

ASP.NET MVC Tutorial

Do original: <http://www.asp.net/mvc/tutorials/mvc-music-store/mvc-music-store-part-3>

Este tutorial é uma tradução do original criado e mantido pela Microsoft. Este material tem fins educativos e é fornecido na maneira que se apresenta.

Parte 3:: Views e ViewModels

Até agora nós temos retornado strings das nossas ações de controller. Isto é legal para termos uma ideia de como controllers funcionam mas não é a forma que gostaríamos de construir uma aplicação Web de verdade. Nós vamos precisar de uma maneira melhor de gerar HTML para retornar aos navegadores que acessam nosso site, uma solução onde possamos usar templates para facilitar a customização deste retorno HTML. Isto é exatamente o que Views fazem.

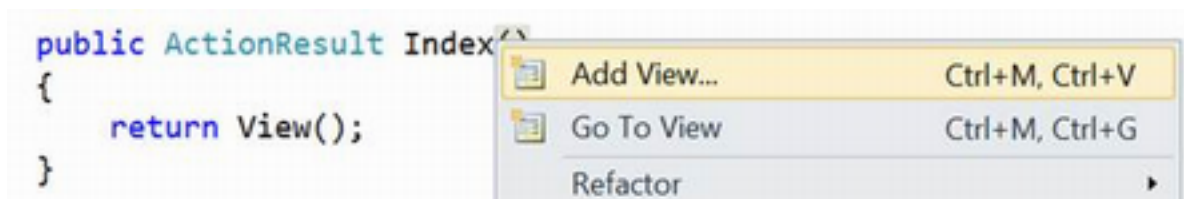
Adicionando uma View

Para usar uma view-template, nós iremos alterar o método Index no HomeController para retornar um ActionResult, e alterar o corpo do método para retornar View() como descrito abaixo:

```
public class HomeController : Controller
{
    //
    // GET: /Home/
    public ActionResult Index()
    {
        return View();
    }
}
```

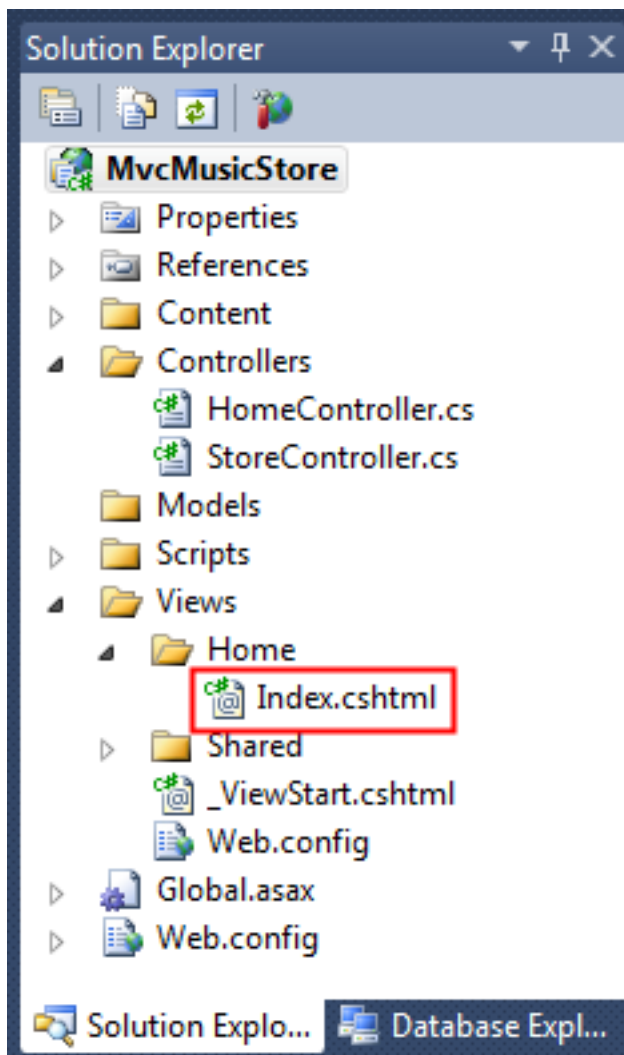
A alteração acima indica que ao invés de retornar uma string nós iremos ao invés usar uma View para gerar o retorno.

Nós iremos agora adicionar um View template para nosso projeto. Para fazer isso nós iremos posicionar o cursor dentro do método (ação) Index e então clicar com o botão direito e selecionar "Add View". Isto irá abrir a janela "Add View":

The 'Add View' dialog box is shown. It has a title bar with a close button. The 'View name' field contains 'Index'. The 'View engine' dropdown is set to 'Razor (CSHTML)'. There is a checkbox for 'Create a strongly-typed view' which is unchecked. Below it is a 'Model class' dropdown. The 'Scaffold template' dropdown is set to 'Empty'. To its right is a checked checkbox for 'Reference script libraries'. Below that is another unchecked checkbox for 'Create as a partial view'. Then a checked checkbox for 'Use a layout or master page:'. Below this is an empty text field for the layout name, followed by a button with three dots. Below that is the text '(Leave empty if it is set in a Razor _viewstart file)'. Then a label 'ContentPlaceHolder ID:' followed by a text field containing 'MainContent'. At the bottom right are 'Add' and 'Cancel' buttons.

A janela “Add View” permite-nos facilmente criar arquivos de templates (view templates). Por padrão a janela “Add View” já popula o nome da View de forma que ele seja o mesmo que método ação do controller. Como nós usamos o menu de contexto “Add View” dentro do método Index() do nosso HomeController a janela “Add View” pré-populou o nome da view com “Index” como padrão. Nós não precisamos mudar nenhuma das opções padrão oferecidas então clique no botão “Add”.

Quando nós clicarmos no botão “Add”, o Visual Studio irá criar um novo template para nós chamado Index.cshtml na pasta \Views\Home, criando as pastas do caminho caso não existam.



O nome e o caminho da pasta do arquivo “Index.cshtml” são importantes e seguem a convenção padrão de nomenclatura do framework ASP.NET MVC. O nome da pasta \Views\Home esta relacionada ao controller que é chamado HomeController. O nome do view template, Index, esta relacionado ao método ação do controller que retorna este view template, neste caso Index().

ASP.NET MVC nos permite evitar de ter que definir manualmente o nome ou o caminho do view template para retornar uma view quando nós usamos a convenção padrão de nomenclatura. Uma chamada ao método View() dentro do método ação Index() no nosso HomeController irá por padrão renderizar o view template \Views\Home\Index.cshtml como retorno.

```
public class HomeController : Controller
{
    //
    // GET: /Home/
    public ActionResult Index()
    {
        return View();
    }
}
```

```
}
```

Visual Studio cria e abre o view template “Index.cshtml” após nós clicarmos no botão “Add” na janela “Add View”. O conteúdo de “Index.cshtml” é exibido abaixo:

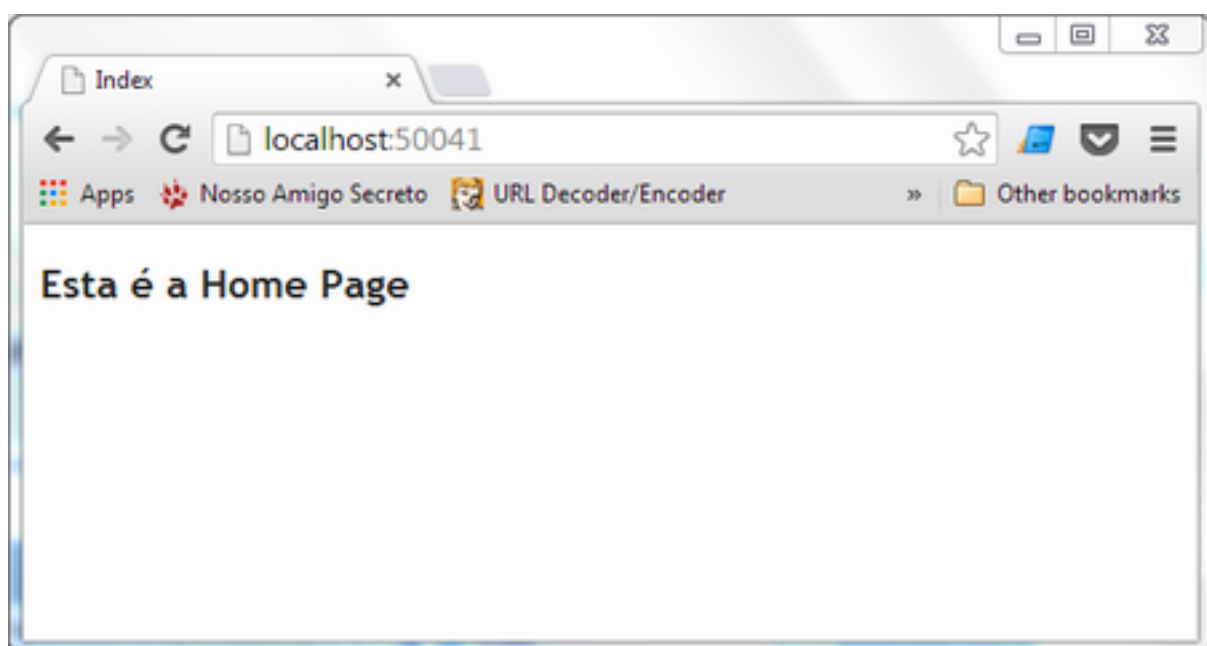
```
@{  
    ViewBag.Title = "Index";  
}  
<h2>Index</h2>
```

