ASP.NET MVC em Ação

Jeffrey Palermo
Ben Scheirman
Jimmy Bogard

Prefácio de Phil Haack

Novatec

CAPÍTULO 1

Introdução ao ASP.NET MVC Framework

Este capítulo aborda:

- Execução do projeto iniciador.
- Progressão por meio de exemplos do tipo Hello World.
- Roteamento básico.
- Teste unitário básico.

Dependendo de quanto tempo vem construindo aplicações web na plataforma Microsoft, você vai se identificar com algumas ou com todas as "dores-de-cabeça" apresentadas em seguida. Na década de 1990, os desenvolvedores construíam websites interativos usando programas executáveis que rodavam em um servidor. Esses programas (Common Gateway Interface [CGI] era uma tecnologia comum naquela época) aceitavam uma requisição de rede e eram responsáveis por criar uma resposta em HTML. Templates eram criados sob demanda, e os programas eram difíceis de serem escritos, depurados e testados. No final da década de 1990, a Microsoft, após um breve flerte com os templates HTX e os conectores IDC, introduziu o Active Server Pages, ou *ASP*. O ASP trouxe os templates para as aplicações web.

A página de servidor era um documento HTML com scripts dinâmicos misturados nele. Apesar de isso ter sido um grande passo rumo a alternativas, o mundo logo presenciou páginas de servidor enormes com código indistinguível da marcação HTML.

No início de 2002, surgiram o ASP.NET e o framework Web Forms. O Web Forms foi uma mudança completa para os desenvolvedores ASP, em parte porque moveu a maioria da lógica do programa para um arquivo de classe (denominado code-behind) e substituiu a marcação HTML por controles de servidor dinâmicos escritos em uma sintaxe XML. Apesar de o desempenho ter aumentado e da experiência de depuração ter melhorado, novos problemas surgiram.

O novo ciclo de vida do evento de postback do lado do servidor causou uma explosão de atividade nos newsgroups, conforme desenvolvedores confusos procuravam por aquele evento *mágico*, no qual seriam adicionadas duas linhas simples de código necessárias para fazer a página funcionar como desejado. `ViewState`, apesar de ser bom na teoria, desmoronou conforme as aplicações aumentavam em complexidade.

Páginas simples ultrapassavam os 100KB de tamanho, já que o estado completo da aplicação tinha que ser armazenado na saída de cada página gerada. As boas práticas de desenvolvimento foram ignoradas quando ferramentas como o Visual Studio encorajaram a colocação de interesses relativos ao acesso de dados, como consultas SQL, dentro da lógica de visualização. Talvez o maior pecado do Web Forms tenha sido o forte acoplamento a tudo que existe dentro do namespace `System.Web`. Não havia esperanças de se realizarem testes unitários em qualquer código do arquivo *code-behind*, e atualmente encontramos métodos `Page_Load` que usam várias árvores para serem construídas. Apesar de as primeiras versões do Web Forms terem alguns reveses, o ASP.NET e, mais amplamente, o .NET Framework fizeram uma grande invasão no mercado de aplicações web. Hoje vemos grandes websites, como *CallawayGolf.com*, *Dell.com*, *Newsweek.com*, *WhiteHouse.gov* e *Match.com* rodando em ASP.NET. A plataforma colocou-se à prova no mercado e, quando combinado com o IIS executado no Windows, o ASP.NET pode suportar facilmente aplicações web complexas que executam em grandes data centers.

O framework ASP.NET MVC alavanca o sucesso do ASP.NET e das Web Forms, impulsionando o ASP.NET adiante como líder no espaço de desenvolvimento de aplicações web. O framework ASP.NET MVC foi introduzido para simplificar as partes complexas do desenvolvimento de aplicações Web Forms, enquanto retém o poder e a flexibilidade do pipeline do ASP.NET. A infraestrutura do ASP.NET e do pipeline de requisição, introduzidos no .NET 1.0, permanecem os mesmos, e o ASP.NET MVC fornece suporte para o desenvolvimento de aplicações ASP.NET usado o padrão de apresentação Modelo-View-Controlador. Os interesses relativos ao modelo de dados, à lógica da aplicação e à apresentação dos dados são claramente separados, com a lógica da aplicação mantida em uma classe isolada das dependências fortes relacionadas com a apresentação dos dados. Páginas de servidor tornam-se simples visualizações (views), que não passam de templates HTML esperando para serem povoados com objetos (modelos) passados a elas pelo controlador. O ciclo de vida do evento postback deixa de existir, e a `ViewState` não é mais necessária. Neste capítulo, vamos avançar rumo às suas primeiras linhas de código construídas sobre o ASP.NET MVC Framework. Após essa introdução, você estará pronto para tópicos mais avançados.

Neste capítulo, e ao longo de todo o livro, assumimos que o leitor esteja familiarizado com o ASP.NET. Se você for novo no ASP.NET, por favor, familiarize-se com o pipeline de requisição do ASP.NET, bem como com o tempo de execução do .NET. Ao longo deste capítulo, conduziremos você por meio da criação de um projeto de aplicação web com o ASP.NET MVC Framework, criando suas primeiras *rotas*, *controladores* e *views*. Vamos esmiuçar a aplicação-padrão e explicaremos cada parte dela. Em seguida, vamos estendê-la, e você criará seu primeiro controlador e sua primeira view. Inicialmente, vamos explorar o padrão MVC e o template da aplicação padrão fornecido com o framework.

Integrando com aplicações ASP.NET Web Forms ou migrando a partir delas

Podemos criar telas que usem o ASP.NET MVC Framework, ao passo que outras telas continuam trabalhando com Web Forms? É claro que podemos. Eles podem ser executados lado a lado até que toda a aplicação use o MVC. Usar o framework MVC não é uma proposição do tipo "tudo ou nada". Existem inúmeras aplicações ASP.NET em produção que usam Web Forms. Se uma equipe de software deseja migrar a aplicação de Web Forms para ASP.NET MVC, é possível fazer uma migração por etapas e executar os dois, lado a lado, no mesmo domínio de aplicação. O ASP.NET MVC não substitui as bibliotecas e funcionalidades centrais do ASP.NET. Em vez disso, ele agrega às funcionalidades existentes do ASP.NET. O UrlRoutingModule, que registramos no arquivo de configuração de rede, avalia um URL entrante, relativo às rotas existentes. Se uma rota correspondente não for encontrada, o ASP.NET seguirá em frente e usará o Web Forms para preencher a requisição, então é muito simples misturar e compatibilizar funcionalidades durante uma migração ou com o propósito de estender a aplicação.

Apesar do fato de que as Web Forms não sumirão tão cedo, acreditamos que controladores, ações e views serão a maneira preferida de se escrever aplicações ASP.NET daqui por diante. Mesmo com a Microsoft mantendo o suporte a ambas as opções (e o desenvolvimento da próxima versão do Web Forms continua ativo), acreditamos que o ASP.NET MVC Framework será favorecido em detrimento do Web Forms, da mesma forma que vemos o C# sendo favorecido em detrimento do VB na documentação em conferências da indústria e nos livros técnicos.

1.1 Desmembrando a aplicação-padrão

Nesta seção, explicaremos o que é o padrão MVC e criaremos nossa primeira aplicação web ASP.NET MVC. Concentraremo-nos primeiro no controlador, pois, na tríade Modelo-View-Controlador (MVC), o controlador está no comando e é responsável por decidir quais objetos-modelo usar e quais views transmitir de volta para o usuário. O controlador está a cargo da coordenação e é o primeiro a ser executado quando a requisição web chega à aplicação. O controlador é responsável por decidir qual resposta é apropriada para a requisição.

