

INFORMÁTICA PARA INTERNET

Como sugestão, configure a opção de leitura de caracteres e pontuação de seu leitor de tela para o grau máximo de leitura, para que os códigos disponibilizados neste material sejam lidos corretamente. No caso do NVDA (NonVisual Desktop Access), para localizar a opção Grau de pontuação/símbolos, acesse Preferências – Configurações – Fala. Altere o padrão Pouco para Tudo. Dessa forma, o leitor passará a ler os segmentos de código em sua totalidade.

Arquitetura *web* em camadas: conceitos de camadas *front-end* e *back-end*

Uma página ou um sistema *web* dinâmico depende invariavelmente da existência de uma parte *back-end*, que executa em servidor cálculos, processamentos ou operações com dados, e uma parte *front-end*, que é alimentada pelo processamento do *back-end* e responsável por montar a visualização ao usuário e permitir a interação com o sistema. Independentemente da linguagem utilizada, a lógica para um sistema *web* sempre acabará seguindo essa premissa essencial.

No caso do .NET Core, a aplicação *web* é desenvolvida a partir do *framework* ASP.NET Core, que permite a programação do *back-end* (normalmente com linguagem C#) e do *front-end* (geralmente com arquivos CSHTML). Um modelo específico de aplicação, foco do ASP.NET desde 2009, é o ASP.NET MVC, baseado em um popular padrão *model view controller* (MVC).

Neste material, você verá como o *front-end* e o *back-end* se comportam no ASP.NET Core, além de estudar o padrão MVC e a sua aplicação no ASP.NET Core MVC. Para melhor entendimento, leia os materiais **Ambiente de desenvolvimento: instalação e configuração** e **Fundamentos de programação e linguagens de *script***, ambos disponíveis nesta unidade curricular.

Aplicações *web* dinâmicas com ASP.NET Core

Pode-se dizer que uma aplicação *web* composta apenas de uma camada *front-end* é essencialmente estática. Sempre que o usuário requisitar a página (por URL [*uniform resource locator*] no navegador, por exemplo), o resultado será o mesmo. Por mais que interações sejam incluídas por meio de JavaScript ou *hiperlinks*, as alterações são apenas superficiais, acontecendo diretamente no *browser*, sem comunicação com um servidor.



Figura 1 – Página com uma postagem do Stack Overflow Fonte: <http://stackoverflow.com>.

A figura 1 exibe um *site* de perguntas e respostas sobre programação. Quase todos os elementos da página são dinâmicos. O *site* é feito com ASP.NET.

Somente na década de 1990 é que as páginas começaram a se tornar realmente dinâmicas com a introdução de linguagens como CGI (*common gateway interface*), Perl, PHP (*hypertext preprocessor*), ASP (*active server pages*) e Java. Essas linguagens mostraram que é possível implementar grandes sistemas com um alto nível de segurança baseados inteiramente na *web* ao usar requisições de *request* e *response* para basicamente todas as operações. O resto é história: muitas outras tecnologias para desenvolvimento *web* foram criadas e aprimoradas, mas pode-se dizer hoje que a programação para Internet é uma das bases da informática.

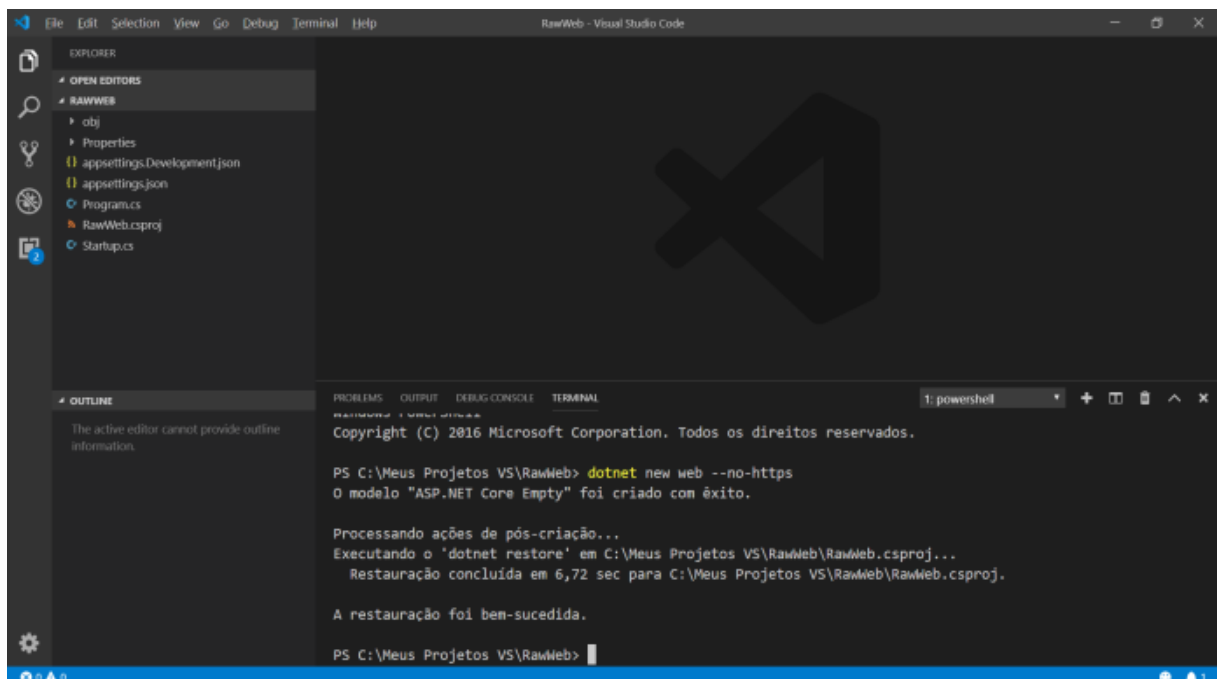
Request e *response* são de fato as duas operações que arantem praticamente qualquer comunicação entre *front-end* e *back-end*.

1. O cliente envia um *request* para um endereço específico (URL ou endereço de IP [*Internet protocol*] etc.) com informações sobre alguma operação necessária.
2. O servidor recebe a informação e processa a operação (esta pode envolver, por exemplo, buscar ou gravar dados em um banco de dados).
3. Com os resultados da operação, o servidor envia uma *response* com as informações desses resultados.

As etapas citadas são a base do protocolo HTTP (*hypertext transfer protocol*) e, conseqüentemente, das aplicações *web* dinâmicas. Sem dúvidas, as respostas de um servidor, geralmente em formato de texto, podem trazer desde pequenas informações (uma palavra ou um número, por exemplo) até complexas páginas HTML (*hypertext markup language*).

Veja a seguir algumas experiências com diferentes tipos de projetos ASP.NET Core para entender o conceito estudado.

1. Com o Visual Studio Code, abriremos uma nova pasta com o nome **RawWeb**. Em seguida, acionaremos o terminal (atalho **Ctrl + `** ou menu “**View > Terminal**”) e, na linha de comando aberta, usaremos o comando `dotnet new web --no-https`, clicando **Enter** no final. O resultado deverá ser semelhante ao da imagem.



The screenshot shows the Visual Studio Code interface with a project named 'RawWeb'. The Explorer sidebar on the left shows the project structure: 'obj', 'Properties', 'appsettings.Development.json', 'appsettings.json', 'Program.cs', 'RawWeb.csproj', and 'Startup.cs'. The Terminal window at the bottom shows the execution of the command `dotnet new web --no-https` in a PowerShell session. The output indicates that the 'ASP.NET Core Empty' model was created successfully, followed by the execution of 'dotnet restore' which completed in 6.72 seconds. The terminal text is as follows:

```
Copyright (C) 2016 Microsoft Corporation. Todos os direitos reservados.

PS C:\Meus Projetos VS\RawWeb> dotnet new web --no-https
O modelo "ASP.NET Core Empty" foi criado com êxito.

Processando ações de pós-criação...
Executando o 'dotnet restore' em C:\Meus Projetos VS\RawWeb\RawWeb.csproj...
  Restauração concluída em 6,72 sec para C:\Meus Projetos VS\RawWeb\RawWeb.csproj.

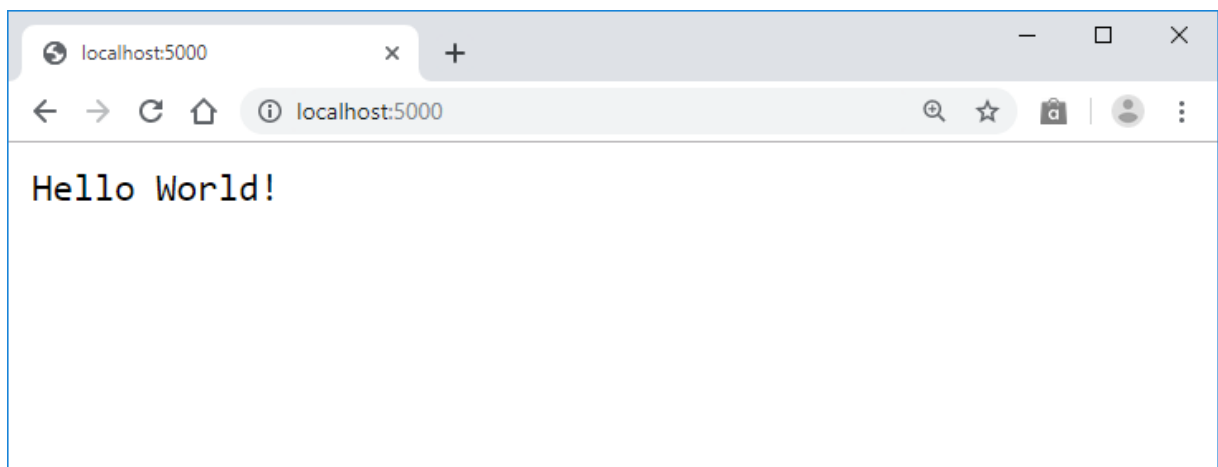
A restauração foi bem-sucedida.

PS C:\Meus Projetos VS\RawWeb>
```

2. Com isso, já teremos uma aplicação *web* simples. Para testá-la, utilizaremos o comando `dotnet run` no terminal.

```
PS C:\Meus Projetos VS\RawWeb> dotnet run
Hosting environment: Development
Content root path: C:\Meus Projetos VS\RawWeb
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

3. Com a tecla **Ctrl** pressionada, clicaremos no endereço fornecido pelo terminal (<http://localhost:5000>). Também é possível digitar esse endereço diretamente no navegador. O resultado não poderia ser mais simples:



4. A resposta é a frase “Hello World!”. Agora, investigaremos o local de onde ela vem. Para isso, voltaremos ao Visual Studio Code e examinaremos o arquivo **Program.cs**. Nesse arquivo, existe a classe responsável por iniciar o servidor portátil Kestrel, usado pelo ASP.NET Core. A aplicação acontece a partir do método **Main()**, no qual encontraremos a seguinte linha:

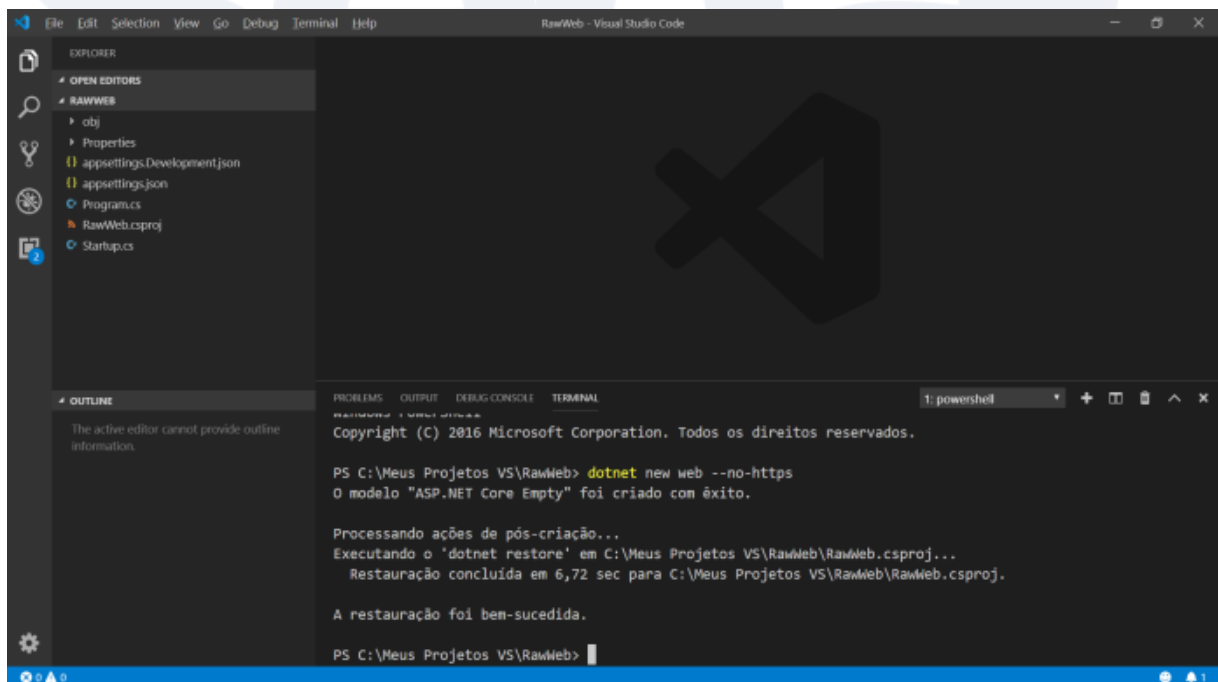
```
CreateWebHostBuilder(args).Build().Run();
```

Após, também encontraremos a configuração para esse CreateWebHostBuilder:

```
Builder(string[] args) =>

    public static IWebHostBuilder CreateWebHost
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
```

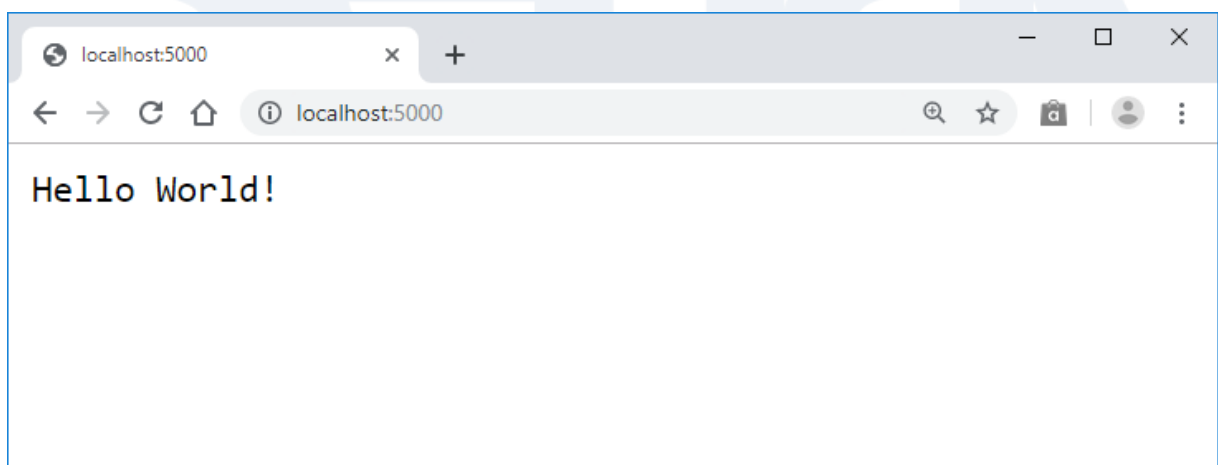
1. Com o Visual Studio Code, abriremos uma nova pasta com o nome **RawWeb**. Em seguida, acionaremos o terminal (atalho **Ctrl + `** ou menu “**View > Terminal**”) e, na linha de comando aberta, usaremos o comando `dotnet new web --no-https`, clicando **Enter** no final. O resultado deverá ser semelhante ao da imagem.



2. Com isso, já teremos uma aplicação *web* simples. Para testá-la, utilizaremos o comando `dotnet run` no terminal.