Esta view está usando a nova sintaxe de templates da Microsoft chamada Razor, que é mais concisa que o modelo de templates Web Forms usado em ASP.NET Web Forms e versões mais antigas de ASP.NET MVC. O renderizador de templates (engine) Web Forms continua disponível em ASP.NET MVC 3 mas muitos desenvolvedores acham Razor um mecanismo de template mais adequado ao desenvolvimento Web em ASP.NET MVC.

As primeiras três linhas definem o título da página usando o comando ViewBag.Title. Nós iremos ver como isso funciona em mais detalhes em breve mas primeiro vamos alterar o cabeçalho da página. Alterar o conteúdo da tag HTML <h2> para “Esta é a Home Page” como mostrado abaixo.

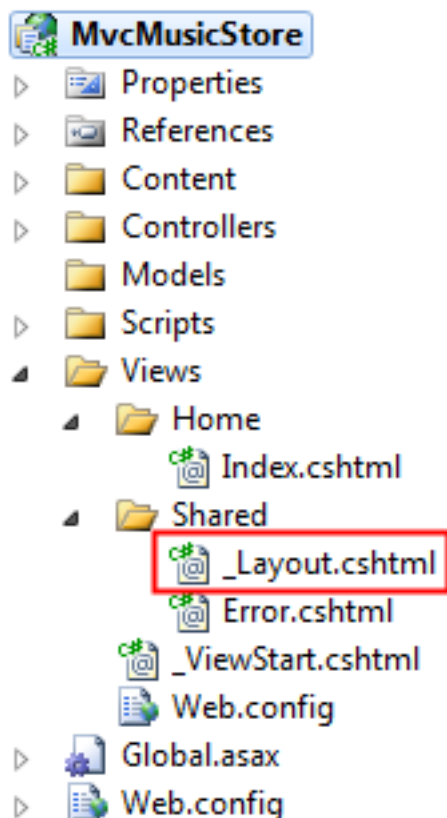
```
@{  
    ViewBag.Title = "Index";  
}  
<h2>Esta é a Home Page</h2>
```

Ao executar a aplicação o novo texto é exibido na nossa home.



Usando um layout padrão para elementos comuns do site

Muitos websites possuem conteúdo comum a diversas páginas, como menus, rodapés, cabeçalhos, logos, folhas de estilo, entre outras coisas. O mecanismo de template Razor possibilita gerenciar este conteúdo compartilhado de forma fácil usando uma página especial chamada `_Layout.cshtml` que foi automaticamente criada pra gente na pasta `Views\Shared`.



Clique duplo no arquivo `_Layout.cshtml` para ver seu conteúdo que é exibido abaixo

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>@ViewBag.Title</title>
  <link href="@Url.Content("~/Content/Site.css")"
rel="stylesheet" type="text/css" />
  <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
type="text/javascript"></script>
```

```

<script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")"
type="text/javascript"></script>
</head>
<body>
    @RenderBody()
</body>
</html>

```

O conteúdo dos nossos outros templates, como o Index.cshtml que criamos a pouco, será exibido pelo comando `@RenderBody()`, e qualquer conteúdo comum que a gente deseje que seja exibido pode ser adicionado acima ou abaixo do comando `@RenderBody()`. Na nossa loja de música nós iremos ter um cabeçalho comum com links da nossa página Home e vitrine para todas as páginas do site, então nós iremos adicionar este cabeçalho no `_Layout.cshtml` logo acima do `@RenderBody()`.

```

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/
css"/>
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"
type="text/javascript"></script>
</head>
<body>
    <div id="header">
        <h1>
            ASP.NET MVC MUSIC STORE</h1>
        <ul id="navlist">
            <li class="first"><a href="/" id="current">Inicio</a></li>
            <li><a href="/Vitrine/">Loja</a></li>
        </ul>
    </div>
    @RenderBody()
</body>
</html>

