conforme uma representação de recurso é transferida para o cliente como o corpo de uma mensagem de resposta HTTP.

11.2. Representação de um Webservice REST



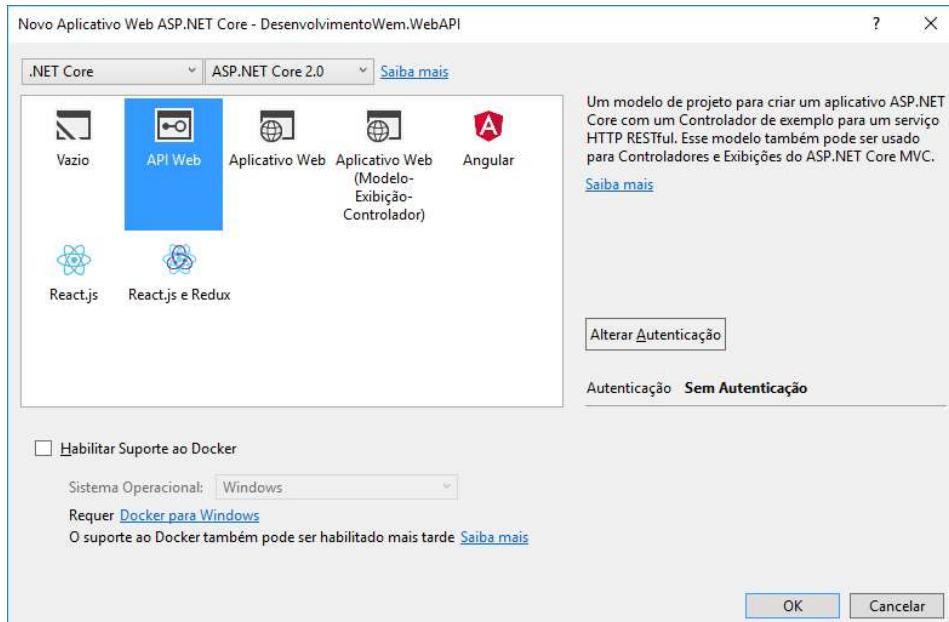
Fonte: <https://smartdogservices.com/http-headers-and-ords-rest-web-services/>

Considere que uma aplicação web necessite se comunicar com outra aplicação. O Webservice é o canal de comunicações entre as aplicações.

Nesta fase da aplicação criaremos um novo projeto representando um webservice. O objetivo é simular uma administradora de cartões de crédito. Nosso novo projeto conterá seu próprio banco de dados, e as informações sobre os pagamentos serão recebidas do cliente.

11.3. Projeto: Criação do projeto WebAPI

Criar um novo projeto (não na mesma solução) chamado **DesenvolvimentoWeb.WebAPI**. Escolher a opção **Web API**.



Vamos criar a pasta **Models** e nesta pasta, a classe **Pagamento**:

```
namespace DesenvolvimentoWem.WebAPI.Models
{
    public class Pagamento
    {
        public int Id { get; set; }
        public int IdEvento { get; set; }
        public string Cpf { get; set; }
        public string NumeroCartao { get; set; }
        public double Valor { get; set; }
        public int Status { get; set; }
    }
}
```

Nesta mesma pasta criar a classe **PagamentoContext**:

```
namespace DesenvolvimentoWem.WebAPI.Models
{
    public class PagamentoContext : DbContext
    {
        public PagamentoContext(DbContextOptions<PagamentoContext>
            options)
            : base(options) { }

        public DbSet<Pagamento> Pagamentos { get; set; }
    }
}
```

Vamos registrar o **DbContext** com o container de injeção de dependência. Estes serviços estarão disponíveis para os controladores. Este trabalho será realizado na classe **Startup.cs**, método **ConfigureServices**:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<PagamentoContext>(options =>
        options.UseSqlServer(Configuration
            .GetConnectionString("ApiConnection")));
    services.AddMvc();
}
```

Abrir o arquivo **appsettings.json**. Incluir o trecho destacado, representando a string de conexão:

```
{
  "ConnectionStrings": {
    "ApiConnection": "Server=(localdb)\\mssqllocaldb;Database=DBPagamentos;ConnectRetryCount=0;
Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  }
}
```

Na pasta **Models**, incluir o arquivo **DBInitializer.cs** com o conteúdo indicado (esta classe, como no outro projeto, será usada para criar o banco de dados):

```
namespace DesenvolvimentoWem.WebAPI.Models
```

```

{
    public static class DbInitializer
    {
        public static void Initialize(PagamentoContext context)
        {
            context.Database.EnsureCreated();
        }
    }
}

```

Na classe **Program.cs**, importar o namespace **Microsoft.Extensions.DependencyInjection**. Em seguida abrir o método Main():