```
PS C:\Meus Projetos VS\RawWeb> dotnet run
Hosting environment: Development
Content root path: C:\Meus Projetos VS\RawWeb
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

3. Com a tecla **Ctrl** pressionada, clicaremos no endereço fornecido pelo terminal (<http://localhost:5000>). Também é possível digitar esse endereço diretamente no navegador. O resultado não poderia ser mais simples:



4. A resposta é a frase “Hello World!”. Agora, investigaremos o local de onde ela vem. Para isso, voltaremos ao Visual Studio Code e examinaremos o arquivo **Program.cs**. Nesse arquivo, existe a classe responsável por iniciar o servidor portátil Kestrel, usado pelo ASP.NET Core. A aplicação acontece a partir do método **Main()**, no qual encontraremos a seguinte linha:

```
CreateWebHostBuilder(args).Build().Run();
```

Após, também encontraremos a configuração para esse **CreateWebHostBuilder**:

```
public static IWebHostBuilder CreateWebHost  
Builder(string[] args) =>  
    WebHost.CreateDefaultBuilder(args)  
        .UseStartup<Startup>();
```

Por enquanto, não se preocupe com o que cada palavra dessas linhas de código significa. O importante é que você note que essa instrução está injetando algumas configurações ao servidor Kestrel. Na última linha, temos a chamada ao método **UseStartup<Startup>()**. Isso nos leva à classe *startup* do projeto. Abrindo **Startup.cs**, nos depararemos com o seguinte código:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace RawWeb
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add s
        ervices to the container.
        // For more information on how to configure your application, visit
        https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method gets called by the runtime. Use this method to
        configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnviro
        nment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.Run(async (context) =>
            {
                await context.Response.WriteAsync("Hello World!");
            });
        }
    }
}
```

Veremos então a classe *startup*, que estará presente em qualquer aplicação ASP.NET Core com mais ou menos configurações do que nesse exemplo. A linha 31 chama a nossa atenção:

```
context.Response.WriteAsync("Hello World!");
```


Conseguimos rastrear o exato momento em que aquela página vista há pouco no *browser* é formada. Trata-se de um texto simples, que é devolvido para a requisição realizada para o servidor (o Kestrel iniciado em **Program.cs**, no caso) e endereçada pela URL <http://localhost:5000> por meio do navegador, o qual interpreta o resultado simplesmente exibindo na tela a frase.

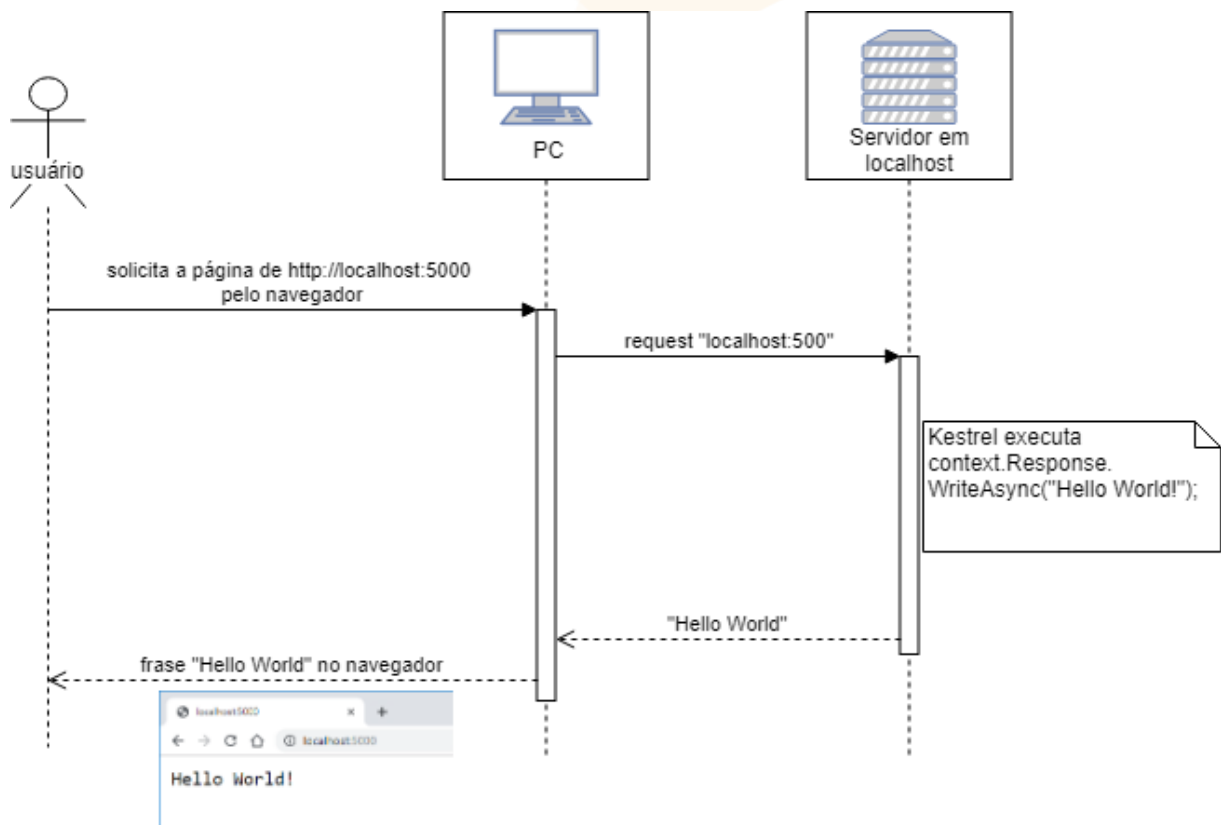


Figura 2 – Processo desde a requisição da página até a resposta do servidor <http://stackoverflow.com>.

O que aconteceu no exemplo foi uma das mais básicas interações entre um *back-end* e um *front-end*. A partir do *front-end*, o usuário requisita uma página, que é formada no *back-end* e devolvida ao *front-end*.

Nesse caso, o servidor é endereçado por *localhost*, o que significa que ele está na mesma máquina em que aconteceu a requisição (a máquina do usuário). Esse servidor poderá estar em um servidor remoto – nesse caso, em vez de *localhost*, será usado um endereço de IP ou um endereço DNS (*domain name system*) (uma URL como <www.senac.br>).

O número à frente de *localhost* é a porta. Cada aplicação de rede que recebe conexões precisa ter uma porta aberta, disponível para ser acessada, seja por aplicações da própria máquina, seja por aplicações externas (nesse caso, são necessárias ainda configurações na rede em roteadores e *firewalls*). Por padrão, o Kestrel usa a porta 5000, mas, se alguma outra aplicação já estiver usando essa porta, pode ser gerado um erro. Assim, é possível configurar uma nova porta no arquivo **Properties/launchSettings.json**, na linha “applicationUrl”.

Para reforçar o conceito, modificaremos o arquivo “**Hello World!**” pelo seguinte texto:

```
$"<html><body><h1>Minha página</h1><p>texto da minha página {DateTime.Now}</p></body></html>"
```

É preciso tomar cuidado para não pular linhas. O código na linha 30 deve ficar como o seguinte (sem considerar quebras de linha):

```
await context.Response.WriteAsync($"<html><body><h1>Minha página</h1><p>texto da minha página {DateTime.Now}</p></body></html>");
```

Agora que já montamos via texto uma página HTML com um título e um parágrafo e incluímos uma informação dinâmica com a hora do servidor (concatenamos `DateTime.Now` na frase “texto da minha página”), salvaremos então a página com **Ctrl + s** ou no menu **File > Save**.

Caso julgue necessário, reveja o tópico **Introdução à programação em C#**, do material **Arquitetura web em camadas: conceitos de camadas *front-end* e *back-end***, da unidade curricular **Monitorar projetos de aplicações web**, para relembrar as diferentes maneiras de incluir uma informação dinâmica (uma variável, por exemplo) dentro de um texto.

DateTime é uma classe do C# para manipular tempo. **Now** é uma propriedade de **DateTime** que traz data e hora exatas do momento em que essa instrução é chamada.

Antes de executar, precisamos parar o servidor que está em execução. Para tanto, clicaremos no terminal do Visual Studio Code para ativá-lo. Em seguida, pressionaremos as teclas **Ctrl + C** para encerrar o servidor – a mensagem “application is shutting down” (que, em tradução livre, significa “a aplicação está encerrando”) deve aparecer.

Agora, usaremos o comando `dotnet build` para compilar a modificação realizada e depois o comando `dotnet run` para rodar novamente o servidor.

O resultado, ao acessar o endereço `<http://localhost:5000>` no *browser*, deverá ser este:

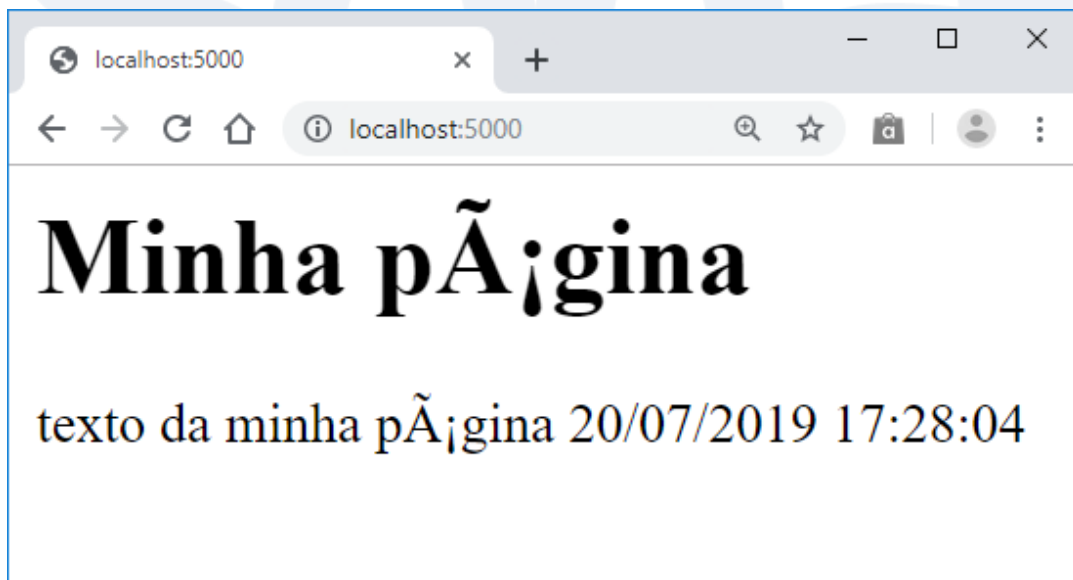


Figura 3 – Página inicial do projeto no navegador

Por enquanto, não nos preocuparemos com acentuações. O importante é notar que existe uma página HTML montada dinamicamente pelo servidor. A prova disso é que, se recarregarmos a página no navegador (geralmente por meio da tecla **F5**), veremos a data e a hora atualizadas.

Outra característica para podermos visualizar e entender como acontecem requisições e respostas é o próprio terminal do Visual Studio Code. Experimente atualizar a página no navegador e verificar as últimas frases que o terminal emite. Provavelmente, aparecerá algo como este exemplo:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 0.1508ms 200
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/favicon.ico
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 0.1665ms 200
```

Tais *logs* (informações de execução) mostram que é iniciado um *request* (o principal), respondido em questão de milissegundos. Em seguida, um novo *request* é feito solicitando a imagem “favicon.ico”, também respondido rapidamente. Favicon é o ícone que aparece na aba da página no navegador. No projeto que estamos desenvolvendo, não configuramos nenhuma imagem que substitui o padrão usado pelo *browser*.

A manipulação de código HTML de acordo com processamentos do servidor é a base das aplicações *web* dinâmicas. Em vez de uma data e uma hora, como fizemos anteriormente, poderíamos consultar um banco de dados e incluir uma lista com informações obtidas dessa consulta. Essa lista seria embutida no HTML como texto e entregue ao usuário, o qual visualizaria tudo em seu *browser*. É claro que nessa programação *server-side* podem aparecer ainda condicionais, laços e outros tipos de decisão que tornam essas páginas dinâmicas.

De fato, nas linguagens mais antigas, programava-se dessa forma, construindo todo o HTML dinamicamente, com texto. A partir de tecnologias como ASP e PHP, introduziu-se a facilidade de programar o HTML em separado, apenas incluindo trechos de programação no meio do código de marcação. Isso facilitava a programação, pois era possível distinguir o que era código estático e o que era código dinâmico.

Observe o exemplo de código ASP a seguir. O PHP segue uma estrutura semelhante.

```
<!DOCTYPE html>
<html>
<body>
<p>Parágrafo fixo na página HTML</p>
<%
response.write("<p style='color:#0000ff'>Parágrafo dinâmico com A
SP</p>")
%>
</body>
</html>
```

Nesse tipo de aplicação, as páginas são interpretadas no servidor, e todas as marcações dinâmicas são preenchidas por meio de processamento em *back-end*. O resultado é uma página que mescla partes estáticas e partes dinâmicas.

Tal abordagem, apesar de muito bem-vinda, acabou trazendo um efeito colateral. Se antes as páginas eram todas feitas pela camada *back-end*, agora a tendência é concentrar as regras de negócio da aplicação na camada de *front-end*. Por vezes, o código acabava ficando confuso, por misturar excessivamente as responsabilidades dos diversos componentes do sistema. A manutenção de sistemas, assim, ficava mais difícil de ser realizada.

Essa desorganização levou à criação de um padrão de projeto em especial: o MVC.

Clique ou toque para visualizar o conteúdo.



O Padrão MVC

Padrões de projeto são modelos de soluções prontas que podem ser usados em problemas comuns de projetos de *software*.

Um padrão constitui não uma implementação pronta, mas, sim, um modelo de resolução documentado que se encaixa em situações específicas resolvendo problemas.

Um dos registros mais importantes de padrões de projeto é a obra *Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos*, de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, autores também conhecidos como *gang of four* (em tradução livre, “ganguê dos quatro”). Entre os vários padrões clássicos descritos no livro, estão os padrões *singleton*, *factory*, *façade*, *command* e *observer*.

Derivados de padrões de projeto, existem ainda alguns padrões de arquitetura de aplicação. A mesma intenção de reutilizar código e separar responsabilidades em um sistema – o que cada código faz – está presente nos padrões de arquitetura, que podem embutir alguns conceitos de padrões de projeto ou mesmo de padrões inteiros.

Um dos padrões de arquitetura mais utilizados é o MVC. Também classificado como padrão composto, o MVC é uma arquitetura de aplicação que define uma separação explícita entre as funcionalidades de dados, de *front-end* e de controle entre essas duas camadas.

O nome *model view controller* deriva exatamente dessa divisão. Veja agora o que cada palavra representa:

***Model* (modelo)**

É a camada responsável pelo acesso aos dados (um banco de dados, por exemplo).



exemplo) e pelas regras de negócio.

View (visão)

É a camada usada para renderizar (mostrar ao usuário) o resultado do processamento de *model* na interface do usuário.

Controller (controle)

É a orquestração entre as duas camadas, recebendo requisições, demandando operações ao *model* e selecionando *views* que devem ser preparadas e enviadas ao usuário.

Freeman & Freeman (2007) comparam o MVC com uma bolacha recheada. É uma analogia divertida e muito válida: um dos lados é o *view*, o outro é o *model* e o recheio é o *controller*, ligando os dois primeiros.

Outro exemplo citado pelos autores é escutar músicas em um aparelho portátil (um *smartphone*, por exemplo). É possível dizer que o aplicativo de música pode funcionar em um padrão MVC com as seguintes características:

- ◆ O *view* é responsável por capturar ações, mostrar a música corrente, listar biblioteca, permitir escolha de faixa etc.
- ◆ O *controller* é responsável por receber as ações vindas da interface visual e repassá-las ao *model*, chamando as operações necessárias dessa camada, como buscar uma música, tocar uma faixa etc.
- ◆ O *model* é responsável por buscar a música na biblioteca, localizar a música na memória do aparelho e tocá-la, bem como processar buscas etc.