O padrão modelo-view-controlador não é novo. Um princípio central do padrão MVC é separar a lógica de controle e a view, ou seja, uma tela. Uma view é responsável apenas por apresentar a interface de usuário. Ao separar a lógica de domínio e desacoplar da view, o acesso aos dados e a outras chamadas a interface de usuário (UI) pode permanecer igual, mesmo quando a lógica e o acesso aos dados mudam dentro da aplicação. A figura 1.1 apresenta um diagrama simples da tríade MVC.

Perceba que o controlador possui uma relação direta com a view e o modelo, mas o modelo não precisa saber nada a respeito do controlador ou da view. A requisição

web será tratada pelo controlador, e este decidirá quais objetos-modelo usar e quais objetos-view transmitir.

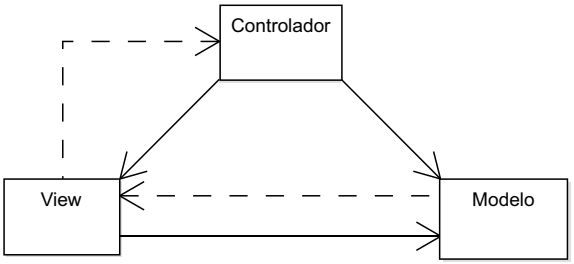

Figura 1.1 – Um diagrama simples para descrever o relacionamento entre o Modelo, a View e o Controlador. As linhas sólidas indicam uma associação direta, e as linhas tracejadas indicam uma associação indireta (gráfico e descrição usados com permissão da Wikipédia).

Para começar, abriremos o Visual Studio 2008 SP1 e criaremos nosso projeto. A edição do Visual Studio faz diferença. Apesar de existirem maneiras de utilizar o ASP.NET MVC Framework sem o SP1, `System.Web.Abstractions.dll` e `System.Web.Routing.dll` fazem parte do GAC (cache de assembly global) no SP1. Você pode usar o Visual Studio 2008 Professional, um Team Edition SKU, ou o Visual Web Developer Express SP1. Perceba que o ASP.NET MVC Framework é construído sobre o Web Application Projects, e, apesar de ser possível fazê-lo funcionar com websites, a experiência de desenvolvimento é otimizada para uso com o Web Application Projects.

NOTA Você já deve ter o Visual Studio 2008 SP1 ou Visual Web Developer 2008 SP1, o .NET 3.5 SP1 e o ASP.NET MVC Framework devidamente instalados para prosseguir. O framework MVC é um pacote independente construído sobre o .NET 3.5 Service Pack 1. Os exemplos deste livro usarão o Visual Studio 2008 SP1, mas você encontrará informações sobre como usar o Visual Web Developer Express 2008 SP1, que é gratuito, no website do ASP.NET MVC: *http://www.asp.net/mvc*

Começaremos no Visual Studio 2008 Professional SP1 criando um novo projeto ASP.NET MVC Web Application. Ao abrir a caixa de diálogo New Project, certifique-se de que o .NET Framework 3.5 esteja selecionado. Se você tiver o .NET Framework 3.0 ou 2.0 selecionado, o Visual Studio filtrará a lista e você não verá o template de projeto para ASP.NET MVC Web Application. Agora que você entende o básico do padrão e como instalar o framework MVC, mergulharemos no nosso primeiro projeto.

1.1.1 Criando o projeto

Criar o seu primeiro projeto ASP.NET MVC Web Application será uma das coisas mais simples que você fará neste capítulo. No Visual Studio 2008, quando o .NET Framework 3.5 está selecionado como framework-alvo, você verá um novo template de projeto chamado ASP.NET MVC Web Application. Escolha esse template. O novo diálogo de projeto se parecerá com o mostrado na figura 1.2.

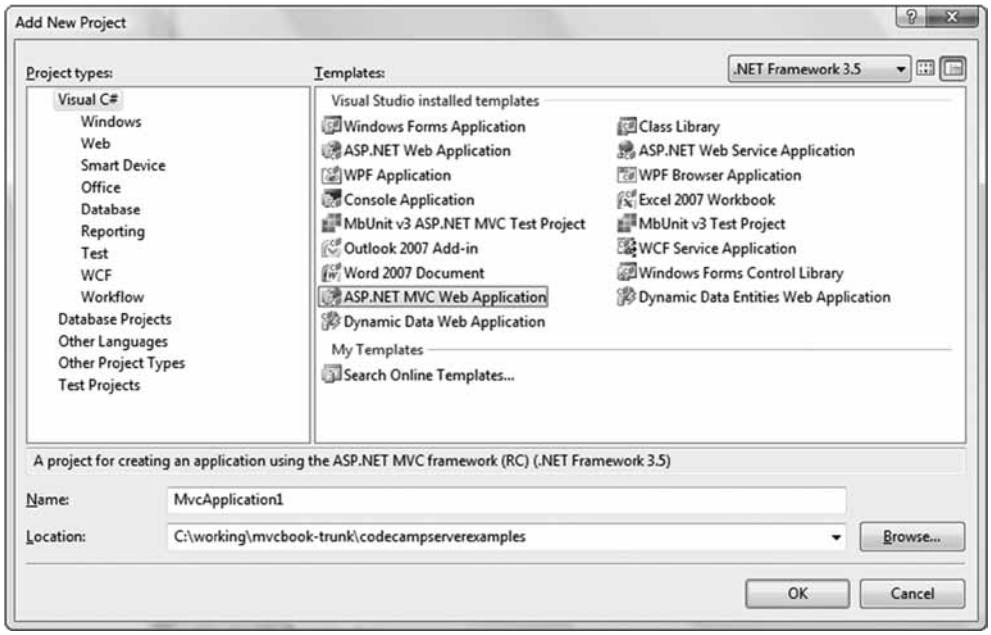

Figura 1.2 – O projeto MVC Web Application é um template de projeto adicionado às seções C# e VB.NET do diálogo New Project. Ele só fica disponível quando você tem o .NET Framework 3.5 selecionado como o framework-alvo.

NOTA Outros starter kits e aplicações de exemplo da comunidade do ASP.NET MVC estão disponíveis no website do ASP.NET. No momento em que escrevemos, *http://www.asp.net/community/projects/* e *http://www.asp.net/mvc/* possuem vários starter kits e aplicações de exemplo para iniciar projetos ASP.NET MVC (bem como starter kits para ASP.NET Web Forms). As opções incluem:

- Kigg Starter Kit – uma aplicação semelhante ao Digg.
- Contact Manager Sample Application.
- Storefront Starter Kit.

Apesar dos starter kits serem bem simples, você também deveria ver starter kits mais completos, como aqueles encontrados em *http://CommunityForMvc.net*. Este site contém um template vazio e outro completo com MvcContrib, StructureMap, NHibernate, NUnit, NAnt, AutoMapper, Tarantino, Naak, NBehave, Rhino Mocks, WatiN, Gallio, Castle, 7zip e outros.

Se estiver começando a desenvolver em .NET, você deve se familiarizar primeiro com o template padrão da Microsoft. Somente após isso, passe a usar um starter kit mais robusto ou as aplicações de exemplo fornecidas pela comunidade para ter um ponto de partida mais avançado. Quando você tiver dominado o framework, considere a possibilidade de contribuir com seus próprios starter kits.

A primeira coisa a se observar é que, em contraste com as estruturas praticamente vazias de um template de projeto padrão do Web Forms, o template MVC padrão cria várias pastas: *Content*, *Controllers*, *Models*, *Scripts* e *Views*. Essas pastas representam parte das convenções de desenvolvimento de aplicações MVC, que, se forem seguidas, podem transformar em um passeio a experiência de desenvolvimento com o framework.

Por enquanto, as pastas mais importantes para você, e com as quais você deve se familiarizar, são as nomeadas de acordo com as suas contrapartes no padrão MVC (as pastas *Model*, *View* e *Controller*). Como era de se esperar, o propósito de cada uma delas é fornecer uma separação lógica das três áreas de interesse do MVC, bem como aproveitar uma funcionalidade interessante do Visual Studio que coloca as classes criadas dentro de namespaces, nomeados de acordo com as pastas em que elas são inseridas.

