



## Programação Web ASP.NET MVC NÚCLEO

© 2017, NOEL LOPES

## PROGRAMA



- introdução ASP.NET MVC Núcleo
- O Model-View Controller (MVC)
- Criação de aplicações ASP.NET MVC Núcleo
- Introdução ao C # e Razor
- Entity Framework Core
- validação de dados
- Layouts e navegação
- Segurança
- Autenticação e autorização
- ASP.NET MVC Núcleo Testes de Automação

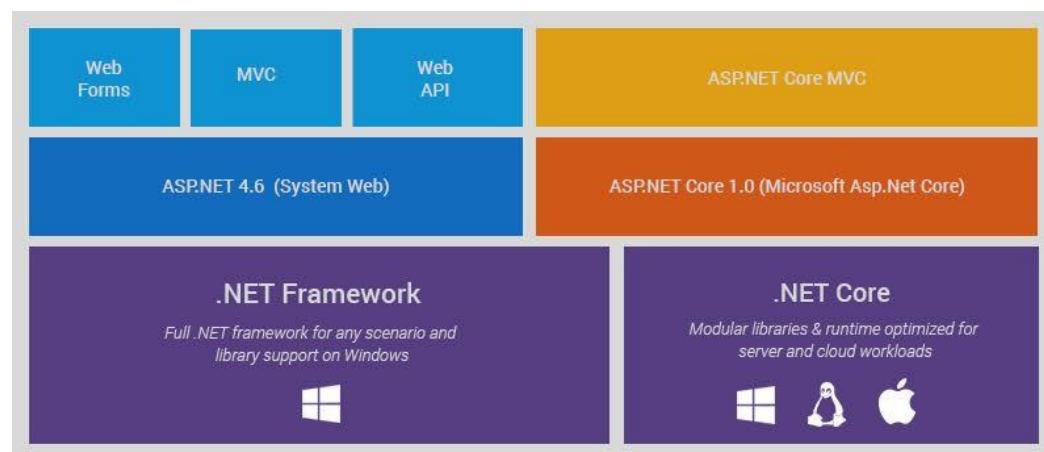
# BIBLIOGRAFIA E RECURSOS

- Adam Freeman, “Pro ASP.NET MVC Núcleo”, 6º edição, Apress de 2016
- <https://www.asp.net/>

## ASP.NET MVC NÚCLEO

- ASP.NET MVC Núcleo é um framework de desenvolvimento de aplicações web da Microsoft que combina a eficácia e arrumação de MVC (MVC) arquitetura, ideias e técnicas de desenvolvimento ágil, e as melhores partes da plataforma .NET.
- Ela enfatiza a arquitetura limpa, padrões de projeto, e capacidade de teste, e não tenta esconder a forma como a Web funciona.

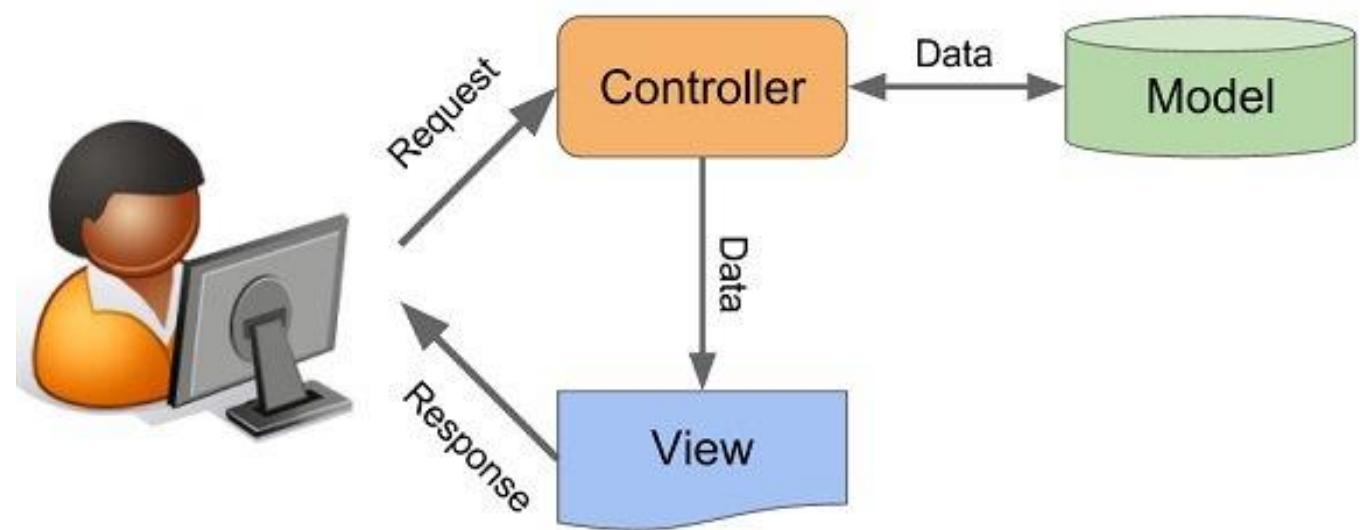
## ASP.NET MVC NÚCLEO



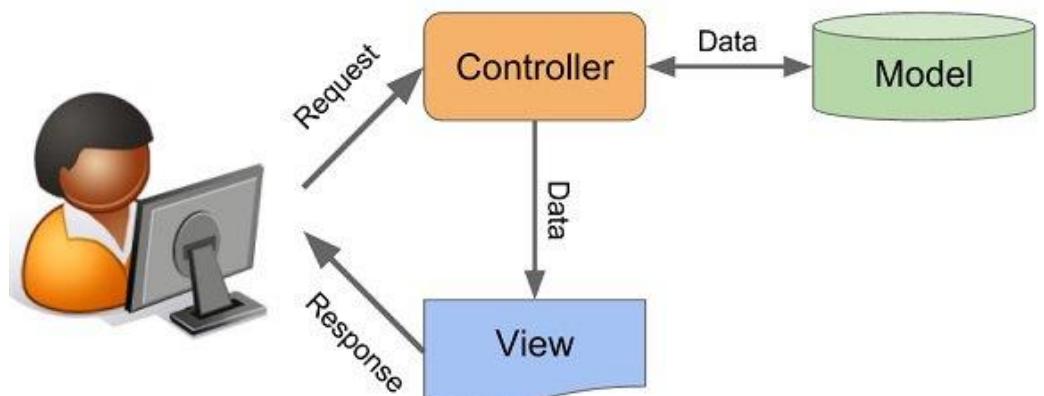
- ASP.NET núcleo é construído em .NET Core, que é uma versão multiplataforma do .NET Framework sem APIs específicas do Windows.
- aplicações web estão cada vez hospedado em recipientes pequenos e simples em plataformas de nuvem, e ao abraçar uma abordagem multi-plataforma Microsoft estendeu o alcance de .NET, possibilitando a implantação de aplicações ASP.NET núcleo ligado a um conjunto mais amplo de ambientes de hospedagem, e, como um bônus, tornou possível para os desenvolvedores criarem aplicações web ASP.NET core no Linux e OS X.

## MVC PADRÃO

- ASP.NET núcleo MVC segue um padrão chamado modelo de viewcontroller (MVC), que orienta a forma de um aplicativo ASP.NET e as interacções entre os componentes nele contidos.

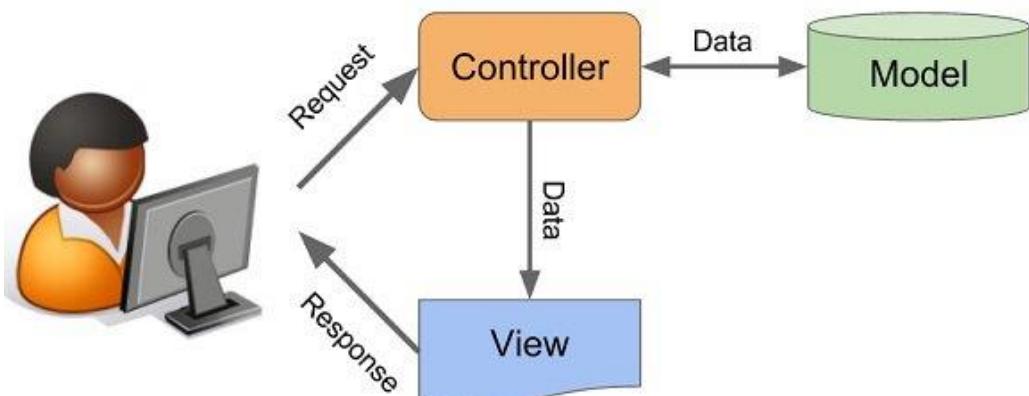


## MVC PADRÃO



- O padrão MVC remonta a 1978 e o projeto Smalltalk na Xerox PARC. Mas ganhou popularidade recentemente como um padrão para aplicações web, pelas seguintes razões:
  - A interação do usuário com um aplicativo que adere ao padrão MVC segue um ciclo natural: o usuário realiza uma ação, e em resposta a aplicação muda seu modelo de dados e fornece uma visão atualizada para o usuário. E, em seguida, o ciclo se repete.
  - Este é um ajuste conveniente para aplicações web entregues como uma série de solicitações e respostas HTTP.
  - aplicações web tornem indispensável combinar diversas tecnologias (bancos de dados, HTML e código executável, por exemplo), geralmente dividida em um conjunto de níveis ou camadas. Os padrões que surgem a partir destas combinações mapear naturalmente sobre os conceitos do padrão MVC.

## MVC PADRÃO



- O padrão de arquitetura MVC ajuda a conseguir a separação de preocupações, separando uma aplicação em três grupos principais de componentes: modelos, visualizações e controladores.
- Usando este padrão, as solicitações do usuário são encaminhadas para um controlador que é responsável por trabalhar com o modelo para executar ações do usuário e / ou recuperar resultados de consultas.
- O controlador escolhe o View para exibir para o usuário, e fornece-o com todos os dados modelo que necessita.

# 01

O padrão MVC ajuda a criar aplicativos que separam os diferentes aspectos do aplicativo (lógica de entrada, lógica de negócios e lógica UI), enquanto fornece um baixo acoplamento entre esses elementos.

# 02

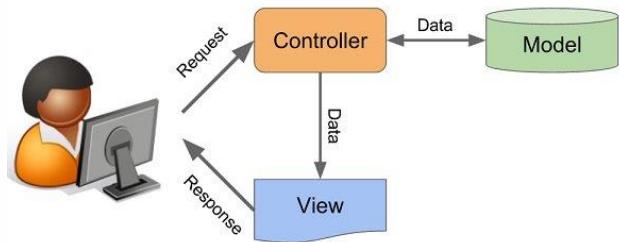
O padrão especifica onde cada tipo de lógica deve ser localizado no app:

- A lógica UI pertence a vista.
- Lógica de entrada pertence ao controlador.
- A lógica de negócios pertence no modelo.

# 03

Esta separação ajuda a gerenciar a complexidade quando você cria um aplicativo, porque ele permite que você trabalhe em um aspecto da implementação de cada vez, sem afetar o código de outro.

MVC PADRÃO



## MODELO

## RESPONSABILIDADES

O modelo em um aplicativo MVC representa o estado da aplicação e qualquer lógica de negócios ou operações que devem ser realizadas por ele.

A lógica de negócios deve ser encapsulado no modelo, junto com toda a lógica de implementação para persistir o estado da aplicação.

vistas fortemente tipados tipicamente usam tipos ViewModel especificamente concebidos para conter os dados para mostrar que em vista; o controlador irá criar e preencher essas instâncias de ViewModel do modelo.

## viewmodel

---

Em um aplicativo web MVC, um ViewModel é um tipo que inclui apenas os dados a View requer para exibição (e talvez para enviar de volta para o servidor).

---

tipos de ViewModel também pode simplificar modelo obrigatório em ASP.NET MVC. tipos ViewModel são geralmente apenas recipientes de dados; qualquer lógica eles podem ter deve ser específico para ajudar o Ver processar dados.

---

# 1

Visualizações são responsáveis por apresentar conteúdo através da interface do usuário.

# 2

Eles usam o mecanismo de exibição Navalha para incorporar código .NET na marcação HTML.

# 3

Deve haver lógica mínima dentro de pontos de vista, e qualquer lógica neles devem se relacionar com apresentação de conteúdo.

# 4

Um modelo de vista nunca deve executar lógica de negócios ou interagir com um banco de dados diretamente. Em vez disso, um modelo de visão deve funcionar apenas com os dados que é fornecido a ele pelo controlador.

## RESPONSABILIDADES VISTA

# RESPONSABILIDADES CONTROLADOR

- Controladores de lidar com interação do usuário, trabalhar com o modelo e, finalmente, selecionar uma exibição para renderizar.
- Controladores são responsáveis por fornecer todos os dados ou objetos são necessários para um modelo de visão para renderizar uma resposta para o navegador.
- Em um aplicativo MVC, a visão só exibe informações; o controlador processa e responde à entrada do usuário e interação. Por exemplo, o controlador manipula dados de rota e valores de consulta cordas, e passa esses valores para o modelo. O modelo pode usar esses valores para consultar o banco de dados.
- O controlador é o ponto de entrada inicial, e é responsável por selecionar quais tipos de modelo para trabalhar e qual visualização render (daí o seu nome - que controla como o aplicativo responde a um determinado pedido).

# 01

Controladores não devem ser excessivamente complicado por muitas responsabilidades. Para manter lógica do controlador de tornar-se excessivamente complexa, use o princípio da responsabilidade única para empurrar a lógica de negócios para fora do controlador e para o modelo de domínio.

# 02

Se você achar que suas ações do controlador frequentemente executar os mesmos tipos de ações, você pode seguir o princípio Não Repeat Yourself movendo essas ações ordinárias em filtros.

## CONTROLADORES

# MVC framework web

## Modelo

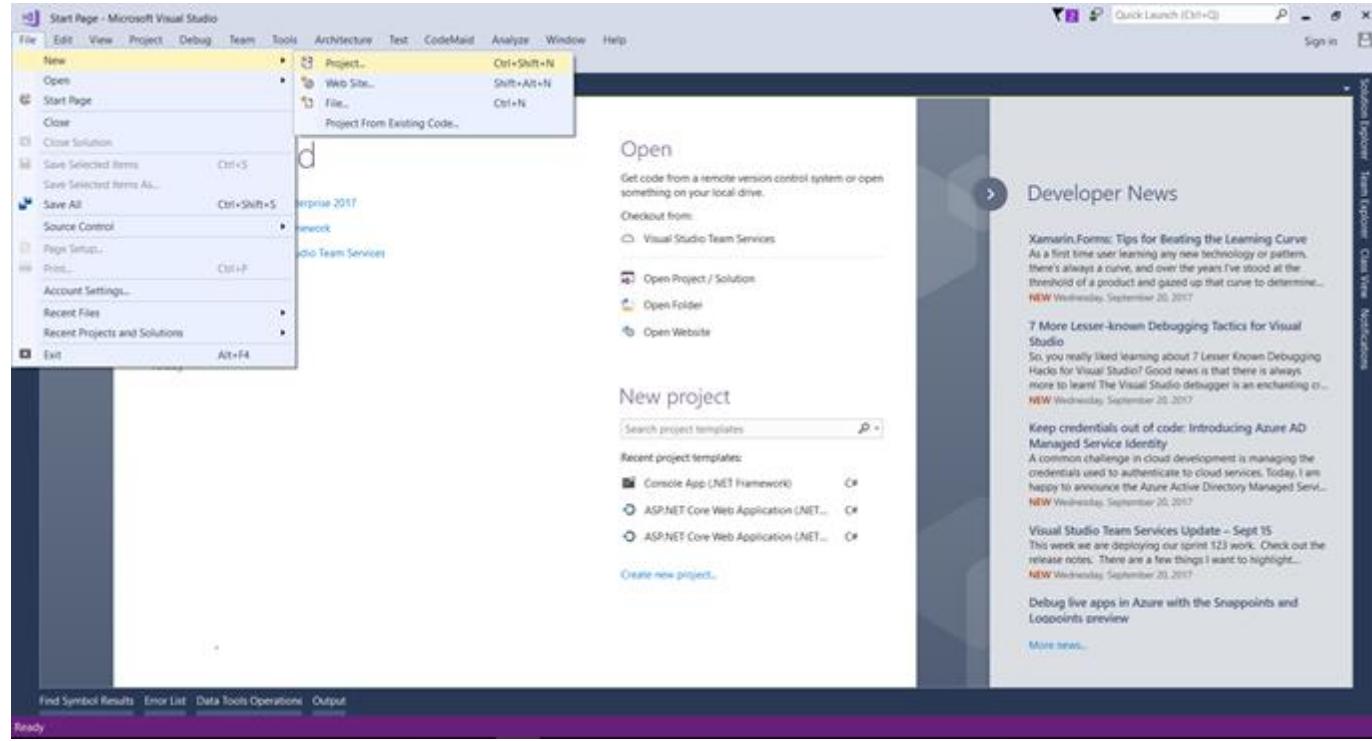
- Data Access Layer (por exemplo, usando uma ferramenta como o Entity Framework ou Nhibernate)

## Visão

- Este é um modelo para gerar dinamicamente HTML.