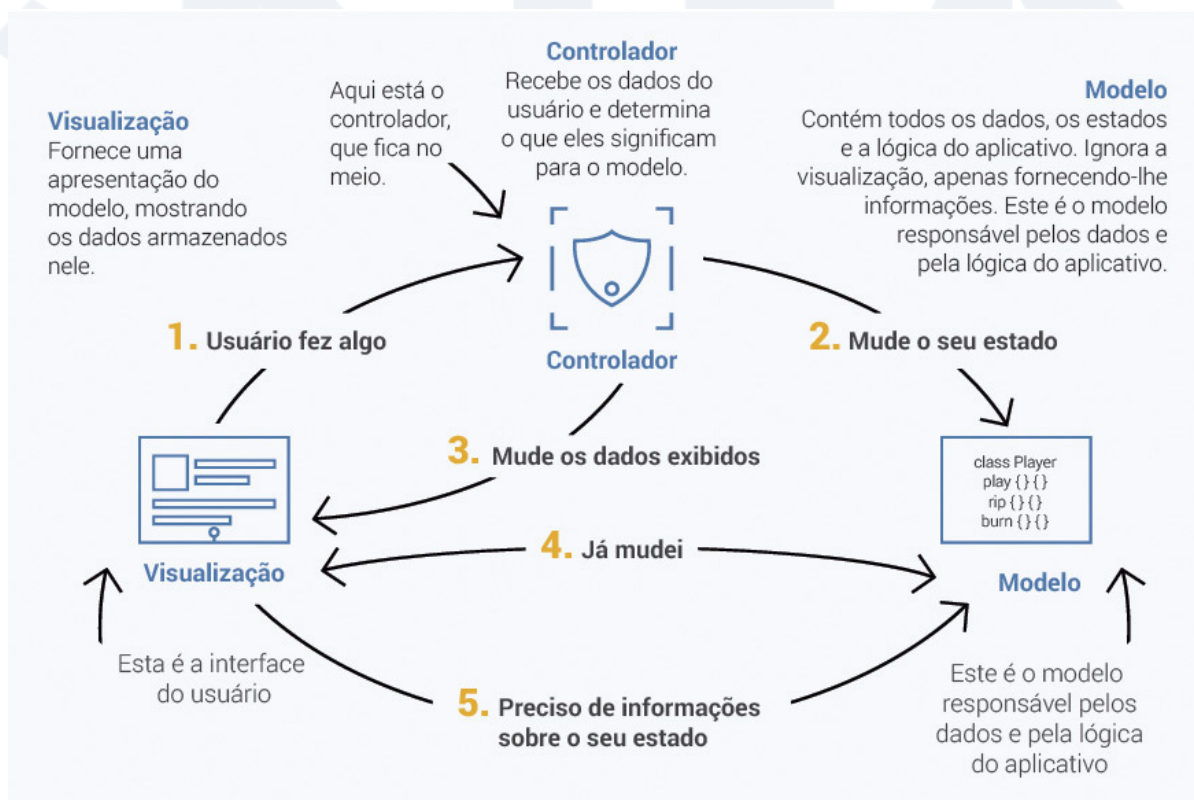


Figura 4 – Exemplo de como o MVC poderia ser aplicado a um tocador de MP3 Fonte: adaptado de Freeman & Freeman (2007).

Clique ou toque para visualizar o conteúdo.

Models

Contêm os dados com os quais o usuário trabalha e constituem a definição do universo em que a aplicação atuará. Como exemplo, imagine uma aplicação de banco: a camada *model* dessa aplicação seria responsável por todo tipo de informação (por exemplo, contas, clientes, limites de crédito, saldos) e também pelas operações de manipulação dessas informações (por exemplo, operações de depósito, saque, extrato, transferência). Cabe à camada *model* preservar a consistência de tais informações, validando as operações.

Um modelo não deve expor os seus detalhes de operação de dados (a conexão com o banco de dados interessa apenas ao *model*, e não às outras camadas), não deve conter lógica que modifique os dados diretamente pela interação do usuário (o *controller* deve fazer essa intermediação) e não deve implementar operações que mostrem diretamente informações ao usuário (o *view* deve fazer esse papel).

Controllers

Fazem a conexão entre o modelo e o *view*, definindo as ações de lógica de negócio (*model*) que respondam adequadamente a uma requisição do usuário (*view*), bem como são responsáveis por repassar a informação recuperada ou resultante de uma operação do *model* para o *view*. Um *controller* não deve conter nem a lógica de persistência de dados (responsabilidade do *model*), nem operações de interface do usuário (responsabilidade do *view*).

Views

Contêm as operações necessárias para mostrar a informação ao usuário e também para capturar informações do usuário de maneira que estas possam ser processadas no sistema. Um *view* não deve conter lógicas complexas (que ficam bem-postas em um *controller*) nem lógicas que manipulem ou armazenem informações (papel do *model*). *Views* podem conter alguma lógica de negócio (especialmente quanto a validações de dados informados pelo usuário), mas devem ser mais simples e menos frequentes que *models*.

Sendo assim, fica **evidente que a separação das camadas tem o propósito de deixar cada uma independente da outra**. Obviamente, existirão conexões, mas, em tese, em um projeto MVC, uma camada poderia ser substituída por outra implementação equivalente sem que as demais camadas precisassem ser refeitas.

Por exemplo, imagine se houvesse todo um sistema desenvolvido para *web* que agora deve ter uma interface *mobile*. As camadas *controller* e *model* poderiam ser mantidas, e a camada *view* poderia ser trocada de uma *view* interface *web* para uma *view* interface *mobile*. A camada nova deveria ser trabalhada para, apesar de totalmente diferente em interface, se comunicar da mesma maneira com a camada *controller*, garantindo consistência do sistema.

Em um código, tais camadas seriam apenas agrupamentos de classes. Algumas classes serão criadas como *models* e terão a responsabilidade de manipular dados e regras. Outras serão criadas para servir como *controllers*. Outras, ainda, serão elaboradas para cuidar de toda a interface visual com o usuário, formando a camada *view*.

No ASP.NET MVC, as camadas ficam claramente separadas, de uma maneira muito característica.



Estrutura de um projeto ASP.NET Core MVC

Lançado em 2009, o ASP.NET MVC tem sido o formato preferido da Microsoft para orientar as suas soluções *web* desde então. Com a chegada do ASP.NET Core, uma nova versão dos projetos MVC também surge, mais leve e portátil.

Observe a seguir as estruturas que aparecem em um projeto ASP.NET Core MVC.

Para tanto, abriremos o Visual Studio Code, criaremos uma nova pasta nele e acionaremos o terminal.

Após, digitaremos o comando `dotnet new mvc --no-https` e teclaremos **Enter**. Essas ações iniciarão a criação da estrutura do projeto e depois a operação de restauração, preparando o ambiente. O resultado deverá ser semelhante ao exibido na imagem a seguir:

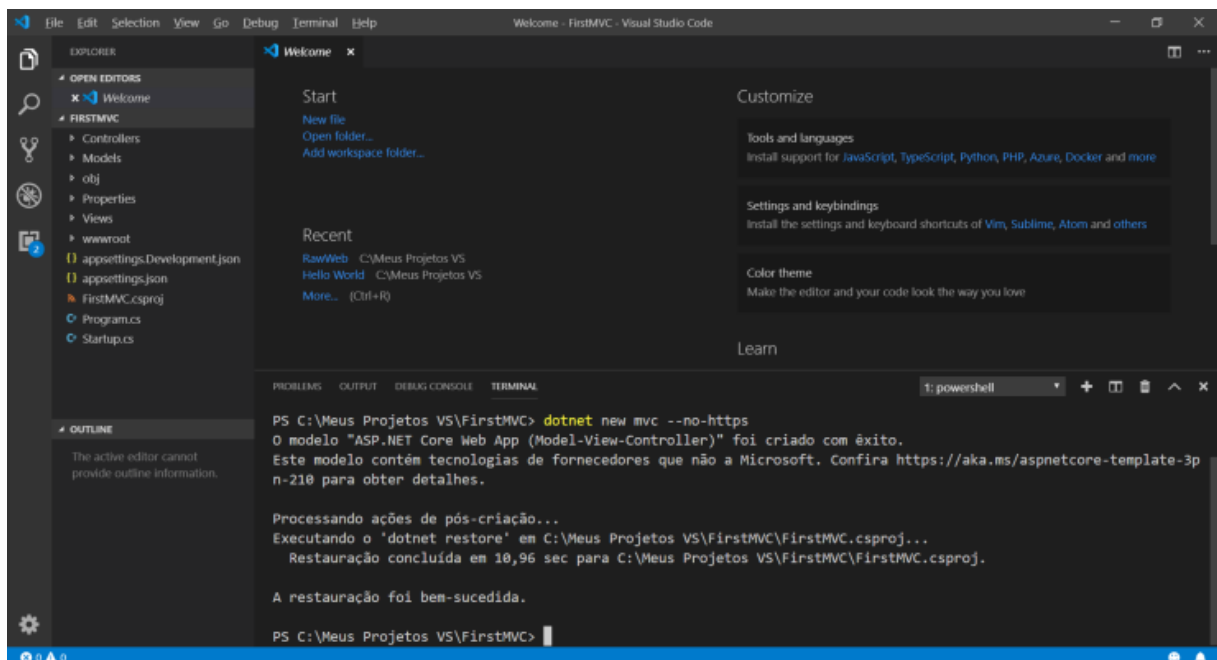


Figura 5 – Resultado da criação de uma aplicação ASP.NET Core MVC

Caso você já tenha criado o projeto FirstMVC a partir do material **Ambiente de desenvolvimento: instalação e configuração**, basta abri-lo novamente. Todavia, caso queira treinar um pouco mais, sinta-se à vontade para criar um novo projeto.

Observe com atenção a aba **Explorer** à esquerda. Com base nos conceitos do padrão MVC de projetos, é possível decifrar três dos diretórios que foram criados no projeto: o primeiro (*controllers*), o segundo (*models*) e o quinto (*views*). Os outros diretórios são *obj* e *properties*, responsáveis por configurações e resultados (compilações) do projeto, e *wwwroot*, usado para armazenar recursos (imagens ou arquivos CSS).

A pasta **Controller** contém um arquivo **HomeController.cs** com a classe **HomeController**. Nessa classe, podem-se verificar três métodos: **Index()**, **Privacy()** e **Error()**. No contexto de *controller*, eles são chamados de ações (o tipo de retorno é **ActionResult**, necessário para ações). As ações serão responsáveis por receber uma requisição e fornecer um resultado ao usuário.

```
namespace FirstMVC.Controllers //todos os controladores estarão
sob esse namespace <NomeDoProjeto>.Controllers
{
    public class HomeController : Controller //classe HomeController é derivada de Controller. Ela será responsável por receber algumas requisições do usuário e lhe entregar respostas
    {
        public IActionResult Index() //ação Index
        {
            return View();
        }

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

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

Essa é a camada de controlador do nosso MVC. Outros controladores poderão estar presentes, mas devem ficar sempre na pasta “**Controllers**”.L.

Expandindo a pasta **Models**, encontraremos apenas o arquivo **ErrorViewModel.cs**, que contém uma classe **ErrorViewModel**. Há, nesse caso, uma classe **ViewModel**, ou seja, uma classe que fornecerá informações ou validações que interessem ao *view*, e não necessariamente ao negócio (ou ao assunto que o sistema trata).

Nesse diretório, ficarão as classes relativas ao modelo de nossas aplicações. **ErrorViewModel** é usado por **HomeController** no método **Error()**.

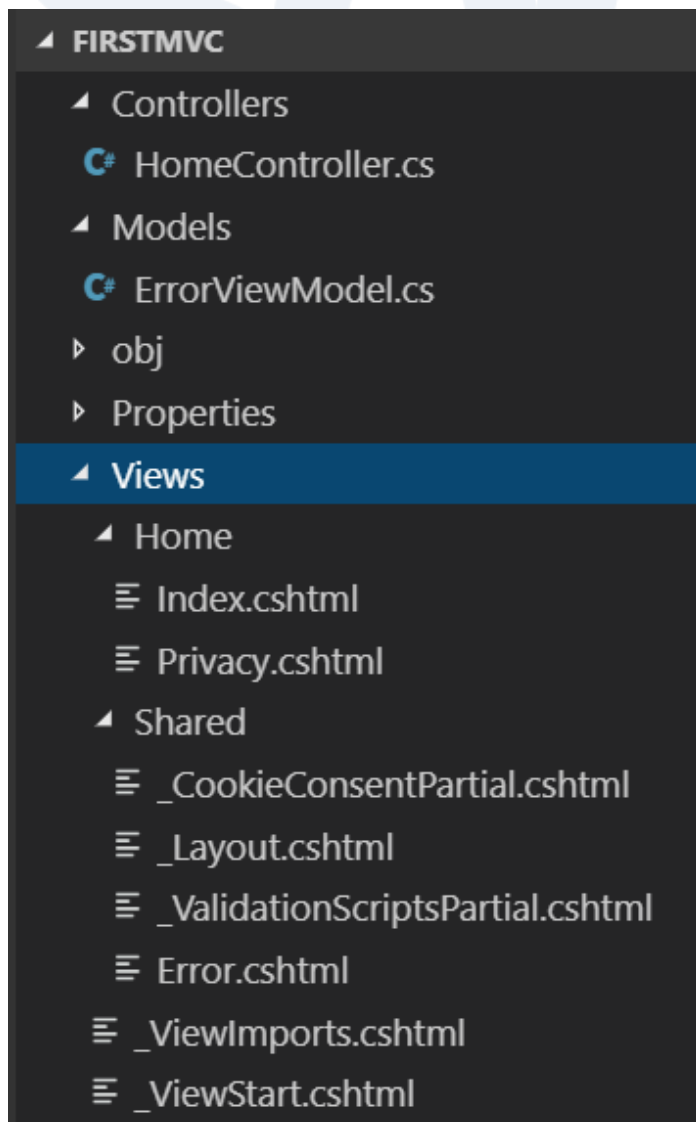


Figura 6 – Diretório views e arquivos .cshtml

Por fim, examinaremos a pasta **Views**. O conteúdo desse diretório é diferente: há arquivos com extensão `.cshtml` e dois subdiretórios *home* e *shared* também contando com arquivos `cshtml`. Essa extensão é o formato que o ASP.NET Core MVC utiliza para disponibilizar a interface HTML para a aplicação. Trata-se de um HTML com modificações para incluir marcações que ajudem a integrar as informações que vêm do *controller*.

Examinaremos agora o código de uma dessas páginas. Primeiramente, abriremos o arquivo **Views/Home/Index.cshtml**

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

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core"
>building Web apps with ASP.NET Core</a>.</p>
</div>
```

Trata-se de um arquivo padrão em HTML, mas com inclusão de marcações iniciadas com o caractere **@**. **ViewData**, que é um meio de transmitir ou definir informações entre os *views*.

Um arquivo mais complexo pode dar uma ideia melhor sobre o que esperar dessas páginas CSHTML. Para isso, teremos que abrir **Views/Shared/Error.cshtml**:

```

@model ErrorViewModel
@{
    ViewData["Title"] = "Error";
}

<h1 class="text-danger">Error.</h1>
<h2 class="text-danger">An error occurred while processing your r
equest.</h2>

@if (Model.ShowRequestId)
{
    <p>
        <strong>Request ID:</strong> <code>@Model.RequestId</code>
    </p>
}

<h3>Development Mode</h3>
<p>
    Swapping to <strong>Development</strong> environment will disp
lay more detailed information about the error that occurred.
</p>
<p>
        <strong>The Development environment shouldn't be enabled for d
eployed applications.</strong>
        It can result in displaying sensitive information from excepti
ons to end users.

        For local debugging, enable the <strong>Development</strong> e
nvironment by setting the <strong>ASPNETCORE_ENVIRONMENT</strong> environm
ent varia
ble to <strong>Development</strong>
        and restarting the app.
    </p>

```

Como o próprio nome diz, essa página deverá ser mostrada quando ocorrer um erro na aplicação. Nela, há uma instrução **@if** no código. Isso significa que, nos *views* de ASP.NET Core MVC, é possível fazer não só condições, mas também referências ao modelo (note onde há **@model** no código), bem como realizar *loops*.

Esse tipo de *view* é um subsistema chamado **Razor**, o qual será abordado adiante neste material.

Agora que já identificamos as partes M, V e C de nosso projeto, compreenderemos melhor como ele funciona e como elas entram em cena. Primeiramente, rodaremos a aplicação acessando o terminal do Visual Studio Code

usando o comando `dotnet run`. Como nos exemplos anteriores, nos resultados desse comando veremos o endereço `<http://localhost:5000<`, que será o endereço utilizado para acessar a aplicação.