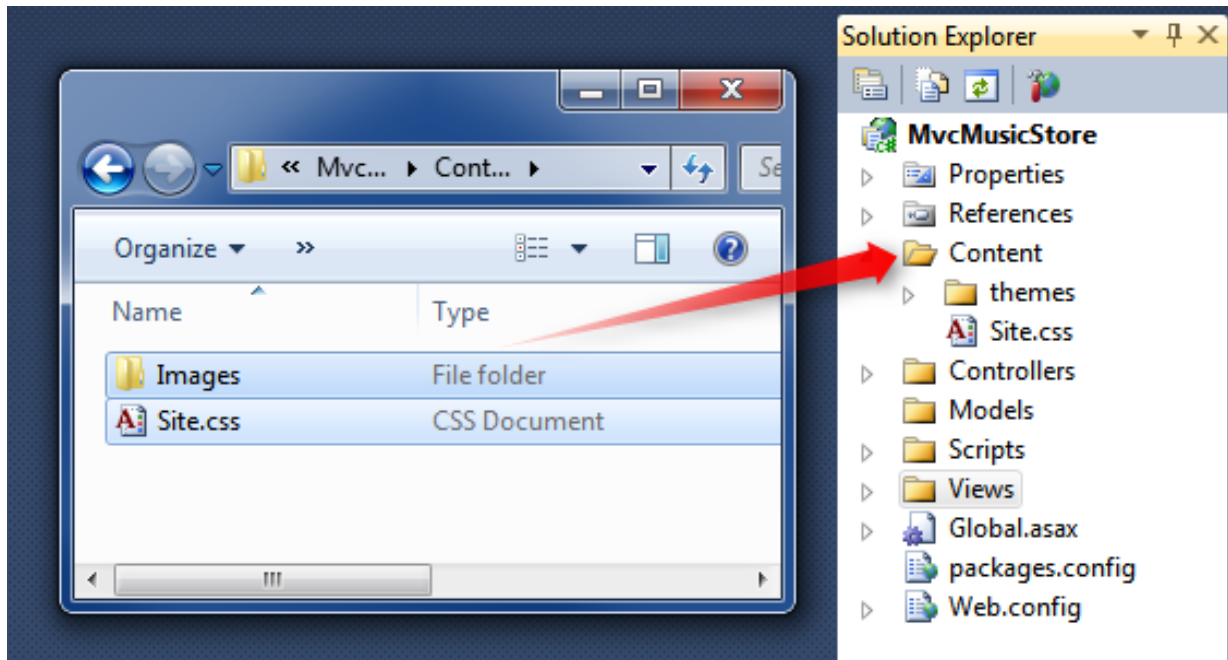
```

Atualizando a folha de estilos (CSS)

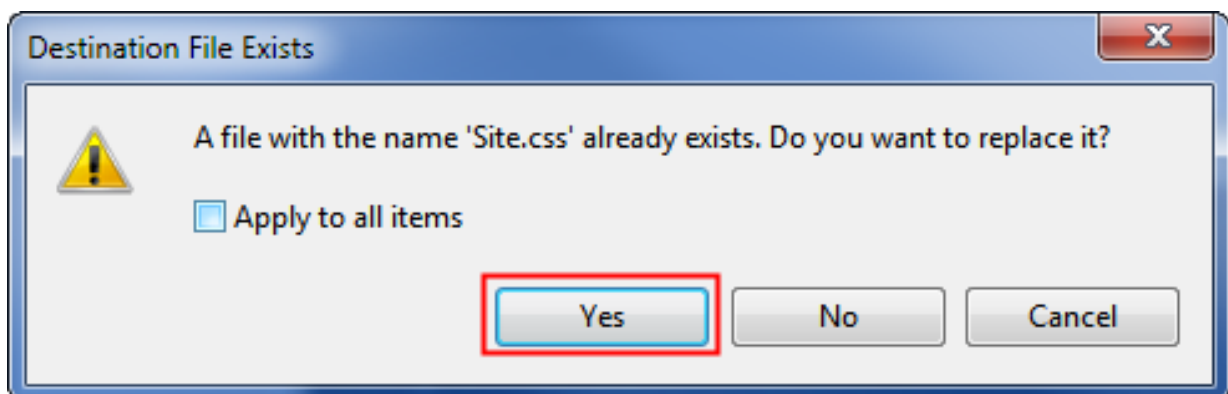
O template de projeto MVC vazio inclui um arquivo CSS bem simples que somente possui estilos usados para exibir mensagens de validação. Nosso designer criou um novo CSS com mais

definições de estilo e imagens para definir a identidade visual do nosso site e nós iremos adicionar isto ao nosso projeto agora.

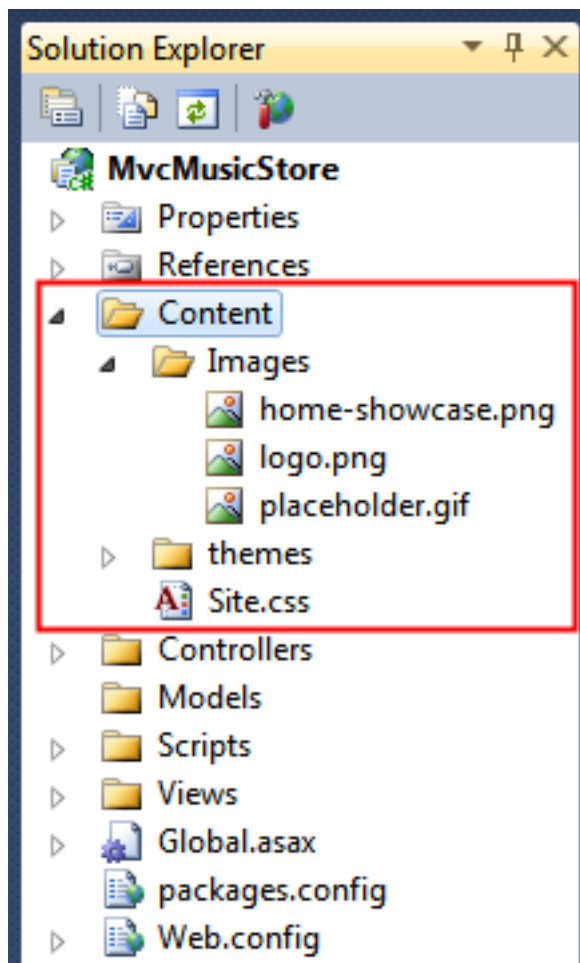
A nova versão do nosso CSS e as imagens associadas estão no material de apoio em <material de apoio>\MvcMusicStore-Assets\Content. Nós iremos copiar o arquivo **Site.css** e pasta **imagens** direto do Windows Explorer para nossa pasta Content na nossa solução.



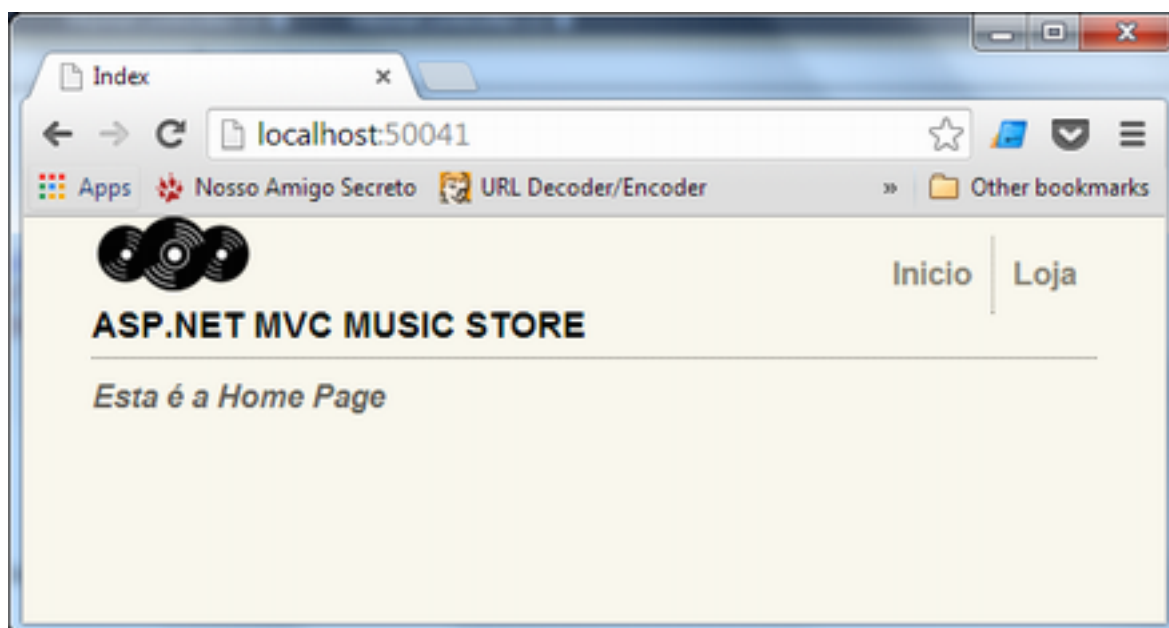
Você será perguntado se deseja sobrescrever os arquivos, diga que sim.



A pasta Content da nossa aplicação irá ficar como exibido abaixo:



Agora vamos executar nossa aplicação e ver como ficou a cara da nossa Home.



- Vamos revisar o que mudou: A ação Index do HomeController esta retornando o template `\Views\Home\Index.cshtml`, apesar de nosso código somente passarmos `"return View()"`. Isto funciona porque estamos usando as convenções de nomenclatura do ASP.NET MVC.
- A página Home esta exibindo uma simples mensagem que nós definimos dentro do template `\Views\Home\Index.cshtml`
- Nossa home esta usando o template `_Layout.cshtml` então a mensagem que nós definimos aparece dentro do layout padrão do site.

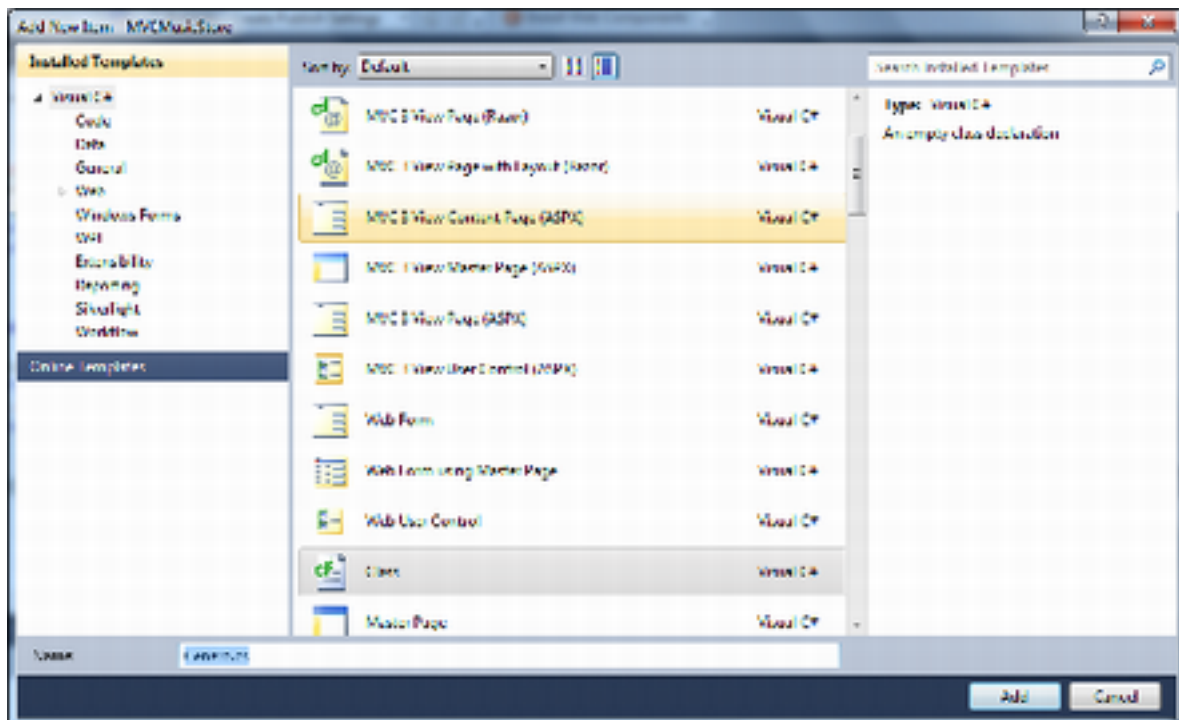
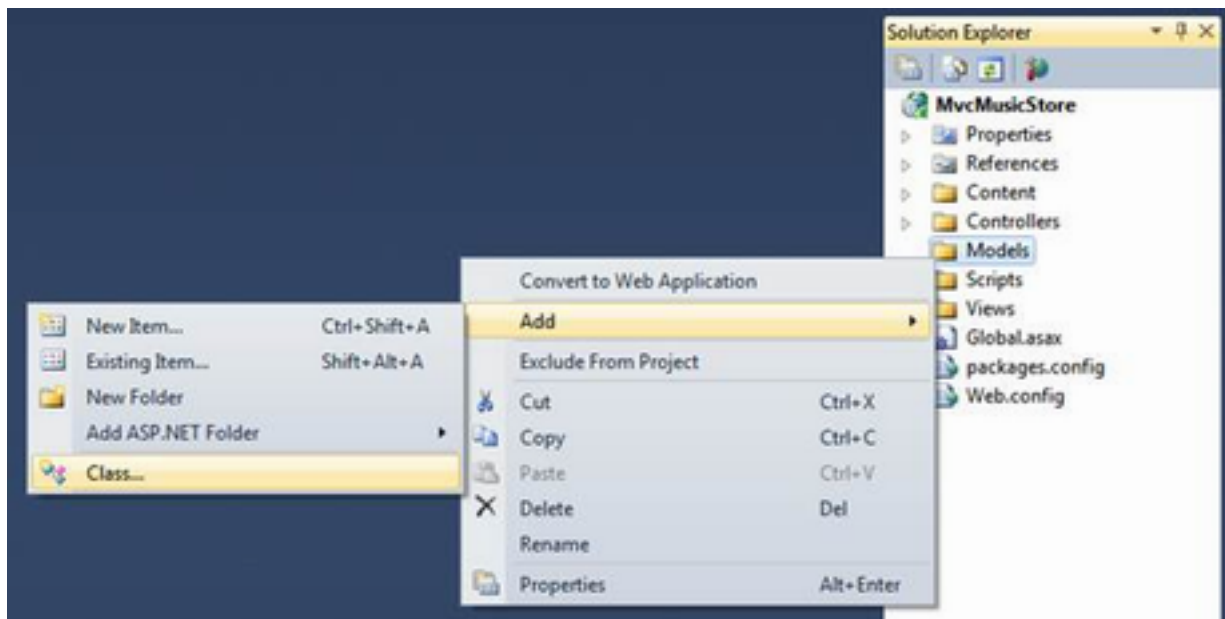
Usando um Model para passar informação para nossa View