A pasta *Controller* é provavelmente a menos interessante. Esta deve conter apenas classes que serão usadas como controladores ou classes-base e interfaces herdadas pelos controladores. A pasta *Views* é especial porque conterá o código que provavelmente será o mais familiar de todos para os desenvolvedores de Web Forms. A pasta *Views* contém as páginas aspx (views), ascx (views parciais) e as páginas-mestras usadas para apresentar os dados. Normalmente, você terá uma pasta para cada controlador dentro da pasta *Views,* contendo as views que serão usadas por um controlador específico, bem como uma pasta *Shared,* que conterá views compartilhadas.

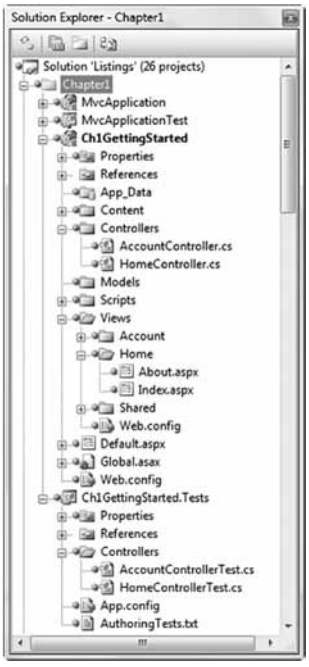

Figura 1.3 – A estrutura-padrão de um projeto de aplicação web com o ASP.NET MVC Framework usa convenções para a disposição de arquivos.

O caminho feliz

Desenvolvedores ASP.NET (e outros desenvolvedores que usam frameworks centralizados em convenções) costumam mencionar o Caminho Feliz. Isso é uma referência à noção de que as convenções do framework MVC tornarão a experiência do desenvolvedor mais prazerosa e relativamente indolor. O framework MVC não exige que você siga qualquer convenção em particular, mas quanto mais se distanciar do Caminho Feliz, maior será o esforço solicitado do desenvolvedor. O projeto MvcContrib aperfeiçoa o caminho, e com certeza você encontrará maneiras de aperfeiçoá-lo em seu sistema. Permanecer no caminho lhe dá um ganho enorme em consistência.

Para a maioria dos projetos não triviais, você provavelmente não colocará seus modelos na pasta *Models*. De forma geral, é uma boa prática manter o seu modelo de domínio em um projeto separado para que outras aplicações possam usá-lo sem gerar uma dependência sobre sua aplicação MVC. Recomendamos que coloque apenas os interesses relativos à apresentação no projeto Web Application.

No projeto-padrão, você deve estar familiarizado com o arquivo *Default.aspx* que lhe é fornecido, e discutiremos em breve por que ele está ali. Primeiro, precisamos entender o conceito de *rota*.

1.1.2 Suas primeiras rotas

As rotas serão discutidas com mais detalhes no capítulo 5; no entanto, você precisa saber o básico de rotas para acompanhar esta seção. Apesar de o Web Forms ordenar uma convenção estrita de URLs, o framework MVC fornece um mecanismo que permite aos desenvolvedores manusear os URLs e fazê-los mapear automaticamente para um objeto no sistema, que possa tratar a requisição web entrante. O roteamento foi adicionado ao ASP.NET no .NET Framework 3.5 Service Pack 1 e está disponível para todas as aplicações ASP.NET. O arquivo *Global.asax.cs* contém rotas básicas que são fornecidas com o projeto MVC Web Application para ajudá-lo a começar o trabalho. Antes de continuarmos, devemos definir o que é uma rota.

Uma rota é uma definição completa de como despachar uma requisição web para um controlador, normalmente usando `System.Web.Mvc.MvcRouteHandler`. Antigamente, tínhamos pouco controle sobre o despacho de mensagens, sem depender de ferramentas externas como filtros ISAPI ou `HttpModules`, cuidadosamente construídos para reescrever URLs. Com o Web Forms, o URL da requisição web estava fortemente acoplado à localização da página de que estava tratando a requisição. Se a página fosse nomeada `Foo.aspx` em uma pasta chamada `Samples`, o URL certamente seria algo como *http://MvcContrib*. Várias equipes recorreram à técnica de reescrever URLs para conseguir algum controle sobre os URLs e como eram produzidos. Com o ASP.NET MVC

Framework, e com o ASP.NET 3.5 em geral, as rotas são cidadãs de primeira classe que podem ser gerenciadas diretamente na aplicação web. Começamos definindo como queremos estruturar os nossos URLs. O template de projeto nos dá algumas rotas para começarmos, como mostra a listagem 1.1.

⇨ **Listagem 1.1 – Rotas-padrão de um projeto novo**

```
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespace Ch1GettingStarted
{
  public class MvcApplication : HttpApplication
  {
    public static void RegisterRoutes(RouteCollection routes)
    {
      routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
      routes.MapRoute(
        "Default", ◄ Nome da rota
        "{controller}/{action}/{id}", ◄ URL com parâmetros
        new {controller = "Home", action = "Index", id = ""} ◄ Valores-padrão dos parâmetros
      );
    }
    protected void Application_Start ()
    {
      RegisterRoutes(RouteTable.Routes);
    }
  }
}
```

As rotas devem ser definidas antes que qualquer requisição seja recebida pela aplicação, então o template de projeto adiciona as rotas ao método `Application_Start` no arquivo `Global.asax.cs`. Mais adiante, você verá que não deixamos as rotas nesse lugar, excetuando as aplicações web mais triviais.

NOTA Seguiremos as velhas boas práticas da separação de interesses (*Separation of Concerns*, ou SoC) e do princípio da responsabilidade única, ou SRP (*Single Responsibility Principle*), transportando as rotas para uma localização dedicada e separada por uma interface. Aprofundaremos esses princípios mais tarde, mas, resumindo, o interesse (ou responsabilidade) do método `Application_Start` é iniciar as operações que devam acontecer no início da vida da aplicação. A abordagem responsável é evitar realizar qualquer tipo de trabalho que deva acontecer no início. Qualquer operação que deva acontecer no início da aplicação deveria residir em classes separadas, que seriam apenas invocadas na ordem apropriada dentro do método `Application_Start`.

Perceba que a parte da rota equivalente ao URL é simplesmente um mecanismo de correspondência da requisição. Se o URL corresponde a uma rota específica, nós especificamos qual controlador deverá tratar a requisição e qual método-ação deverá ser executado. Você pode criar quantas rotas quiser, mas uma rota já é fornecida para você. Essa rota segue o template `{controller}/{action}/{id}`.

A rota cujo template é `{controller}/{action}/{id}` é uma rota genérica e pode servir para inúmeras requisições web. Tokens são denotados pela inclusão de `{chaves}`, e a palavra englobada pelas chaves corresponde a um valor que o framework MVC compreende. Os valores mais comuns que nos interessarão são `controller` e `action`. O valor de rota `controller` é um valor especial que a classe `System.Web.Mvc.MvcHandler` usa para invocar a interface `IControllerFactory`. Esta será a rota que usaremos no restante do capítulo, então ficaremos satisfeitos com um URL no formato *http://MvcContrib.org/nome_controlador/nome_ação*. O manipulador básico de rotas será uma instância de `IRouteHandler` denominada `MvcRouteHandler`. Temos controle total e poderíamos fornecer nossa própria implementação de `IRouteHandler` se quiséssemos, mas deixaremos isso para um capítulo posterior.

Antes de criarmos nosso primeiro controlador, vamos examinar o que há de diferente no arquivo *web.config* em um projeto MVC Web Application. As diferenças são fáceis de detectar.