## Controlador

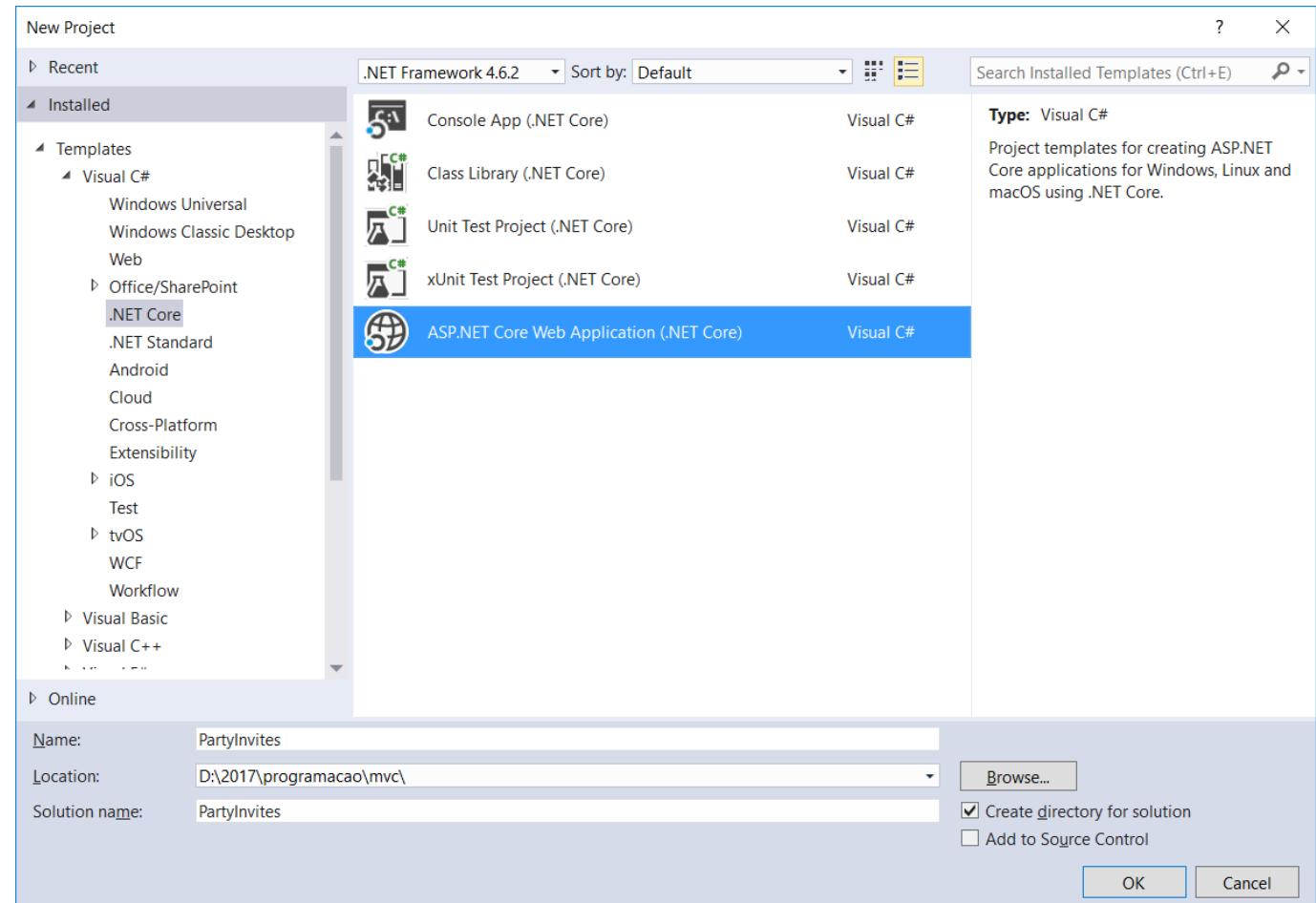
- Gerencia a relação entre a vista eo modelo. Ele responde a entrada do usuário, fala com o modelo, e decide que ver para renderizar



# Criar um novo projeto ASP.NET MVC NÚCLEO DE USAR VISUAL STUDIO

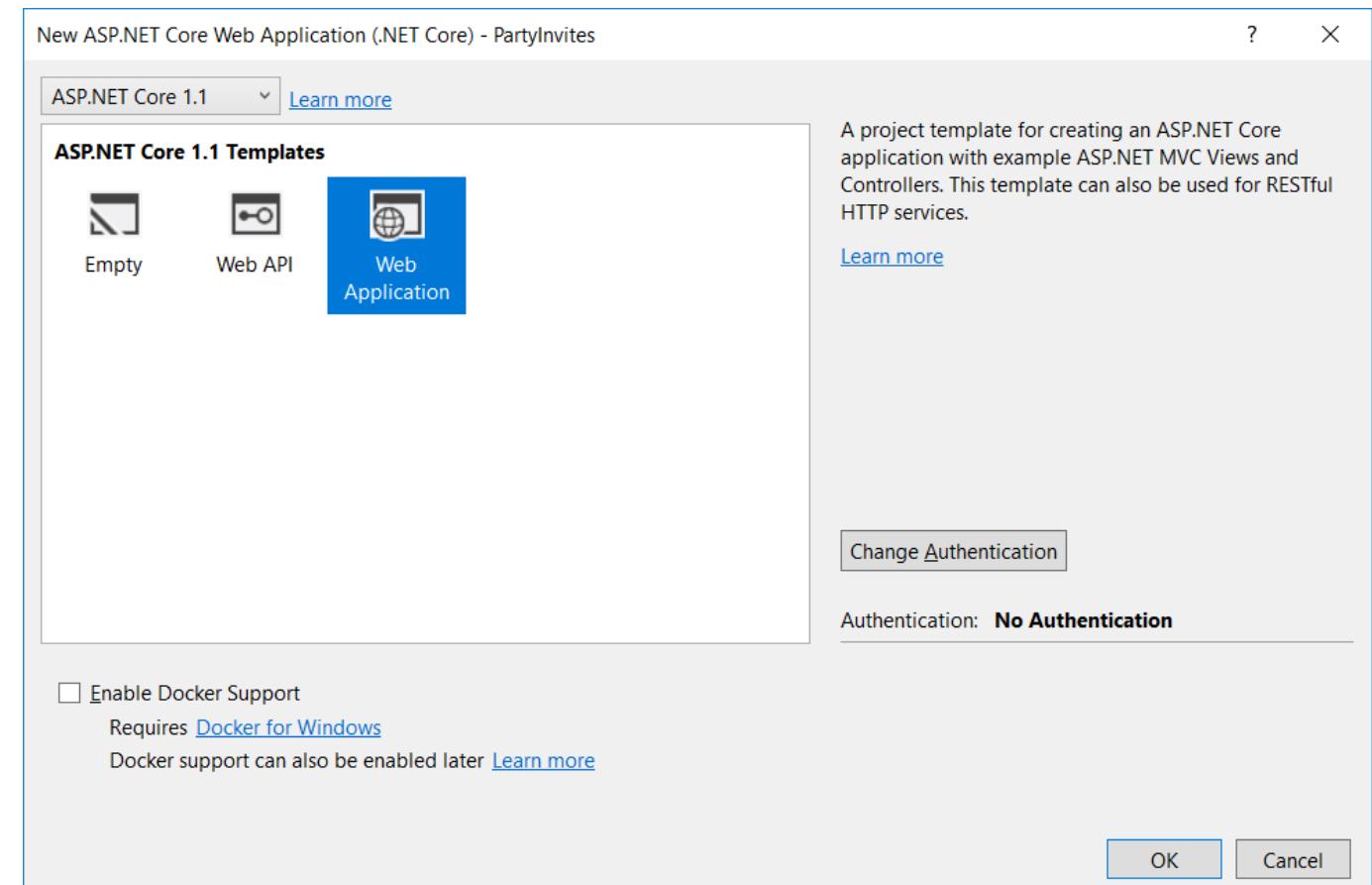
# Criar um novo projeto ASP.NET MVC

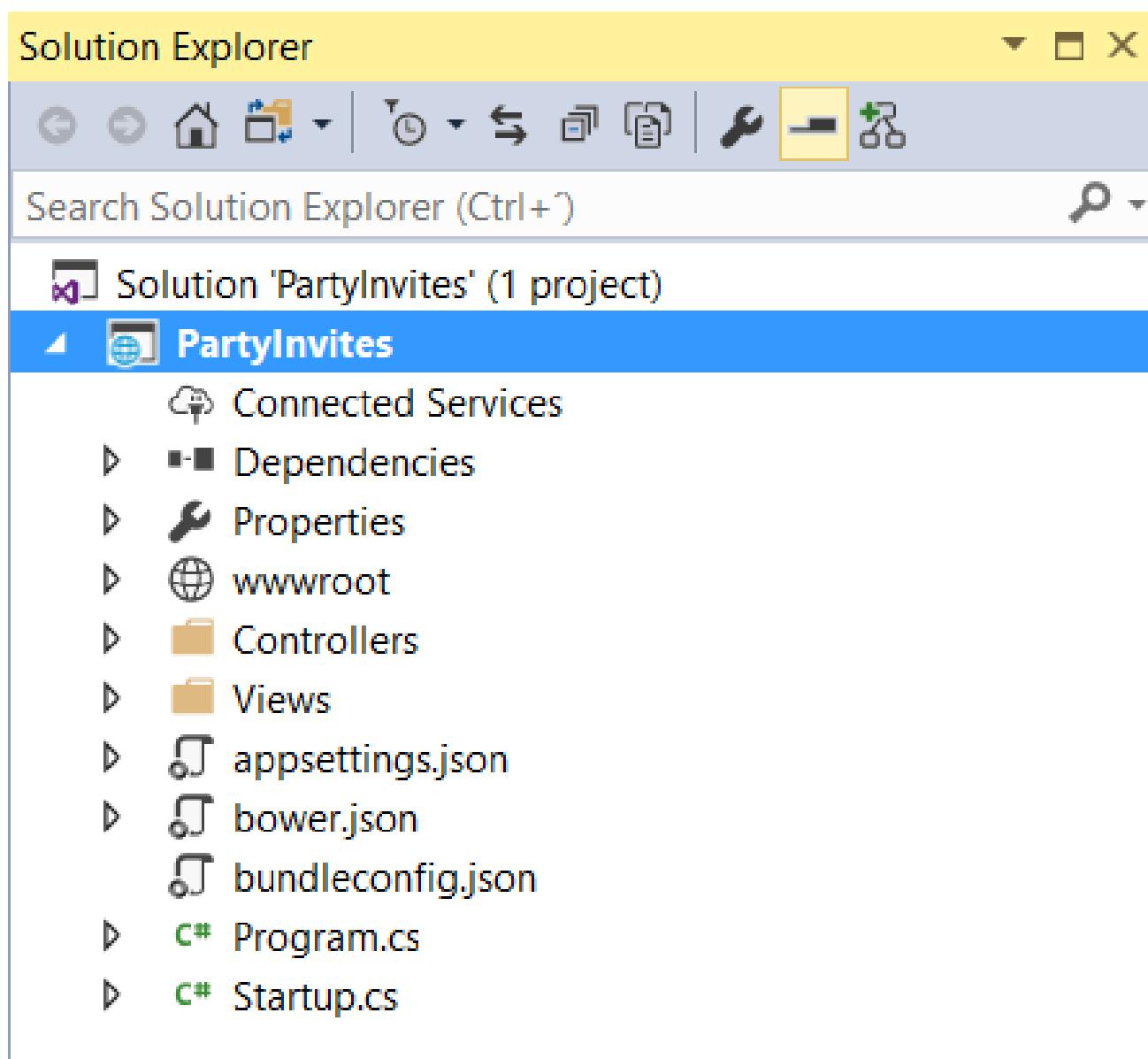
## NÚCLEO DE USAR VISUAL STUDIO



# Criar um novo projeto ASP.NET MVC

## NÚCLEO DE USAR VISUAL STUDIO





EXPLORER SOLUÇÃO

## ESTRUTURA DE APLICAÇÃO MVC

DIRETÓRIO	FINALIDADE
/ Modelos	Dados (de negócios) objetos
/ Visualizações	arquivos de modelo de UI para renderização de saída (por exemplo, HTML)
/ Controllers	classes do controlador que lidam com pedidos de URL
/ Views / Shared	Layouts e pontos de vista que não são específicos para um único controlador.
/Views/_ViewImports.cshtml	Contém os namespaces que serão incluídos em arquivos de visão Navalha.
/Views/_ViewStart.cshtml	layout padrão para o mecanismo de exibição Navalha.
/ wwwroot	O conteúdo estático (por exemplo CSS, imagem, arquivos javascript)
/ Wwwwroot / lib	Terceiros JavaScript e CSS pacotes

## ESTRUTURA DE APLICAÇÃO MVC

DIRETÓRIO	FINALIDADE
/ áreas	Áreas são uma forma de particionar um aplicativo grande em pedaços menores.
/ Dependências	Fornece detalhes de todos os pacotes de um projeto depende.
/ Componentes	Este é o local onde as classes de componentes vista, os quais são utilizados para exibir características de auto-contido, como carrinhos de compras, são definidos.
/Dados	Este é o local onde as classes de contexto do banco de dados são definidos
/ Migrações	Este é onde os detalhes de esquemas de banco de dados são armazenados de forma que os bancos de dados podem ser atualizados.
/bower.json	Lista de pacotes gerenciados pelo gerenciador de pacotes Bower. Este arquivo está oculta por padrão.
/project.json	opções de configuração básicas para o projeto, incluindo os pacotes NuGet ele usa
/Program.cs	Esta classe configura a plataforma de hospedagem para a aplicação
/Startup.cs	Esta classe configura a aplicação

[PartyInvites](#)[Home](#)[About](#)[Contact](#)

# Executando o aplicativo

ASP.NET Core

Windows

Linux

OSX



Learn how to build ASP.NET apps that can run anywhere.

[Learn More](#)



## Application uses

- Sample pages using ASP.NET Core MVC
- Bower for managing client-side libraries
- Theming using Bootstrap

## How to

- Add a Controller and View
- Manage User Secrets using Secret Manager.
- Use logging to log a message.
- Add packages using NuGet.
- Add client packages using Bower.
- Target development, staging or production environment.