Um template de view que exhibe somente HTML estático não faz um site muito interessante. Para criar um site dinamico, nós temos que passar informação das nossas ações do controller para nossos templates de view.

No padrão Model-View-Controller (MVC), o termo Model refere-se a objetos que representam os dados da aplicação. Em geral um model corresponde a dados de tabelas no banco de dados mas isto não é uma regra.

Métodos de ação do controller que retornam um ActionResult podem passar um objeto model para uma view. Isto permite que o controller coloque toda a informação necessária para gerar uma resposta e passe esta informação para um template de view para gerar a resposta HTML esperada. Tudo isso de uma maneira bem limpa, e separando a responsabilidade de cada camada. Isto é mais fácil de entender com um exemplo prático que vamos fazer a seguir.

Primeiro nós iremos criar algumas classes de Model para representar gênero e album dentro da nossa loja. Vamos começar criando a classe Gênero. Clique com o botão direito na pasta "Models" dentro do projeto, escolha a opção "Add Class", e nomeie o arquivo "Genero.cs".



Depois adicione uma propriedade `Nome` do tipo `string`, pública, para a classe que acabamos de criar:

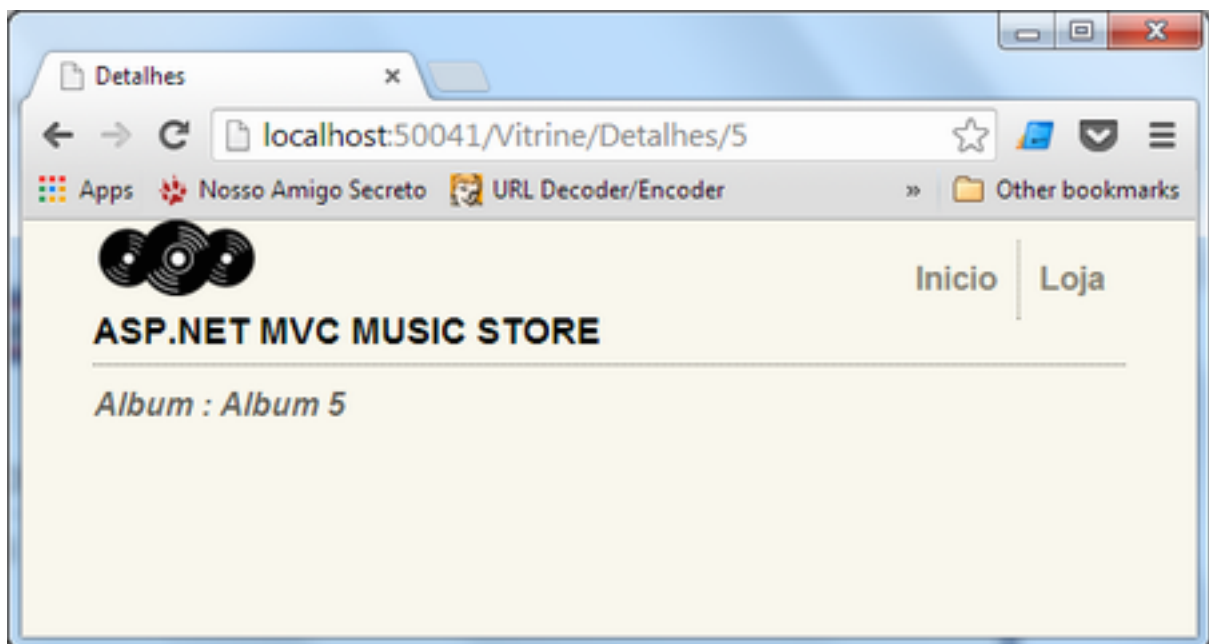
```
public class Genero
{
    public string Nome { get; set; }
}
```

Nota: Em caso de você estar pensando, a notação {get; set;} esta fazendo uso da funcionalidade de propriedades auto-implementadas do C#. Isto nos traz os beneficios das propriedades sem que nós tenhamos que declarar um campo para esta propriedade.

Em seguida repita os passos anteriores e crie uma classe chamada Album (arquivo Album.cs) que possui propriedades Título e Genero.