```

public static void Main(string[] args)
{
    //BuildWebHost(args).Run();
    var host = BuildWebHost(args);
    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services
                .GetRequiredService<PagamentoContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services
                .GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, ex.Message);
            throw;
        }
    }
    host.Run();
}

```

Em seguida, executar esta aplicação e verificar se o banco de dados foi criado.

11.3.1. Definindo o controller para o serviço

Adicionar um **controller** na pasta **Controllers** chamado **PagamentosController**. A opção deve ser **Api Controller**. Em seguida substituir o conteúdo pelo indicado abaixo:

```

namespace DesenvolvimentoWem.WebAPI.Controllers
{
    [Produces("application/json")]
    [Route("api/Pagamentos")]
    public class PagamentosController : Controller
    {
        private readonly PagamentoContext _context;

        public PagamentosController(PagamentoContext context)
        {
            this._context = context;
        }
    }
}

```

Adicionar agora os métodos com os serviços:

```

namespace DesenvolvimentoWem.WebAPI.Controllers
{
    [Produces("application/json")]
    [Route("api/Pagamentos")]
    public class PagamentosController : Controller
    {
        private readonly PagamentoContext _context;

        public PagamentosController(PagamentoContext context)
        {
            this._context = context;
        }

        //HTTP GET - Listar Todos os pagamentos
        [HttpGet]
        public IEnumerable<Pagamento> GetAll()
        {
            return _context.Pagamentos.ToList<Pagamento>();
        }

        //HTTP GET - Busca por um pagamento
        [HttpGet("{id}", Name = "GetPagamento")]
        public IActionResult GetById(long id)
        {
            var item = _context.Pagamentos.FirstOrDefault(p => p.Id == id);
            if (item == null)
            {
                return NotFound();
            }
            return new ObjectResult(item);
        }
}

```

```

//HTTP POST - Inclusão de um pagamento
public IActionResult PostPagamento([FromBody]Pagamento pagto)
{
    var pagamento = _context.Pagamentos.FirstOrDefault(p =>
        p.IdEvento == pagto.IdEvento && p.Cpf.Equals(pagto.Cpf));

    if (pagamento == null)
    {
        pagto.Status = 1;
        _context.Pagamentos.Add(pagto);
        _context.SaveChanges();
        return CreatedAtRoute("GetPagamento", new { id =
pagto.Id }, pagto);
    }
    else
    {
        return BadRequest();
    }
}

}

```

Para deixar o projeto mais limpo, vamos remover o controller que havia sido gerado: o **ValuesController**. Este era um controller de exemplo produzido pelo template do Visual Studio, e não precisamos mais dele.

11.3.2. Executando a aplicação

Podemos executar a aplicação para que o banco de dados seja gerado, mas não temos nenhum registro.

Se a aplicação for executada, você verá que a url apontará para o controller que apagamos:

/api/values

Podemos alterar este controle no arquivo **launchSettings.json**, na pasta Properties. Abra este arquivo, e altere as propriedades assinaladas abaixo:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:58378/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "api/pagamentos",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "DesenvolvimentoWem.WebAPI": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "api/pagamentos",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:58379/"
    }
  }
}
```

Mesmo que esta configuração não seja realizada, no browser devemos indicar a url do controller que estamos trabalhando.

O resultado da execução é mostrado a seguir:



Nada de muito interessante, mas temos um array vazio, indicando que não temos nada no banco de dados. Se incluirmos um pagamento manualmente, por exemplo, teremos os dados do pagamento no formato **JSON**.

Nosso próximo passo é configurar a aplicação para que consuma este webservice. O participante poderá realizar o pagamento do evento.

11.3.3. Consumindo o webservice na aplicação web

Vamos realizar as alterações necessárias no projeto para contemplar o recurso de pagamento.

Na pasta **Models**, criar a classe **PagamentoEvento**. Esta classe deve ter as mesmas propriedades da classe **Pagamento**, descrita no webservice (esse procedimento é importante para evitar configurações desnecessárias para o nosso propósito):

```
namespace DesenvolvimentoWeb.Projeto02.Models
{
    public class PagamentoEvento
    {
        public int IdEvento { get; set; }
        public string Cpf { get; set; }
        public string NumeroCartao { get; set; }
        public double Valor { get; set; }
        public int Status { get; set; }
    }
}
```

