1

Getting Started

Jon Galloway

COPYRIGHTED MATERIAL

WHAT'S IN THIS CHAPTER?

- Understanding ASP.NET MVC
- An overview of ASP.NET MVC 4
- Creating MVC 4 applications
- How MVC applications are structured

This chapter gives you a quick introduction to ASP.NET MVC, explains how ASP.NET MVC 4 fits into the ASP.NET MVC release history, summarizes what's new in ASP .NET MVC 4, and shows you how to set up your development environment to build ASP.NET MVC 4 applications.

This is a Professional Series book about a version 4 web framework, so we're going to keep the introductions short. We're not going to spend any time convincing you that you should learn ASP.NET MVC. We're assuming that you've bought this book for that reason, and that the best proof of software frameworks and patterns is in showing how they're used in real-world scenarios.

A QUICK INTRODUCTION TO ASP.NET MVC

ASP.NET MVC is a framework for building web applications that applies the general Model View Controller pattern to the ASP.NET framework. Let's break that down by first looking at how ASP.NET MVC and the ASP.NET framework are related.

How ASP.NET MVC Fits in with ASP.NET

When ASP.NET 1.0 was first released in 2002, it was easy to think of ASP.NET and Web Forms as one and the same thing. ASP.NET has always supported two layers of abstraction, though:

- `System.Web.UI`: The Web Forms layer, comprising server controls, ViewState, and so on
- `System.Web`: The plumbing, which supplies the basic web stack, including modules, handlers, the HTTP stack, and so on

The mainstream method of developing with ASP.NET included the whole Web Forms stack — taking advantage of drag-and-drop server controls and semi-magical statefulness, while dealing with the complications behind the scenes (an often confusing page life cycle, less than optimal HTML that was difficult to customize, and so on).

However, there was always the possibility of getting below all that — responding directly to HTTP requests, building out web frameworks just the way you wanted them to work, crafting beautiful HTML — using handlers, modules, and other handwritten code. You could do it, but it was painful; there just wasn't a built-in pattern that supported any of those things. It wasn't for lack of patterns in the broader computer science world, though. By the time ASP.NET MVC was announced in 2007, the MVC pattern was becoming one of the most popular ways of building web frameworks.

The MVC Pattern

Model-View-Controller (MVC) has been an important architectural pattern in computer science for many years. Originally named *Thing-Model-View-Editor* in 1979, it was later simplified to *Model-View-Controller*. It is a powerful and elegant means of separating concerns within an application (for example, separating data access logic from display logic) and applies itself extremely well to web applications. Its explicit separation of concerns does add a small amount of extra complexity to an application's design, but the extraordinary benefits outweigh the extra effort. It has been used in dozens of frameworks since its introduction. You'll find MVC in Java and C++, on Mac and on Windows, and inside literally dozens of frameworks.

The MVC separates the user interface (UI) of an application into three main aspects:

- **The Model:** A set of classes that describes the data you're working with as well as the business rules for how the data can be changed and manipulated
- **The View:** Defines how the application's UI will be displayed
- **The Controller:** A set of classes that handles communication from the user, overall application flow, and application-specific logic

MVC AS A USER INTERFACE PATTERN

Notice that we've referred to MVC as a pattern for the UI. The MVC pattern presents a solution for handling user interaction, but says nothing about how you will handle other application concerns like data access, service interactions, etc. It's helpful to keep this in mind as you approach MVC: It is a useful pattern, but likely one of many patterns you will use in developing an application.

MVC as Applied to Web Frameworks

The MVC pattern is used frequently in web programming. With ASP.NET MVC, it's translated roughly as:

- **Models:** These are the classes that represent the domain you are interested in. These domain objects often encapsulate data stored in a database as well as code used to manipulate the data and enforce domain-specific business logic. With ASP.NET MVC, this is most likely a Data Access Layer of some kind, using a tool like Entity Framework or NHibernate combined with custom code containing domain-specific logic.
- **View:** This is a template to dynamically generate HTML. We cover more on that in Chapter 3 when we dig into views.
- **Controller:** This is a special class that manages the relationship between the View and the Model. It responds to user input, talks to the Model, and decides which view to render (if any). In ASP.NET MVC, this class is conventionally denoted by the suffix *Controller*.