No *browser*, teremos a seguinte página:

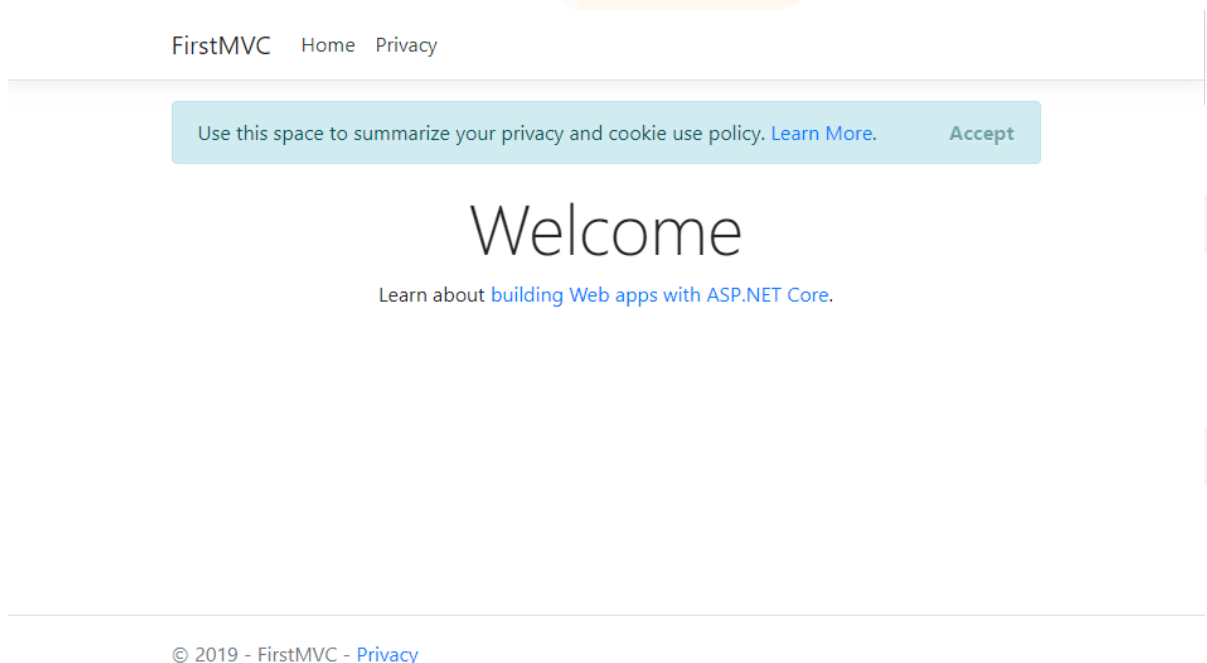


Figura 7 – Página do projeto FirstMVC

É importante saber de onde vem o “*welcome*”. Caso ache necessário, retorne a **Views/Home/Index.cshtml**. Clique no *link Privacy* e confira o resultado. Compare-o com o que há no *view (Views/Home/Privacy.cshtml)*.

Agora, antes de prosseguir, relembremos como chegamos até aqui:

1. O usuário, ao clicar no *link*, faz uma solicitação pela URL `<http://localhost:5000/Home/Privacy<`.
2. Essa solicitação (*request*) é enviada para o servidor Kestrel (o qual foi iniciado no momento que fizemos **dotnet run**).
3. O servidor localiza o *controller*, que deve se encarregar dessa requisição. A URL traz *home*. Por isso, o controle que cuidará dessa requisição é o **HomeController**.

4. Em **HomeController**, há um método chamado **Privacy**. A URL traz **Privacy** no final, e por isso esse método é chamado.
5. Na linha “return View();” desse método, internamente resgata-se um arquivo CSHTML que tenha nome **Privacy** (a ação) e que esteja em uma pasta **Home** (o *controller*) dentro da pasta **Views**. Encontra-se, assim, **Views/Home/Privacy.cshtml**.
6. O servidor então processa o **Privacy.cshtml**, preenchendo as partes dinâmicas e montando a página.
7. O servidor, por fim, retorna (uma *response*) para o cliente com o HTML pronto, que é exibido no *browser*.

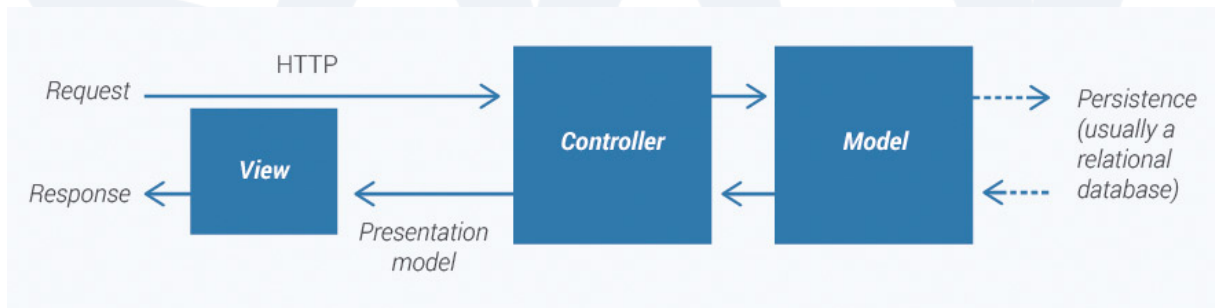


Figura 8 – Funcionamento de requisições em uma aplicação ASP.NET Core MVC Fonte: adaptado de Freeman (2017).

Privacy.cshtml não contém *links* nem o texto “use this space to summarize your privacy and cookie use policy”, entre várias outras coisas que estão na página. Não contém nem mesmo uma marcação `<html>`. Por que isso ocorre?

Essas marcações partem de um sistema de layouts próprio do ASP.NET MVC. Em resumo, o que acontece é que **Privacy.cshtml** constitui apenas o miolo de uma página, sendo o entorno dela definido em **Views/Shared/_Layout.cshtml**.

Por que a URL `<http://localhost:5000>` mandou para o *view* **Index**, se nem **Home** nem **Index** estão no endereço?

Isso acontece porque há a definição de endereços padrão. No caso, a URL ficou definida como a página inicial.

Clique ou toque para visualizar o conteúdo.

Você está pronto para o primeiro desafio?



Para começar, modifique o conteúdo de **Privacy.cshtml**, incluindo um texto utilizando as tags `<p>` e `<h1>`, e teste no seu navegador. Talvez você tenha que realizar um novo `dotnet build`.

Em suma, nos projetos ASP.NET Core MVC, os *controllers* são classes C# derivadas de `AspNetCore.Mvc.Controller`. Cada método público dessas classes é uma ação associada a uma URL. Quando uma requisição é enviada à URL associada ao método de ação, o código desse método é executado, realizando algumas operações no modelo do domínio e, no fim, selecionando a visão que deve ser mostrada ao cliente.

Ao implementar uma aplicação ASP.NET MVC, algumas regras precisam ser seguidas. São elas:

- ◆ As classes de *controller* precisam ser nomeadas com a terminação **Controller** (**HomeController**, no nosso exemplo).
- ◆ Os *views* devem necessariamente estar na pasta **/Views/NomeDoController** ou em **/Views/Shared**. O MVC busca pelo *view* primeiramente na pasta correspondente ao *controller* e, caso não encontre, em *shared*. Essa pasta é adequada para recursos que serão reutilizados por outros processos, como *layouts* e *partial views*.
- ◆ Recursos de *view* reutilizáveis iniciam com “_”. Veja no projeto os exemplos `_CookieConsentPartial.cshtml`, `_Layout.cshtml` e outros.

Clique ou toque para visualizar o conteúdo.

Você está pronto para criar a sua primeira página em MVC?



Crie uma nova página no projeto FirstMVC, bem como uma ação chamada **About**. A página deve mostrar uma página com as suas informações pessoais (nome completo, idade, sexo e uma frase que o defina). Para tanto, inclua um método de ação e **HomeController** com o nome **About**. Em **/Views/Home**, crie um novo arquivo CSHTML com o nome **About**. Caso necessário, utilize como modelo a página index já implementada no projeto. Use no HTML elementos como `<h1>`, `<p>`, ``. Por fim, acesse a URL `http://localhost:5000/Home/About` para testar. Caso o navegador não encontre nada, verifique os nomes do método e do CSHTML criados.

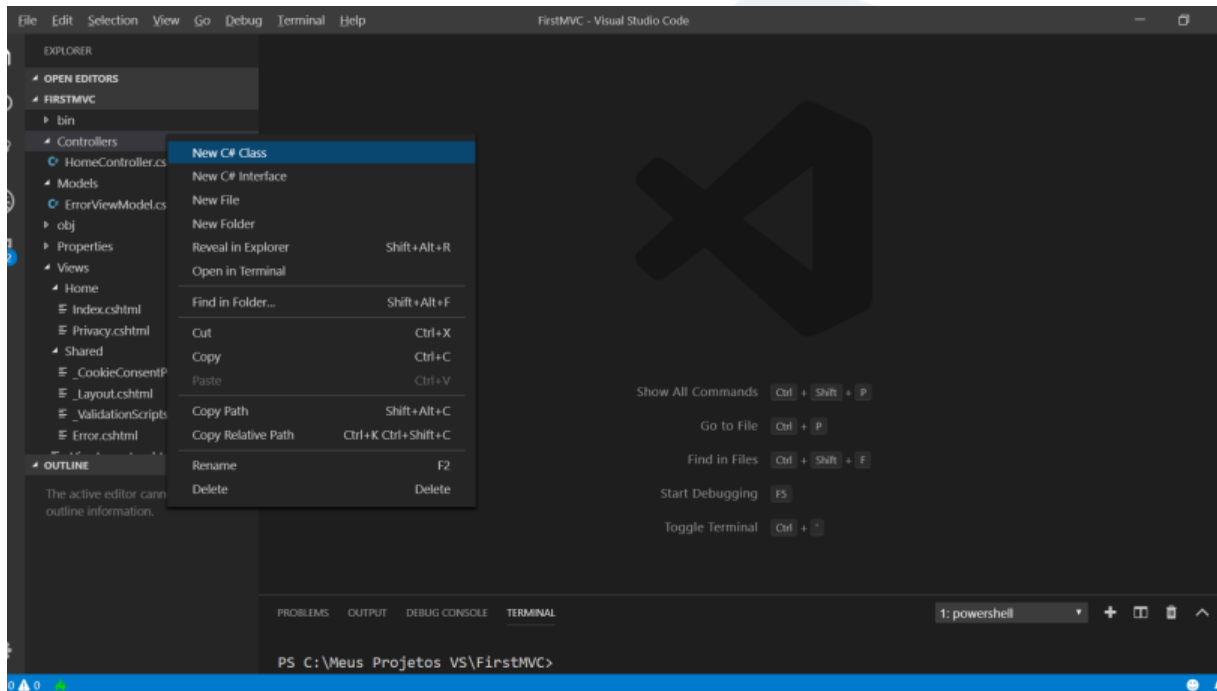
Explorando *views*

Ainda em nosso projeto FirstMVC, faremos algumas experiências, separando-as convenientemente em um *controller* próprio.

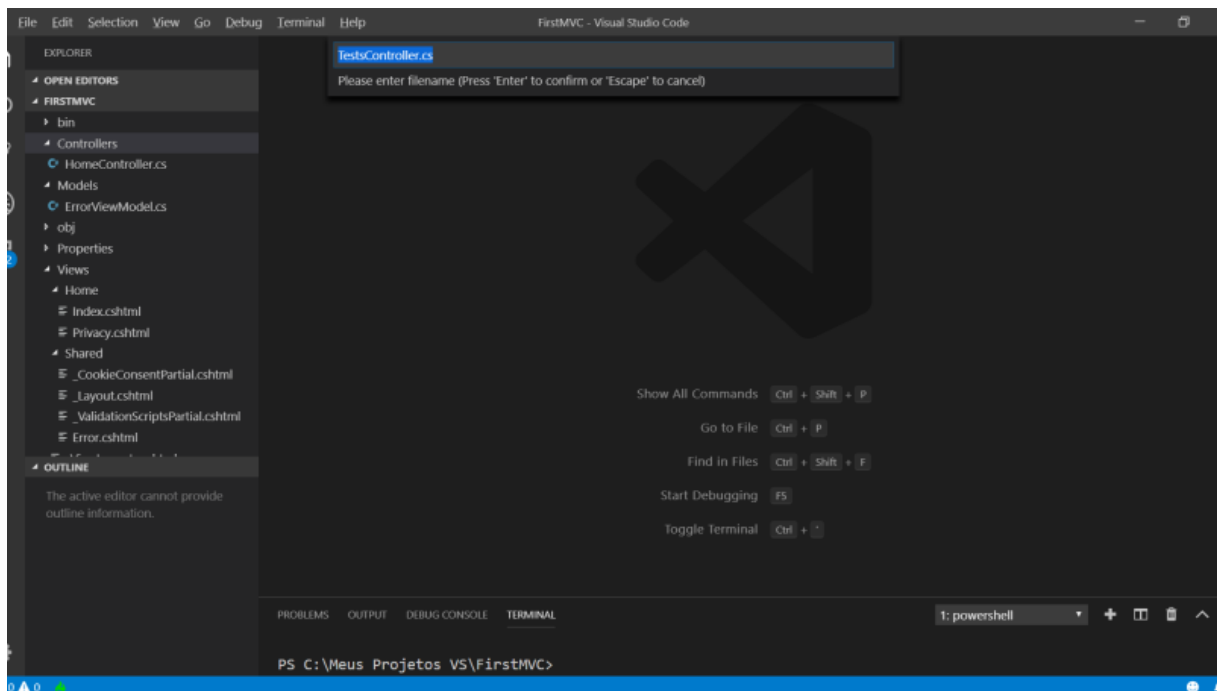
Se as mudanças do desafio anterior prejudicaram o seu projeto, remova a alteração, volte a funcionar o projeto e, depois dos novos exemplos citados, tente fazer o desafio novamente.

Caso você não tenha instalado a extensão C# Extensions, instale-a antes de prosseguir. Veja o conteúdo **Ambiente de desenvolvimento: instalação e configuração** para mais informações.

Clique na pasta **Controllers** e escolha **New C# Class**.



A barra superior se expandirá para você digitar o nome do arquivo. Então, digite **TestsController.cs** e depois clique em **Enter**.



O arquivo **TestsController.cs** será criado com o seguinte conteúdo:

```
namespace FirstMVC.Controllers
{
    public class TestsController
    {

    }
}
```

Apenas o **namespace** e a definição da classe estão presentes. Ainda precisamos das cláusulas *using*. Inicialmente, utilizaremos as mesmas cláusulas que estavam presentes em **HomeController**. Copie-as daquele arquivo e cole-as neste.

Também incluiremos a derivação na classe *controller*, acrescentando “: Controller” à frente do nome da classe. Depois, salvaremos com **Ctrl + S**.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using FirstMVC.Models;

namespace FirstMVC.Controllers
{
    public class TestsController
    {

    }
}
```

Após, verificaremos se está tudo certo realizando um “dotnet build”. Se a mensagem for “Compilação com êxito”, está tudo certo. Caso haja mensagens em vermelho, verifique a linha em que aconteceu o erro e cheque novamente se o seu código está igual ao exemplo anterior.

Veja a seguir um exemplo de erro emitido pelo *build*. No caso, TestsController.cs(14,5) indica o arquivo em que a falha aconteceu (a linha 14, no exemplo, e a coluna 5). Em seguida, a mensagem de erro indica a falha.