```
public class Album
{
    public string Titulo { get; set; }
    public Genero Genero { get; set; }
}
```

Agora nós podemos modificar a VitrineController para usar Views que exibem informação dinamica a partir do nosso Model. Se (para fins de demonstração somente) nós nomearmos nossos albums baseados no ID da requisição HTTP, nós poderíamos exibir esta informação como na view abaixo.



Nós iremos começar mudando o método de ação Detalhes da Vitrine de forma que ele mostre a informação de um único album. Adicione uma declaração “using” no topo da classe **VitrineControllers** para incluir o namespace MvcMusicStore.Models, desta forma nós não precisamos digitar o nome completo da classe (MvcMusicStore.Models.Album) cada vez que nós quisermos usar a classe album. A seção “usings” desta classe desta estar agora como abaixo.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```
using System.Web.Mvc;  
using MvcMusicStore.Models;
```

Em seguida nós iremos atualizar a ação `Detalhes` do controller de forma que ela retorna um `ActionResult` ao invés de uma string como nós fizemos com o método `Index` do `HomeController`.

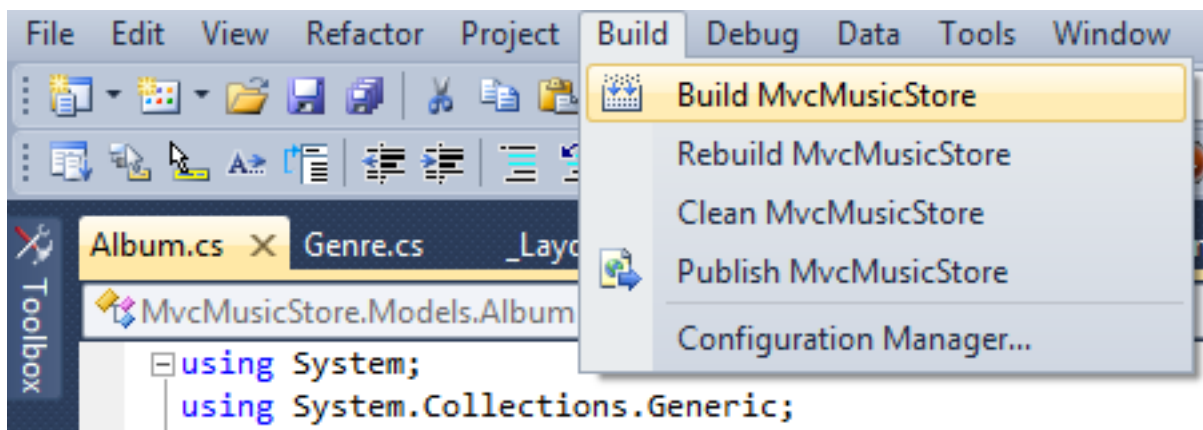
```
public ActionResult Details(int id)
```

Agora nós poderemos alterar a lógica para retornar um objeto `Album` para a view. Mais tarde neste tutorial nós iremos recuperar os dados do banco de dados mas por hora nós iremos trabalhar com dados falsos (fake/dummy) pra começar.

```
public ActionResult Details(int id)  
{  
    var album = new Album { Titulo = "Album " + id };  
    return View(album);  
}
```

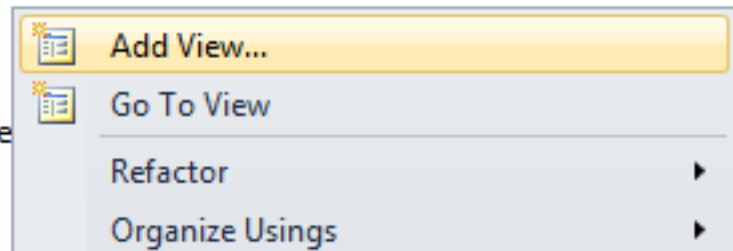
Nota: Se você não tem familiaridade com C# você pode achar que o tipo de dados "var" é um tipo dinâmico. Isto não é verdade pois isto é resolvido em tempo de compilação, a partir do objeto sendo atribuído a variável o compilador entende que a variável é do tipo `Album` e compila a aplicação fazendo esta substituição em tempo de compilação.

Agora vamos criar um template de View que usa nosso `Album` para gerar uma resposta HTML. Antes de nós fazermos isso nós precisamos construir o projeto para que a janela "Add View" saiba sobre nossa classe `Album` recém criada. Você pode construir o projeto selecionando o menu `Debug` -> `Build MvcMusicStore` (ou você pode executar isso via atalho com `Ctrl-Shift-B` para construir o projeto).



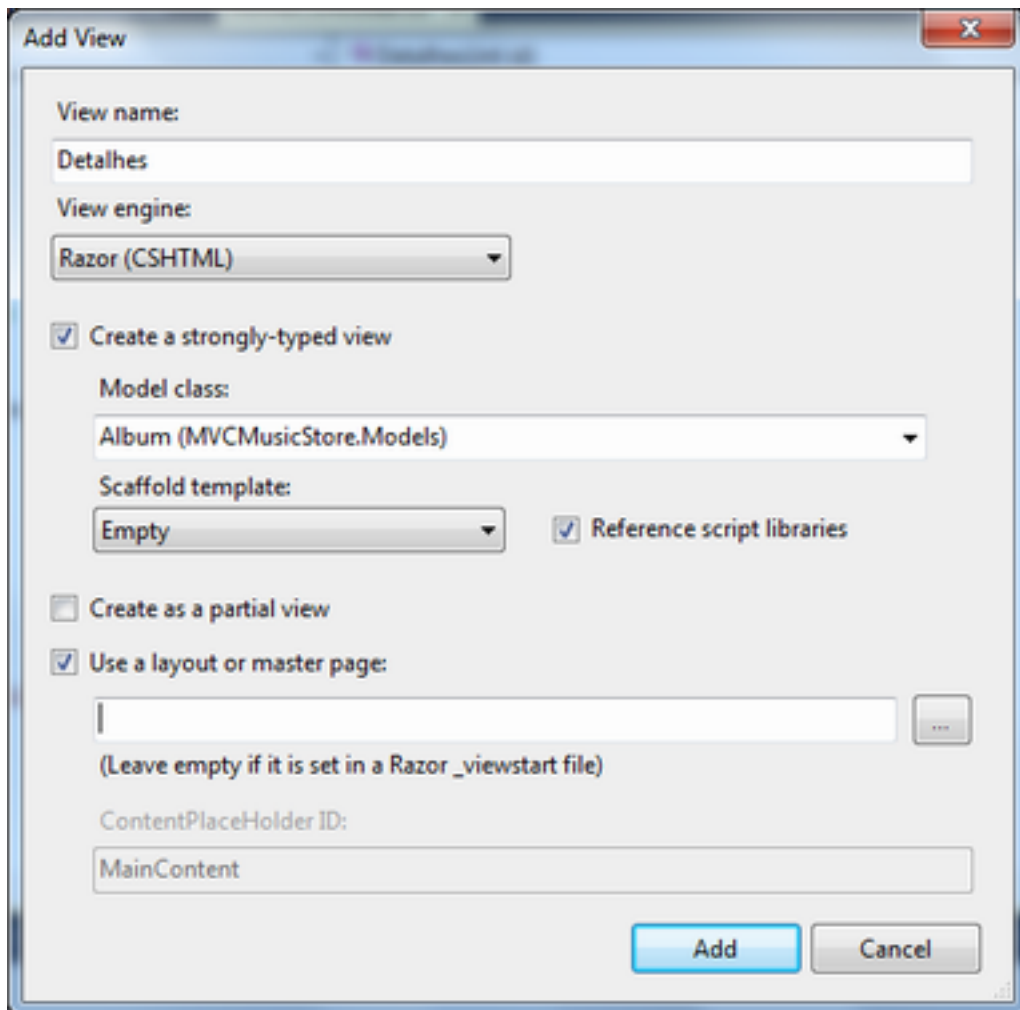
Agora que nós criamos e configuramos nossas classes de dados nós estamos prontos para construir nosso template. Clique com o botão direito dentro do método Details e selecione “Add View...” no menu de contexto.

```
//  
// GET: /Store/Details/5  
  
public ActionResult Details(int id)  
{  
    var album  
  
    return View  
}
```



Nós iremos criar um novo view template como nós fizemos antes como o HomeController. Como estamos criando o template a partir do StoreController ele será gerado por padrão no arquivo `\Views\Store\Index.cshtml`.

Diferente de antes, nós iremos marcar o checkbox “Create a strongly-typed” view. Nós iremos então selecionar nossa classe “Album” dentro da lista “View data-class”. Isto irá fazer com que a janela “Add View” crie um template que espera como entrada um objeto Album.