> **NOTE** *It's important to keep in mind that MVC is a high-level architectural pattern, and its application varies depending on use. ASP.NET MVC is contextualized both to the problem domain (a stateless web environment) and the host system (ASP.NET).*
>
> *Occasionally I talk to developers who have used the MVC pattern in very different environments, and they get confused, frustrated, or both (confustrated?) because they assume that ASP.NET MVC works the exact same way it worked in their mainframe account processing system 15 years ago. It doesn't, and that's a good thing — ASP.NET MVC is focused on providing a great web development framework using the MVC pattern and running on the .NET platform, and that contextualization is part of what makes it great.*
>
> *ASP.NET MVC relies on many of the same core strategies that the other MVC platforms use, plus it offers the benefits of compiled and managed code and exploits newer .NET language features, such as lambdas and dynamic and anonymous types. At its heart, though, ASP.NET applies the fundamental tenets found in most MVC-based web frameworks:*
>
> - *Convention over configuration*
> - *Don't repeat yourself (aka the "DRY" principle)*
> - *Pluggability wherever possible*
> - *Try to be helpful, but if necessary, get out of the developer's way*

The Road to MVC 4

In the three short years since ASP.NET MVC 1 was released in March 2009, we've seen four major releases of ASP.NET MVC and several more interim releases. In order to understand ASP.NET

MVC 4, it's important to understand how we got here. This section describes the contents and background of each of the three major ASP.NET MVC releases.

ASP.NET MVC 1 Overview

In February 2007, Scott Guthrie ("ScottGu") of Microsoft sketched out the core of ASP.NET MVC while flying on a plane to a conference on the East Coast of the United States. It was a simple application, containing a few hundred lines of code, but the promise and potential it offered for parts of the Microsoft web developer audience was huge.

As the legend goes, at the Austin ALT.NET conference in October 2007 in Redmond, Washington, ScottGu showed a group of developers "this cool thing I wrote on a plane" and asked if they saw the need and what they thought of it. It was a hit. In fact, many people were involved with the original prototype, codenamed *Scalene*. Eilon Lipton e-mailed the first prototype to the team in September 2007, and he and ScottGu bounced prototypes, code, and ideas back and forth.

Even before the official release, it was clear that ASP.NET MVC wasn't your standard Microsoft product. The development cycle was highly interactive: There were nine preview releases before the official release, unit tests were made available, and the code shipped under an open source license. All these highlighted a philosophy that placed a high value on community interaction throughout the development process. The end result was that the official MVC 1.0 release — including code and unit tests — had already been used and reviewed by the developers who would be using it. ASP.NET MVC 1.0 was released on 13 March 2009.

ASP.NET MVC 2 Overview

ASP.NET MVC 2 was released just one year later, in March 2010. Some of the main features in MVC 2 included:

- UI helpers with automatic scaffolding with customizable templates
- Attribute-based model validation on both client and server
- Strongly typed HTML helpers
- Improved Visual Studio tooling

There were also lots of API enhancements and "pro" features, based on feedback from developers building a variety of applications on ASP.NET MVC 1, such as:

- Support for partitioning large applications into *areas*
- Asynchronous controllers support
- Support for rendering subsections of a page/site using `Html.RenderAction`
- Lots of new helper functions, utilities, and API enhancements

One important precedent set by the MVC 2 release was that there were very few breaking changes. I think this is a testament to the architectural design of ASP.NET MVC, which allows for a lot of extensibility without requiring core changes.

ASP.NET MVC 3 Overview

ASP.NET MVC 3 shipped just 10 months after MVC 2, driven by the release date for Web Matrix. Some of the top features in MVC 3 included:

- The Razor view engine
- Support for .NET 4 Data Annotations
- Improved model validation
- Greater control and flexibility with support for dependency resolution and global action filters
- Better JavaScript support with unobtrusive JavaScript, jQuery Validation, and JSON binding
- Use of NuGet to deliver software and manage dependencies throughout the platform

Since these MVC 3 features are relatively recent and are pretty important, we'll go through them in a little more detail.