```
Controllers\TestsController.cs(14,5): error CS1519: Token inválido  
"}" na classe, estrutura ou declaração de membro de interface [C:\Meus Projetos VS  
\FirstMVC\FirstMVC.csproj]  
0 Aviso(s)  
1 Erro(s)
```

Com o projeto sem erros, criaremos então a ação para o nosso primeiro teste.

Clique ou toque para visualizar o conteúdo.



Hora atual

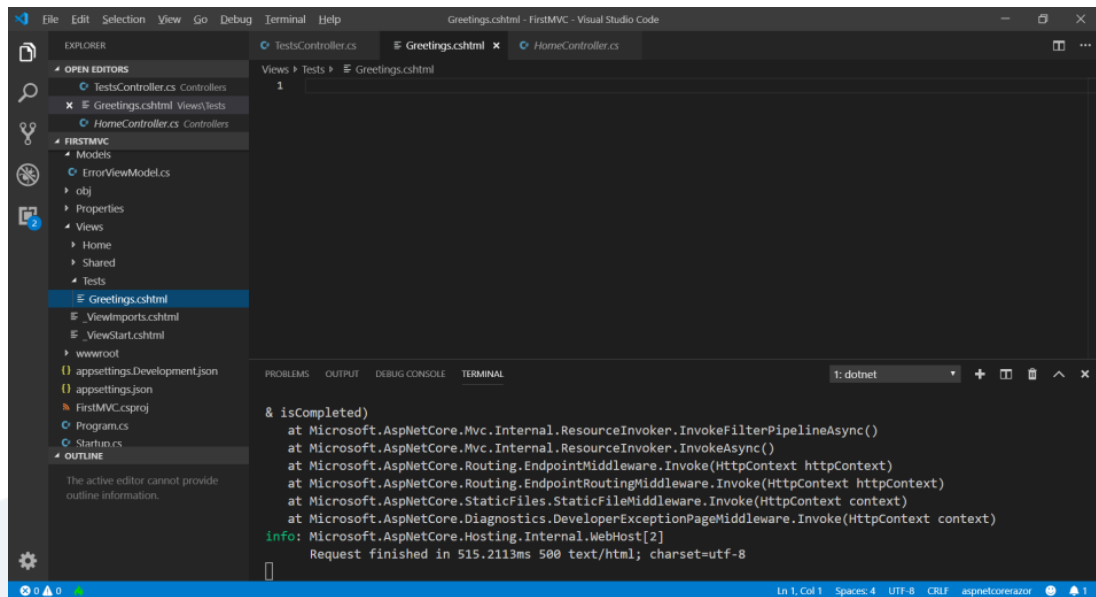
Agora, faremos uma página que mostre a hora atual e, com base nesta, escreva “Bom dia!”, “Boa tarde!” ou “Boa noite!”.

Para criar a ação, declararemos um método, o qual será chamado de **Greetings**:

```
public class TestsController : Controller
{
    public IActionResult Greetings()
    {
        return View();
    }
}
```

Experimente digitar algo (por exemplo, `IAct...`) e pressionar **Ctrl + Espaço**. O Visual Studio Code deve oferecer algumas sugestões do que se trata, entre elas `IActionResult`. Esse nada mais é do que o IntelliSense, também chamado de autocompletamento.

Assim, estamos definindo uma ação simples que apenas chamará uma visão. Criaremos então um *view* CSHTML. Para tanto, criaremos antes uma pasta filha de *views* clicando sobre ela com o botão direito do *mouse* e selecionando **New folder**. Então, digitaremos o nome “Tests” (que é como o *controller* se chama). Em seguida, clicando com o botão direito do *mouse* sobre **Tests**, selecionaremos **New file** e o nomearemos como “Greetings.cshtml”. Confira na imagem a seguir, na aba **Explorer**, à esquerda, a estrutura correta:

Figura 9 – Criando **Greetings.cshtml**

Em **Greetings.cshtml**, incluiremos o seguinte código:

```
@{
    DateTime date = DateTime.Now;
    string hourMinutes = date.ToString("HH:mm:ss");
}

<p>A hora certa é: @hourMinutes </p>
```

O que fizemos foi incluir um bloco de código C# usando **@{ }**. Todo o código dentro das chaves dessa estrutura é considerado *back-end*. Nesse exemplo, estamos declarando uma variável *date*, passando a ela o valor atual da data e da hora, e, na linha de baixo, estamos declarando uma nova variável, a qual trará apenas a hora certa, no formato **horas:minutos:segundos**.

Dentro da *tag* **<p>**, é possível usar qualquer uma das variáveis definidas no bloco **@{ }** anterior. No caso, usaremos a variável que mostra a hora exata utilizando **@hoursMinutes**.

Em seguida, salvaremos o arquivo e, no terminal, usaremos os comandos **dotnet build** e, depois, caso não tenha havido nenhum erro, **dotnet run**. No navegador, acessaremos **<http://localhost:5000/Tests/Greetings>**.

O resultado será o seguinte:

FirstMVC Home Privacy

Use this space to summarize your privacy and cookie use policy. [Learn More.](#)

Accept

A hora certa é: 23:34:25 !

Figura 10 – Página de **Tests/Greetings** mostrando a hora exata do servidor

No texto, surge a hora exata de acordo com o servidor (que, no caso, é a máquina que você utiliza).

Agora, voltando ao **Greetings.cshtml**, identificaremos o período do dia (manhã, tarde ou noite) e incluiremos uma mensagem de saudação conforme o resultado. Para tanto, acrescentaremos um código de acordo com o trecho a seguir:

```
@{
    DateTime date = DateTime.Now;
    string hourMinutes = date.ToString("HH:mm:ss");
}

<p>A hora certa é: @hourMinutes</p>
<p>
@if(date.Hour >= 5 && date.Hour <= 12)
{
    <strong>Bom dia!</strong>
}
else
{
    if(date.Hour >= 13 && date.Hour <= 18)
    {
        <strong>Boa tarde!</strong>
    }
    else
    {
        <strong>Boa noite!</strong>
    }
}
</p>
```

Estamos usando uma condicional **if**. Novamente, precisamos da diretiva **@** antes do primeiro **if** para definir que estamos em um trecho de código que será executado no servidor. O **@if** está dentro de uma *tag* **<p>**, ou seja, o resultado dessa condição preencherá essa *tag*.

Primeiramente, verificaremos a variável *date*, inicializada no topo do arquivo, percebendo se a hora da data está entre 5 e 12. Em caso afirmativo, escreveremos em negrito “Bom dia!” na tela.

Em *else*, há um *if* aninhado que verificará se a hora está entre 13 e 18. Em caso positivo, escreveremos “Boa tarde!”. Em caso negativo, escreveremos “Boa noite!”. Para testar, podemos alterar o horário atual do sistema operacional e fazer *refresh* (**F5**) na página.

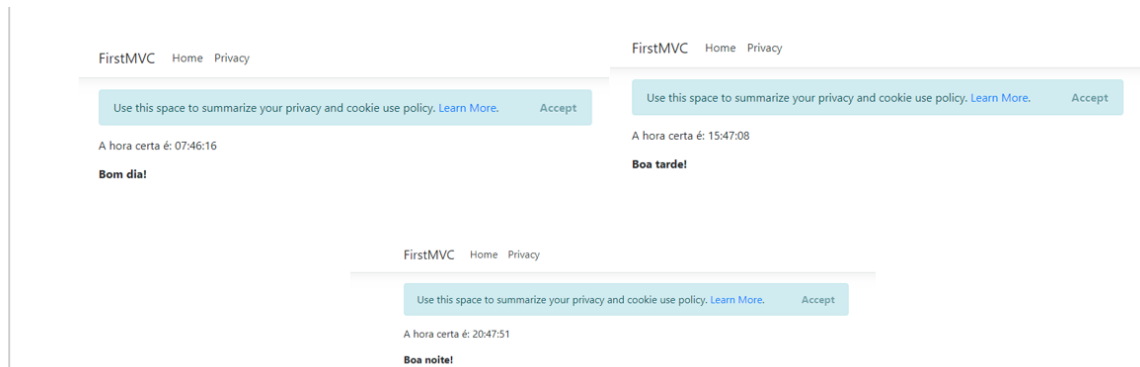


Figura 11 – Testes com a página **Greetings**

A definição de regras em *views* deve ser usada apenas em situações simples e que reflitam diretamente em elementos de *view*. Não se devem carregar os arquivos CSHTML de verificações e de regras, pois estas devem estar principalmente em *controllers* e em *models*.

Boas-vindas

Agora, criaremos uma página com o nome do usuário e com a mensagem de saudação “boas-vindas”. Para tanto, utilizaremos parâmetros informados na própria URL. A primeira alteração que faremos é, em **TestsController**, criar a ação **Welcome**, abaixo de **Greetings**.

```
public class TestsController : Controller
{
    public IActionResult Greetings()
    {
        return View();
    }

    public IActionResult Welcome(string name)
    {
        return Content(name);
    }
}
```

Para conseguir obter uma informação passada pela URL, precisamos incluir um parâmetro no método. No caso, incluímos um parâmetro do tipo *string* chamado *name* no método **Welcome**. Depois, retornamos, em vez de uma chamada ao método **View()**, uma chamada ao método **Content()**, passando *name* como argumento.

O método **Content()** retorna diretamente um valor bruto em vez de pesquisar um CSHTML em *views*. Existem vários outros retornos que podem ser usados em ações do MVC. São alguns deles:

- ◆ **Content()** – Retorna um conteúdo genérico.
- ◆ **Json()** – Retorna um texto em formato JSON (JavaScript *object notation*) de representação de objeto.
- ◆ **Redirect()** – Redireciona uma ação.
- ◆ **File()** – Permite disponibilizar um arquivo para *download*.
- ◆ **PartialView()** – Retorna um *view* parcial, ou seja, um trecho reutilizável de *view*.
- ◆ **View()** – Retorna um *view* criado no projeto.

Para executar o exemplo citado, pode ser necessário encerrar e após reiniciar o Kestrel. Para isso, clique no terminal, use **Ctrl + C** e depois execute `dotnet build` seguido de `dotnet run` (não se esqueça de antes salvar o arquivo editado).

Depois, no *browser*, use o endereço `<http://localhost:5000/Tests/Welcome>`. A página ficará totalmente em branco. Agora, inclua um parâmetro com `?name=<Algum Nome>` na URL (por exemplo, `<http://localhost:5000/Tests/Welcome?name=Fulano>`).

O resultado na tela deve ser apenas a palavra “Fulano” ou o nome que você digitou no parâmetro. Veja novamente o código de **TestsController** e note que *name*, o parâmetro da URL, é mapeado para o parâmetro da ação **Welcome**, que recebe o valor informado (“Fulano”) e o utiliza dentro do método.

Contudo, não queremos algo tão simples assim. Portanto, integraremos um *view* bem formado a essa informação que recebemos do parâmetro.

Para tanto, utilizaremos **ViewBags**.

Primeiramente, alteraremos a nossa ação **Welcome** de acordo com o trecho de código a seguir:

```
public IActionResult Welcome(string name)
{
    ViewBag.Name = name;
    return View();
}
```

No caso, definimos **ViewBag.Name** e passamos a esse atributo o valor do parâmetro *name* recebido pelo método **Welcome**.

Ao alterar um *controller*, será necessário um novo *build* para aplicar as modificações no sistema. Use sempre **Ctrl + C** no terminal para finalizar um *run* em execução e execute `dotnet run` para recompilar e rodar a aplicação.

Agora, já podemos fazer um *view* completo dessa ação. Na pasta **Views/Tests**, clicaremos no botão direito do *mouse* e depois selecionaremos **New file**. Em seguida, criaremos **Welcome.cshtml**.

O conteúdo desse arquivo será o seguinte:

```
<h1>Seja bem vindo(a) @ViewBag.Name</h1>
<p>este é um site em construção.</p>
<p>estamos aprendendo ASP.NET MVC</p>
```

Veja que não precisamos declarar **ViewBag** aqui (utilizada diretamente em `@ViewBag.Name`). Prosseguindo, rodaremos a aplicação e, no navegador, informaremos um nome para o parâmetro *name*, como no teste anterior.

O resultado deve ser semelhante ao da figura a seguir (no exemplo, usando o endereço `<http://localhost:5000/Tests/Welcome?name=Fulano>`)

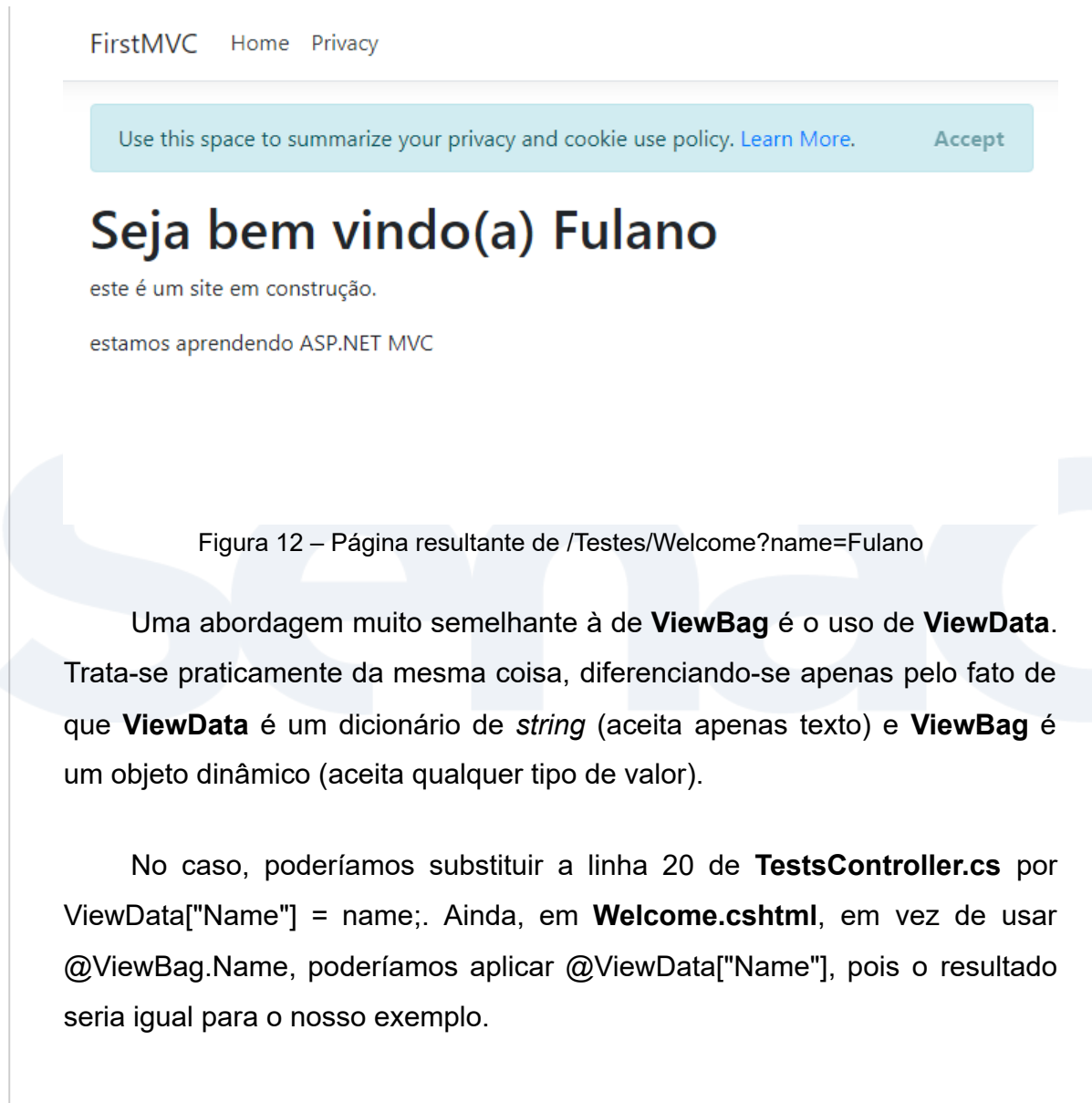


Figura 12 – Página resultante de /Testes/Welcome?name=Fulano

Uma abordagem muito semelhante à de **ViewBag** é o uso de **ViewData**. Trata-se praticamente da mesma coisa, diferenciando-se apenas pelo fato de que **ViewData** é um dicionário de *string* (aceita apenas texto) e **ViewBag** é um objeto dinâmico (aceita qualquer tipo de valor).

No caso, poderíamos substituir a linha 20 de **TestsController.cs** por `ViewData["Name"] = name;`. Ainda, em **Welcome.cshtml**, em vez de usar `@ViewBag.Name`, poderíamos aplicar `@ViewData["Name"]`, pois o resultado seria igual para o nosso exemplo.

Existem duas ações para **TestsController**. Em ambas, é usado o mesmo leiaute de página com uma barra superior de menu, uma faixa azul alertando sobre política de privacidade e um rodapé com informação de *copyright*. Para descobrir a razão de isso acontecer, abriremos o arquivo **/Views/Shared/_Layout.cshtml**.

Não se assuste com a quantidade de códigos. Algumas coisas, em especial para os nossos estudos, podem ser ignoradas. As *tags* `<environment>` não são próprias do HTML. Elas são, sim, criadas pelo ASP.NET Core para definir o que deve aparecer quando a aplicação está em ambiente de desenvolvimento e quando está em ambiente de produção.

Apenas para esclarecer, na listagem a seguir retiraremos as *tags* <environment exclude="Development">, pois os conteúdos delas não são mostrados no nosso ambiente, que é de desenvolvimento.