Quando nós clicamos no botão “Add” nosso template será criado em `\Views\Store\Detalhes.cshtml` com o código abaixo:

```
@model MvcMusicStore.Models.Album
@{
    ViewBag.Title = "Detalhes";
}
<h2>Detalhes</h2>
```

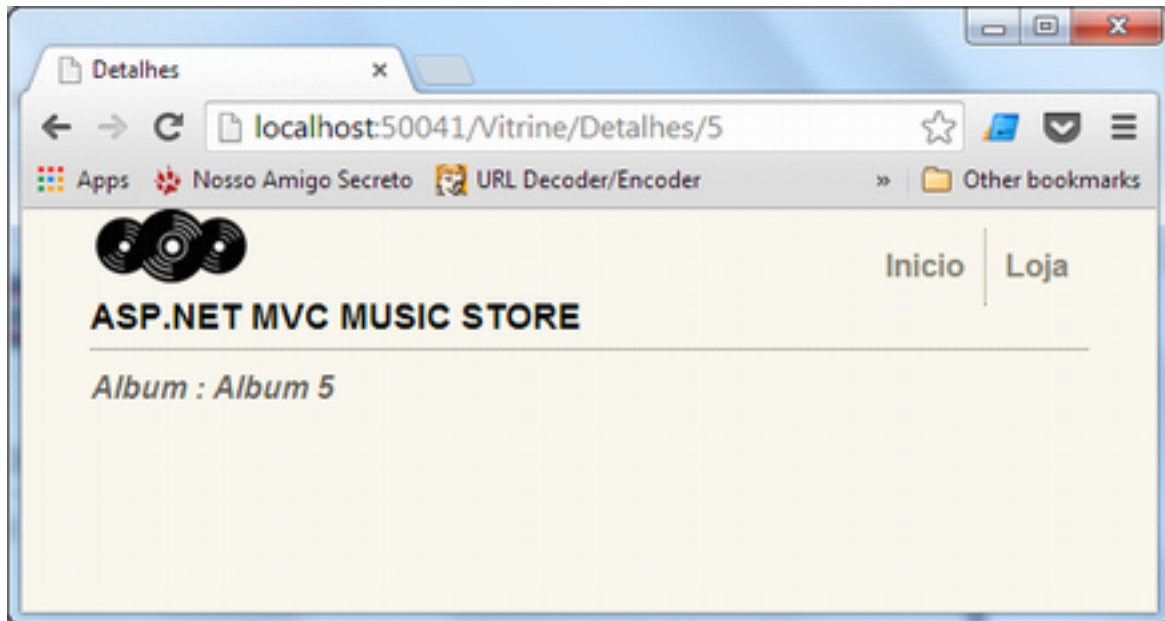
Observe a primeira linha, isto indica que esta view é fortemente tipada para nossa classe Album. A view engine Razor entende que esta view irá receber um objeto Album como tipo de dados, assim nós podemos facilmente acessar as propriedades deste Model além de ter o benefício de usar o IntelliSense (auto completção de código) no Visual Studio.

Altere a conteúdo da tag `<h2>` conforme abaixo para que ela exiba o valor da propriedade Titulo do Album.

```
<h2>Album: @Model.Titulo</h2>
```

Observe que o IntelliSense é acionado quando você digita ponto após a palavra chave @Model exibindo as propriedades e métodos que a classe Album possui.

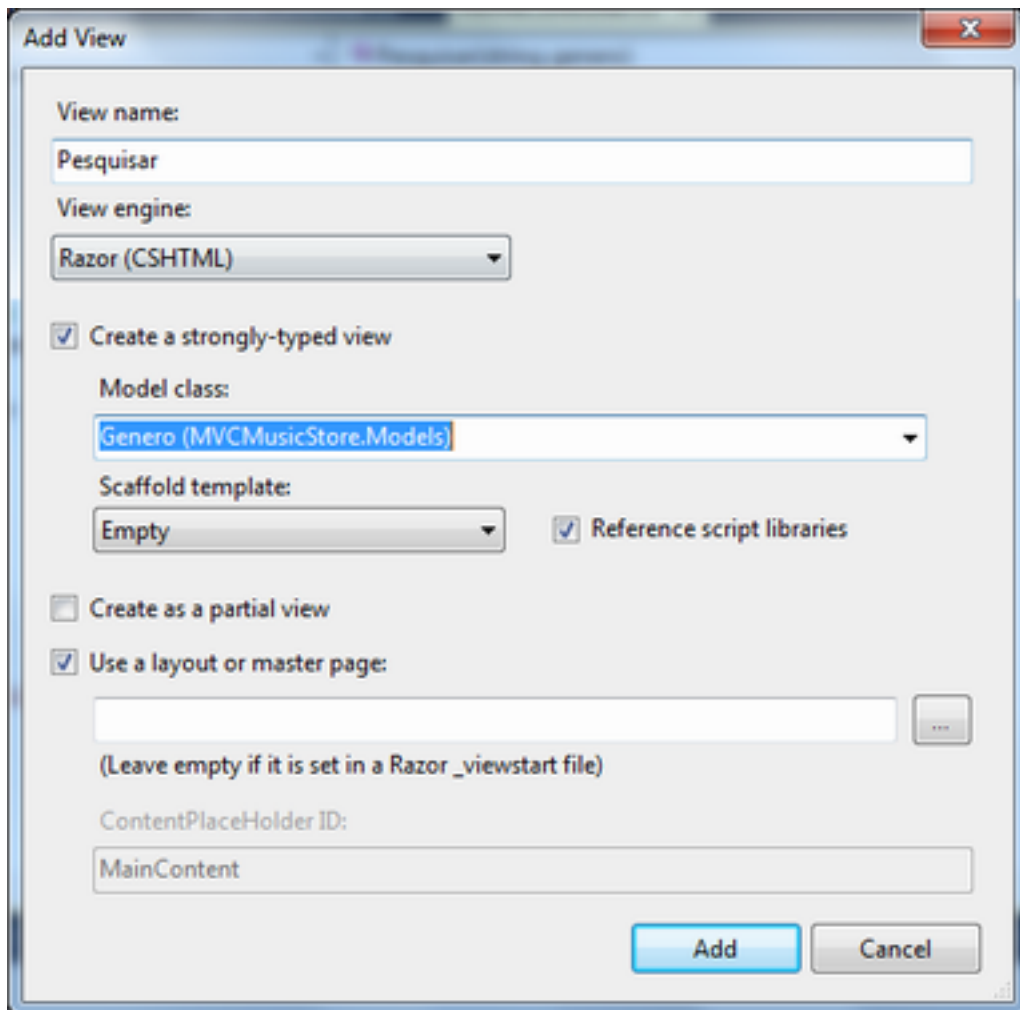
Agora vamos reexecutar nosso projeto e visitar a URL /Vitrine/Detalhes/5 . Nós iremos ver detalhes de um Album como abaixo.



Agora nós iremos fazer uma alteração similar no método de ação Pesquisar da Vitrine. Altere o método para que ele retorne um ActionResult, e modifique a lógica do método para que ele crie um novo objeto Genero e retorne para a view conforme abaixo.

```
public ActionResult Browse(string genero)
{
    var generoModel = new Genero { Nome = genero };
    return View(generoModel);
}
```