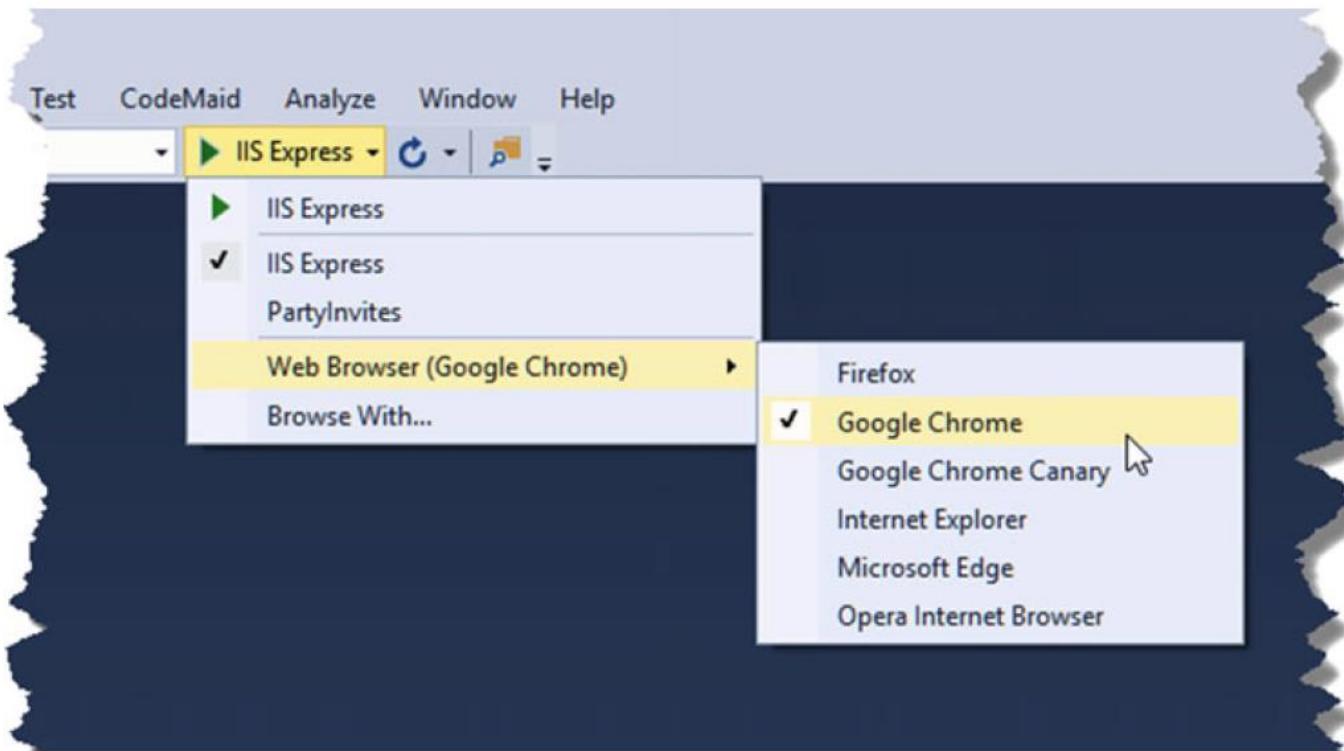
## Overview

- Conceptual overview of what is ASP.NET Core
- Fundamentals of ASP.NET Core such as Startup and middleware.
- Working with Data
- Security
- Client side development
- Develop on different platforms
- Read more on the documentation site

## Run & Deploy

- Run your app
- Run tools such as EF migrations and more
- Publish to Microsoft Azure Web Apps

# ALTERAR O NAVEGADOR



## CONTROLADORES

solicitações de entrada são tratadas pelos controladores. No ASP.NET núcleo MVC, os controladores são apenas C # classes (geralmente herdados da classe Microsoft.AspNetCore.Mvc.Controller, que é incorporada no MVC classe base controlador).

classes do controlador manipular a comunicação do usuário, o fluxo geral do aplicativo e lógica específica da aplicação. Cada método público em um controlador é exigível como um ponto final HTTP. Os métodos dentro de um controlador são chamados de ações do controlador. Seu trabalho é para responder a solicitações de URL executar as ações apropriadas, e retornar uma resposta de volta para o navegador ou usuário que invocou a URL.

A convenção MVC é colocar controladores na pasta controladores, que Visual Studio criado quando se configurar o projeto.

## Convenção sobre configuração

***“Sabemos que, por agora, como construir uma aplicação web. Vamos rolar essa experiência para a armação, não temos a configurar absolutamente tudo novamente “.***

---

nome da classe de cada controlador termina com Controller (eg ProductController, HomeController) no diretório controladores.

---

Visualizações que os controladores usam ao vivo em um subdiretório do diretório principal vistas e são nomeados de acordo com o nome do controlador (por exemplo, a vista para o ProductController é / Views / Product)

---

## CONTROLADORES

```
classe pública HomeController : Controlador {  
    público IActionResult Index () {  
        Retorna Visão(); }  
  
    público IActionResult Contato() {  
        Ver dados[ "Mensagem "] = "Contatos ..." ;  
        Retorna Visão(); }}
```

Solution Explorer

Search Solution Explorer (Ctrl+)

Solution 'PartyInvites' (1 project)

PartyInvites

- Connected Services
- Dependencies
- Properties
- wwwroot

View in Browser (Google Chrome) Ctrl+Shift+W

Browse With...

Cleanup Selected Code

Collapse Recursively

Add

- Controller...
- New Item... Ctrl+Shift+A
- Existing Item... Shift+Alt+A
- New Scaffolded Item...
- New Folder
- Docker Support
- Class...

Scope to This

New Solution Explorer View

Exclude From Project

Cut Ctrl+X

Copy Ctrl+C

Delete Del

Rename

Open Folder in File Explorer

Properties Alt+Enter

# ADICIONAR UM CONTROLADOR

## Add MVC Dependencies



### **Minimal Dependencies**

Adds the minimal packages and references to start using ASP.NET MVC Core.

### **Full Dependencies**

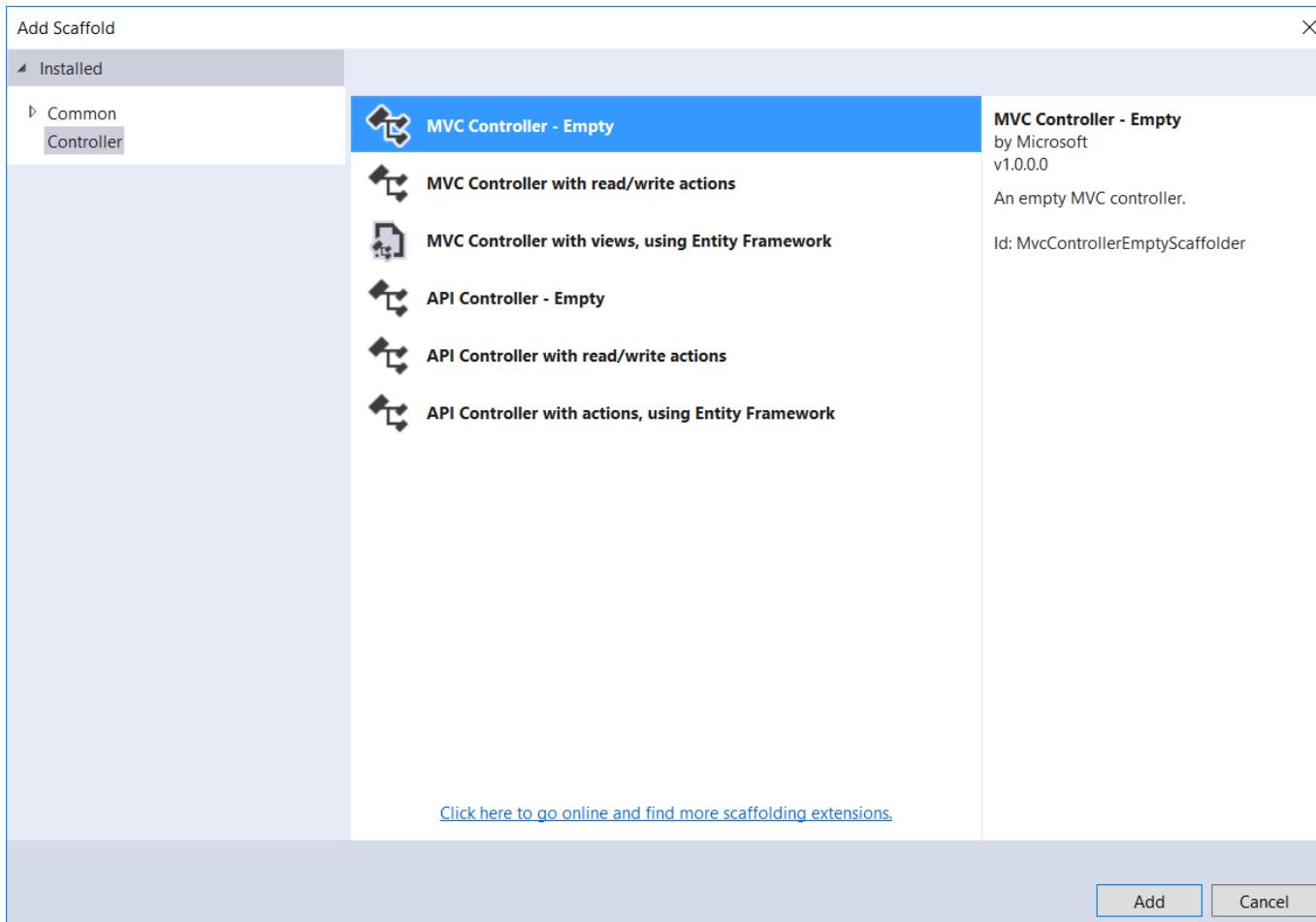
Adds packages and configurations, as well as a default layout, error page, script libraries, and script bundling to your application.

Add

Cancel

ADICIONAR UM CONTROLADOR

# ADICIONAR UM CONTROLADOR



Add Controller X

Controller name:

Add Cancel

ADICIONAR UM CONTROLADOR

## CONTROLADOR DE AÇÕES

```
classe pública HelloWorldController : Controlador {  
    //  
    // GET: / HelloWorld /  
    public string Índice () {  
        Retorna "Esta é a minha ação padrão ..."; }  
  
    //  
    // GET: / HelloWorld / Bem-vindo /  
    public string Bem vinda () {  
        Retorna "Método de ação Bem-vindo ..."; }}
```

## ROTAS COMPREENSÃO

- aplicações MVC usar o ASP.NET *sistema de roteamento*, que decide como URLs são mapeadas para controladores e ações. Uma rota é uma regra que é usada para decidir como um pedido é tratado.
- O formato para encaminhamento é definida no Startup.cs Arquivo.

```
app.UseMvc (rotas => {
    routes.MapRoute (
        nome: "padrão" , Modelo: "{ controller = Início} / {action = Índice} / {id}?" );
});
```

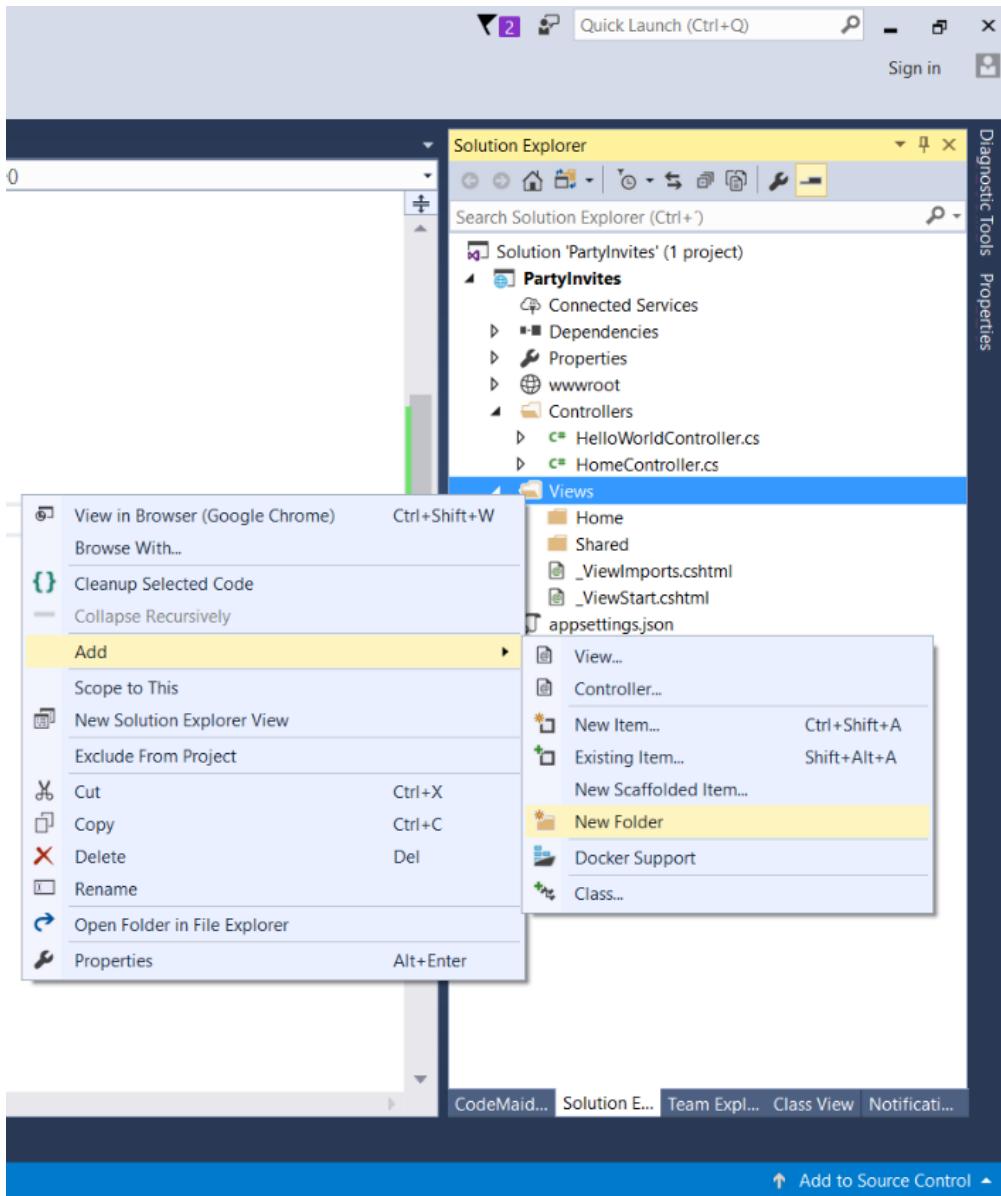
## VISTAS

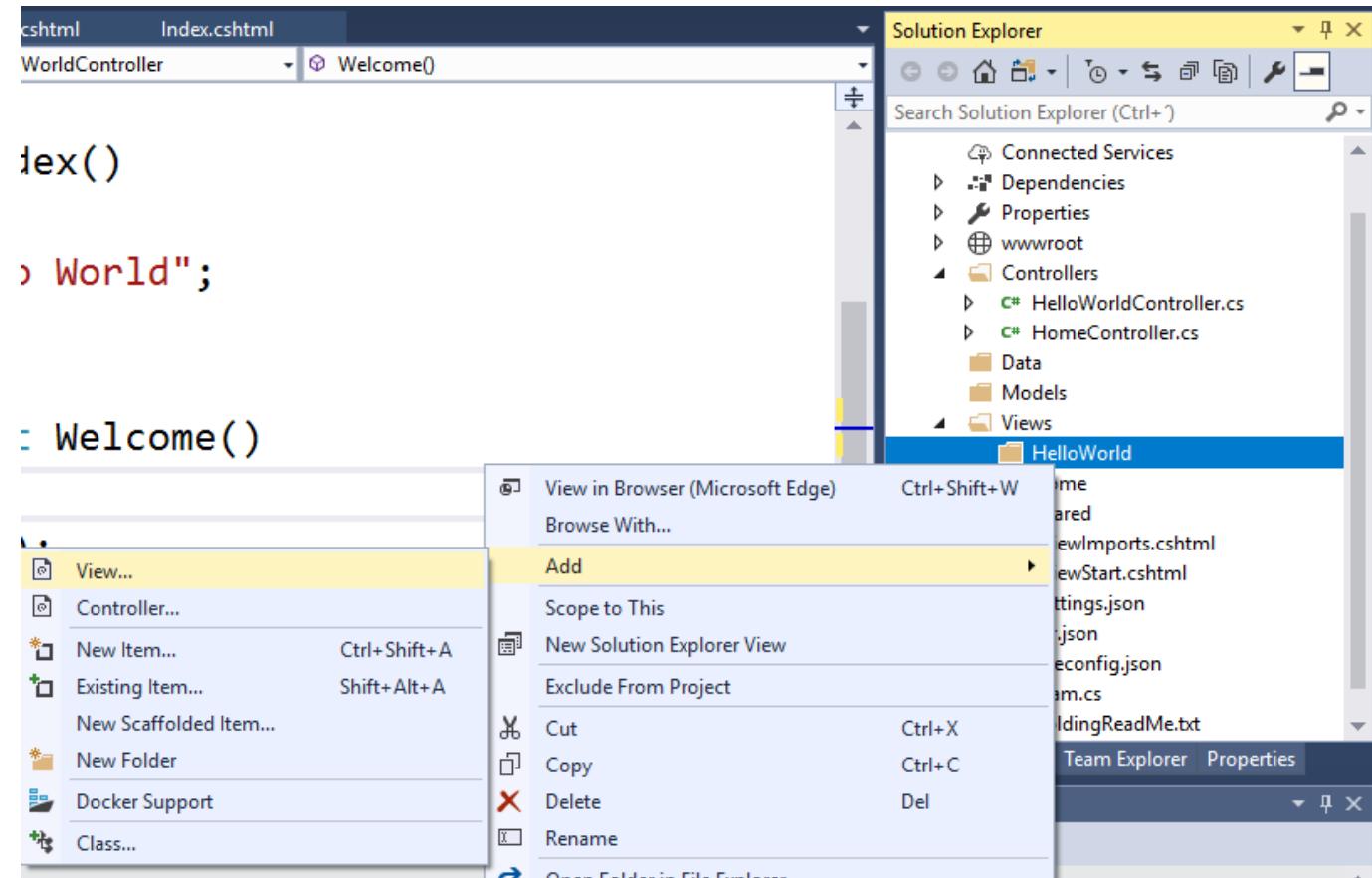
- A vista é responsável por fornecer a interface de usuário (UI) para o usuário. Depois que o controlador tenha executado a lógica apropriada para o URL solicitado, ele delega a visualização para a vista.
- Visualizações são armazenados na pasta Views, organizados em subpastas. Visualizações que estão associados com o controlador inicial, por exemplo, são armazenados em uma pasta chamada Views / Home. Vistas que não são específicos para um único controlador são armazenados em uma pasta chamada Views / Shared.
- Visual Studio cria a Casa e pastas partilhadas automaticamente quando o modelo Aplicativo Web é usado e coloca em alguns pontos de vista de espaço reservado para começar o projeto começou.

```
público ViewResult Bem vinda() {  
    Retorna Visão(); }
```