> **NOTE** *This feature summary is included for developers with previous MVC experience who are anxious to hear what's changed in the newer versions.*
>
> *If you're new to ASP.NET MVC, don't be concerned if some of these features don't make a lot of sense right now; we'll be covering them in a lot more detail throughout the book. You're welcome to skip over them now and come back to this chapter later.*

Razor View Engine

Razor is the first major update to rendering HTML since ASP.NET 1 shipped almost a decade ago. The default view engine used in MVC 1 and 2 was commonly called the Web Forms view engine, because it uses the same ASPX/ASCX/MASTER files and syntax used in Web Forms. It works, but it was designed to support editing controls in a graphical editor, and that legacy shows. An example of this syntax in a Web Forms page is shown here:

```
<%@ Page Language="C#" MasterPageFile="</Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowseViewModel>"
%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
   Browse Albums
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

   <div class="genre">
       <h3><em><%: Model.Genre.Name %></em> Albums</h3>
```

```
        <ul id="album-list">
            <% foreach (var album in Model.Albums) { %>

            <li>
                <a href="<%: Url.Action("Details", new { id = album.AlbumId }) %>">
                    <img alt="<%: album.Title %>" src="<%: album.AlbumArtUrl %>" />
                    <span><%: album.Title %></span>
                </a>
            </li>

            <% } %>
        </ul>

    </div>

</asp:Content>
```

Razor was designed specifically as a view engine syntax. It has one main focus: *code-focused templating for HTML generation*. Here's how that same markup would be generated using Razor:

```
@model MvcMusicStore.Models.Genre

@{ViewBag.Title = "Browse Albums";}

<div class="genre">
   <h3><em>@Model.Name</em> Albums</h3>

   <ul id="album-list">
       @foreach (var album in Model.Albums)
       {
           <li>
               <a href="@Url.Action("Details", new { id = album.AlbumId })">
                   <img alt="@album.Title" src="@album.AlbumArtUrl" />
                   <span>@album.Title</span>
               </a>
           </li>
       }
   </ul>
</div>
```

The Razor syntax is easier to type, and easier to read. Razor doesn't have the XML-like heavy syntax of the Web Forms view engine.

We've talked about how working with the Razor syntax feels different. To put this in more quantifiable terms, let's look at some of the team's design goals in creating the Razor syntax:

- **Compact, expressive, and fluid:** Razor's (ahem) sharp focus on templating for HTML generation yields a very minimalist syntax. This isn't just about minimizing keystrokes — although that's an obvious result — it's about how easy it is to express your intent. A key example is the simplicity in transitions between markup and code. You can see this in action when writing out some model properties in a loop:

```
@foreach (var album in Model.Albums)
{
   <li>
       <a href="@Url.Action("Details", new { id = album.AlbumId })">
```

```
                <img alt="@album.Title" src="@album.AlbumArtUrl" />
                <span>@album.Title</span>
            </a>
        </li>
    }
```

- **Not a new language:** Razor is a syntax that lets you use your existing .NET coding skills in a template in a very intuitive way. Scott Hanselman summarized this pretty well when describing his experiences learning Razor:

> "I kept […] going cross-eyed when I was trying to figure out what the syntax rules were for Razor until someone said stop thinking about it, just type an "at" sign and start writing code and I realize that there really is no Razor."
>
> —Hanselminutes #249: On Webmatrix with Rob Conery
>
> http://hanselminutes.com/default.aspx?showid=268

- **Easy to learn:** Precisely because Razor is not a new language, it's easy to learn. You know HTML, you know .NET; just type HTML and hit the `@` sign whenever you need to write some .NET code.
- **Works with any text editor:** Because Razor is so lightweight and HTML-focused, you're free to use the editor of your choice. Visual Studio's syntax highlighting and IntelliSense features are nice, but it's simple enough that you can edit it in any text editor.
- **Good IntelliSense:** Though Razor was designed so that you shouldn't *need* IntelliSense to work with it, IntelliSense can come in handy for things like viewing the properties your model object supports. For those cases, Razor does offer nice IntelliSense within Visual Studio, as shown in Figure 1-1.