Apenas procure por "Routing" ou "MVC". Uma diferença visível é que um novo `IHttpModule` é registrado no arquivo de configuração. Observamos a existência do `UrlRoutingModule` na listagem 1.2.

⇨ **Listagem 1.2 – Adição exclusiva ao arquivo web.config**

```
<add name="UrlRoutingModule" type="System.Web.Routing.UrlRoutingModule,
  System.Web.Routing, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
```

O `UrlRoutingModule` avalia uma requisição e verifica se ela corresponde a alguma rota armazenada na `RouteTable`. Se a rota corresponder, ele redefine o manipulador-padrão (`IHttpHandler`) no caso dessa requisição, de forma que o framework MVC acabe por manipular a requisição. Vamos examinar nosso primeiro controle como um meio de tratar uma rota para o URL `/home`. Na próxima seção veremos como todas as peças do projeto iniciador (starter) se encaixam.

1.1.3 Adotando o projeto iniciador

Vamos passar rapidamente pelo projeto iniciador, observando cada porção de código fornecida. Isso servirá de exemplo sobre como juntar o código ao escrever uma aplicação com a camada de apresentação fornecida e potencializada pelo ASP.NET MVC Framework. Antes de observarmos o código, execute a aplicação web pressionando CTRL + F5 e você deverá ver uma tela que lembra a figura 1.4.

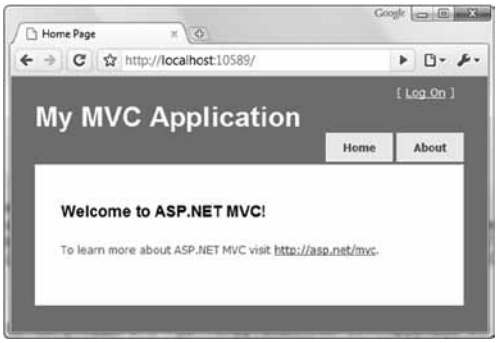

Figura 1.4 – O projeto iniciador vem com um layout básico e CSS.

O projeto iniciador vem com alguma navegação, um Log On e conteúdo. O CSS fornece formatação simples sobre o XHTML. Perceba que o URL na barra de endereço é `/`. `"/home"` também trará a mesma página, já que nossa rota especifica "home" como o controlador-padrão. Esse URL não tem uma extensão, portanto, se estiver planejando executar sua aplicação no IIS 6, deverá adicionar um mapeamento-curinga ou instalar um filtro ISAPI que forneça essa funcionalidade. A implantação no IIS 6 será vista com mais detalhes no capítulo 10.

Já que você está se familiarizando com o pipeline de requisição do ASP.NET, revisaremos rapidamente como essa requisição encontra seu caminho até um controlador ASP.NET MVC. Aqui está um esboço de como a requisição caminha pelo ASP.NET até o controlador e, em seguida, pela view:

1. A requisição chega em */Home*.
2. O IIS determina que a requisição deve ser tratada pelo ASP.NET.
3. O ASP.NET dá uma chance a todos os `HttpModules` para que modifiquem a requisição.
4. O `UrlRoutingModule` determina que o URL corresponde a uma rota configurada na aplicação.
5. O `UrlRoutingModule` pega o `IHttpHandler` apropriado a partir do `IRouteHandler` usado na rota correspondente (`MvcRouteHandler`, normalmente) como o manipulador da requisição.
6. O `MvcRouteHandler` constroi e retorna um `MvcHandler`.
7. O `MvcHandler`, que implementa `IHttpHandler`, executa `ProcessRequest`.
8. O `MvcHandler` usa `IControllerFactory` para obter uma instância de `IController` que usará o "`controller`" para rotear os dados da rota `{controller}/{action}/{id}`.
9. O `HomeController` é encontrado, e seu método `Execute` é invocado.

10. O `HomeController` invoca a ação `Index`.
11. A ação `Index` adiciona objetos ao dicionário `ViewData`.
12. O `HomeController` invoca o `ActionResult` retornado pela ação, que exibe uma view.
13. A view `Index` na pasta Views exibe os objetos de `ViewData`.
14. A view, derivada de `System.Web.Mvc.ViewPage`, executa seu método `ProcessRequest`.
15. O ASP.NET exibe a resposta no navegador.

Esses passos representam simplificadamente a vida de uma requisição tratada pelo ASP.NET MVC Framework. Se estiver curioso sobre os detalhes, consulte o código-fonte em *http://www.codeplex.com/aspnet*. Os 15 passos são suficientes para compreender como escrever código, baseado no ASP.NET MVC Framework, e na maior parte do tempo você deverá prestar atenção apenas no controlador e na view. Você já viu a rota no projeto iniciador. Vamos dar uma olhada em `HomeController`, mostrado na listagem 1.3.

⇨ Listagem 1.3 – O HomeController

```
using System.Web.Mvc;
namespace Ch1GettingStarted.Controllers
{
  [HandleError]
  public class HomeController : Controller ❶
  {
    public ActionResult Index() ◄ Ação-padrão do controlador
    {
      ViewData ["Message"] = "Welcome to ASP.NET MVC!"; ❷
      return View(); ◄ Retorna a view-padrão da ação
    }
    public ActionResult About() ◄ Outro método-ação
    {
      return View();
    }
  }
}
```

Perceba como é simples o controlador. Não há muito código gerado para destrincharmos, e cada método-ação retorna um objeto derivado de `ActionResult`. Esse controlador deriva de `System.Web.Mvc.Controller` ❶. Provavelmente você considerará esta classe-base adequada, mas existem outras que você pode escolher no projeto MvcContrib, e, conforme o tempo passa, é bem provável que a comunidade disponibilize muitas outras. Também pode ser uma boa prática criar o seu próprio *supertipo de camada* para usar em sua aplicação.

Dentro de cada método-ação, normalmente você colocará objetos em um dicionário chamado `ViewData` ❷. Esse dicionário será passado à view que está sendo exibida. O controlador pode fornecer qualquer objeto que a view solicite por meio desse dicionário `ViewData`; o objeto primário que a view exibirá deverá ser atribuído à propriedade `Model` do `ViewData`. Isso pode ser feito automaticamente, passando-se o objeto para o método `View()` do controlador. No projeto iniciador, os objetos são strings simples, mas, em sua aplicação, você usará objetos mais complexos, como os mostrados na figura 1.5.

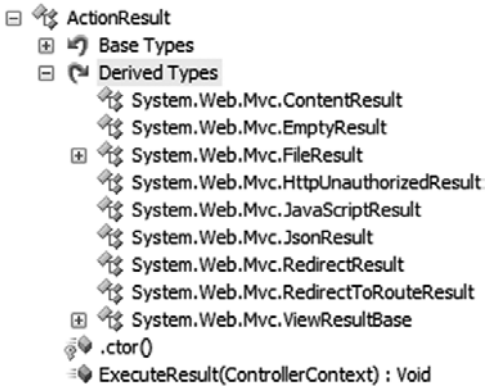

Figura 1.5 – Classes que derivam de ActionResult. Essa imagem foi retirada do .NET Reflector da Red Gate.

Cada ação-padrão retorna o resultado do método `View()`, que retorna um objeto `System.Web.Mvc.ViewResult`. Essa subclasse de `ActionResult` provavelmente será um resultado bem comum, já que sua aplicação terá várias telas. Em alguns casos, você pode usar os outros tipos de `ActionResult` mostrados na figura 1.5. A ação do seu controlador pode retornar qualquer tipo. A classe-base `Controller` chamará `ToString()` em seu objeto e retornará essa string em um objeto `ContentResult`. Em seguida, verificaremos a view mostrada na listagem 1.4, que pode ser encontrada dentro do projeto no seguinte caminho: */Views/Home/Index.aspx*.