MODIFICAÇÃO o  
controlador para  
renderizar uma visão

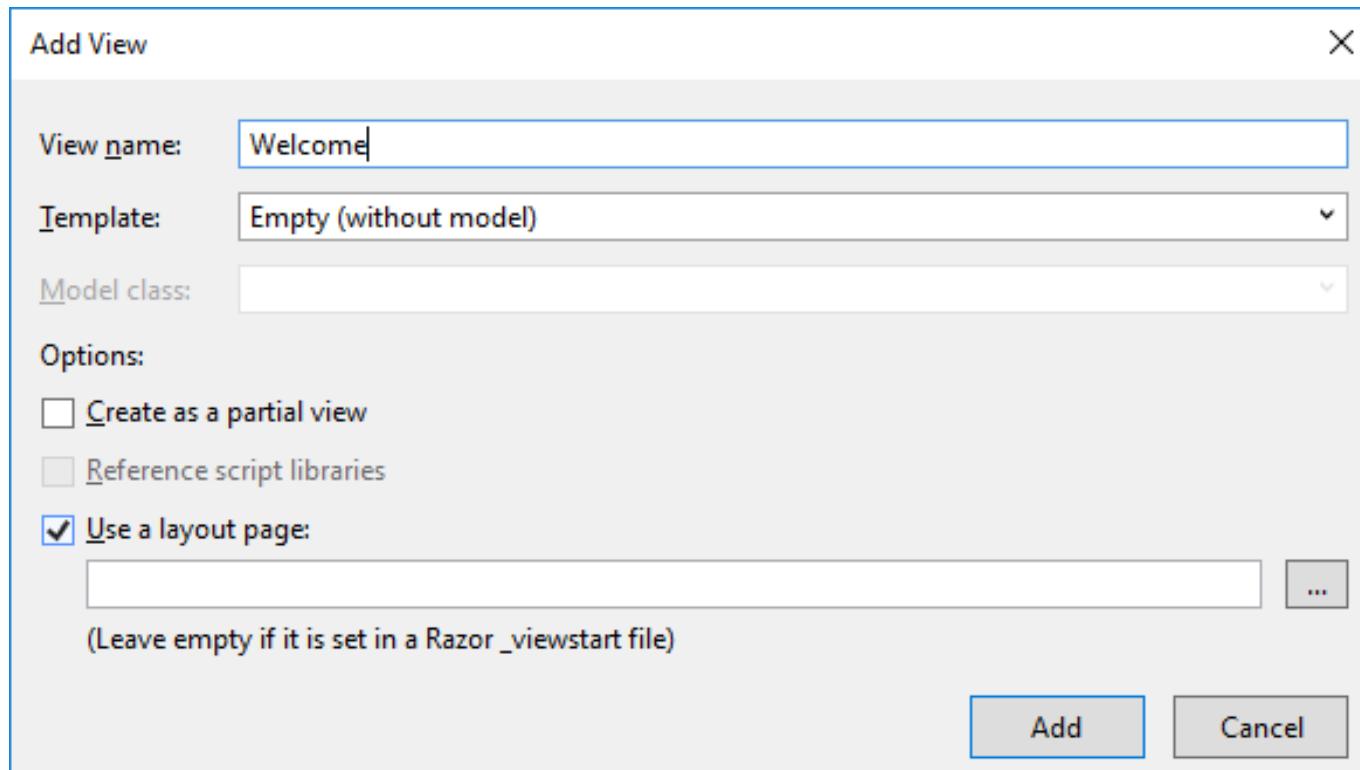
## Adicionando uma exibição





## Criando uma View

# Criando uma View



## VISTAS

MVC utiliza a convenção de nomenclatura para encontrar a exibição automaticamente. A convenção é que a visão tem o nome do método de ação e está contido em uma pasta com o nome do controlador.

```
@ {
```

```
    Ver dados[ "Título" ] = "Bem vinda" ; }
```

```
< h2 > Bem vinda </ h2 >
```

```
< p > Olá da ação Bem-vindo (Olá Mundo Controller) </ p >
```

# Os métodos de ação Tipos de retorno

- Além das e ViewResult métodos de ação objetos podem retornar outros resultados. Por exemplo, se o método retorna uma RedirectResult, o navegador será redirecionado para outro URL. Se ele retorna um HttpUnauthorizedResult, I forçar o utilizador ligar-se. Estes objectos são colectivamente conhecidas como resultados de acção. O sistema de resultado de ação permite encapsular e reutilizar respostas comuns em ações.

## adicionar Dynamic SAÍDA

- O ponto inteiro de uma plataforma de aplicações web é construir e exibição *dinâmico* saída. No MVC, é o trabalho do controlador para construir alguns dados e passá-lo para a vista, que é responsável por renderizar para HTML.
- Um modo para transmitir dados a partir do controlador para a vista é usando o ViewBag objeto, que é um membro da classe de base controlador. ViewBag é um objeto dinâmico para o qual você pode atribuir propriedades arbitrárias, tornando esses valores disponíveis em qualquer ponto de vista é posteriormente processado.

## Passando dados do controlador para THE VIEW

Os dados para a vista é fornecido quando o ViewBag.Message propriedade é atribuído um valor. o mensagem propriedade não existia até o momento em que foi atribuído um valor. Isto permite que a passagem de dados a partir do controlador para a vista de uma maneira livre e fluido, sem a necessidade de definir classes antes do tempo.

```
público IActionResult Index () {  
    int hora = Data hora .Now.Hour;  
  
    corda mensagem;  
  
    E se (H> = 7 && horas <12) {  
        message = "Bom Dia" ; } else if (Hora> = 12 && hora  
<20) {  
        message = "Boa tarde" ; } outro {  
  
        message = "Boa noite" ; }  
  
    ViewBag.Message = mensagem;  
  
    Retorna Visão(); }
```

```
@{
```

```
Ver dados[ "Título" ] = "Casa" ; }
```

```
< h2 > Casa </ h2 > < p > @ ViewBag.Message mundo !!! </ p > < p > Nós  
vamos ter uma festa emocionante. </ p >
```

RECUPERAR UM  
VALOR VIEWBAG  
dados na exibição

# mODELOS

- Modelos representar o domínio da aplicação se concentra. Eles descrevem os dados, bem como as regras de negócio para a forma como os dados podem ser alterados e manipulados.
- *o modelo* é a parte mais importante da aplicação. É uma representação dos objetos do mundo real, processos e regras que definem o assunto, conhecido como o *domínio*, da aplicação. O modelo, muitas vezes referida como um *modelo de domínio*, contém os objectos C # (conhecido como *objectos de domínio*) que compõem o universo da aplicação e os métodos que os manipulam.
- Os pontos de vista e controladores de expor o domínio para os clientes de um modo consistente, e de uma aplicação MVC bem concebido começa com um modelo bem concebido, que é, em seguida, o ponto focal como controladores e as vistas são adicionados.
- A convenção MVC é que as classes que compõem um modelo são colocados dentro de uma pasta chamada modelos.

```
namespace PartyInvites.Models {  
  
    classe pública GuestResponse {  
  
        public string nome { obter ; conjunto ; }  
  
        public string Telefone { obter ; conjunto ; }  
  
        public string O email { obter ; conjunto ; }  
  
        bool pública ? WillAttend { obter ; conjunto ; }}}
```

mODELOS

```
namespace PartyInvites.Controllers {  
  
    classe pública HomeController : Controlador {  
  
        // ...  
  
        público ViewResult Responda Por Favor() {  
  
            Retorna Visão(); } } }
```

Répondez dobra vous  
do s'il (RSVP)  
CONTROLADOR

## *VISTA fortemente tipada*

Uma visão fortemente tipificada se destina a prestar  
um tipo de modelo específico

PartyInvites.Models @model. GuestResponse

@{

Ver dados[ "Título" ] = "Répondez S'il Vous Plait" ; }

< h2 > Répondez S'il Vous Plait </ h2 >

```
@{
```

```
    Ver dados[ "Título" ] = "Casa" ; }
```

```
< h2 > Casa </ h2 >
```

```
< p > @ ViewBag.Message mundo !!! </ p >
```

```
< p > Nós vamos ter uma festa emocionante. </ p >
```

```
< um asp-action =" RSVP " > Répondez S'il Vous Plait </ um >
```

LIGAÇÃO  
métodos de  
ação

# LIGAÇÃO métodos de ação

```
@{  
    Ver dados[ "Título" ] = "Casa" ; }
```

```
< h2 > Casa </ h2 >
```

```
< p > @ ViewBag.Message mundo !!! </ p >
```

```
< p > Nós vamos ter uma festa emocionante. </ p >
```

```
< um asp-action =" RSVP " > Répondez S'il Vous Plait </ um >
```

- o atributo `asp-action` é um *tag helper* atributo, que é uma instrução para Navalha que será executada quando a vista é processado.
- A `href` atribuir ao elemento que contém uma URL para um método de ação será processado.
- Há um princípio importante no trabalho aqui, que é que você deve usar os recursos fornecidos pelo MVC para gerar URLs, ao invés de hard-code-los em seus pontos de vista. Quando o ajudante tag criou o `href` atributo do elemento, ele inspecionou a configuração do aplicativo para descobrir o que o URL deve ser. Isso permite que a configuração do aplicativo a ser alterado para suportar diferentes formatos de URL sem a necessidade de atualizar quaisquer pontos de vista.

## Criando uma View FORM

o `asp-action` atributo usa configuração de roteamento de URL do aplicativo para definir o atributo de ação para uma URL que terá como alvo um método de ação específico.

```
< forma asp-ação =" Responda Por Favor" método = "Post">
<! - ... ->
</ Formato >
```

## Criando uma View FORM

Cada elemento é associado com a propriedade modelo usando o **asp-for** atributo, que é outro atributo tag helper. o **asp-for** atributo no elemento **label** define o valor do atributo **for**. o **asp-for** atributo no elemento de entrada define os elementos de identificação e nome.

```
< forma asp-ação =" Responda Por Favor" método = "Post">
< mesa > < tr > < td > < etiqueta asp-for =" Nome "> Nome: </ rótulo >

</ td > < td > < entrada para asp- =" Nome "/> </ td
>

</ tr >

<! - ... ->

</ mesa >

</ Formato >
```

## Criando uma View FORM

```
< forma asp-ação =" Responda Por Favor" método = "Post">
< mesa >
<! - ... ->

< tr > < td > < etiqueta asp-for =" WillAttend " > eu vou </ rótulo >

</ td > < td > < selecione asp-for =" WillAttend " >

< valor da opção = ""> Escolha uma opção </ opção >
< valor da opção = " true "> participar da festa </ opção >
< valor da opção = " false "> não ir a festa </ opção >
</ selecionar >
</ td >
</ tr >

<! - ... ->
</ mesa >
</ Formato >
```

## Criando uma View FORM

```
< forma asp-ação =" Responda Por Favor" método = "Post">
  < mesa >
    <! - ... ->
    < tr > < td > < botão tipo = "Submit">
      Enviar resposta
      </ botão >
      </ td > < td > </ td
      >
      </ tr >

    </ mesa >
</ Formato >
```

# Solicitações GET e POST

aplicações Web geralmente usam solicitações GET para leituras e solicitações POST para as gravações (que normalmente incluem atualizações, cria e elimina).

Um pedido GET representa uma operação independente só de leitura. Você pode enviar uma solicitação GET para um servidor repetidamente sem efeitos nocivos, porque um GET não devem mudar de estado no servidor. Além disso, você pode marcar a solicitação GET, porque todos os parâmetros estão na URL (assim, os valores de entrada forma são preservados). Um pedido GET é o que um navegador emite normalmente cada vez que alguém clica em um link.

Realizar uma operação de criar, apagar ou editar em resposta a um pedido GET (ou para essa matéria, qualquer outra operação que muda de dados) abre uma brecha de segurança. Um pedido POST geralmente modifica o estado no servidor, e repetindo o pedido pode produzir efeitos indesejáveis (*por exemplo faturamento duplo*).

```
classe pública HomeController : Controlador {  
  
    // ...  
  
    [HttpGet]  
    público ViewResult Responda Por Favor() {  
        Retorna Visão(); }  
  
    [HttpPost]  
    público ViewResult VPSD ( GuestResponse resposta) {  
        // TODO: Resposta convidado loja  
        Retorna Visão(); }  
  
    // ...  
}
```

GET VS POST

# modelo de ligação

- *vinculativo modelo* é uma característica útil MVC no qual os dados de entrada é analisado e os pares de chave / valor no pedido de HTTP são utilizados para preencher as propriedades de tipos de modelo de domínio. Ele elimina a moagem e do trabalho de lidar com solicitações HTTP diretamente e permite trabalhar com C # objetos em vez de lidar com valores de dados individuais enviadas pelo navegador.
- Modelo de ligação libertar-nos da tarefa tediosa e propensa a erros de ter que inspecionar uma solicitação HTTP e extrair todos os valores de dados que são necessários.

## EXEMPLO ACADEMIC

Por agora, vamos armazenar dados em uma coleção na memória de objetos. Isso não é útil em uma verdadeira aplicação, porque os dados de resposta serão perdidos quando o aplicativo é parado ou reiniciado

```
// Nunca faça isso !!!  
// Este é apenas para demonstração / fins acadêmicos  
  
classe pública Repositório {  
    private static Lista < GuestResponse > Respostas =  
        Novo Lista < GuestResponse > ();  
  
    public static IEnumerable < GuestResponse > Respostas {  
        obter {  
            Retorna respostas; }  
  
        public static void AddResponse ( GuestResponse resposta) {  
            responses.Add (resposta); }  
    }  
}
```

```
namespace PartyInvites.Controllers {  
    classe pública HomeController : Controlador {  
        // ...  
  
        [HttpPost]  
        público ViewResult Responda Por Favor( GuestResponse resposta ) {  
            Repositório .AddResponse (resposta);  
  
            Retorna Visão( "Obrigado" , resposta);  
        }  
  
        // ...  
  
    }  
}
```

AÇÃO de RSVP  
POST

o Thanks.cshtml exibição usa Navalha para exibir conteúdo com base no valor do GuestResponse

Propriedades que eu passados para o método de exibição na Responda Por Favor método de acção. A navalha @ modelo expressão específica do tipo de modelo de domínio com a qual a vista é fortemente tipado.