```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-s
cale=1.0" />

  <title>@ViewData["Title"] - FirstMVC</title>

  <environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bo
otstrap.css" />
  </environment>
  <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
  <header>
    <nav class="navbar navbar-expand-sm navbar-toggleable-sm
navbar-light bg-white border-bottom box-shadow mb-3">
      <div class="container">
        <a class="navbar-brand" asp-area="" asp-controll
er="Home" asp-action="Index">FirstMVC</a>
        <button class="navbar-toggler" type="button" dat
a-toggle="collapse" data-target=".navbar-collapse" aria-controls="navbarSupportedCo
ntent"
          aria-expanded="false" aria-label="Toggle
navigation">

          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="navbar-collapse collapse d-sm-inline
-flex flex-sm-row-reverse">
          <ul class="navbar-nav flex-grow-1">
            <li class="nav-item">
              <a class="nav-link text-dark" asp-ar
ea="" asp-controller="Home" asp-action="Index">Home</a>
            </li>
            <li class="nav-item">
              <a class="nav-link text-dark" asp-ar
ea="" asp-controller="Home" asp-action="Privacy">Privacy</a>
            </li>
          </ul>
        </div>
      </nav>
    </header>
    <div class="container">
      <partial name="_CookieConsentPartial" />
      <main role="main" class="pb-3">
        @RenderBody()
      </main>
    </div>
  </body>
</html>

```

```

        <footer class="border-top footer text-muted">
            <div class="container">
                @ 2019 - FirstMVC - <a asp-area="" asp-controller="H
ome" asp-action="Privacy">Privacy</a>
            </div>
        </footer>

        <environment include="Development">
            <script src="~/lib/jquery/dist/jquery.js"></script>
            <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.j
s"></script>

        </environment>
        <script src="~/js/site.js" asp-append-version="true"></sc
ript>

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

```

A partir daí, já podemos notar algumas características. São elas:

- ◆ Finalmente, a *tag* <html> foi encontrada. Todo o entorno de um *view* está definido pelo *leiaute*.
- ◆ @ViewData["Title"] usa uma informação repassada pelo *view*. Reveja **Index.cshtml** para um exemplo.
- ◆ A *tag* <partial name="_CookieConsentPartial" /> é importante. Ela define a renderização de um *view* parcial, ou seja, um *view* que, na verdade, é um trecho reutilizável. Reveja **Views/Shared/_CookeConsentPartial.cshtml** para conferir o conteúdo (texto que indica o uso de *cookies* que aparecem no topo de todas as páginas).
- ◆ Dentro da *tag* <main>, existe a função mais notória de **_Layout**: @RenderBody(). É a partir dela que o conteúdo do *view* CSHTML é incluído, embutido na página. Quando **/Tests/Welcome** é chamado, por exemplo, o método de ação encontra **Welcome.cshtml**, o qual é processado, e depois o seu conteúdo é embutido exatamente nessa linha, ou seja, nessa chamada ao método **RenderBody()**.
- ◆ Por fim, é possível notar @RenderSection(). Tal método é usado para embutir outras informações (no caso, inclusão de *scripts*).

Também podemos alterar o nosso *view* para simplesmente rejeitar qualquer *leiaute*. Para tanto, podemos experimentar executar isso com **Welcome.cshtml**, fazendo a seguinte modificação:

```
@{
    Layout = null;
}

<h1>Seja bem vindo(a) @ViewData["Name"]</h1>
<p>este é um site em construção.</p>
<p>estamos aprendendo ASP.NET MVC</p>
```

Dentro do bloco de marcação **@**, definimos a propriedade **Layout** como nula. Isso retira a vinculação desse *view* com o **_Layout.cshtml**. O resultado é uma página muito mais simples (salve e acesse a ação **Welcome** para verificar):

Seja bem vindo(a) Fulano

este é um site em construção.

estamos aprendendo ASP.NET MVC

Figura 13 – Página **Welcome** sem nenhum *leiaute*

Também podemos criar um *leiaute* novo e utilizá-lo em páginas específicas. Para testar, clicaremos com o botão direito na pasta **Views/Shared**, selecionaremos **New file** e criaremos um novo arquivo chamado **_Layout2.cshtml**. Neste, incluiremos o seguinte código HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Another Layout</title>
  </head>
  <body>
    <header>
      <h2 style="color: red">Página de tests</h2>
    </header>
    <main style="color: blue">
      @RenderBody()
    </main>
  </body>
</html>
```

Veja que o código é muito mais simples que o do leiaute original, definindo apenas um cabeçalho em vermelho e o texto do conteúdo em azul. É importante que **RenderBody()** esteja presente.

Voltando para **Welcome**, modificaremos a primeira instrução por:

```
Layout = "_Layout2";
```

Observe ainda que não precisamos informar a extensão .cshtml. O resultado é a nossa página com um novo leiaute:

Página de tests

Seja bem vindo(a) Fulano

este é um site em construção.

estamos aprendendo ASP.NET MVC

Figura 14 – Página **Welcome** usando o novo leiaute

Também podemos aplicar esse novo layout à visão **View/Tests/Greetings.cshtml** e separar as duas porções de nosso FirstMVC entre o que está no *controller* **Home** e o que está sob **Tests**.

Clique ou toque para visualizar o conteúdo.

Você se sente confiante para continuar o exercício? Em caso afirmativo, mãos à obra!



No *controller* **Tests**, crie uma nova *action* que receba dois valores inteiros por parâmetro – valores que virão da URL – e some-os. Repasse o resultado obtido a um *view* e mostre na página o texto “O resultado da soma de [valor 1 informado] e [valor 2 informado] é [resultado]”, substituindo [valor 1 informado] e [valor 2 informado] pelos valores que o usuário informou e [resultado] pelo valor do resultado final. Utilize **ViewBag** e **ViewData** para transmitir o valor. Na página, use **Layout2**.

A partir de agora, integraremos um pouco mais os *controllers* e usaremos *models* em novos exemplos.

Criando *model* e integrando com *view*

Para começar, iniciaremos um novo projeto para explorar outras características do ASP.NET Core MVC.

O nosso projeto consiste em cadastrar os alunos e as suas notas. O *software* deve:

- ◆ Permitir a inclusão de um aluno, sendo informados nome, idade, disciplina (entre matemática, português e ciências) e média final
- ◆ Listar os alunos em tela, marcando em verde os aprovados e em vermelho os reprovados

Por ora, não nos ocuparemos com o banco de dados (a persistência será simulada). Começaremos criando uma nova pasta e um novo projeto MVC. Você já sabe os passos: no Visual Studio Code, abra uma pasta nova, acione o terminal e use o comando `dotnet new mvc --no-https`. Aqui, usaremos o nome “NotasAluno” para a pasta, mas você pode usar outro nome, desde que fique atento para alterar corretamente os *namespaces* e os *usings* quando necessário.

Utilizaremos a estrutura já definida no projeto MVC, a qual, como anteriormente, apresenta a página padrão MVC de boas-vindas. Antes de tudo, modificaremos o arquivo **Views/Shared/_Layout.cshtml**. Apague todo seu conteúdo e substitua pelo código abaixo:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-sca
le=1.0" />

  <title>@ViewData["Title"] - NotasAlunos</title>
</head>
<body>
  <header>
    <h2>Cadastro Escolar</h2>
  </header>
  <div>
    <main role="main">
      @RenderBody()
    </main>
  </div>

  <footer>
    <div>
      © 2019 - NotasAlunos
    </div>
  </footer>
</body>
</html>

```

Em **View/Shared**, podemos excluir as visões parciais **_CookieConsentPartial.cshtml** e **_ValidationScriptsPartial.cshtml**, pois elas não serão usadas no projeto. Em **Views/Home**, excluiríamos também **Privacy.cshtml**. Em **HomeController**, eliminaremos a ação **Privacy()**, pois ela não será mais utilizada.

_ViewImports.cshtml e **_ViewStart.cshtml** são *views* de configuração usados implicitamente pelos outros *views*. Portanto, é melhor não removê-los. **_ViewImports** é responsável por adicionar referências úteis aos *views*, e **_ViewStart** define configurações padrões para todos os *views* (por exemplo, o *layout* padrão).

Clique ou toque para visualizar o conteúdo.



Cadastrar alunos

A primeira funcionalidade de nosso sistema será cadastrar alunos. Para tanto exploraremos os princípios de *model* no MVC, construindo um modelo para “aluno” um pequeno repositório temporário de dados.

Com o botão direito do *mouse* sobre a pasta **Models**, escolheremos **New C class** (fornecido pela extensão C# Extensions) e, na barra superior, informaremos nome **Aluno.cs**. Uma classe básica deve ser criada com *namespace* e com declaração de classe **Aluno**. Em seguida, criaremos, dentro da classe, uma propriedade para nome do aluno:

```
public string Nome {get; set;}
```

Após, incluiremos propriedades para a idade (*int*), a disciplina (*string*) e a nota (*double*).

Por padrão, propriedades são nomeadas com a primeira letra maiúscula.

Como resultado, a classe deve estar assim:

```
namespace NotasAlunos.Models
{
    public class Aluno
    {
        public string Nome {get; set;}
        public int Idade {get; set;}
        public string Disciplina {get; set;}
        public double Nota {get; set;}
    }
}
```

Estamos criando um *domain model* (modelo de domínio), ou seja, uma classe que representa uma entidade ou uma operação relativa à regra de negócio de nossa aplicação. Além disso, precisaremos de informações de alunos. Segundo a descrição da aplicação, são informações necessárias: nome, idade, disciplina e nota.

Model é considerado a parte mais importante de um projeto MVC justamente por tratar das regras da aplicação. No nosso exemplo, não há nada complexo, pois as informações necessárias são poucas e as regras são simples.

Agora, em **HomeController**, incluiremos uma ação para mostrar como o *model Aluno* pode ser usado em conjunto com o *view*. No *controller*, adicionaremos uma ação **Detalhe** sem parâmetros e que retorne **View()**.

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

Em **Views/Home**, criaremos um *view* **Detalhe.cshtml**. Nele, poderíamos mostrar as informações de alunos usando **ViewData** ou **ViewBag**. Isso, de fato, funcionaria, mas o ASP.NET Core MVC tem uma maneira muito melhor de realizar a integração do *view* com as informações do *model*: o *model binding*, ou seja, a construção de um *view fortemente tipado* a partir de uma classe de modelo.

Em **Detalhe.cshtml**, incluiremos uma nova diretiva, qual seja **@model**:

```
@model Aluno
```

Essa diretiva indica que o *view* usará as informações do modelo **Aluno** em suas páginas. A partir de então, poderemos usar várias outras *tags* e auxiliares que ligam diretamente as informações da página às propriedades do modelo.

Depois, incluiremos um título para a página usando uma tag `<h2>`, que estática. Para esse tipo de informação, podemos usar `@Model` para acessar as suas propriedades. Então, usaremos `@Model.Nome`:

```
@model Aluno

<h2>Detalhes do Aluno @Model.Nome</h2>
```

Após, utilizaremos *labels* e campos *input* para mostrar as informações de um aluno:

```
@model Aluno

<h2>Detalhes do Aluno @Model.Nome</h2>
<p>
    <label asp-for="Nome">Nome do aluno:</label>
    <input asp-for="Nome" />
</p>
<p>
    <label asp-for="Idade">Idade:</label>
    <input asp-for="Idade" />
</p>
<p>
    <label asp-for="Disciplina">Disciplina:</label>
    <input asp-for="Disciplina" />
</p>
<p>
    <label asp-for="Nota">Nota:</label>
    <input asp-for="Nota" />
</p>
```

O ASP.NET Core 2 introduz algumas *tags* e alguns atributos auxiliares que ligam diretamente os elementos do HTML às propriedades do *model*. Repare na *tag* `<input asp-for="Nome" />`. O atributo *asp-for* não é próprio do HTML, mas é usado dentro do MVC para indicar que o *input* lerá e interpretará a propriedade “Nome” do *model* (**Aluno**) e montará um *input* adequado.

Preste muita atenção ao adicionar os nomes informados. Caso seja colocado o nome de uma propriedade que não existe no modelo, o sistema emitirá um erro. Por exemplo, se você usar `<inputv asp-for="NotaFinal"/>` em vez de "Nota", obterá seguinte falha ao tentar acessar a página:

A imagem mostra uma página branca de erro com a mensagem "An error occurred during the compilation of a resource required to process this request. Please review the following specific error details and modify your source code appropriately". Abaixo, está o caminho para o arquivo `Detalhes.cshtml`. Mais abaixo, tem-se outra mensagem: "Aluno does not contains a definition for NotaFinal and no extension method NotaFinal". Por fim, ainda mais abaixo, há o trecho `<input asp-for="NotaFinal" />` escrito em vermelho.

An error occurred during the compilation of a resource required to process this request. Please review the following specific error details and modify your source code appropriately.

C:\Meus Projetos VS\NotasAlunos\Views\Home\Detalhe.cshtml

'Aluno' does not contain a definition for 'NotaFinal' and no extension method 'NotaFinal' accepting a first argument of type 'Aluno' could be found (are you missing a using directive or an assembly reference?)

18. `<input asp-for="NotaFinal" />`

[Show compilation source](#)

Figura 15 – Página de erro

O aviso de falha informa no título que houve um erro na compilação do CSHTML qual seja "Aluno' não contém a definição para 'NotaFinal'". Em suma, não há uma propriedade ou um método com esse nome no nosso modelo. Recomenda-se atentar sempre para as mensagens de erro que possam surgir na aplicação.

Para testar esse `view`, criaremos um aluno em **HomeController**, no método **Detalhe()**:

```
public IActionResult Detalhe()
{
    Aluno aluno = new Aluno();
    aluno.Nome = "Fulano de Tal";
    aluno.Idade = 15;
    aluno.Disciplina = "Ciências";
    aluno.Nota = 9.5;

    return View(aluno);
}
```

Veja que informamos por parâmetro para o método **View()** o objeto **Aluno** que acabamos de criar. É isso que faz com que o CSHTML seja informado de que objeto e/ou dados ele precisa usar.

Agora já podemos testar usando nosso dotnet run no terminal do Visual Studio Code e acessando <http://localhost:5000/Home/Detalhe> (não se esqueça de antes salvar todas as alterações).

Cadastro Escolar

Detalhes do Aluno Fulano de Tal

Nome do aluno:

Idade:

Disciplina:

Nota:

© 2019 - NotasAlunos

Figura 16 – Resultado da página **Detalhe**

O resultado não deve ser muito diferente do da imagem anterior. Ainda voltaremos a usar esse *view* para realizar alteração no nosso aluno, mas agora já estamos cientes de como as informações podem passar de uma camada para outra.

no MVC.



POSTs: cadastrando a partir de um formulário

Chegou o momento de avançar e criar a funcionalidade de cadastrar alunos. Para tanto, precisaremos de uma nova ação que mostrará o formulário de cadastro. Em **HomeController**, adicionaremos um método **Cadastro()** que retorne **ActionResult**, como na listagem a seguir:

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

Em **Views/Home**, criaremos o *view* **Cadastro.cshtml** usando, com anterioridade, **@model Aluno** e construindo assim um *view* fortemente tipado.