⇨ Listagem 1.4 – Uma view simples

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage" %>
<asp:Content ID="indexTitle" ContentPlaceHolderID="TitleContent" runat="server">
  Home Page
</asp:Content>
<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">
  <h2><%= Html.Encode(ViewData ["Message"]) %></h2>
  <p>
  To learn more about ASP.NET MVC visit
  <a href="http://asp.net/mvc" title="ASP.NET MVC Website">http://asp.net/mvc</a>.
  </p>
</asp:Content>
```

A view mostrada na listagem 1.4 é a exibida no navegador, mostrada na figura 1.4. Com o framework MVC, arquivos de marcação não usam arquivos code-behind. Como a view usa o mecanismo de templates do Web Forms, você até poderia utilizá-los, mas por padrão apenas um arquivo simples de marcação é gerado.

Essa view usa uma página-mestra (master page), como você pode ver no atributo `MasterPageFile` na diretiva `Page`. A página-mestra pode ser especificada pelo controlador por uma questão de compatibilidade com vários mecanismos de views, mas alguns deles suportam a especificação do layout pela view, que é o caso do mecanismo de views Web Forms, o mecanismo-padrão que acompanha o framework MVC.

NOTA `ViewResult` usa a interface `IViewEngine`, a qual é uma abstração que permite o uso de qualquer mecanismo de apresentação de uma view. Mecanismos de views serão abordados com mais profundidade posteriormente, mas algumas alternativas podem ser encontradas no projeto open source MvcContrib.

No corpo dessa view, as tags do lado do servidor estão extraindo objetos de `ViewData` e passando-os junto com o HTML. A responsabilidade da view é pegar objetos do `ViewData` e passá-los adiante, para serem consumidos pelo usuário. A view não decide o que exibir, e sim apenas como exibir. O controlador já decidiu o que precisa ser exibido.

Na listagem 1.5, examine o código do layout. Você verá imediatamente que essa é uma página-mestra simples, não muito diferente daquelas encontradas no Web Forms. A diferença é que as páginas-mestras de projetos MVC não precisam usar arquivos *code-behind*.

⇨ **Listagem 1.5 – A página-mestra do projeto iniciador**

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/
  TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
  <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" />
  </title>
  <link href="../../Content/Site.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <div class="page">
  <div id="header">
    <div id="title">
      <h1>My MVC Application</h1>
    </div>
    <div id="logindisplay">
      <% Html.RenderPartial("LogOnUserControl"); %> ◄ Exibe outra view
    </div>
```

```
      <div id="menucontainer">
        <ul id="menu">
          <li><%= Html.ActionLink("Home", "Index", "Home")%></li> ◄ Exibe hiperlinks
          <li><%= Html.ActionLink("About", "About", "Home")%></li>
        </ul>
      </div>
      </div>
        <div id="main">
        <asp:ContentPlaceHolder ID="MainContent" runat="server" /
        <div id="footer">
        </div>
      </div>
      </div>
    </body>
  </html>
```

Aqui, a página-mestra está encarregada da navegação. Ela usa *auxiliares de views* (`Html.ActionLink` neste caso) para exibir os links. Auxiliares de views (view helpers) estão disponíveis para as necessidades dinâmicas mais comuns e para todos os elementos de formulário. Outros auxiliares de views estão disponíveis no MvcContrib, e fabricantes de componentes não ficam para trás no sentido de oferecer auxiliares de views comerciais.

Agora que você já verificou como o código do projeto iniciador se encaixa, vejamos como testar o código do controlador. O código da view ainda precisará ser testado com uma ferramenta como o Selenium, Watir ou WatiN, mas o código do controlador pode ser facilmente orientado a testes, pois ele é desacoplado da view e do tempo de execução do ASP.NET. Quando começar um novo projeto, um diálogo lhe perguntará qual framework de teste unitário deseja usar.

Se estiver usando o Visual Studio 2008 Professional, o *Visual Studio Unit Test* já estará listado e selecionado. Os frameworks mais comuns de testes unitários possuem templates que são exibidos na lista após serem instalados. Por enquanto, veremos como usar o MSTest (Visual Studio Unit Test), mas recomendamos o uso do NUnit. Se você estiver apenas iniciando no mundo dos testes automatizados, qualquer framework bem difundido servirá. A listagem 1.6 exibe um método de teste do MSTest incluso no template do projeto de teste-padrão.

⇨ **Listagem 1.6 – O teste unitário da ação Index**

```
using System.Web.Mvc;
using Ch1GettingStarted.Controllers;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace Ch1GettingStarted.Tests.Controllers
```

```
{
  [TestClass]
  public class HomeControllerTest
  {
    [TestMethod]
    public void Index()
    {
      HomeController controller = new HomeController();
      ViewResult result = controller.Index() as ViewResult;
      ViewDataDictionary viewData = result.ViewData;
      Assert.AreEqual("Welcome to ASP.NET MVC!",viewData["Message"]);
    }
  }
}
```

Acredite ou não, já percorremos o projeto iniciador do ASP.NET em sua totalidade, e agora você conhece o básico desse novo framework. Obviamente, veremos tópicos mais complexos ao longo deste livro, e se algum deles ao longo do caminho não "entrar na sua cabeça", por favor, abra o Visual Studio e investigue-o enquanto lê. Trabalhar com o código diretamente, enquanto lê este texto, dará a você uma compreensão sólida dessa tecnologia. Na verdade, agora é uma boa hora para baixar os códigos de exemplo deste livro e abrir a aplicação-padrão do seu IDE.

1.2 Seu primeiro controlador ASP.NET MVC desde o princípio

Observe a listagem 1.7 para entender como uma requisição web é processada pelo controlador. Perceba que o único requisito é implementar a interface `IController`.

⇨ **Listagem 1.7 – Nosso primeiro controlador**

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
namespace MvcApplication.Controllers
{
  public class HelloWorld1Controller : IController ❶
  {
    public void Execute(RequestContext requestContext)
    {
      requestContext.HttpContext.Response.Write(
        "<h1>Hello World1</h1>");
    }
  }
}
```

Semelhante a tudo no ASP.NET MVC Framework, há muito pouco que o desenvolvedor precisa fazer para criar funcionalidades customizadas. No caso do controlador, o único – vou repetir – o único requisito é que a classe implemente a interface `IController` ❶. Essa interface requer apenas que você implemente um único método, `Execute`. A forma como você vai tratar a requisição fica completamente por sua conta. No controlador da listagem 1.7, estamos violando intencionalmente todos os princípios racionais da programação, bem como a *Lei de Demeter*, para que consigamos escrever a mensagem "Hello World" na tela o mais rápido possível. Nesse caso, escolhi não fazer uso de uma view. Em vez disso, estou formulando uma marcação HTML incompleta e direcionando-a ao fluxo de resposta. Vamos executar o exemplo – repare na saída, mostrada na figura 1.6. No código da solução que vem com o livro, você pode encontrar o `HelloWorldControllerTest` que ilustra como realizar o teste unitário de um controlador simples como esse.

Figura 1.6 – Nossa aplicação web executando no navegador. Perceba o URL simples e a ausência da extensão .aspx.

A listagem 1.7 mostra o poder completo e absoluto que você tem quando cria uma classe controladora. É muito importante ter controle completo; no entanto, na maior parte do tempo, estaremos trabalhando com uma porção de cenários que se repetirão inúmeras vezes. Para esses cenários, o produto fornece uma classe-base que nos dá funcionalidades extras. A classe-base para esses controladores comuns é `System.Web.Mvc.Controller`. Ela implementa o método `Execute` para nós e usa os valores de rota para chamar diferentes *métodos-ação*, dependendo dos valores-padrão de URL e rota.

NOTA `System.Web.Mvc.Controller` é apenas uma das opções de classe-base que podem ser escolhidas para os seus controladores. Como mencionado anteriormente, normalmente é apropriado criar seu próprio *supertipo de camada* para todos os seus controladores. Esse tipo pode herdar de `System.Web.Mvc.Controller`, implementar `IController` ou derivar de qualquer outra classe-base controladora.

Nosso primeiro uso da classe-base `Controller` precisará de apenas um método-ação – adotaremos a convenção da ação-padrão e a chamaremos de `Index`. Observe na listagem 1.8 com o que nosso controlador se parece, enquanto usamos a classe-base `Controller`. Essa classe-base implementa a interface `IController` para nós e fornece a capacidade de invocar métodos-ação, baseada no objeto `Route` atual.

⇨ **Listagem 1.8 – Usando a classe-base `Controller`**