Para acessar o valor de uma propriedade no objeto de domínio, use Model.PropertyName. Por exemplo, para obter o valor do Nome propriedade, chamada Model.Name.

PartyInvites.Models @model. GuestResponse

@ {

Ver dados[ "Título" ] = "Obrigado" ; }

< h2 > Obrigado, @ Model.Name !!! </ h2 >

@ E se (Model.WillAttend == nulo ) {

:@: Obrigado pela sua resposta. Quando você decidir se você pode vir para a festa, dá-me uma chamada. } else if (Model.WillAttend == verdade ) {

:@: Obrigado pela sua resposta. Quando você decidir se você pode vir para a festa, dá-me uma chamada. } outro { // Model.WillAttend == false

:@: Lamento saber que você não pode fazê-lo, mas os tanques por nos informar. }

```
namespace PartyInvites.Controllers {  
    classe pública HomeController : Controlador {  
  
        // ...  
  
        público ViewResult Lista de convidados() {  
            Retorna Visão( Repositório .Responses);  
        }  
  
        // ...  
    }}
```

## AÇÃO GUESTLIST

```
@modelo IEnumerable <PartyInvites.Models. GuestResponse >
```

```
@{  
    Ver dados[ "Título" ] = "Lista de convidados" ; }
```

```
< h2 > Lista de convidados </ h2 >
```

```
< mesa > < thead > < tr > < td > Nome </ td > < td > telefone </ td >  
    < td > O email </ td > < td > Participará  
    </ td >
```

```
    </ tr >  
</ thead > < tbody >  
    <! - ... ->  
</ tbody >  
</ mesa >
```

VISTA GUESTLIST

```
para cada ( var r dentro Modelo) {  
    < tr > < td > @ r.Name </ td > < td > @  
        r.Phone </ td > < td > @  
        r.Email </ td > < td >  
  
        @ E se (R.WillAttend == verdade ) { @: Sim } else if (R.WillAttend  
        == falso ) { @: Não } outro {  
  
            @: Não sei}  
  
    </ td >  
}</ tr >  
}
```

## VISTA GUESTLIST

```
@modelo IEnumerable <PartyInvites.Models. GuestResponse >
```

```
@{  
    Ver dados[ "Título" ] = "Lista de convidados" ; }  
  
< h2 > Lista de convidados </ h2 >  
  
@ E se (Model.Count () == 0) {  
    < p >  
        Ninguém ainda confirmar que ele / ela virá para a festa. Seja o primeiro a  
fazer isso e eu vou oferecer-lhe uma bebida especial.  
  
    </ p > < um asp-action =" RSVP " > Répondez S'il Vous Plait < uma >  
  
} outro {  
    <! - ... ->  
}
```

## VISTA GUESTLIST

## GUESTLIST CONTROLADOR

```
namespace PartyInvites.Controllers {
    classe pública HomeController : Controlador {
        // ...

        público ViewResult Lista de convidados() {
            Retorna Visão( Repositório .Responses.Where (r => r.WillAttend == verdade );
        }

        // ...
    }
}
```

## ADIÇÃO VALIDAÇÕES

01

Sem validação, os usuários podem inserir dados sem sentido ou até mesmo enviar um formulário vazio. Em um aplicativo MVC, você normalmente irá aplicar a validação para o modelo de domínio em vez de na interface do usuário. Isso significa que você definir a validação em um único lugar, mas ela tem efeito em qualquer lugar do aplicativo que a classe modelo é usado.

02

MVC suporta regras de validação declarativos definidos com atributos do namespace System.ComponentModel.DataAnnotations, o que significa que restrições de validação são expressas usando o padrão C # atributo características.

03

MVC detecta automaticamente os atributos e os usa para validar os dados durante o processo de ligação modelo.

## ADIÇÃO VALIDAÇÕES

```
utilização System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {
    classe pública GuestResponse {
        [ Requeridos (ErrorMessage = "Por favor, insira seu nome" )]
        public string nome { obter ; conjunto ; }

        [ Requeridos (ErrorMessage = "Por favor, digite seu número de telefone" )]
        public string Telefone { obter ; conjunto ; }

        [ Requeridos (ErrorMessage = "Por favor, indique o seu endereço de e-mail" )] [ Expressão regular ( "+\n        \\@. + \\.. +" , ErrorMessage = "Por favor insira um endereço de e-mail válido" )]
        public string O email { obter ; conjunto ; }

        [ Requeridos (ErrorMessage = "Por favor, especifique se irá comparecer" )]
        bool pública ? WillAttend { obter ; conjunto ; }}}
```

## Adicionar validação

Se o ModelState.IsValid

propriedade retorna falso, então eu sei que existem erros de validação. O objecto devolvido pela

ModelState propriedade fornece detalhes de cada problema que tem sido encontrado, mas não precisamos entrar nesse nível de detalhe, porque podemos contar com um recurso útil que automatiza o processo de perguntar ao usuário para resolver quaisquer problemas chamando o método Ver sem nenhum parâmetro.

[HttpPost]

```
público ViewResult Responda Por Favor( GuestResponse resposta) {  
    E se (ModelState.IsValid) {  
        Repositório .AddResponse (resposta);  
        Retorna Visão( "Obrigado" , resposta);  
    } outro {  
        // Há erros de validação  
        Retorna Visão(); }}}
```

## Adicionar validação

Quando MVC torna um ponto de vista, Navalha tem acesso aos detalhes de quaisquer erros de validação associados com o pedido, e auxiliares de tag pode acessar os detalhes para exibir erros de validação para o usuário. o `asp-validation-s`

atributo é aplicado a um elemento div, e exibe uma lista de erros de validação quando a vista é processado.

```
< forma asp-ação =" Responda Por Favor" método = "Post">
    <! - ... ->

    < div asp-validation-summary =" Todos "> </ div >

</ Formato >
```

## DESTACANDO campos inválidos

- Quando há campos inválidos, os dados inseridos são preservados e exibidos novamente. Este é outro benefício de ligação modelo, e simplifica o trabalho com dados do formulário.

```
. campo-validação de erros {  
    cor : # f00 ; }  
  
. -Field-validation válido {  
    exibição : Nenhum ; }  
  
. input-validation-error {  
    fronteira : 1px solid # f00 ;  
    cor de fundo : #taxa ; }  
  
. validação de resumo-erros {  
    espessura da fonte : negrito ;  
    cor : # f00 ; }  
  
. -Validation-summary válida {  
    exibição : Nenhum ; }
```

## BOOTSTRAP

Bootstrap, é framework CSS originalmente desenvolvido pelo Twitter que se tornou um grande projeto de código aberto em seu próprio direito e que se tornou um dos pilares de desenvolvimento de aplicações web. A convenção é que os pacotes Bootstrap e outro CSS de terceiros e JavaScript são instalados no wwwroot / lib pasta

```
@ {
```

```
    Ver dados[ "Título" ] = "Casa" ; }
```

```
< h2 > Casa </ h2 >
```

```
< p > @ ViewBag.Message mundo !!! </ p >
```

```
< p > Nós vamos ter uma festa emocionante. </ p >
```

```
< p > Você está vindo para a festa? </ p > < um asp-action =" Responda Por Favor" classe = "Btn btn-primária">
```

```
    Enviar resposta
```

```
</ uma >
```

```
< um asp-action =" Lista de convidados" classe = "Btn btn-info">
```

```
    Ver quem está vindo para a festa
```

```
</ uma >
```

# BOOTSTRAP

Bootstrap define classes que podem ser usadas para formas de estilo.

<https://bootstrapcreative.com/resources/introduction-to-3-classes-css/>  
[índice/](#)

```
< div classe = "Painel-primário painel" estilo = " margem superior- : 2ex ; ">
    < div classe = "Text-centro-rubrica painel">
        < h2 > Répondez S'il Vous Plait </ h2 >
    </ div >

    < Formato asp-action = "RSVP" método = "Post">
        < div classe = "Panel-body">
            <! - ... ->

            < div asp-validation-summary = "Todos"> </ div >
        </ div > < div classe = "Painel-footer">

            < botão tipo = "Submit" classe = "Btn btn-primária">
                Enviar resposta
            </ botão >
        </ div >

    </ Formato >
</ div >
```

## BOOTSTRAP

o forma-grupo classe é usada para denominar o elemento que contém o rótulo e a entrada associada ou elemento select.

```
< div classe = "Form-grupo">  
  < etiqueta asp-for =" Nome "> Nome: </ rótulo >  
  < entrada para asp- =" Nome " classe = "Form-controle" /> </ div >
```

```
< div classe = "Form-grupo">  
  < etiqueta asp-for =" telefone "> Telefone: </ rótulo >  
  < entrada para asp- =" telefone " classe = "Form-controle" /> </ div >
```

```
< div classe = "Form-grupo">  
  < etiqueta asp-for =" O email "> O email: </ rótulo >  
  < entrada para asp- =" O email " classe = "Form-controle" /> </ div >
```

```
<! - ... ->
```

# BOOTSTRAP

T CAPAZES

```
< mesa classe = "Table-listrado mesa">
    < thead > < tr > < ^> Nome </ ^> < ^> telefone </ ^>
        < ^> O email </ ^> < ^> vai participar </
            ^>
        </ tr >
    </ thead >

    <! - ... ->
</ mesa >
```

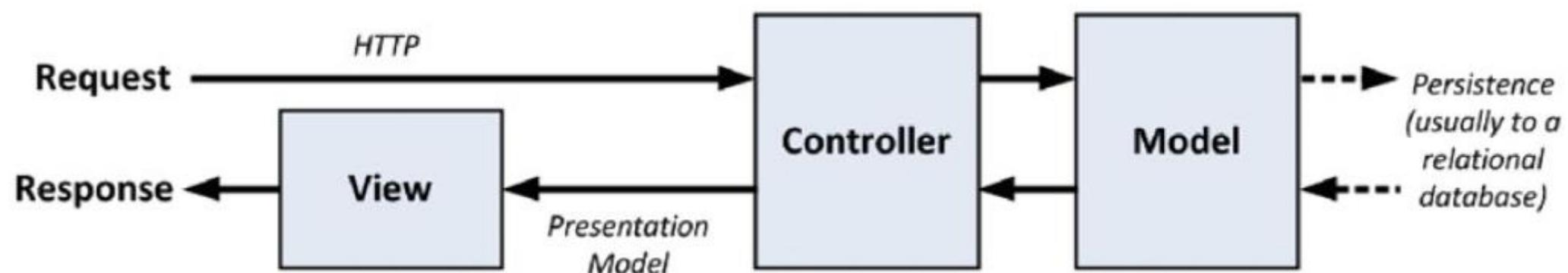
## Usando um tema BOOTSTRAP DIFERENTE

- Faça o download e renomear os arquivos CSS de <https://bootswatch.com/flatly/>
- Adicione os arquivos para a pasta CSS
- Mudar o Comum / \_Layout.cshtml Arquivo

```
< nomes ambiente = " Desenvolvimento ">  
    < ligação rel = "Stylesheet" href = "~ / Css / flatly_bootstrap.css" /> < ligação rel = "Stylesheet" href  
    = "~ / Css / Site.css" /> </ meio Ambiente >
```

```
< nomes ambiente = " Encenação, Produção ">  
    < ligação rel = "Stylesheet" href = "~ / Css / flatly_bootstrap.min.css" /> < ligação rel = "Stylesheet" href = "~ / css /  
    site.min.css" asp-append-version = " true"/> </ meio Ambiente >
```

# A IMPLEMENTAÇÃO DE ASP.NET MVC



## mODELOS

Veja os  
modelos

Representam apenas dados passados do controller para a view

modelos de  
domínio

Conter os dados em um domínio do negócio, juntamente com as operações, transformações e regras para criar, armazenar e manipular esses dados, referidos coletivamente como a lógica do modelo.

## modelos de domínio

### Um modelo deve

- Contêm os dados de domínio
- Conter a lógica para criar, gerenciar e modificar os dados de domínio
- Fornecer uma API limpa que expõe os dados e operações modelo nele

### Um modelo não deve

- Expor detalhes de como os dados do modelo é obtida ou geridos (em outras palavras, os detalhes do mecanismo de armazenamento de dados não deve ser exposto a controladores e views)
- Conter a lógica que transforma o modelo baseado na interação do usuário (porque isso é o trabalho do controlador)
- Conter a lógica para a exibição de dados para o usuário (que é o trabalho do ponto de vista)

## CONTROLADORES

Um controlador deve

- Conter as ações necessárias para atualizar o modelo baseado na interação do usuário

O controlador não deve

- Conter a lógica que gere a aparência de dados (que é o trabalho do ponto de vista)
- Conter a lógica que gere a persistência de dados (que é o trabalho do modelo)

## VISTAS

### Visualizações deve

- Contêm a lógica e marcação exigida a apresentação de dados para o usuário

### Visualizações não deve

- Conter lógica complexa (isto é melhor colocado num controlador)
- Conter lógica que cria, armazena, ou manipula o modelo de domínio



1

Navalha é o mecanismo de exibição responsável por incorporando dados em documentos HTML.



2

Navalha fornece recursos que tornam mais fácil para trabalhar com o resto do ASP.NET MVC Núcleo usando C # declarações.



3

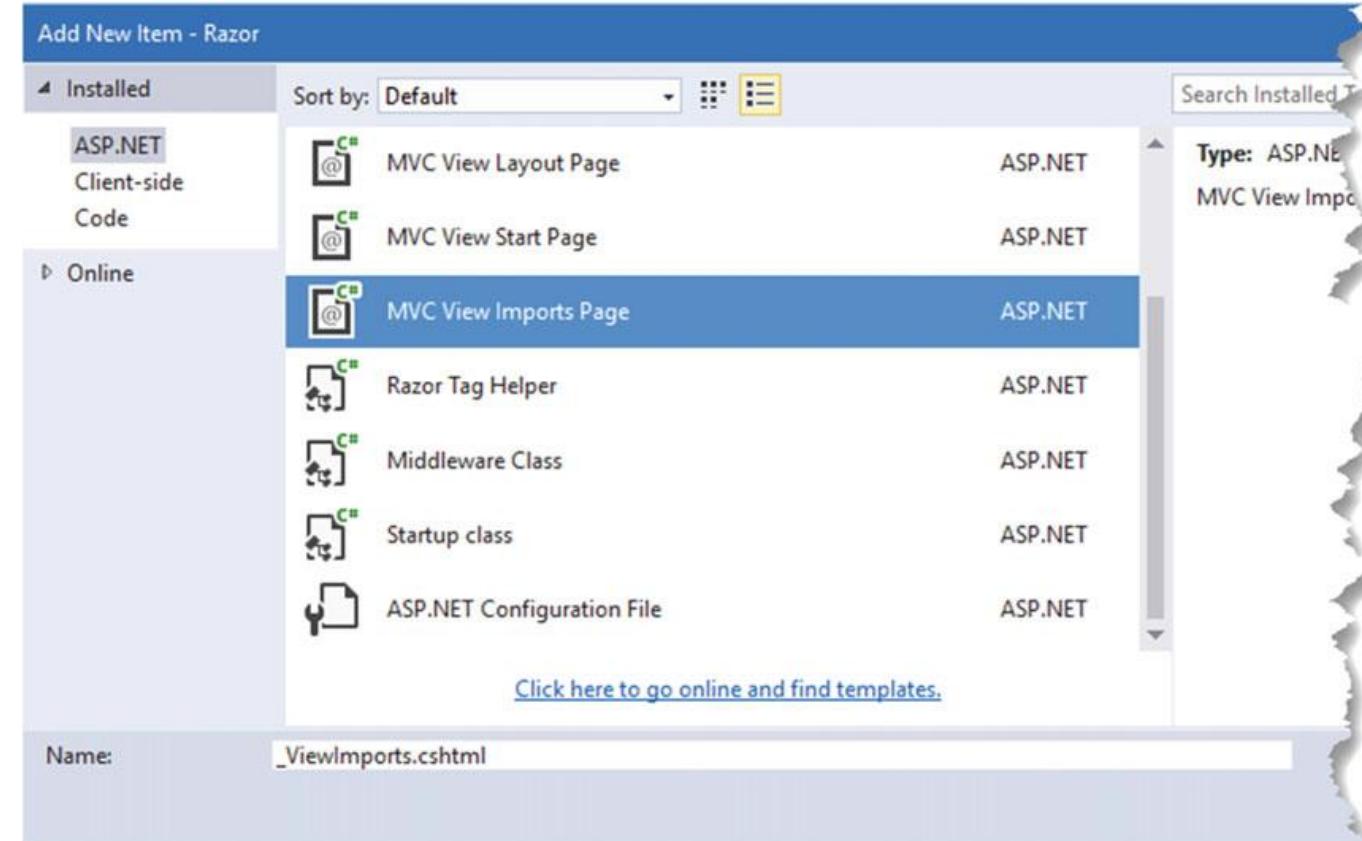
expressões de barbear são adicionados ao HTML estático em arquivos de visão. As expressões são avaliadas para gerar respostas às solicitações do cliente.