No arquivo **ListarParticipantes.cshtml**, definir um link direcionando o usuário para o pagamento:

```
@model IEnumerable<DesenvolvimentoWeb.Projeto02.Models.Participante>
 @{
    ViewData["Title"] = "ListarParticipantes";
}

<h2>Listar Participantes</h2>
```

```

<form asp-action="ListarParticipantes" asp-controller="Eventos"
method="get">

    <label class="control-label col-sm-1">ID Evento</label>
    <div class="col-sm-5">
        <select id="id" name="idEvento" asp-
items="(SelectList)ViewBag.Eventos"
            class="form-control col-sm-6"></select>
    </div>
    <div class="form-group">
        <input type="submit" value="Buscar" class="btn btn-default" />
    </div>
</form>

<table class="table">
    <thead>
        <tr>

            <th>
                @Html.DisplayNameFor(model => model.Nome)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Email)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Cpf)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.DataNascimento)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>

                <td>
                    @Html.DisplayFor(modelItem => item.Nome)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Email)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Cpf)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.DataNascimento)
                </td>
                <td>
                    @Html.ActionLink("Efetuar Pagamento",
                    "EfetuarPagamento",

```

```

                new { id = item.Id, idEvento = item.IdEvento
})
        </td>
    </tr>
}
</tbody>
</table>

```

Vamos agora definir o action **EfetuarPagamento**, responsável por gerar a view para fornecer os dados do cartão de crédito, e a versão POST, que efetiva o pagamento no webservice:

```

//Acesso ao Webservice
public IActionResult EfetuarPagamento(int id, int idEvento)
{
    var participante = Context.Participantes
        .FirstOrDefault(p => p.Id == id);
    var evento = Context.Eventos.FirstOrDefault(e => e.Id ==
        idEvento);

    PagamentoEvento pagamento = new PagamentoEvento();
    pagamento.IdEvento = evento.Id;
    pagamento.Cpf = participante.Cpf;
    pagamento.Valor = evento.Preco;

    return View(pagamento);
}

[HttpPost]
public async Task EfetuarPagamento(
    PagamentoEvento pagamento)
{
    //a ser implementado
    try
    {
        HttpClient client = new HttpClient();
        client.BaseAddress = new Uri("http://localhost:49860/");
        client.DefaultRequestHeaders.Accept.Add(new
            MediaTypeWithQualityHeaderValue("application/json"));

        string json = JsonConvert.SerializeObject(pagamento);
        HttpContent content = new StringContent(json,
            Encoding.Unicode, "application/json");

        var response = await client.PostAsync("api/pagamentos",
            content);

        if (response.IsSuccessStatusCode)
        {

```

```

        return RedirectToAction("ListarParticipantes");
    }
    else
    {
        string msg = response.StatusCode + " - " +
            response.ReasonPhrase;
        ViewBag.Erro = msg;
        return View("Erro");
    }
}
catch (Exception ex)
{
    ViewBag.Erro = ex.Message;
    return View("Erro");
}
}

```

Duas observações importantes:

- O projeto referente ao webservice deve estar em execução, para que possamos acessá-lo.
- O link mostrado neste exemplo deve ser alterado para o link correto no momento da sua execução, pois o número da porta certamente mudará de projeto para projeto.

Nós indicamos uma view chamada **Erro**. Vamos criá-la na pasta **Views/Shared**:

```

@{
    ViewBag.Title = "Erro";
}

<h2>Erro</h2>
<div class="alert alert-danger">
    @ViewBag.Erro
</div>

```

Por último, vamos adicionar a view para efetuar o pagamento, usando o template **Create** com o model **PagamentoEvento**. Observe as alterações realizadas no início da view:

```

@model DesenvolvimentoWeb.Projeto02.Models.PagamentoEvento

 @{
    ViewData["Title"] = "EfetuarPagamento";
}

```

```

<h2>EfetuarPagamento</h2>

<h4>PagamentoEvento</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="EfetuarPagamento">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <input type="hidden" asp-for="IdEvento" class="form-control" />
                @*<label asp-for="IdEvento" class="control-label"></label>
                <input asp-for="IdEvento" class="form-control" />
                <span asp-validation-for="IdEvento" class="text-danger"></span>*@
            </div>
            <div class="form-group">
                <label asp-for="Cpf" class="control-label"></label>
                <input asp-for="Cpf" class="form-control" />
                <span asp-validation-for="Cpf" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="NumeroCartao" class="control-label"></label>
                <input asp-for="NumeroCartao" class="form-control" />
                <span asp-validation-for="NumeroCartao" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Valor" class="control-label"></label>
                <input asp-for="Valor" class="form-control" />
                <span asp-validation-for="Valor" class="text-danger"></span>
            </div>
            @*<div class="form-group">
                <label asp-for="Status" class="control-label"></label>
                <input asp-for="Status" class="form-control" />
                <span asp-validation-for="Status" class="text-danger"></span>
            </div>*@
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

```

Testar a aplicação, fornecendo os dados do cartão. Após a inclusão, testar o webservice para ver se a lista de pagamentos no formato Json aparece no browser.