```
using System.Web.Mvc;
namespace MvcApplication.Controllers
{
  public class HelloWorld2Controller : Controller ◄ Herda de Controller
  {
    public string Index()
    {
      return "<h1>Hello World2</h1>";
    }
  }
}
```

O método-ação público `Index` é tudo que é necessário para esse controlador ser invocado via web. Métodos-ação de conteúdo simples não precisam retornar `ActionResult`. Retornar qualquer outro tipo resultará naquele objeto sendo passado como conteúdo para o fluxo de resposta.

Se apontarmos nosso navegador para `/HelloWorld2`, veremos que nosso controlador envia uma resposta ao navegador igual à mostrada na figura 1.7:

Figura 1.7 – A página web tem uma saída semelhante à anterior. O resultado final é o mesmo, ainda que a implementação do controlador tenha mudado.

Agora que sabemos como modelar um controlador, exploraremos nossa primeira view.

1.3 Nossa primeira view

Recorde que o ASP.NET MVC Framework usa uma convenção para localizar views. A convenção é encontrar um arquivo .aspx em uma árvore de diretórios que corresponda a */Views/nome_controlador/nome_ação.aspx*. Em nosso próximo exemplo, modificaremos o controlador invocando um método da classe-base `Controller` chamado `View()`. Vamos vincular o modelo, que é uma string com o texto "Hello World", a uma entrada do objeto `ViewDataDictionary` na propriedade `ViewData` da classe-base `Controller`. Essa instância de `ViewDataDictionary` será encaminhada à view. Apesar de

`ViewData` ser um `ViewDataDictionary`, recomendamos que você dependa apenas da interface `IDictionary<string, object>` se você estiver substituindo o mecanismo de views. Mecanismos de views serão discutidos mais detalhadamente no capítulo 4. Na listagem 1.9, vemos que nossa ação retorna `ActionResult` em vez de `string`. Após o retorno de um método-ação, o `ActionResult` é executado, realizando o comportamento apropriado que, neste caso, seria exibir uma view. Examine a listagem 1.9 para entender a implementação atual de nosso controlador. `ViewData` contém o objeto que será encaminhado à view. O método `View()` também permite passar um único objeto à view que fica, então, acessível pelo `ViewData.Model`, que exploraremos posteriormente.

⇨ **Listagem 1.9 – Usando uma view para exibir o modelo**

```
using System.Web.Mvc;
namespace MvcApplication.Controllers
{
  public class HelloWorld3Controller : Controller
  {
    public ActionResult Index()
    {
      ViewData.Add("text", "Hello World3"); ◄ Adiciona objetos a ViewData
      return View();
    }
  }
}
```

Se estiver acompanhando este exemplo, deverá criar uma pasta *HelloWorld3* dentro de */Views* em seu projeto, conforme mostra a figura 1.8.

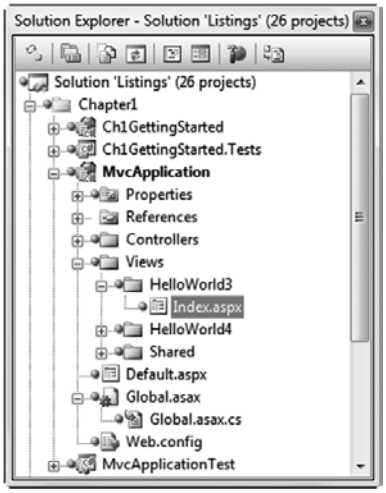

Figura 1.8 – A localização correta da pasta HelloWorld3 é dentro da pasta /Views. A fábrica-padrão de views usa essa convenção. Você pode redefinir esse comportamento, se desejar.

Em seguida, adicione uma view ao projeto dentro de */Views/Helloworld3*. Você pode usar o diálogo New Item do menu Project e selecionar MVC View Page; uma forma mais rápida é usar o menu de contexto (botão direito do mouse) sobre a ação e selecionar Add View..., como mostra a figura 1.9. Essa ferramenta criará uma view com o nome correto dentro da pasta correta. Seu projeto deverá agora estar semelhante à figura 1.8. Nossa nova view, *Index.aspx*, reside na pasta *HelloWorld3*.

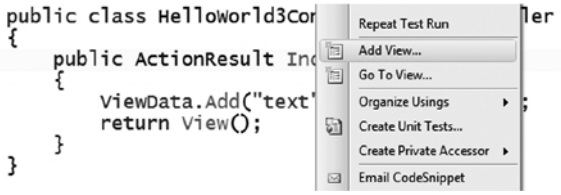

Figura 1.9 – Adicionando a view ao nosso projeto, por meio do menu de contexto.

O nosso código de marcação dentro da view será bem simples. Afinal, essa aplicação é tão trivial que tudo que ela faz é mostrar "Hello World3" na tela em letras grandes. Usaremos os operadores do lado do servidor, `<%` e `%>`, para extrair o modelo (que é uma string) do dicionário de `ViewData` e exibi-lo na tela. A listagem 1.10 mostra a marcação para a view. A classe-base é `System.Web.Mvc.ViewPage`. Esta é uma diferença importante com relação ao Web Forms.

⇨ Listagem 1.10 – Acessando ViewData de dentro da view

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <div>
    <h1><%=ViewData ["text"]%></h1>
  </div>
</body>
</html>
```

O fluxo de nosso controlador para nossa view é simples. O controlador indica um ou mais objetos que serão encaminhados à view, então, opcionalmente, especifica o nome da view. O framework MVC localiza a view, instancia-a, transmite o `ViewData` e faz a view passar a si própria ao fluxo de resposta. A classe-base `ViewPage` suporta totalmente o processo de transmissão de dados, mas `ViewState`, postbacks e eventos de postback do lado do servidor não mais acontecem. Eventos de exibição ainda são disparados, pois `ViewPage` deriva de `System.Web.UI.Page`.

A única responsabilidade da view é transformar objetos passados a ela em HTML. Esse é um ponto-chave do SoC, uma prática importante que a Microsoft está focalizando com este produto. O controlador não sabe como a view está fazendo a formatação. O único ponto de acoplamento é que o controlador nomeia a view, seja implicitamente com o nome da ação, seja explicitamente com o método `View()`. Não importa que o `System.Web.Mvc.WebFormViewEngine` esteja executando. O mesmo controlador poderia funcionar muito bem com uma view do NVelocity (suportado pelo MvcContrib), já que o único acoplamento do controlador com a view é o nome da view. Por enquanto, encaminharemos "Hello World3" à view, e a figura 1.10 mostra a página desenhada. Esse exemplo mostra como botar um caso simples para funcionar, mas existem questões de manutenção, às quais você deve atentar enquanto constroi uma aplicação com esse framework.

Figura 1.10 – A saída é semelhante à dos exemplos anteriores, exceto que agora temos uma página HTML completa e não apenas um fragmento HTML.

1.4 Assegurando a manutenção da aplicação

Você deve ter se encolhido de medo quando viu que o objeto de modelo de domínio estava sendo inserido em um `IDictionary`, com uma chave de `string` como a forma de identificação do objeto na view. O .NET nos condicionou a querer que tudo seja fortemente tipado, então identificadores em string são algumas vezes vistos como negativos. Esse ponto é subjetivo e controverso, e você tem várias opções, dependendo de suas preferências.