NAVALHA

## IMPORTAÇÕES VISTA

- Por padrão, todos os tipos que são referenciados em uma visão Navalha fortemente digitado deve ser qualificado com um espaço de nomes (por exemplo, **PartyInvites.Models @model. GuestResponse** )
- Alternativamente, você pode especificar um conjunto de espaços de nomes que devem ser procurados os tipos, adicionando um arquivo vista importações para o projeto. O arquivo vista importações é colocado na pasta Views e é nomeado \_ ViewImports.cshtml.
- Você também pode adicionar uma **@ utilização** expressão para os arquivos de visão individuais, o que permite que tipos para ser usado sem espaços de nomes em uma única visualização.



## CRIAR UMA PÁGINA Vista IMPORTAÇÕES

## TRABALHANDO COM LAYOUTS

Um projeto real pode ter dezenas de pontos de vista, e alguns pontos de vista terá conteúdo compartilhado. Duplicação de conteúdo compartilhado em vista torna-se difícil de gerir, especialmente quando você precisa fazer uma mudança e tem que rastrear todos os pontos de vista que precisam ser alterados. Uma abordagem melhor é usar um layout Navalha, que é um modelo que contém conteúdo comum e que pode ser aplicado a um ou mais pontos de vista. Quando você faz uma alteração em um layout, a mudança afetará automaticamente todos os pontos de vista que o utilizam.

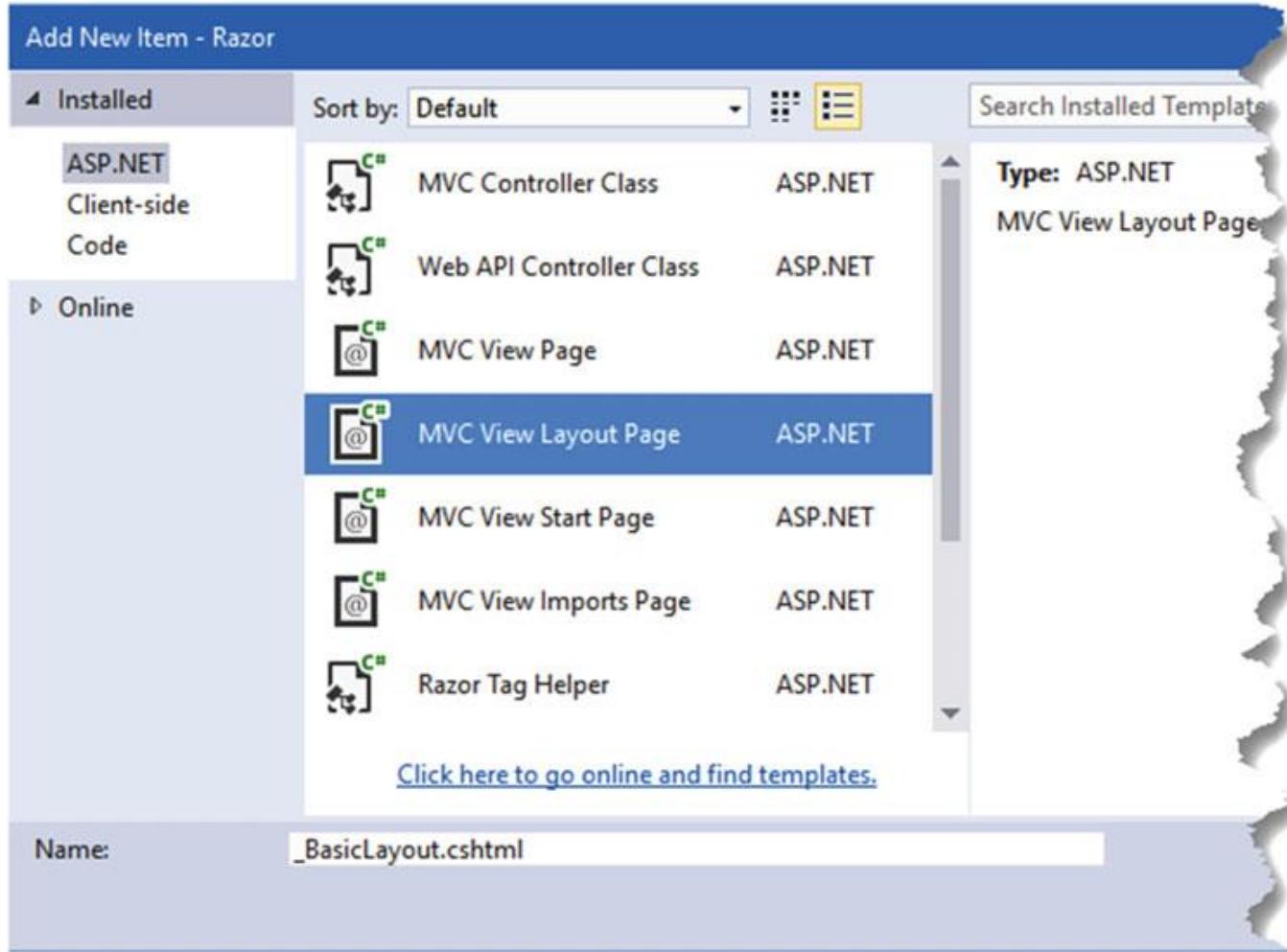
```
@ {
```

```
layout = "_Layout" ; }
```



## TRABALHANDO COM LAYOUTS

- Layouts são tipicamente compartilhada por visões usadas por vários controladores e são armazenados em uma pasta chamada **Views / Shared**, que é um dos locais que Navalha olha em quando se tenta encontrar um arquivo.
- Como importar arquivos de visualização, os nomes de arquivos de layout começar com um sublinhado, porque eles não são destinadas a ser devolvido diretamente para o usuário.



Criar uma página de exibição de layout

## LAYOUTS

```
<!DOCTYPE html> <html> <cabeça> <meta nome = "Viewport" conteúdo = "Width =  
device-width" /> <título> @ ViewBag.Title </título>
```

```
</cabeça>
```

```
<corpo>
```

```
  <div>
```

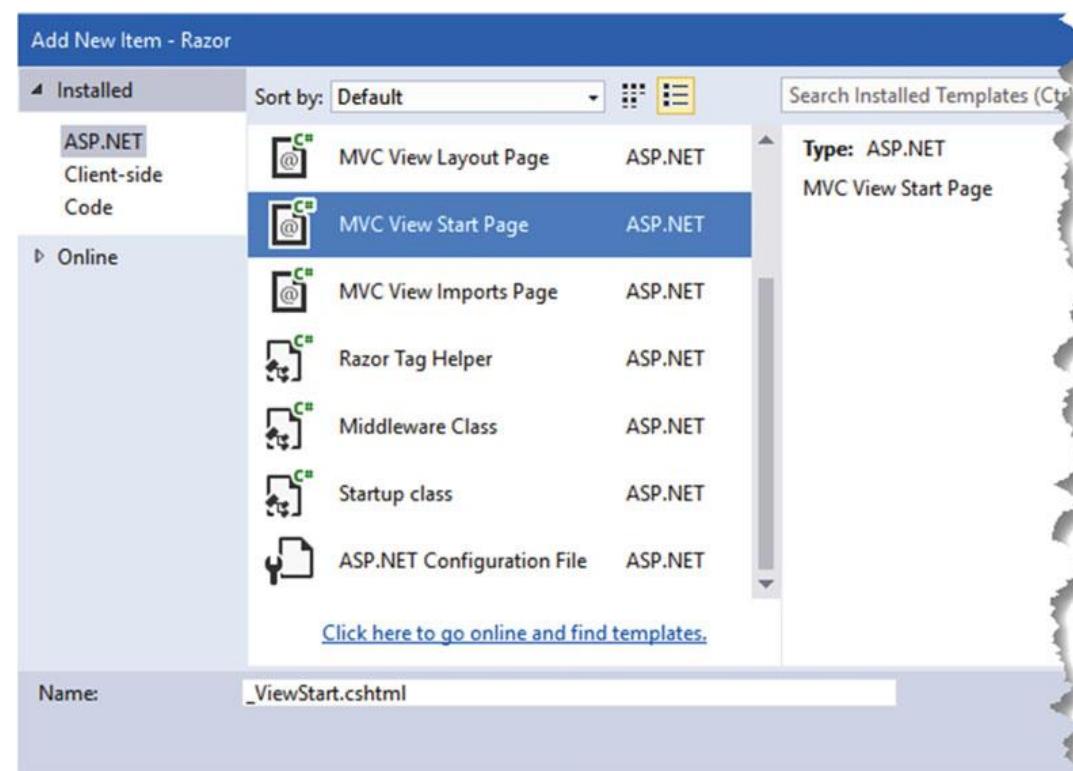
```
    @RenderBody()
```

```
  </div>
```

```
</corpo>
```

```
</html>
```

## CRIAR UM COMEÇO página de visualização



PROJETO SPORTS STORE

## Modelo Produto

```
namespace SportsStore.Models {  
  
    classe pública produtos {  
  
        int público ID do produto { obter ; conjunto ; }  
  
        public string nome { obter ; conjunto ; }  
  
        public string Descrição { obter ; conjunto ; }  
  
        decimal pública Preço { obter ; conjunto ; }  
  
        public string Categoria { obter ; conjunto ; }}}}
```

## interface de repositório

```
namespace SportsStore.Models {  
    interface pública IProductRepository {  
        IEnumerable <Product> Produtos { obter ; } } }  
    
```

## REPOSITORY FALSIFICAÇÃO

```
namespace SportsStore.Models {  
  
    classe pública FakeProductRepository : IProductRepository {  
  
        público IEnumerable <Product> Produtos => Novo Lista <Product> {  
  
            Novo Product {Name = "Futebol" , Preço = 25},  
  
            Novo Product {Name = "Prancha de surfe" , Preço = 179},  
  
            Novo Product {Name = "Tênis de corrida" , Preço = 95}  
  
        }; }}
```

SERVIÇO

COMPONENTES

---

MVC enfatiza o uso de componentes de baixo acoplamento, o que significa que você pode fazer uma mudança em uma parte da aplicação sem ter que fazer alterações correspondentes em outros lugares.

---

Esta abordagem categoriza partes da aplicação como serviços, que fornecem características que outras partes do uso do aplicativo.

---

A classe que fornece um serviço pode, então, ser alterados ou substituídos sem a necessidade de mudanças nas classes que utilizam.

## Configuração de serviços

```
classe pública Comece {  
    // ...  
  
    public void ConfigureServices (serviços IServiceCollection) {  
        services.AddMvc ();  
  
        services.AddTransient <IProductRepository, FakeProductRepository> (); } }
```

## INJEÇÃO DE DEPENDÊNCIA

```
classe pública ProductController : Controlador {  
    privado repositório IProductRepository;  
  
    público ProductController (repositório IProductRepository) {  
        esta .repository = repositório;  
    }  
  
    público Lista ViewResult () => View (repository.Products); }
```

## EXIBIÇÃO DE LISTA

```
@model IEnumerable <Product>

{@
    Ver dados[ "Título" ] = "Produtos" ; }

@ para cada (P var dentro Modelo) {
    < div > < h3 > @ p.Name </ h3 >
        @ p.Description
        < h4 > @ P.Price.ToString ( "C" ) </ h4 >
    </ div >
}
```

# ENTIDADE estrutura do núcleo (EF CORE)

- EF Core é uma estrutura de mapeamento objeto-relacional (ORM). Um quadro ORM apresenta as tabelas, colunas e linhas de um banco de dados relacional através de objetos regulares.

[HTTPS://DOCS.MICROSOFT.COM/EN-US/EF/](https://docs.microsoft.com/en-us/ef/)

## Criando as classes DATABASE

A classe de contexto de banco de dados é a ponte entre a aplicação e o EF Core e fornece acesso a dados do aplicativo usando objetos do modelo.

```
classe pública ApplicationDbContext : DbContext {  
    público ApplicationDbContext (DbContextOptions <ApplicationDbContext> opções): base (opções) {}  
  
    público DbSet <Produto> Produtos { obter ; conjunto ; }  
}
```

## EF repositório do produto

```
classe pública EFProductRepository : IProductRepository {  
    privado ApplicationDbContext contexto;  
  
    público EFProductRepository (contexto ApplicationDbContext) {  
        esta .context = contexto;  
    }  
  
    público IEnumerable <Produto> Produtos => context.Products; }
```

## Cadeia de ligação (APPSETTINGS.JSON)

```
{  
  "Exploração madeireira": {  
    "IncludeScopes": false,  
    "LogLevel": {  
      "Padrão": "Atenção"  
    },  
  
    "ConnectionStrings": {  
      "ConnectionStringSportsStore":  
        "Servidor = (LocalDB) \\ mssqllocaldb; banco de dados = SportsStore; Trusted_Connection = True; MultipleActiveResultSets = true"  
    }  
}
```

## CONFIGURAÇÃO EF DATABASE

```
utilização Microsoft.EntityFrameworkCore;

// ...

public void ConfigureServices (serviços IServiceCollection) {
    services.AddMvc (); services.AddTransient < IProductRepository , EFProductRepository > ();
    services.AddDbContext < ApplicationDbContext > (Options => options.UseSqlServer (Configuration.GetConnectionString ( "ConnectionStringSportsStore"
        )));
}; }
```

## DADOS SEED

```
classe estática pública SeedData {  
    public static void EnsurePopulated (IServiceProvider appServices) {  
  
        ApplicationDbContext contexto = (ApplicationDbContext) appServices.GetService ( tipo de (ApplicationDbContext));  
  
        E se (Context.Products.Any ()) return;  
  
        context.Products.AddRange (  
            Novo Product {Name = "Caiaque" , Descrição = "Um barco para uma pessoa" Categoria = "Esportes Aquáticos" , Preço = 275},  
            Novo Product {Name = "Colete salva-vidas" , Descrição = "Protecção e elegante" Categoria = "Esportes Aquáticos" , Preço = 48.95m},  
            Novo Product {Name = "Bola de futebol" , Descrição = "Tamanho e peso FIFA-aprovado" Categoria = "Futebol" , Preço = 19.50m},  
            Novo Product {Name = "Canto Flags" , Descrição = "Dê o seu campo de jogo um toque profissional" Categoria = "Futebol" , Preço = 34.95m},  
            Novo Product {Name = "Estádio" , Descrição = "Estádio de 35.000 lugares Plano-embalados" Categoria = "Futebol" , Preço = 79500},  
            Novo Product {Name = "Pensando Cap" , Descrição = "Melhorar a eficiência do cérebro em 75%" Categoria = "Xadrez" , Preço = 16},  
            Novo Product {Name = "Cadeira instável" , Descrição = "Secretamente dar o seu adversário uma desvantagem" Categoria = "Xadrez" , Preço = 29.95m},  
            Novo Product {Name = "Tabuleiro de Xadrez Humano" , Descrição = "Um jogo divertido para a família" Categoria = "Xadrez" , Preço = 75},  
            Novo Product {Name = "Bling-Bling Rei" , Descrição = "Banhado a ouro, cravejado de diamantes Rei" Categoria = "Xadrez" , Preço = 1200}  
        );  
        context.SaveChanges (); }}
```

## DADOS SEED

```
public void configurar ( IApplicationBuilder aplicativo, IHostingEnvironment env, ILoggerFactory loggerFactory) {  
    // ...  
  
    SeedData .EnsurePopulated (app.ApplicationServices);  
}
```

## DADOS SEED

- ASP.NET MVC NÚCLEO dois

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

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

# migração de banco

---

Entity Framework Core é capaz de gerar o esquema para o banco de dados usando as classes do modelo através de um recurso chamado migrações.