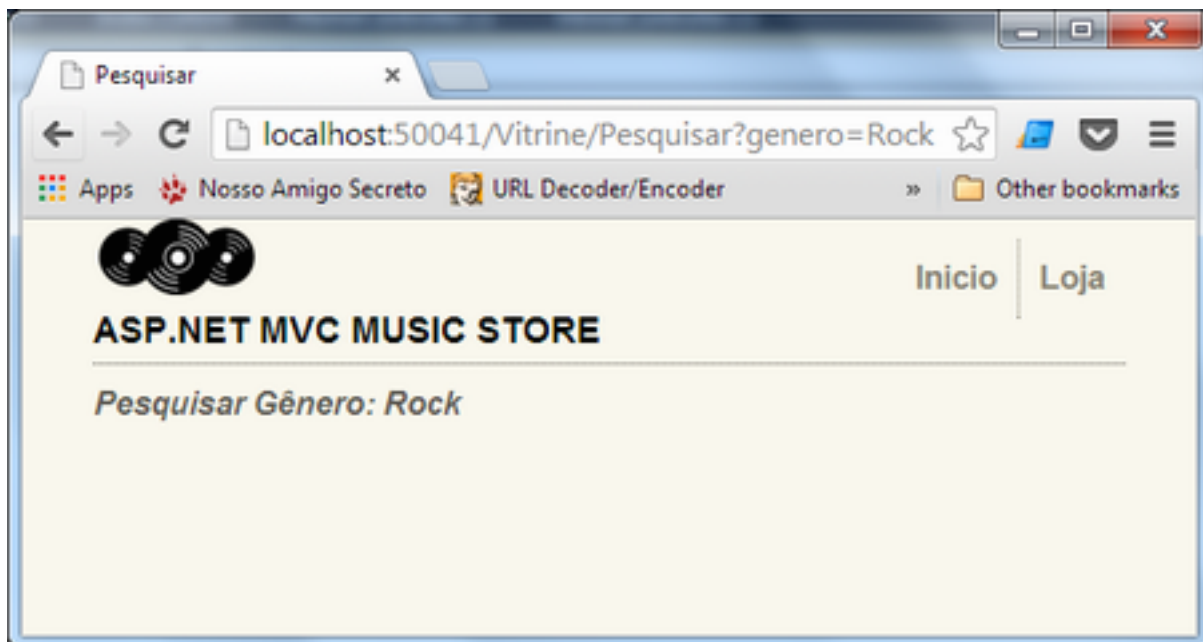
Clique com o botão direito no método Pesquisar e selecione “Add View...” a partir do menu de contexto, então adicione um view fortemente tipada com a classe Genero.



Altere o elemento `<h2>` no código da view (em `/Views/Store/Pesquisar.cshtml`) para que ele exiba informação sobre Genero.

```
@model MvcMusicStore.Models.Genero
@{
    ViewBag.Title = "Pesquisar";
}
<h2>Pesquisar Gênero: @Model.Nome</h2>
```

Agora vamos executar novamente o projeto e acessar a URL `/Vitrine/Pesquisar?Genero=Disco`. Nós iremos ver a página Pesquisar conforme abaixo.



Por último, vamos fazer uma alteração um pouco mais complexa ao método de ação Store Index e ao seu template para exibir uma lista de todos os generos de nossa loja. Nós iremos fazer isso usando uma lista de Genero como nosso model ao invés de um único objeto Genero.

```
public ActionResult Index()
{
    var generos = new List<Genero>
    {
        new Genero { Nome = "Disco"},
        new Genero { Nome = "Jazz"},
        new Genero { Nome = "Rock"}
    };
    return View(generos);
}
```

Clique com o botão direito no método de ação Store Index e selecione “Add View” como fizemos anteriormente, selecione Genero como classe de Model e aperte o botão “Add”.

Add View

View name:

View engine:

☒ Create a strongly-typed view

Model class:

Scaffold template:

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

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

ContentPlaceHolder ID:

Primeiro nós iremos mudar a declaração `@model` para indicar que esta view estara esperando vários objetos do tipo `Genero` ao invés de somente um. Mude a primeira linha de `/Vitrine/Index.cshtml` como abaixo:

```
@model IEnumerable<MvcMusicStore.Models.Genero>
```

Isto fala pro Razor (view engine) que o model desta view é uma coleção de objetos. Nós estamos usando `IEnumerable<Genero>` ao invés de `List<Genero>` porque é mais generico, permitindo que a gente mude mais tarde o tipo de dados do nosso model para qualquer objeto que implemente a interface `IEnumerable`.

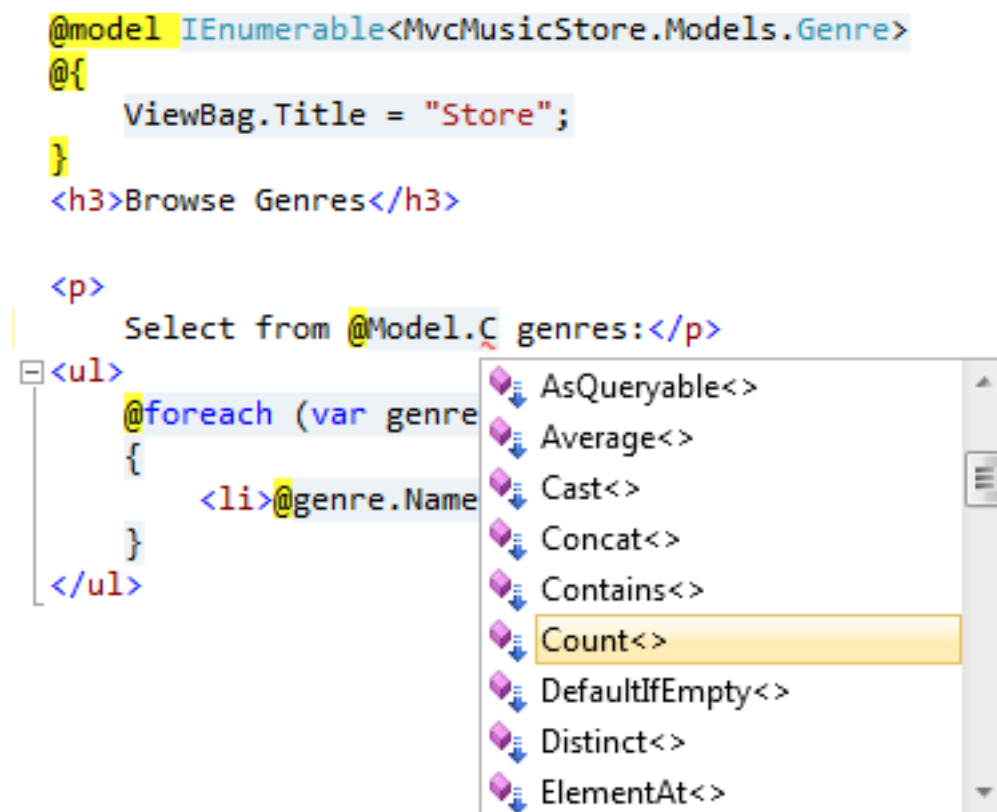
Em seguida nós iremos listar cada objeto `Genero` num loop como mostrado abaixo.

```

@model IEnumerable<MVCMusicStore.Models.Genero>
@{
    ViewBag.Title = "Loja";
}
<h3>Pesquisar Gênêros</h3>
<p> Escolha a partir @Model.Count() gêneros:</p>
<ul>
    @foreach (var genero in Model)
    {
        <li>@genero.Nome</li>
    }
</ul>

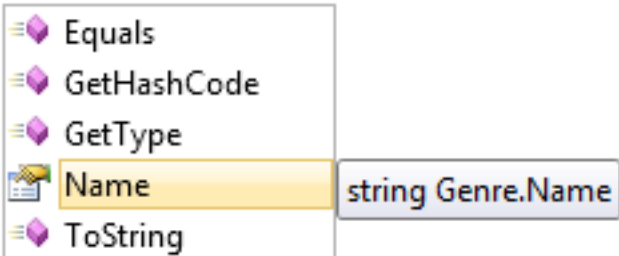
```

Perceba que nós temos suporte completo ao IntelliSense ao digitar este código, então quando nós digitamos “@Model” nós vemos todos os métodos e propriedades implementadas por uma classe IEnumerable do tipo Genero.