NOTA Usar um dicionário para passar objetos entre diferentes partes de uma aplicação (tipicamente chamado de *saco de propriedades*) permite que várias partes da aplicação fiquem fracamente acopladas. O lado negativo é que os objetos que saem do saco de propriedades precisam passar por operações de conversão (casting), antes de serem usados.

O uso de um `IDictionary<string, object>` para `ViewData` é o mecanismo-padrão na classe-base `Controller`. Conforme o número de objetos encaminhados à view aumenta, é fácil criar auxiliares nela, que evitem operações de conversão espalhadas por todo lado. No

entanto, muitas ações do controlador encaminharão apenas um objeto para a view e, para esses casos, podemos fazer uso da propriedade `ViewData.Model`, bem como das views fortemente tipadas. Para o nosso próximo exemplo, realizaremos uma pequena alteração no controlador e na view para permitir uma referência fortemente tipada.

A listagem 1.11 mostra o nosso controlador descartando o uso do `IDictionary<string, object>` e passando o objeto de modelo de domínio diretamente ao método `View()`. Isso funciona bem em vários cenários, inclusive em aplicações de grande porte com o uso de um modelo de view, que é um objeto de apresentação modelado especificamente para ser usado em uma única view. No caso de views compostas, pode ser necessário usar `ViewData` como um dicionário, mas as views parciais, abordadas no capítulo 4, também podem dar suporte a views compostas.

NOTA Na listagem 1.11, tivemos que transformar a string "Hello World4" em um objeto, quando a passamos ao método `View()`. Isso é necessário porque uma das sobrecargas de `View()` aceita uma string como parâmetro único, sendo usada para especificar uma view em particular que será exibida.

⇨ Listagem 1.11 – Passando o modelo de apresentação ao método View

```
using System.Web.Mvc;
namespace MvcApplication.Controllers
{
  public class HelloWorld4Controller : Controller
  {
    public ActionResult Index()
    {
      return View((object)"Hello World4");  ◄ Conversão realizada, para que a sobrecarga correta seja chamada
    }
  }
}
```

Temos disponível uma propriedade `Model` (um atalho para `ViewData.Model`) que é o tipo declarado como sendo o parâmetro genérico de `ViewPage<T>` na tag `Inherits`. Na listagem 1.12 optamos por `System.Web.Mvc.ViewPage<string>`. A classe-base genérica nos permite tipar fortemente a propriedade `ViewData.Model`, porque a propriedade `ViewData` é do tipo `ViewDataDictionary<T>`, e a propriedade `Model` é do tipo `T`.

⇨ Listagem 1.12 – View fortemente tipada com página-mestra

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<string>"
MasterPageFile="~/Views/Shared/HelloWorld.Master"%>
<%@ Import Namespace="MvcApplication.Controllers"%>
<asp:Content ID="Content2" ContentPlaceHolderID="Main" runat="server">
  <h1><%=Model%>! I'm strongly typed in a layout!</h1>
</asp:Content>
```

Nesse exemplo, a view escolheu a página-mestra, mas, como dissemos anteriormente, você também pode escolher a página-mestra dentro da ação do controlador. A escolha é sua, mas nem todos os mecanismos de views suportam a especificação da página-mestra pela view, então, caso exista a possibilidade de precisar alterar o mecanismo de views posteriormente, talvez seja melhor considerar especificar a página-mestra no controlador. O lado negativo disso é que você aumenta o acoplamento do controlador com as views.

Páginas-mestras funcionam da mesma forma que no Web Forms para criação de templates, mas o postback do lado do servidor e os mecanismos de `ViewState` são irrelevantes. Exibição é a única responsabilidade da view e da página-mestra. A listagem 1.13 mostra nossa página-mestra, que esboça a estrutura da página. O layout declara `System.Web.Mvc.ViewMasterPage` como o tipo-base.

⇨ **Listagem 1.13 – O layout da nossa view**

```
<%@ Master Language="C#" AutoEventWireup="true"
  Inherits="System.Web.Mvc.ViewMasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>Hello!!</title>
  <asp:ContentPlaceHolder ID="head" runat="server">
  </asp:ContentPlaceHolder>
</head>
<body>
  <div style="border: solid 4px red">
    <asp:ContentPlaceHolder ID="Main" runat="server">
    </asp:ContentPlaceHolder>
  </div>
</body>
</html>
```

Se executarmos o projeto (CTRL + F5) e apontarmos o nosso navegador para */HelloWorld4*, observaremos a nossa view fortemente tipada e a página-mestra em ação. Para dar destaque, demos uma ordem contundente, como mostra a figura 1.11. A maioria das views que usaremos neste livro usarão a classe-base `ViewPage<T>` para permitir uma propriedade `Model` fortemente tipada e o uso facilitado de modelos de view, que serão abordados no capítulo 2.

Parabéns! Você acabou de criar seu primeiro controlador, e ele lida perfeitamente com suas próprias responsabilidades. Deste ponto em diante, não cobriremos mais cada pequeno passo de desenvolvimento com o framework MVC. Se você está lendo este livro, já deve estar bem experiente com o ASP.NET, então abordaremos apenas

os itens que são novos no ASP.NET MVC Framework. Usaremos as melhores práticas e técnicas avançadas de desenvolvimento de software ao longo dos exemplos. A primeira prática importante que abordaremos será o teste unitário. Como o framework MVC nos permite manter os nossos controladores separados das views, podemos testar facilmente a lógica de controle. Consideramos os testes uma parte importante para iniciar os trabalhos com o framework MVC, então tocaremos no assunto agora e manteremos uma forte ênfase em testes ao longo do livro. Apesar do fato de não reproduzirmos todos os testes unitários neste livro, você poderá encontrar mais testes unitários nos exemplos do livro, que podem ser baixados do website da editora (*www.manning.com/ASP.NETMVCinAction*).

Figura 1.11 – Esta é a nossa view finalizada, completa com layout, tipagem forte e um pouco de CSS para "dar uma apimentada".

1.5 Testando as classes controladoras

Para esta seção, você precisará ter instalado o framework de testes unitários NUnit. Ele pode ser encontrado em *http://www.nunit.org*, e é gratuito. Após instalar a última versão, abra a interface gráfica do NUnit e selecione Tools > Options > Visual Studio (à esquerda) e marque a opção que diz "Enable Visual Studio Support". Isso facilitará trabalhar com o NUnit de dentro do Visual Studio.

Agora retorne ao projeto de teste unitário que criamos anteriormente. Você precisará remover as referências a MSTest e adicionar uma referência a NUnit. Nós criamos a classe `HelloWorld4ControllerTester` para abrigar os testes unitários que verificam se o controlador funciona corretamente. Veja na figura 1.12 que estamos agrupando a afixação dos testes unitários de controladores em uma pasta `Controllers`. Conforme você navega pelas listagens de código, perceberá que criamos stubs de várias classes de que o controlador necessita para funcionar. Esse exemplo de teste unitário é o mais simples que você verá neste livro. A listagem 1.14 descreve a afixação do teste do NUnit para o `HelloWorld4Controller`.

⇨ **Listagem 1.14 – Afixação de teste unitário com o NUnit**

```
using System.Web.Mvc;
using MvcApplication.Controllers;
using NUnit.Framework;
```