```
@model Aluno

<form method="POST">
  <p>
    <label asp-for="Nome">Nome:</label>
    <input asp-for="Nome" />
  </p>
  <p>
    <label asp-for="Idade">Idade:</label>
    <input asp-for="Idade" />
  </p>
  <p>
    <label asp-for="Disciplina">Disciplina:</label>
    <select asp-for="Disciplina">
      <option value="Português">Português</option>
      <option value="Matemática">Matemática</option>
      <option value="Ciências">Ciências</option>
    </select>
  </p>
  <p>
    <label asp-for="Nota">Nota:</label>
    <input asp-for="Nota" />
  </p>
  <input type="submit" value="Cadastrar" />
</form>
```

Para prosseguir, teremos que encerrar o servidor Kestrel, caso esteja iniciado usando **Ctrl + D** no terminal do Visual Studio Code e também depois um novo `dotnet run`.

Caso `@model` seja omitido, podem ocorrer erros já no momento da compilação do projeto. Isso reforça o aspecto *server-side* de recursos como o *asp-for* e o *model binding*.

O resultado, ao acessar a URL `<http://localhost:5000/Home/Cadastro>`, será semelhante ao da imagem a seguir:

Cadastro Escolar

Nome:

Idade:

Disciplina:

Nota:

© 2019 - NotasAlunos

Figura 17 – Página de cadastro **Detalhe**

É um bom exercício compreender o que o ASP.NET Core gera a partir do *model binding*. Uma das formas de fazer isso é clicar com o botão direito do *mouse* em uma área vazia da página, no navegador, e solicitar **Exibir código-fonte**, disponível na maioria dos navegadores.

Você conseguirá visualizar o HTML bruto, transformado após os processamentos das *tags* e dos atributos de apoio do Razor. Outra forma é inspecionar clicando no botão **F12** em *browsers* como Chrome, Firefox e Opera.

Há uma seção em que o código da página é mostrado. É possível interagir passando com o *mouse* pelos elementos e destacando-os. Esse recurso será muito útil especialmente no trabalho com *front-end*.

Cadastro Escolar

Nome:

Idade:

Disciplina: Português

Nota:

© 2019 - NotasAlunos

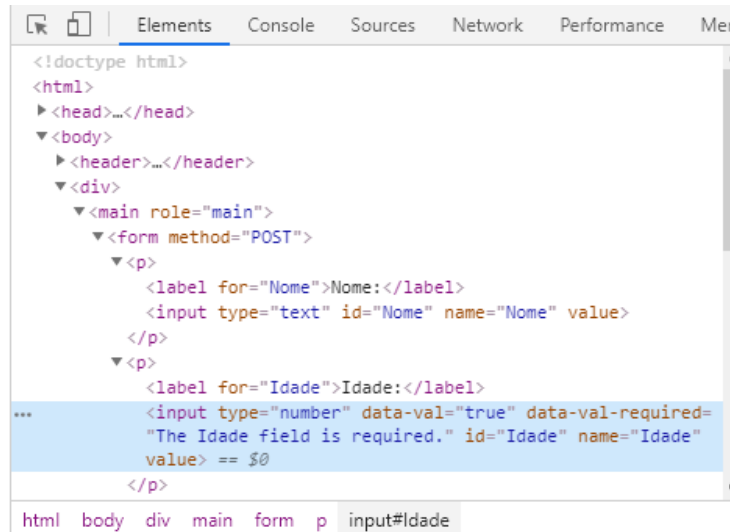


Figura 18 – Página de cadastro desenvolvida

A imagem mostra, no navegador Chrome, que a ferramenta de desenvolvimento, acessada pela tecla F12, está aberta ao lado da página de cadastro desenvolvida. Na aba de desenvolvimento, vê-se o código HTML, com o qual se pode interagir. A imagem ainda mostra o elemento “*input* idade” selecionado no código e destacado no visual da página.

O campo “Idade” (figura 18) foi adequadamente incluído no HTML como um *input* do tipo *number*.

Você deve inclusive ter notado que a tag `<form>` não continha um atributo *action* mas apenas um `method = “POST”`. Para fazer o **POST**, precisamos definir uma nova ação específica no nosso **HomeController**. A assinatura desse método deve ficar assim:

```
[HttpPost]
public IActionResult Cadastro(Aluno aluno)
{
    return View();
}
```

Ademais, duas características são importantes. São elas:

1. O atributo **[HttpPost]** foi utilizado sobre a assinatura do método. Ele indicará o caminho que o *request* deve fazer. Isso significa que está sendo explicitamente indicado que essa ação será usada para uma postagem de dados.

2. Um parâmetro do tipo **Aluno** foi incluído para o método. Isso ocorre devido ao *model binding*. De fato, a ação receberá um objeto **Aluno** preenchido com todas as informações que o usuário digitou no cadastro.

O nosso cadastro nos direcionará, após a gravação ter sido realizada com sucesso, até agora, para o próprio cadastro. Porém, podemos criar um *view* que mostre uma mensagem para o usuário de que o cadastro foi bem-sucedido. Faremo isso incluindo em **Views/Home** um *view* Concluido.cshtml. É muito simples:

```
<h2>Cadastro concluído com sucesso.</h2>
<p><a asp-action="Cadastro">Realizar novo Cadastro</a>
</p>
```

Apesar de simples, há uma novidade: ao usar a *tag* `<a>`, incluímos o atributo *asp-action* com valor “Cadastro”. Isso indica ao ASP.NET que este deve criar um *link* que redirecione diretamente para a ação **Cadastro** de **HomeController**. Caso quiséssemos fazer isso manualmente, seria bem simples: bastaria indicar `href="/Home/Cadastro"`.

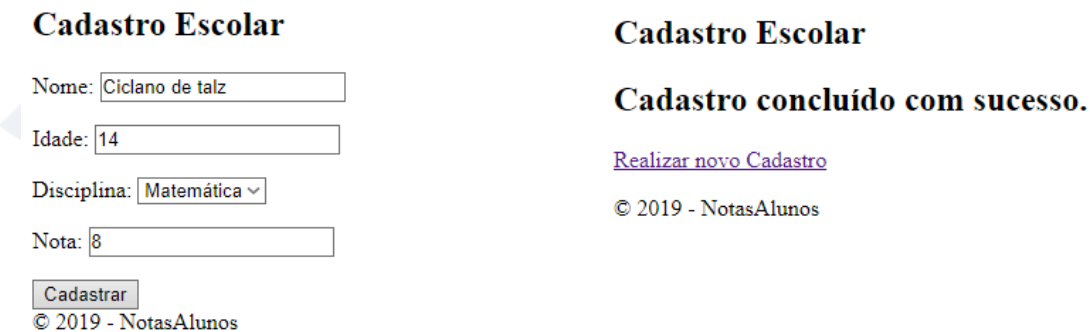
Agora, precisamos modificar a nossa ação **POST** de cadastro para mandá-la para o *view* “Concluido”. Para isso, modificaremos um elemento no método:

```
public IActionResult Cadastro(Aluno aluno)
{
    return View("Concluido");
}
```

Em vez de apenas **View()**, passamos por parâmetro um texto. `View("Concluido")` indica ao ASP.NET Core MVC que, em vez de buscar por `Cadastro.html`, ele deve procurar por `Concluido.cshtml`. Isso flexibiliza o uso de *view* e possibilita reutilizar alguns recursos.

Preste atenção ao colocar acentos nos códigos. No exemplo anterior `Concluido.cshtml` não está acentuado. Caso seja utilizado `View("Concluído")`, sistema não encontrará o *view*, emitindo erro. De forma geral, recomendam-se evita acentuações durante a programação. Elas devem aparecer apenas dentro de texto prontos (valores), não em instruções, *tags* ou nomes de arquivos ou de classes.

É hora de testar o fluxo do nosso cadastro. Para tanto, como sempre finalizaremos o Kestrel, caso esteja rodando, e usaremos `dotnet run` novamente. A URL agora é `<localhost:5000/Home/Cadastro>`. Inseriremos então as informações no campos do cadastro e clicaremos em **Cadastrar**.



Cadastro Escolar

Nome:

Idade:

Disciplina:

Nota:

© 2019 - NotasAlunos

Cadastro Escolar

Cadastro concluído com sucesso.

[Realizar novo Cadastro](#)

© 2019 - NotasAlunos

Figura 19 – Cadastro preenchido (à esquerda) e mensagem de sucesso do cadastro (à direita)

Detalhe

Você pode testar o *link* “Realizar novo cadastro” para comprovar que o redirecionamento está correto.

Como saber se tudo está cadastrado corretamente?

A resposta mais curta e óbvia é: não há como saber. Realmente, não há nenhum cadastro sendo realizado, pois estamos apenas fazendo o processo de direcionar para uma página de conclusão.

Clique ou toque para visualizar o conteúdo.

Você está se sentindo seguro quanto à programação? Em caso afirmativo, aceite o próximo desafio!



Verifique se as informações recebidas no parâmetro **Aluno** estão corretas. Para isso, compare o conteúdo de **ViewBag** ou **ViewData** com as informações recebidas e altere o `view.concluido.cshtml` para mostrar os valores de **ViewBag/ViewData**. Os valores exibidos devem ser os mesmos que você informou no cadastro.

Agora, montaremos a nossa camada de persistência. Como comentado no início do projeto, ainda não usaremos banco de dados. Ao contrário, faremos uma simulação usando uma lista simples de alunos. As operações serão de inclusão e de listagem dos alunos cadastrados.

A camada para fazer isso é a camada *model*. Nenhuma persistência deve estar presente nem em *controller*, nem em *view*. Por isso, clicaremos com o botão direito do *mouse* sobre a pasta **Models** e selecionaremos a opção **New C# class**. Em seguida, informaremos o nome `BaseDados`.

Após criar o arquivo, ainda faremos duas alterações. A primeira delas é deixar a classe estática, incluindo a palavra `static` logo antes de `class`.

Classes estáticas não precisam ser instanciadas com `new` e estão ativas durante todo o tempo de vida da aplicação.

A segunda alteração é criar dentro da classe uma lista também estática, chamada "alunos". O código deve ficar assim:

```
namespace NotasAlunos.Models
{
    public static class BaseDados
    {
        private static List<Aluno> alunos = new List<Aluno>();
    }
}
```

Armazenaremos os objetos de **Aluno** nessa lista. Em seguida, implementaremos dois métodos para manipular esses dados. O primeiro deles será o de inclusão; e o segundo, o de listagem.

```
namespace NotasAlunos.Models
{
    public static class BaseDados
    {
        private static List<Aluno> alunos = new List<Aluno>();

        public static void Incluir(Aluno aluno)
        {
            alunos.Add(aluno);
        }

        public static List<Aluno> Listar()
        {
            return alunos;
        }
    }
}
```

A nossa camada de persistência está suficientemente simples para não tirarmos o foco do ASP.NET Core MVC.

Antes de prosseguir, faremos um comando dotnet build. O resultado não deve ser animador:

```
Models\BaseDados.cs(5,24): error CS0246: O nome do tipo ou do namespace "List<>" não pode ser encontrado (está faltando uma diretiva using ou uma referência de assembly?) [C:\Meus Projetos VS\NotasAlunos\NotasAlunos.csproj]
```

O que aconteceu?

No caso, o erro indica a linha 5, que é a declaração de `List<int> alunos`. Ele cita na mensagem que `List<>` não é um tipo reconhecido – embora saibamos que tal tipo existe, sim.

É necessária, no caso em questão, uma instrução *using* no topo do arquivo para que o compilador tenha conhecimento desse recurso.

O Visual Studio Code pode ajudar sugerindo o *using* a ser realizado. Clique no tipo **List** e visualize uma lâmpada amarela na parte esquerda. Ao clicar na lâmpada, você terá algumas opções. Uma delas é **using System.Collections.Generic;**, a qual deverá solucionar o problema.

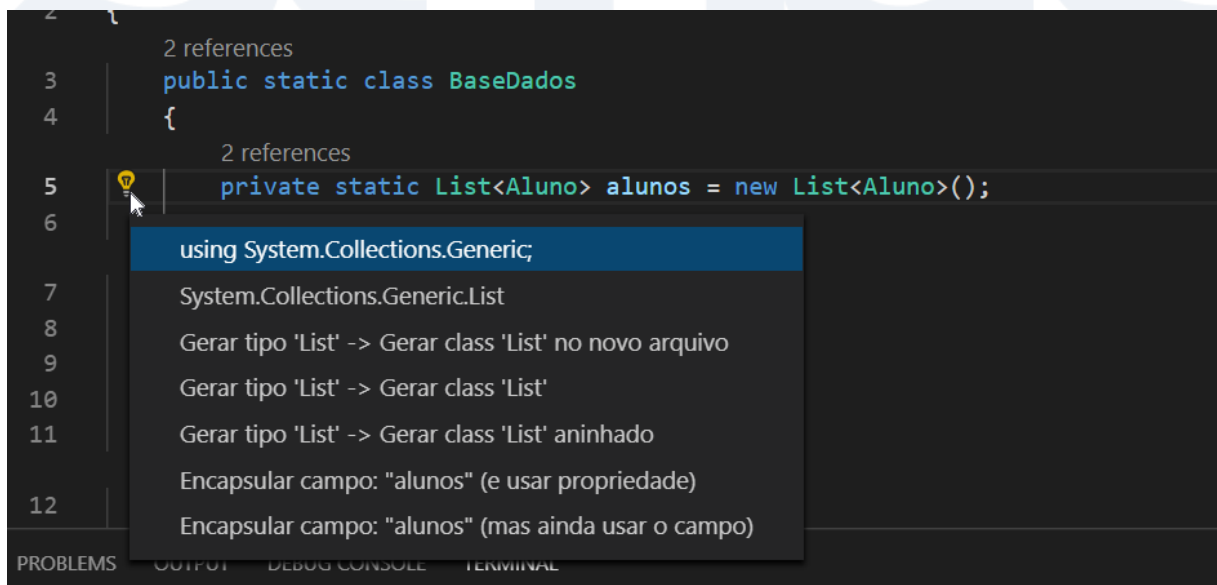


Figura 20 – Recurso de sugestões do Visual Studio Code

As dicas de autocompletamento do editor ajudam a reduzir o tempo de desenvolvimento. O Visual Studio Community é muito mais completo nesse aspecto, mas o Visual Studio Code deve ser suficiente para nós durante o curso.

Agora, utilizaremos essa classe para salvar o aluno que foi cadastrado na ação **Cadastro** de **HomeController**. Por isso, voltando a **HomeController.cs**, localize o método **[HttpPost]Cadastro()** e faça a seguinte alteração:

```
[HttpPost]
public IActionResult Cadastro(Aluno aluno)
{
    BaseDados.Incluir(aluno);
    return View("Concluido");
}
```

Com isso, concluímos o nosso cadastro. Ainda não está claro se a inclusão realmente aconteceu, mas isso será esclarecido no momento de fazer a listagem dos alunos.

Senac

Lista de alunos a partir de model

A listagem dos alunos e a indicação de que eles estão aprovados ou reprovados são uma parte importante de nosso sistema. Para tanto, já temos um primeiro recurso: o método **Listar()** da classe **BaseDados()**. Portanto, criaremos agora ações no *controller* para localizar essa listagem e *views* para mostrar o resultado dessas operações.

Em **HomeController**, adicionaremos um método de ação chamado **Listagem**, sem parâmetros. Nele, utilizaremos o método **Listar** (vindo de **BaseDados**) e o repassaremos ao método **View()** no retorno.

```
public IActionResult Listagem()
{
    List<Aluno> alunos = BaseDados.Listar();
    return View(alunos);
}
```

O código completo de **HomeController**, após as alterações, deve ficar assim (apenas para conferir):

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using NotasAlunos.Models;

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

        public IActionResult Detalhe()
        {
            Aluno aluno = new Aluno();
            aluno.Nome = "Fulano de Tal";
            aluno.Idade = 15;
            aluno.Disciplina = "Ciências";
            aluno.Nota = 9.5;

            return View(aluno);
        }

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

        [HttpPost]
        public IActionResult Cadastro(Aluno aluno)
        {
            BaseDados.Incluir(aluno);
            return View("Concluido");
        }

        public IActionResult Listagem()
        {
            List<Aluno> alunos = BaseDados.Listar();
            return View(alunos);
        }

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
        public IActionResult Error()
        {

```

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

Também criaremos um *view* em **Views/Home** chamado **Listagem.cshtml**. A primeira particularidade desse *view* é o seu *model*, o qual precisa lidar com uma lista de **Aluno** em vez de um único objeto. Por isso, o *view* iniciará com a seguinte instrução:

```
@model List<Aluno>
```

Assim, o *model* desse *view* será um objeto **List** em que cada elemento é um objeto do tipo **Aluno**.