---

Quando você preparar uma migração, EF núcleo cria uma classe C # que contém os comandos SQL necessários para preparar o banco de dados. Se você precisar modificar suas classes de modelo, então você pode criar uma nova migração que contém os comandos SQL necessários para refletir as alterações.

---

Desta forma, você não precisa se preocupar com manualmente escrevendo e testando comandos SQL e pode se concentrar apenas nas classes C # modelo na aplicação.

---

comandos EF núcleo são realizadas usando o console do pacote.

## PREPARAR O banco de dados para a sua primeira utilização

- Add-migração inicial
- Atualizar o banco de dados

## ENCAMINHAMENTO

```
public void Configurar (IApplicationBuilder aplicativo, IHostingEnvironment env) {  
    // ...  
  
    app.UseMvc (rotas => {  
        routes.MapRoute (  
            nome: "padrão" , Modelo: "{Controller = Produtos} / {action = Lista} / {id}?" );  
  
    });  
  
    // ...  
}
```

## APOIO PAGINATION

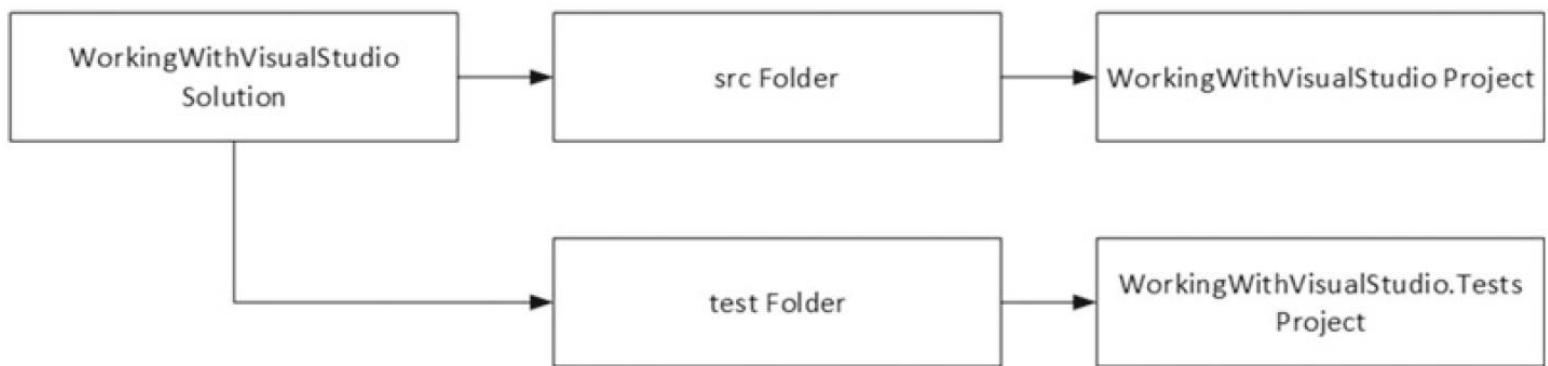
```
classe pública ProductController : Controlador {
    privado repositório IProductRepository;
    int público PageSize = 4;

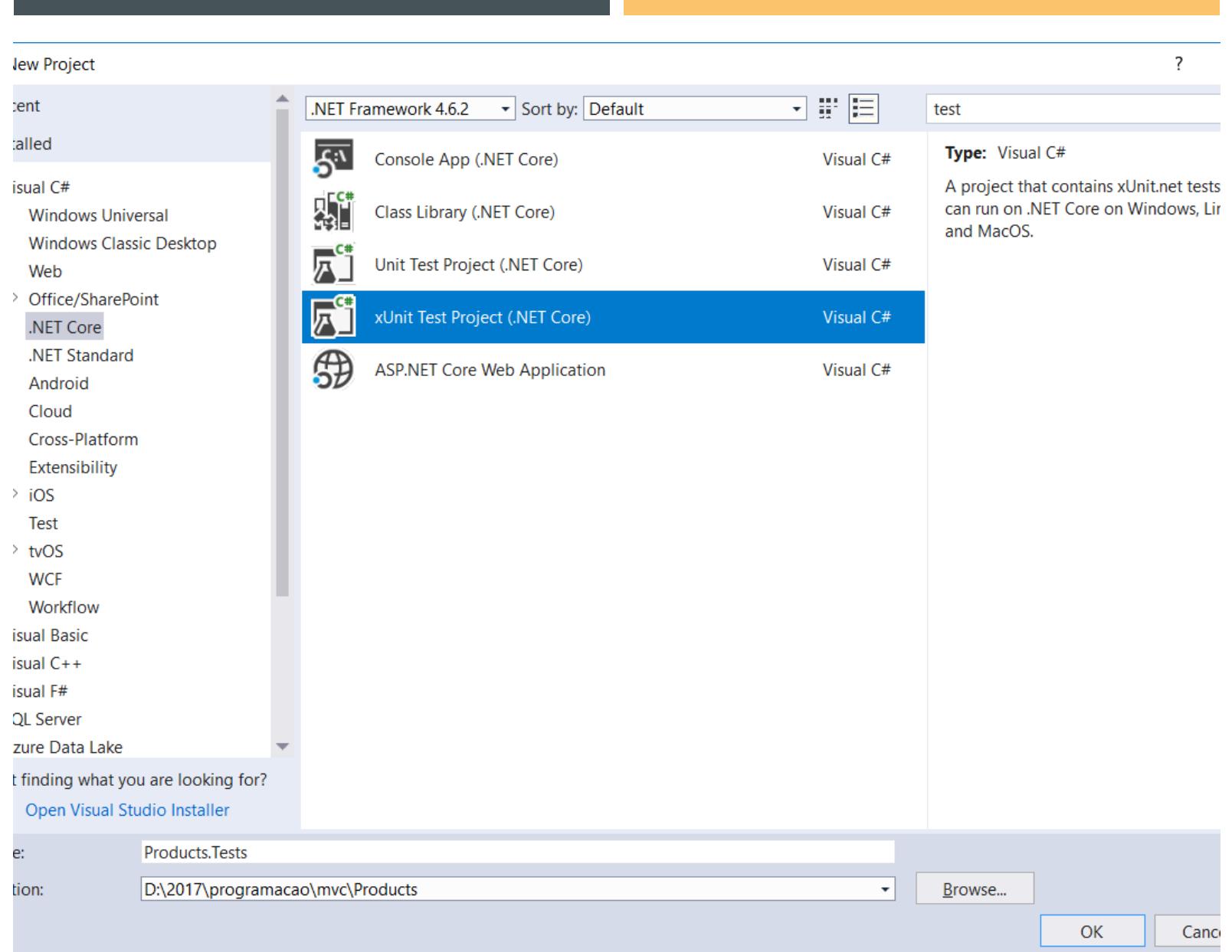
    público ProductController (repositório IProductRepository) {
        esta .repository = repositório;
    }

    público Lista ViewResult ( int page = 1 ) => View (
        repository.Products
            . OrdenarPor ( p => p.Price )
            . Ir ((Página - 1) * PageSize)
            . Tome (PageSize)
    );
}
```

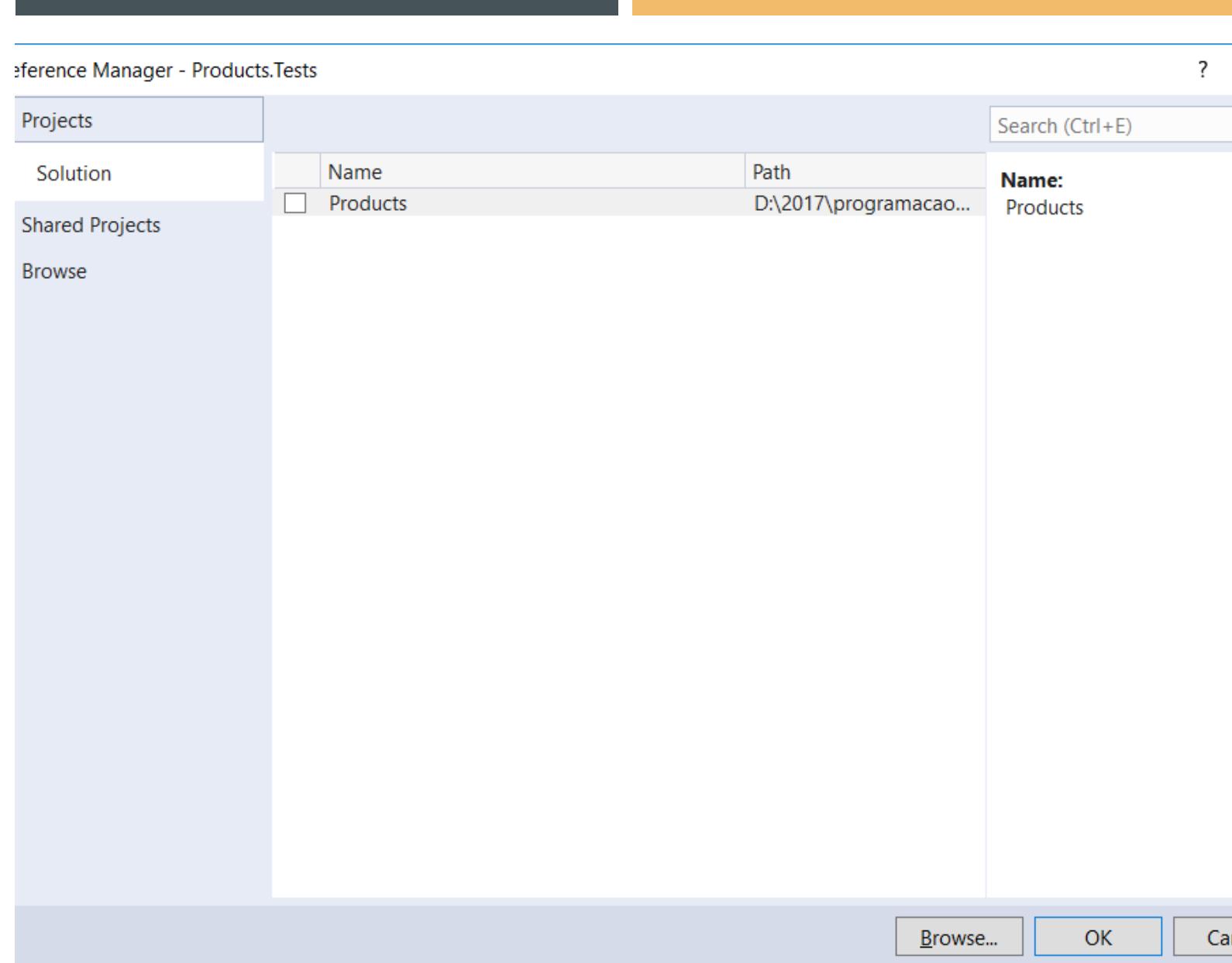
## O teste de unidade

- Testes de unidade são usados para validar o comportamento dos componentes e recursos individuais em um aplicativo, e ASP.NET Core e ASP.NET MVC núcleo foram concebidos para torná-lo tão fácil como possível configurar e executar testes de unidade para aplicações web.
- Para aplicações ASP.NET do núcleo, geralmente você cria um projeto Visual Studio separada para armazenar os testes de unidade, cada um dos quais é definido como um método em uma classe C#. Usando um projeto separado significa que você pode implantar seu aplicativo sem também implantar os testes.





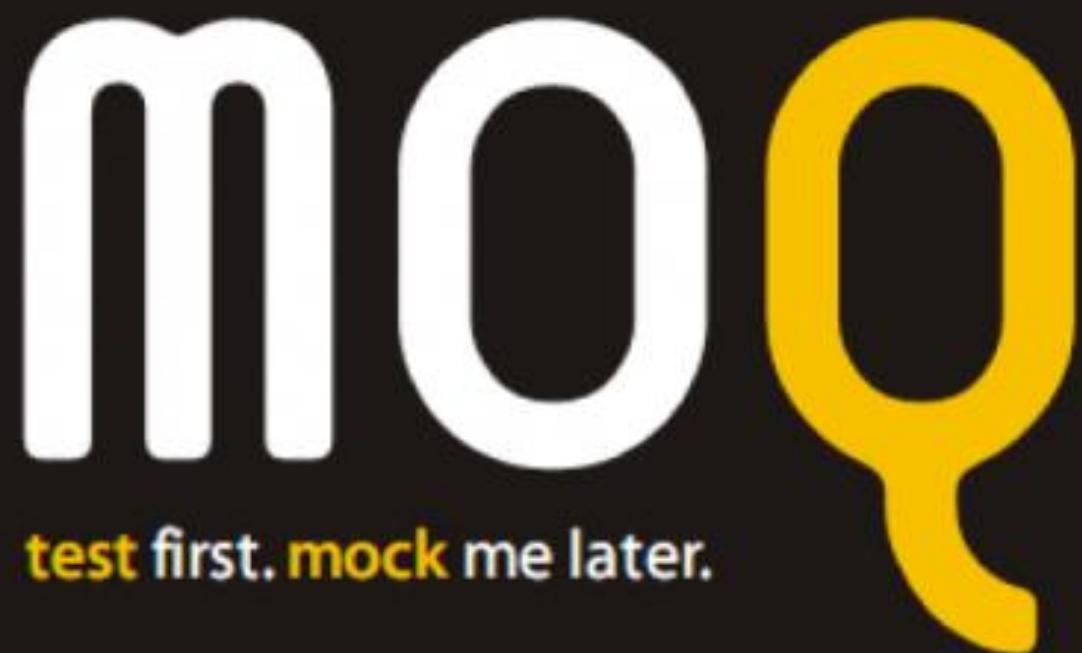
## testes de unidade



```
classe pública ProductControllerTests {  
    [Facto]  
    public void CanPaginate () {  
        // Organizar //  
        Lei // Assert  
    }  
}
```

## ZOMBAR

- Moq cria implementações falsas de componentes de uma aplicação.
- Ela torna mais fácil para criar componentes falsos para isolar partes da aplicação para testes de unidade.
- <https://www.nuget.org/packages/moq>



[Facto]

```
public void CanPaginate () {  
    // Organizar  
    Mock <IProductRepository> simulada = Novo Mock <IProductRepository> (); mock.Setup (m =>  
        m.Products) .Returns ( Novo Produtos[] {
```

```
        Novo Product {ProductID = 1, Name = "P1"},  
        Novo Product {ProductID = 2, Name = "P2"},  
        Novo Product {ProductID = 3, Name = "P3"},  
        Novo Product {ProductID = 4, Name = "P4"},  
        Novo Product {ProductID = 5, Name = "P5"});
```