```
using NUnit.Framework.SyntaxHelpers;
namespace MvcApplicationTest.Controllers
{
  [TestFixture]
  public class HelloWorld4ControllerTester
  {
    [Test]
    public void IndexShouldRenderViewWithStringViewData()
    {
      var controller = new HelloWorld4Controller(); ❶
      var viewResult = (ViewResult) controller.Index(); ❷
      Assert.That(viewResult.ViewName, Is.EqualTo("")); ❸
      Assert.That(viewResult.ViewData.Model, Is.EqualTo("Hello World4"));
    }
  }
}
```

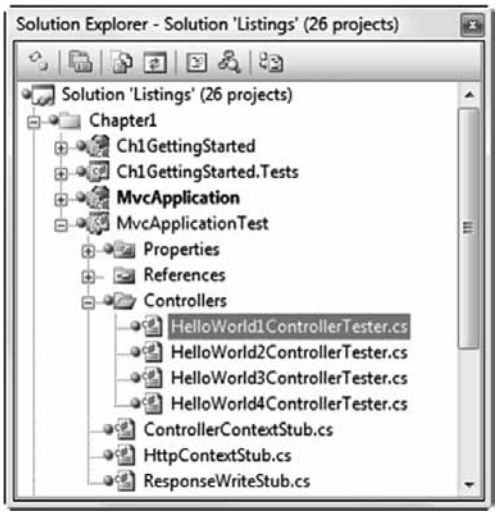

Figura 1.12 – No seu projeto de teste unitário, após adicionar uma referência a nunit.framework.dll, você estará pronto para adicionar uma afixação de teste.

Vamos examinar nosso teste unitário simples dentro de `HelloWorld4ControllerTester`. Nosso teste é o `Index-ShouldRenderViewWithStringViewData`. Criamos uma instância da classe que está sendo testada, `HelloWorld4Controller` ❶. Chamamos, então, nosso método `Index`❷, capturando o `ActionResult` retornado. O nosso teste espera que isso seja uma instância de `ViewResult`, então transformamos o retorno como tal. Se o código retornar o tipo errado, o teste falhará de maneira apropriada. No final do teste, asseguramos facilmente as nossas expectativas ❸. Os testes unitários costumam seguir um fluxo de *disposição*, *ação* e *afirmação*, e esse teste é um exemplo perfeito disso. Este foi um teste unitário bem simples, e o capítulo 3 abordará o teste unitário de controladores

com mais profundidade. A figura 1.13 mostra o teste sendo executado com o UnitRun do JetBrains. UnitRun também é uma funcionalidade do ReSharper.

Criar testes unitários automatizados para todo o código e executar esses testes a cada compilação do software ajudará a certificar que, conforme a aplicação cresce em complexidade, o software continua manutenível. Geralmente, conforme a aplicação cresce, ela fica mais difícil de manter. Um pacote de testes automatizados ajuda a contra-atacar a tendência natural à entropia. Felizmente, é fácil testar controladores com o framework MVC. Na verdade, a equipe da Microsoft usou o desenvolvimento orientado a testes (*test-driven development*, ou TDD), enquanto desenvolvia o framework.

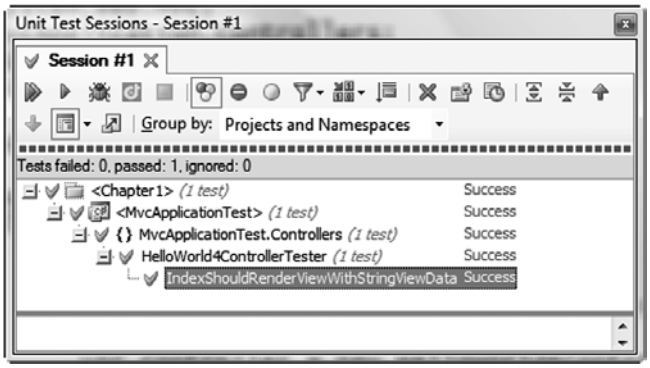

Figura 1.13 – Quando executamos esse teste unitário com o JetBrains ReSharper, ele passa, conforme esperado.

NOTA Junto com o framework MVC, a Microsoft encapsulou parte do código do ASP.NET e forneceu classes abstratas para algumas das APIs-chave, como `HttpResponseBase`, `HttpRequestBase` e a mais importante de todas, `HttpContextBase`. Uma pesquisa no *Google* revelará como inúmeras pessoas tiveram problemas para testar o `HttpContext` por causa dos seus membros de tipo `static` e `sealed`. Fornecer classes abstratas para essas APIs-chave diminui o acoplamento a elas, aumentando a capacidade de teste.

Mencionamos os testes unitários várias vezes no texto, e é importante entender a conexão deles com o TDD. O desenvolvimento orientado a testes é um estilo de desenvolvimento em que testes unitários (e outros testes) são criados antes de se escrever o código que fará com que esses testes passem. Neste capítulo, nós não aderimos estritamente ao processo de TDD, na tentativa de focalizar nas áreas-chave do framework MVC sem adicionar a sobrecarga mental de um novo processo de desenvolvimento.

É boa prática garantir que, conforme você escreva seus testes unitários, eles não invoquem algum banco de dados ou serviço web. Isso ajuda a manter a compilação da parte de testes executando de maneira rápida e garante a manutenibilidade, evitando a inclusão de uma dependência a um sistema externo que não é possível garantir que não vá mudar. É razoável executar uma compilação de um projeto que contenha 2000 testes automatizados em 5 segundos; se vários dos seus testes unitários envolverem

um banco de dados, a sua compilação provavelmente levará muito mais tempo para finalizar. Outros testes que integrem com elementos externos, como um banco de dados, continuam sendo valiosos, mas, em alguns casos, cada um deles pode levar vários segundos para ser compilado, então você vai querer se concentrar em manter os testes de controladores no nível da unidade. Para auxiliar nessa tarefa, você pode "simular" (mock) as dependências dos controladores.

1.6 Resumo

Vimos como é fácil iniciar os trabalhos com o ASP.NET MVC Framework e, daqui por diante no livro, os exemplos não serão tão triviais. Agora você sabe como adicionar uma rota à aplicação e sabe que a rota define qual controlador e ação deverão ser invocados para um determinado URL. Uma vez que o controlador seja invocado, um método-ação fica encarregado de determinar o que deve ser passado à view, baseado na requisição em questão. A view pega os objetos passados a ela e formata-os usando um template de view. A view não toma decisões sobre os objetos passados, mas apenas formata-os para serem exibidos. Essa separação de interesses contribui para uma aplicação mais manutenível do que temos visto com o Web Forms.

Na maior parte do livro, usaremos o CodeCampServer em nossos exemplos. Além de estar incluso nos downloads deste livro, o CodeCampServer pode ser encontrado em *http://CodeCampServer.org*. É uma aplicação ASP.NET MVC Framework que pode hospedar uma conferência de um grupo de usuários. Ela usa uma *Arquitetura em Cebola* desacoplada, projeto orientado ao domínio, o ASP.NET MVC Framework e o NHibernate para mostrar com o que uma aplicação corporativa de verdade se parece. Possui um processo de construção completo com NAnt, que é monitorado por um servidor de compilação como o CruiseControl.Net ou o JetBrains TeamCity. A aplicação como um todo tem o objetivo de ser um exemplo vivo de como se escrever aplicações de verdade com o ASP.NET MVC Framework. Desde sua concepção, mais voluntários têm se juntado ao projeto, e agora ele é um esforço comunitário completo. Ele viverá muito além do tempo de vida deste livro, então a cópia do código que você está recebendo com este livro é uma foto instantânea de um determinado momento do projeto. Convidamos você a se juntar ao projeto, conforme continua a progredir.

Usaremos essa aplicação do mundo real em nossos exemplos conforme avançamos. Isso significa que você deve acompanhar o ritmo com que os conceitos e padrões são discutidos. Como uma equipe de autores, decidimos que poderíamos fornecer mais valor agregado com exemplos avançados da vida real, para levar o leitor a fazer mais pesquisa do que com exemplos simplificados. Escolhemos não comprometer a questão do design de software, mesmo que isso torne o livro um pouco mais difícil de ser escrito. O primeiro tópico que abordaremos em profundidade, no próximo capítulo, é a parte de Modelo do padrão Modelo-View-Controlador.