Para listar os alunos, usaremos a *tag* `<table>`.

```
@model List<Aluno>

<h2> Listagem dos Alunos Cadastrados </h2>

<table>
  <thead>
    <tr>
      <th>Nome</th>
      <th>Idade</th>
      <th>Disciplina</th>
      <th>Nota</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td></td>
      <td></td>
      <td></td>
      <td></td>
    </tr>
  </tbody>
</table>
```

É importante se questionar: como os valores de nomes de vários alunos podem ser incluídos se é necessário montar todas as linhas da tabela e nem ao menos se sabe quantos alunos foram cadastrados?

Quando o comando `@if` foi mencionado, havia a possibilidade de realizar laços nos *views* Razor. É o que precisamos aqui: um laço que percorra todos os itens da lista (o próprio *model*) e que monte cada uma das linhas automaticamente. Assim, utilizaremos `@foreach`:

```
@model List<Aluno>

<h2> Listagem dos Alunos Cadastrados </h2>

<table>
  <thead>
    <tr>
      <th>Nome</th>
      <th>Idade</th>
      <th>Disciplina</th>
      <th>Nota</th>
    </tr>
  </thead>
  <tbody>
    @foreach (Aluno a in Model)
    {
      <tr>
        <td>@a.Nome</td>
        <td>@a.Idade</td>
        <td>@a.Disciplina</td>
        <td>@a.Nota</td>
      </tr>
    }
  </tbody>
</table>
```

O laço `@foreach (Aluno a in Model)` percorre todos os itens que estão na lista passada por parâmetro ao *view*. A cada iteração, o objeto “a”, do tipo *Aluno*, recebe um item da lista. Com isso, podemos usar “a” e suas propriedades para colocar valor para as colunas da linha na tabela.

Então, precisamos referenciar a variável com `@a`. Isso porque já estamos entre *tags* HTML (`<tr>` e `<td>`) que estão dentro do laço (após `{` e antes de `}`).

Para testar, salvaremos todos os arquivos, utilizaremos o comando `dotnet run` e acessaremos `<localhost:5000/Home/Listagem>`.

Cadastro Escolar

Listagem dos Alunos Cadastrados

Nome Idade Disciplina Nota
© 2019 - NotasAlunos

Figura 21 – Resultado de **Home/Listagem** (nenhum registro)

O resultado pode ser um pouco decepcionante, já que a listagem não traz nenhum aluno. Porém, isso já era esperado, pois a nossa persistência não persiste tanto assim. Ela grava na memória, em um *list*, alguns objetos cadastrados. Ao encerrar a aplicação, naturalmente, a lista é excluída.

Por isso, para testar essa funcionalidade, primeiro passe por `/Home/Cadastro` e cadastre ao menos um aluno. Depois volte a `/Home/Listagem` e verifique se ele foi incluído.

Para facilitar a navegação, incluiremos *links* em alguns *views*. Em `/Home/Views/Concluido.cshtml`, adicionaremos o seguinte *link* no final do *view*.

```
<p><a asp-action="Listagem">Ir Para Listagem</a></p>
```

Ainda, no final de `/Home/Views/Listagem.cshtml`, incluiremos o seguinte *link*:

```
<a asp-action="Cadastro">Cadastrar novo aluno</a>
```

Agora, você poderá navegar mais facilmente entre as páginas e cadastrar vários alunos para verificar a listagem.

Cadastro Escolar

Cadastro concluído com sucesso.

[Realizar novo Cadastro](#)

[Ir Para Listagem](#)

© 2019 - NotasAlunos

Cadastro Escolar

Listagem dos Alunos Cadastrados

Nome	Idade	Disciplina	Nota
Falano de Tal	10	Português	8
Ciclano de tal	13	Ciências	9,5

[Cadastrar novo aluno](#)
© 2019 - NotasAlunos

Figura 22 – Páginas de confirmação e de listagem com novos *links* de navegação

Um dos requisitos de nosso sistema é que ele diferencie os alunos aprovados dos reprovados. A tentação inicial é lançar as comparações (realmente simples) diretamente no *view*. Contudo, essa não é uma boa ideia.

1. Se outro *view* depender de *status* de aprovação/reprovação para alguma funcionalidade, a comparação ficará repetida.
2. Estando repetida, a manutenção é prejudicada. Imagine se houver a troca do limite da média para aprovação. Todos os pontos que fazem a comparação precisariam ser alterados, e certamente um deles ficaria para trás.
3. A camada exata de regra de negócio é a camada *model*.

Pensando assim, modificaremos o nosso modelo **Aluno**. Para tanto, abriremos o arquivo `/Models/Aluno.cs` e incluiremos ao final o seguinte método:

```
public bool EstaAprovado()
{
    return Nota >= 6;
}
```

Estamos definindo aqui a regra de aprovação. O aluno estará aprovado se ele tiver nota maior ou igual a 6. Nesse caso, o método retorna `true`; . Caso contrário, ele retorna `false` .

O código completo de **Aluno** ficou assim:

```
namespace NotasAlunos.Models
{
    public class Aluno
    {
        public string Nome {get; set;}
        public int Idade {get; set;}
        public string Disciplina {get; set;}
        public double Nota {get; set;}

        public bool EstaAprovado()
        {
            return Nota >= 6;
        }
    }
}
```

Agora, utilizaremos essa lógica na listagem para deixar a linha escrita em vermelho ou em verde, de acordo com o resultado do aluno. A nossa intenção é usar *style* na *tag* `<tr>` para alterar a cor da fonte (por exemplo, `<tr style="color:red;">` quando o aluno reprovar). Para isso, adicionaremos uma condicional dentro do laço *foreach*.

```
@model List<Aluno>

<h2> Listagem dos Alunos Cadastrados </h2>

<table>
  <thead>
    <tr>
      <th>Nome</th>
      <th>Idade</th>
      <th>Disciplina</th>
      <th>Nota</th>
    </tr>
  </thead>
  <tbody>
    @foreach (Aluno a in Model)
    {
      string color;
      if(a.EstaAprovado())
      {
        color = "color:green";
      }
      else
      {
        color = "color:red";
      }

      <tr style="@color">
        <td>@a.Nome</td>
        <td>@a.Idade</td>
        <td>@a.Disciplina</td>
        <td>@a.Nota</td>
      </tr>
    }
  </tbody>
</table>
<a asp-action="Cadastro">Cadastrar novo aluno</a>
```

No momento, estamos criando uma variável local (*color*) do tipo *string*, a qual, caso o aluno esteja aprovado, receberá “color:green” ou, em caso contrário, “color:red”. O valor dessa variável é usado no atributo *style* de `<tr>`. Alteramos, assim, dinamicamente o estilo das linhas da tabela.

É hora de testar novamente. Lembre-se de que sempre depois de uma alteração em *model* ou *controller* é necessário parar o servidor Kestrel, caso esteja sendo executado, e rodá-lo novamente com `dotnet run` (que também já faz a compilação).

Cadastro Escolar

Listagem dos Alunos Cadastrados

Nome	Idade	Disciplina	Nota
Fulano de Tal	10	Matemática	5,75
Ciclano de Tal	12	Português	8,5
Fulano de Tal	15	Português	10
Beltrano de Tal	9	Ciência	4

[Cadastrar novo aluno](#)

© 2019 - NotasAlunos

Figura 23 – Resultado da listagem já com a lógica de destaque dos aprovados e dos reprovados

Uma sugestão para diminuir o código utilizado para a lógica das cores no *view* é usar uma comparação ternária.

Observe este código:

```
string color;
    if(a.EstaAprovado())
    {
        color = "color:green";
    }
    else
    {
        color = "color:red";
    }

    <tr style="@color">
```

No trecho correspondente a **Listagem.cshtml**, em vez de usar o código citado anteriormente, poderíamos utilizar o seguinte código:

```
string color = a.EstaAprovado() ? "color:green" : "color:red";
<tr style="@color">
```

A funcionalidade permanece a mesma, mas o código fica menos poluído visualmente. Teste novamente e confira.

Clique ou toque para visualizar o conteúdo.

Se você está seguro quanto aos seus conhecimentos, o que acha de incluir a nova funcionalidade na listagem?



Inclua uma coluna com um *link* para a ação **Detalhe()** implementada e **NomeController**. A ação de detalhe deverá ser modificada para detalhar o aluno da linkada. Para tanto, adicione um parâmetro no método **Detalhe()** com o índice do aluno. Em vez de ser criado um novo **Aluno()**, será recuperada a listagem de alunos obtido o aluno de acordo com o índice informado por parâmetro. Veja o exemplo:

```
List<Aluno> todosAlunos = BaseDados.Listar();  
Aluno aluno = todosAlunos[indice];
```

No *link* da listagem, informe esse parâmetro. Talvez por isso, asp-action não se lequado.

Melhorando o leiaute

Já deve ter ficado claro que uma aplicação *web* é um vai e vem de operações que acontecem no servidor (mandadas para o *front-end*) e de comandos que acontecem no navegador (mandados para o *back-end*).

Agora, para finalizar, será estudado o leiaute no *front-end*. O intuito é melhorar um pouco o visual da página, incluir uma imagem de logo e trocar as fontes. Para tanto, são necessários CSS e imagens.

Confira o conteúdo **CSS e linguagem de marcação HTML** desta unidade curricular para saber mais detalhes sobre CSS.

Você deve lembrar que, ao fazermos a nossa limpeza inicial, a pasta **wwwroot** foi mantida. É lá que ficam os recursos extras como CSS, imagens e *scripts*.

Primeiramente, verificaremos a pasta **wwwroot** pela aba **Explorer** do Visual Studio Code. Nessa pasta, está o arquivo `site.css`. Nós até poderíamos criar outro arquivo CSS clicando com o botão direito do *mouse* na pasta < **New File**, mas, no caso, apenas apagaremos o conteúdo de `site.css` e colocaremos o nosso próprio estilo.

Portanto, modificaremos o conteúdo de `site.css` pelo seguinte conteúdo:

```
html{
  font-family: 'Lucida Sans', Verdana, sans-serif;
  font-size: 14px;
}

header{
  border-bottom: 2px solid #6f6f6f;
}
header img
{
  float:left;
  margin: 0 20px 0 10px;
}
footer{
  position: absolute;
  bottom:0;
}
```

Sem se ater muito a explicações sobre CSS, na folha de estilo citada estamos definindo um estilo de fonte para todo o HTML, uma borda para o <header> da página, algumas ações de posicionamento para uma imagem (a ser utilizada como logo dentro do *header*) e um reposicionamento do <footer> para o fim da página.

Agora, em **Views/Shared/_Layout.cshtml**, utilizaremos esse CSS. Para tanto, devemos incluir a *tag* <link rel="stylesheet" href="~/css/site.css" /> dentro da *tag* <head>.

```
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scal
e=1.0" />
  <link rel="stylesheet" href="~/css/site.css" />
  <title>@ViewData["Title"] - NotasAlunos</title>
</head>
```

A notação “~” no endereço indica que o recurso será buscado na raiz do projeto. Observe que em nenhum momento utilizamos o nome **wwwroot** para indicar o endereço de site.css. Tal endereço fica implícito e é tratado como o diretório raiz da aplicação.

Em **wwwroot**, incluiremos uma imagem para o logo. Para isso, criaremos primeiramente uma pasta **images** e salvaremos a imagem nela.

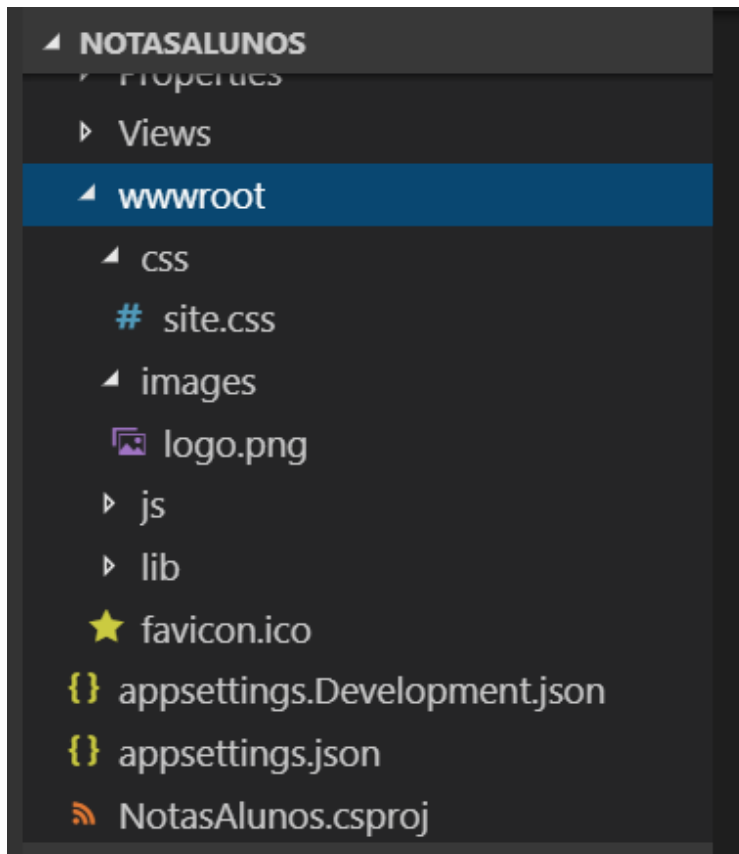


Figura 24 – Estrutura da pasta **wwwroot** após as modificações

Agora, faremos algumas mudanças no *header*, incluindo um logo e *links* para recursos do nosso sistema.

```
<header>
  
  <h1>Cadastro Escolar</h1>
  <nav>
    <a href="/Home/Index">Index</a>
    <a href="/Home/Cadastro">Cadastro</a>
    <a href="/Home/Listagem">Listagem</a>
  </nav>
</header>
```

Em ``, utilizamos uma referência sem o “~”. Também é válido usar “/” como a raiz do projeto. A diferença é que “~” é tratado via servidor e eventualmente trocado pela notação “/”, que se refere a cliente. Os *links* com `<a>` também estão em seus formatos básicos, sem atributo *asp-action*.

Com essas alterações, teremos um visual mais interessante:



Cadastro Escolar

[Index](#) [Cadastro](#) [Listagem](#)

Listagem dos Alunos Cadastrados

Nome	Idade	Disciplina	Nota
Fulano de Tal	10	Matemática	5,75
Ciclano de Tal	12	Português	8,5
Fulano de Tal	15	Português	10
Beltrano de Tal	9	Ciência	4

[Cadastrar novo aluno](#)

© 2019 - NotasAlunos

Figura 25 – Visual da aplicação modificado

Clique ou toque para visualizar o conteúdo.

Você está pronto para o último desafio? Ele é opcional.



A página **Index.cshtml** foi deixada de lado, continuando com o seu conteúdo original. Modifique-a para que ela mostre uma mensagem de boas-vindas adequada ao sistema e inclua *links* para as ações disponíveis.

Considerações finais

O ASP.NET é apenas uma das muitas tecnologias que permitem desenvolver aplicações *web* dinâmicas. Uma coisa em comum entre essas tecnologias é a dinâmica de *request-response*, na qual pedidos são realizados pelo *front-end* (o

navegador), processados pelo *back-end* (o servidor) e devolvidos ao *front-end* como resultado. Linguagens clássicas como PHP, Python e Ruby evoluíram sob essa mecânica e sob padrões como o MVC, usado pelo ASP.NET e largamente difundido (*vide frameworks* como CakePHP, Laravel e Ruby on Rails).

A programação de uma aplicação *web* precisa considerar essencialmente como enviar uma informação do cliente ao servidor (parâmetros e **POSTS** são alternativas), como processar tal informação e como retorná-la ao usuário (a separação de *model*, *view* e *controller* ajuda a não bagunçar os assuntos). Os mecanismos do Razor e o poder da linguagem C# são bons auxiliares na programação com ASP.NET Core MVC.

O próximo passo é evoluir a aplicação. Ainda há vários recursos importante disponíveis que não foram expostos neste material e uma integração com o banco de dados, que dará real funcionalidade ao sistema.