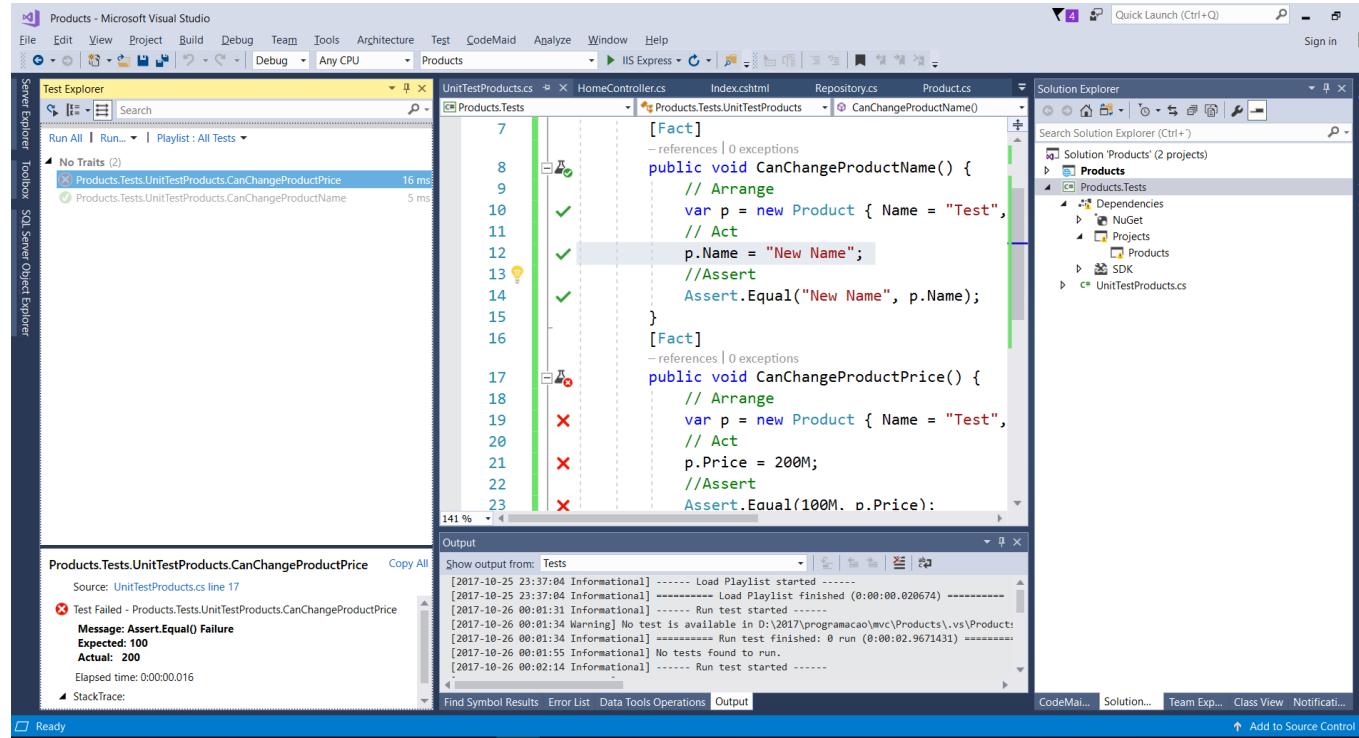
```
controlador ProductController = Novo ProductController (mock.Object); controller.PageSize = 3;
```

```
// Lei // Assert
```

```
}
```

```
utilização System.Linq ;  
  
[Facto]  
public void CanPaginate () {  
    // Organizar // ...  
  
    // Aja  
    IEnumerable <Product> resultado = controller.List (2) .ViewData.Model Como IEnumerable <Product>;  
  
    // Assert  
    Produto [] prodArray = resultado. ToArray () ; Assert.True  
(prodArray.Length == 2); Assert.Equal ( "P4" , ProdArray [0]  
.Nome); Assert.Equal ( "P5" , ProdArray [1] .Nome);  
}  
}
```

execução de  
testes



## testes de unidade

- Quando um teste falhar, é sempre uma boa idéia para verificar a precisão do teste antes de olhar para as metas de TI componentes, especialmente se o teste é novo ou foi recentemente modificado.

## PAGINGINFO viewmodel

```
namespace SportsStore.Models.ViewModels {  
    classe pública PagingInfo {  
        int público TotalItems { obter ; conjunto ; }  
        int público Itens por página { obter ; conjunto ; }  
        int público Pagina atual { obter ; conjunto ; }  
        int público TotalPages =>  
            ( int ) Math.Ceiling (( decimal ) TotalItems / ItemsPerPage);  
    }  
}
```

## AJUDANTES TAG

```
namespace SportsStore.Infrastructure {  
    [HtmlTargetElement ( "Div" , Atributos = "Page-modelo" )]  
    classe pública PagingLinksTagHelper : TagHelper {  
        privado IUrlHelperFactory urlHelperFactory;  
  
        público PagingLinksTagHelper (IUrlHelperFactory helperFactory) {urlHelperFactory = helperFactory; }  
  
        [ViewContext]  
        [HtmlAttributeNotBound]  
        público ViewContext ViewContext { obter ; conjunto ; }  
  
        público PagingInfo PageModel { obter ; conjunto ; }  
  
        public string PageAction { obter ; conjunto ; }  
  
        // ...  
    }  
}
```

## AJUDANTES TAG

```
classe pública PageLinkTagHelper : TagHelper {  
    // ...  
  
    override void pública Processo (TagHelperContext contexto, saída TagHelperOutput) {  
        IUrlHelper UrlHelper = urlHelperFactory.GetUrlHelper (ViewContext);  
  
        resultado TagBuilder = Novo TagBuilder ( "Div" );  
        para ( int i = 1; i <= PageModel.TotalPages; i ++ ) {  
            TagBuilder tag = Novo TagBuilder ( "uma" ); tag.Attributes [ "Href" ] = UrlHelper.Action (PageAction, Novo {Page = i});  
            tag.InnerHtml.Append (i.ToString ());  
  
            result.InnerHtml.AppendHtml (tag); }  
  
        output.Content.AppendHtml (result.InnerHtml); }}
```

## Produtos Lista VISTA MODELO

```
namespace SportsStore.Models.ViewModels {  
    classe pública ProductsListViewModel {  
        público IEnumerable <Product> Produtos { obter ; conjunto ; }  
        público PagingInfo PagingInfo { obter ; conjunto ; }}}}
```

## CONTROLADOR DE PRODUTOS

```
público Lista ViewResult ( int page = 1 ) =>
    Visão(
        Novo ProductsListViewModel {
            Produtos = repository.Products
                . OrdenarPor ( p => p.Price )
                . Ir ((Página - 1) * PageSize)
                . Tome (PageSize),
            PagingInfo = Novo PagingInfo {
                CurrentPage = página, ItemsPerPage
                = PageSize,
                TotalItems = repository.Products.Count ()}
        );
    );
```

## Adicionando o AJUDANTES INFRASTRUCTE TAG

```
@ utilização Loja de esportes @ utilização SportsStore.Models @ utilização SportsStore.Models.ViewModels
```

```
@addTagHelper SportsStore.Infrastructure. *, SportsStore
```



```
[REDACTED]
```

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

# VISÃO

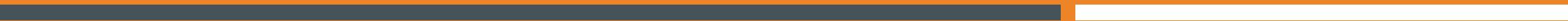
```
@model ProductsListViewModel

@{
    Ver dados[ "Título" ] = "Produtos" ;
}

@ para cada (P var dentro Model.Products) {
    < div classe = "Bem bem-sm">
        < h3 >@ p.Name </ h3 >
        @ p.Description
        < h4 >@ P.Price.ToString ( "C" ) </ h4 >
    </ div >
}

< div página-modelo ="@ Model.PagingInfo" page-action =" List "> </ div >
```

## LIVROS DO PROJETO



## CLASSE AUTOR

```
namespace Books.Models {  
    classe pública Autor {  
        int público AuthorID { obter ; conjunto ; }  
        public string nome { obter ; conjunto ; }  
        público ICollection < Livro > Livros { obter ; conjunto ; }}}
```

## MODELO LIVRO

```
namespace Books.Models {  
  
    classe pública Livro {  
  
        int público BookID { obter ; conjunto ; }  
  
        public string título { obter ; conjunto ; }  
  
        público Autor Autor { obter ; conjunto ; }  
  
        int público AuthorID { obter ; conjunto ; }  
  
        público ICollection <BookCategory> Categorias { obter ; conjunto ; }}}}
```

## categoria de modelo

```
namespace Books.Models {  
    classe pública Categoria {  
        int público Categoria ID { obter ; conjunto ; }  
        public string nome { obter ; conjunto ; }  
        público ICollection <BookCategory> Livros { obter ; conjunto ; }}}}
```

## MODELO BOOKCATEGORY

```
namespace Books.Models {  
    classe pública BookCategory {  
        int público BookID { obter ; conjunto ; }  
        público Livro livro { obter ; conjunto ; }  
        int público Categoria ID { obter ; conjunto ; }  
        público Categoria Categoria { obter ; conjunto ; }}}}
```

## CONTEXTO DATABASE

```
classe pública BooksDbContext : DbContext {  
    público BooksDbContext (  
        DbContextOptions < BooksDbContext > Opções): base (opções) {}  
  
    público DbSet < Autor > Autores;  
    público DbSet < Livro > Livros;  
    público DbSet < Categoria > categorias;  
    público DbSet < BookCategory > BooksCategories;  
}
```

## DATABASE CONTEXTO: API fluente PARA EF NÚCLEO

```
classe pública BooksDbContext : DbContext {  
    // ...  
  
    protected override void OnModelCreating (ModelBuilder modelBuilder) {  
        modelBuilder.Entity <BookCategory> ()  
            . Haskey (bc => Novo {Bc.BookId, bc.CategoryId});  
  
        modelBuilder.Entity <BookCategory> ()  
            . HasOne (bc => bc.Book)  
            . Withmany (b => b.BookCategories)  
            . HasForeignKey (bc => bc.BookId);  
  
        modelBuilder.Entity <BookCategory> ()  
            . HasOne (bc => bc.Category)  
            . Withmany (c => c.BookCategories)  
            . HasForeignKey (bc => bc.CategoryId);  
    }  
}
```

## Cadeia de ligação (APPSETTINGS.JSON)

```
{  
  "Exploração madeireira": {  
    "IncludeScopes": false,  
    "LogLevel": {  
      "Padrão": "Atenção"  
    }  
  },  
  
  "ConnectionStrings": {  
    "ConnectionStringBooks":  
      "Servidor = (LocalDB) \\mssqllocaldb; banco de dados = Books; Trusted_Connection = True; MultipleActiveResultSets = true"  
  }  
}
```

## CONFIGURAÇÃO EF DATABASE

```
utilização Microsoft.EntityFrameworkCore;  
  
// ...  
  
public void ConfigureServices (serviços IServiceCollection) {  
    services.AddMvc ();  
    services.AddDbContext < BooksDbContext > (Options =>  
        options.UseSqlServer (Configuration.GetConnectionString ( "ConnectionStringBooks" ))  
    );  
}
```

## PREPARAR O banco de dados para a sua primeira utilização

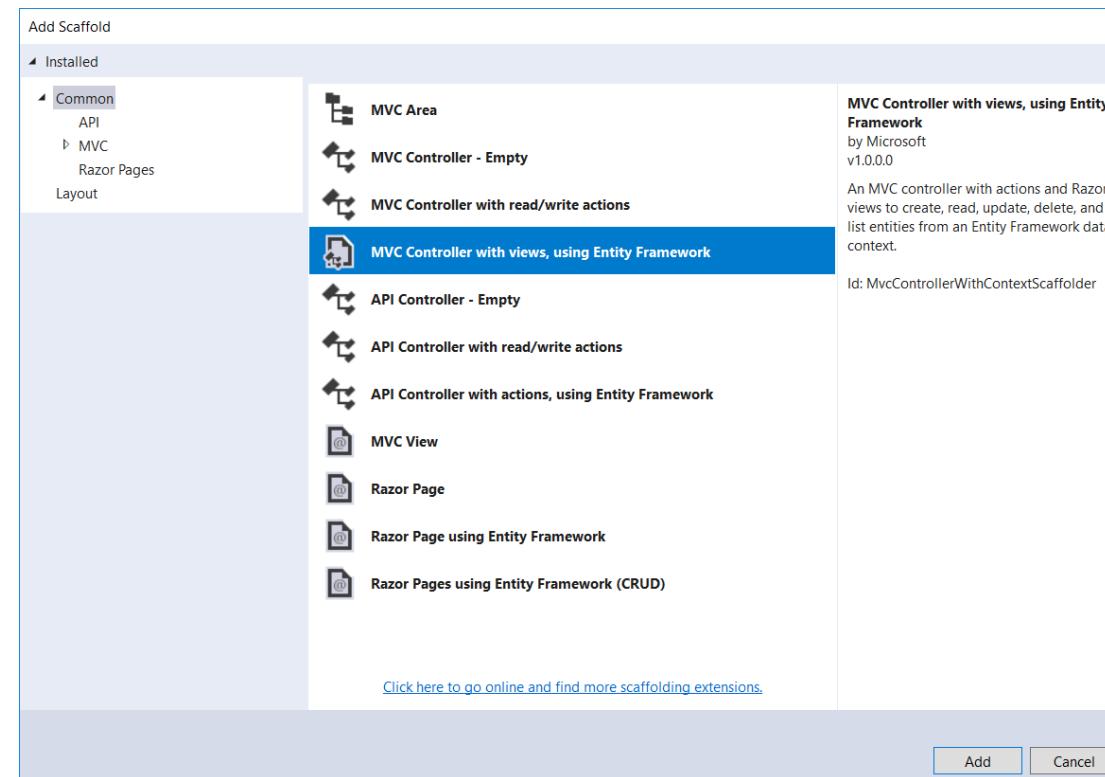
- Add-migração inicial
- Atualizar o banco de dados

# ANDAIME

Você pode sentar e pensar sobre o problema que você quer resolver, e escrever as classes C # simples, tais como **Álbum**, **ShoppingCart**, e **Do utilizador**, para representar os objetos primários envolvidos. Quando estiver pronto, você pode usar ferramentas fornecidas pela MVC para construir os controladores e vistas para o índice padrão, criar, editar e excluir cenários para cada um dos objetos de modelo. O trabalho de construção é chamado

*andaime.*

# ANDAIME



# ANDAIME

Add MVC Controller with views, using Entity Framework X

Model class: Book (Books.Models) ▼

Data context class: BooksDbContext (Books.Data) ▼ **+**

Views:

Generate views

Reference script libraries

Use a layout page:  
 ...

(Leave empty if it is set in a Razor \_viewstart file)

Controller name: BooksController

**Add** **Cancel**

PROJETO SPORTS STORE

DENOMINANDO a lista dos produtos

```
@ para cada (P var dentro Model.Products) {  
    < div classe = "Bem bem-sm">  
        < h3 > < Forte > @ p.Name </ Forte > < período classe = "Pull-direito rótulo label-primária" > @ P.Price.ToString ( "C" ) </ período >  
        @ p.Description  
    </ h3 >  
    < / div >  
}  
  
< div página-metodo =" @ Model.PagingInfo " page-action =" List "> </ div >
```

```
classe pública PageLinkTagHelper : TagHelper {  
    // ...  
  
    bool pública CssClassesEnabled { obter ; conjunto ; } = falso ;  
    public string CssClassPage { obter ; conjunto ; }  
    public string CssClassPageNormal { obter ; conjunto ; }  
    public string CssClassPageSelected { obter ; conjunto ; }  
}
```

## AJUDANTES TAG

```
override void pública Processo (TagHelperContext contexto, saída TagHelperOutput) {
    IUrlHelper UrlHelper = urlHelperFactory.GetUrlHelper (ViewContext);

    resultado TagBuilder = Novo TagBuilder ( "Div" );
    para ( int i = 1; i <= PageModel.TotalPages; i ++ ) {
        TagBuilder tag = Novo TagBuilder ( "uma" ); tag.Attributes [ "Href" ] = UrlHelper.Action (PageAction, Novo {Page = i});
        tag.InnerHtml.Append (i.ToString ());
        if (PageClassesEnabled) {
            tag.AddCssClass (PageClass);
            tag.AddCssClass (? i == PageModel.CurrentPage PageClassSelected: PageClassNormal);
        }
        result.InnerHtml.AppendHtml (tag);
    }
    output.Content.AppendHtml (result.InnerHtml); }}
```

## DENOMINANDO a lista dos produtos

```
@ para cada (P var dentro Model.Products) {
    < div classe = "Bem bem-sm">
        < h3 > < Forte > @ p.Name </ Forte > < período classe = "Pull-direito rótulo label-primária" > @ P.Price.ToString ( "C" ) </ período >
        < / h3 >
        @ p.Description
    < / div >
}

< div página-modelo = "@ Model.PagingInfo" page-action =" Lista"
    page-aulas-habilitado = "True" página de classe = "Btn" página de classe normal = "Btn-default"
    -Page-classe selecionada = "Btn-primário" classe = "Btn-grupo pull-right" > < / div >
```

## vistas parciais

```
@model produtos
```

```
< div classe = "Bem bem-sm">  
    < h3 > < Forte > @ Model.Name </ Forte > < período classe = "Pull-direito rótulo label-primária" > @ Model.Price.ToString ( "C" ) </ período >  
    < /h3 >  
    @ Model.Description  
< / div >
```

## UTILIZAÇÃO vistas parciais

```
@model ProductsListViewModel
```

```
@{
```

```
    Ver dados[ "Título" ] = "Produtos" ; }
```

```
@ para cada (P var dentro Model.Products) {
```

```
    @ Html.Partial ( "Sumario de produtos" , P)
```

```
}
```