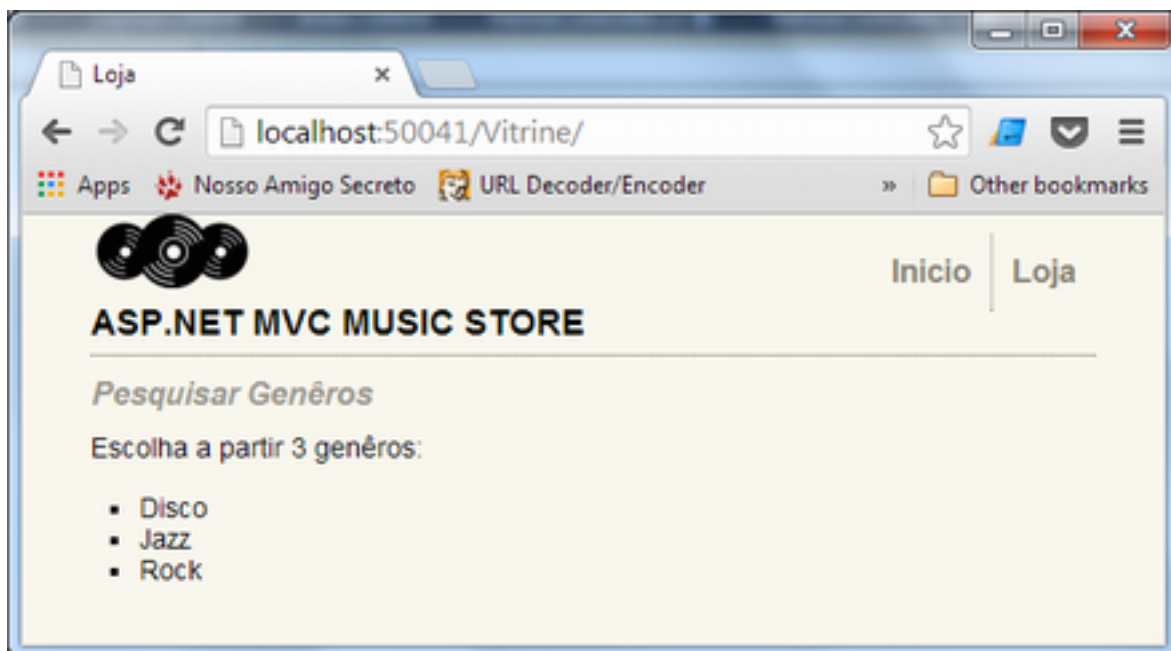
Dentro do nosso loop “foreach”, o Visual Studio sabe que cada item é do tipo Genero, então nós vemos o IntelliSense para cada tipo Genero.

```
<ul>
  @foreach (var genre in Model)
  {
    <li>@genre.</li>
  }
</ul>
```



Em seguida, a funcionalidade de scaffolding (funcionalidade que investiga um objeto e gera um conjunto de comportamentos) examina o objeto `Genero` e determina que cada um tem uma propriedade `Nome`, então ele gera automaticamente um código de listagem que exibe a propriedade `Nome` para cada um dos objetos. Esta funcionalidade também gera links/ações para `Edit`, `Details`, e `Delete` para cada item. Nós iremos tirar vantagem disso mais tarde no administrativo da nossa loja mas por hora nós só queremos ter um lista simples.

Quando nós executamos a aplicação e navegamos para `/Vitrine` nós vemos que tanto a contagem dos `Generos` e a lista dos `Generos` é exibida.



Adicionando links entre páginas

Nossa url `/Vitrine` que lista os generos atualmente lista somente os nomes dos generos no formato texto. Vamos mudar isso de forma que ao invés de somente texto nós iremos criar um link com o

nome do gênero apontando para URL /Vitrine/Pesquisar de forma que ao clicar num gênero de música como “Disco” nós iremos navegar para a URL /Vitrine/Pesquisar?genero=Disco. Nós podemos atualizar nossa view template em \Views\Vitrine\Index.cshtml para criar estes links usando o código abaixo (**não precisa digitar porque nós vamos fazer isso um pouco diferente**):

```
<ul>
    @foreach (var genero in Model)
    {
        <li><a href="/Vitrine/Pesquisar?genero=@genero.Nome">@genero.Nome</a></li>
    }
</ul>
```

Isto funciona mas isto poderia levar a um problema depois pois isto está fixo no código. Por exemplo se nós quizessemos renomear o Controller nós teríamos que buscar em todo o nosso código por links que teriam que ser alterados.

Uma alternativa que nós podemos usar é usar a ajuda de um método helper HTML. ASP.NET MVC inclui métodos helper que estão disponíveis a partir do código do nosso template de view para executar uma variedade de tarefas de formatação comuns como esta. O método helper `Html.ActionLink()` é um método que torna fácil construir um link HTML tomando cuidado para garantir que as URLs estão com o encoding correto.

`Html.ActionLink()` tem vários métodos de sobrecarga que permitem especificar tanta informação o quanto você precise para os seus links. No caso mais simples nós iremos fornecer somente o texto do link e o método de ação do controller a ser acionado quando o cliente clicar no link. Por exemplo, nós podemos fazer um link para o método `Index()` do controller `Store` na página de detalhes da loja com o texto “Vá para a home da loja” usando a seguinte chamada:

```
@Html.ActionLink("Vá para a home da loja", "Index")
```

Nota: Neste caso nós não especificamos o nome do controller porque nós estamos somente linkando para outra ação dentro do mesmo controller que está renderizando a view.

Nossos links para a página de busca (Pesquisar) irão precisar passar um parametro então nós iremos usar um outro método de sobrecarga de `Html.ActionLink` que recebe três parâmetros:

- 1. Texto do link que irá exibir o nome do gênero
- 2. Nome do método de ação do controller
- 3. Parametros usados no roteamento, especificando o nome (Genero) e o valor do parâmetro

Após tudo isso é assim que estes links serão definidos na nossa view Home da loja (Vitrine Index):

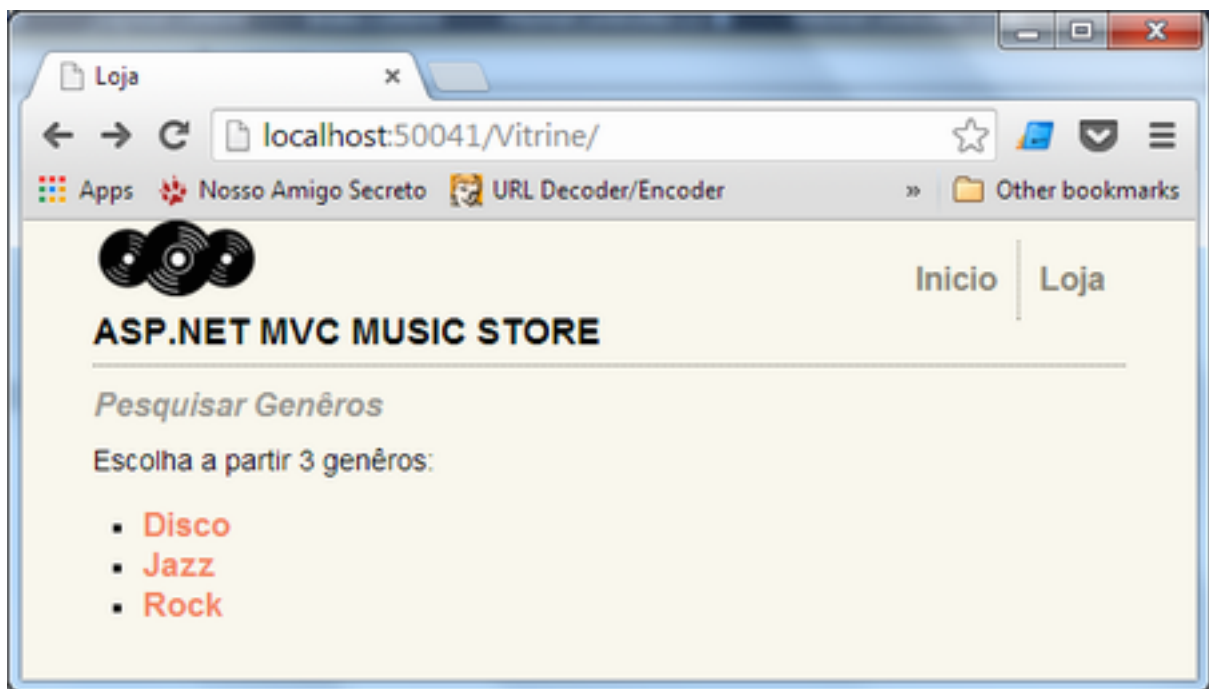
```
<ul>
    @foreach (var genero in Model)
```

```

    {
        <li>@Html.ActionLink(genero.Nome, "Pesquisar", new { genero =
genero.Nome })</li>
    }
</ul>

```

Agora quando nós executamos nosso projeto novamente e acessamos a url /Vitrine/ nós iremos ver uma lista de gêneros. Cada gênero é um hiperlink, quando clicamos neles o navegador ira abrir a url /Vitrine/Pesquisar?genero=<nome do genero>



O HTML para a lista de generos vai ficar assim:

```

<ul>
    <li><a href="/Vitrine/Pesquisar?genero=Disco">Disco</a>
</li>
    <li><a href="/Vitrine/Pesquisar?genero=Jazz">Jazz</a>
</li>
    <li><a href="/Vitrine/Pesquisar?genero=Rock">Rock</a>
</li>
</ul>

```