FIGURE 1-1

This is just a quick highlight of some of the reasons that Razor makes writing View code really easy. We'll talk about Razor in a lot more depth in Chapter 3.

Validation Improvements

Validation is an important part of building web applications, but it's never fun. I've always wanted to spend as little time as possible writing validation code, as long as I was confident that it worked correctly.

MVC 2's attribute-driven validation system removed a lot of the pain from this process by replacing repetitive imperative code with declarative code. However, support was focused on a short list of top validation scenarios. There were plenty of cases where you'd get outside of the "happy path" and have to write a fair amount more code. MVC 3 extended the validation support to cover most scenarios you're likely to encounter. For more information on validation in ASP.NET MVC, see Chapter 6.

.NET 4 Data Annotation Support

MVC 2 was compiled against .NET 3.5 and thus didn't support any of the .NET 4 Data Annotations enhancements. MVC 3 picks up some new, very useful validation features available due to .NET 4 support. Some examples include:

- MVC 2's `DisplayName` attribute wasn't localizable, whereas the .NET 4 standard `System.ComponentModel.DataAnnotations Display` attribute is.
- `ValidationAttribute` was enhanced in .NET 4 to better work with the validation context for the entire model, greatly simplifying cases like validators that compare or otherwise reference two model properties.

Streamlined Validation with Improved Model Validation

MVC 3's support for the .NET 4 `IValidatableObject` interface deserves individual recognition. You can extend your model validation in just about any conceivable way by implementing this interface on your model class and implementing the `Validate` method, as shown in the following code:

```
public class VerifiedMessage : IValidatableObject {
   public string Message { get; set; }
   public string AgentKey { get; set; }
   public string Hash { get; set; }

   public IEnumerable<ValidationResult> Validate(
       ValidationContext validationContext) {
       if (SecurityService.ComputeHash(Message, AgentKey) != Hash)
           yield return new ValidationResult("Agent compromised");
   }
}
```

Unobtrusive JavaScript

Unobtrusive JavaScript is a general term that conveys a general philosophy, similar to the term REST (Representational State Transfer). The high-level description is that unobtrusive JavaScript doesn't intermix JavaScript code in your page markup. For example, rather than hooking in via event attributes like `onclick` and `onsubmit`, the unobtrusive JavaScript attaches to elements by their ID or class, often based on the presence of other attributes (such as HTML5 `data-` attributes).

Unobtrusive JavaScript makes a lot of sense when you consider that your HTML document is just that — a document. It's got semantic meaning, and all of it — the tag structure, element attributes, and so on — should have a precise meaning. Strewing JavaScript gunk across the page to facilitate interaction (I'm looking at you, `__doPostBack`!) harms the content of the document.

MVC 3 added support for unobtrusive JavaScript in two ways:

- Ajax helpers (such as `Ajax.ActionLink` and `Ajax.BeginForm`) render clean markup for the `FORM` tag, wiring up behavior leveraging extensible attributes (`data-` attributes) and jQuery.

- Ajax validation no longer emits the validation rules as a (sometimes large) block of JSON data, instead writing out the validation rules using `data-` attributes. While technically I considered MVC 2's validation system to be rather unobtrusive, the MVC 3 system is that much more — the markup is lighter weight, and the use of `data-` attributes makes it easier to leverage and reuse the validation information using jQuery or other JavaScript libraries.

jQuery Validation

MVC 2 shipped with jQuery, but used Microsoft Ajax for validation. MVC 3 completed the transition to using jQuery for Ajax support by converting the validation support to run on the popular jQuery Validation plugin. The combination of Unobtrusive JavaScript support (discussed previously) and jQuery validation using the standard plugin system means that the validation is both extremely flexible and can benefit from the huge jQuery community.

Client-side validation was turned on by default for MVC 3 projects, and can be enabled site-wide with a `web.config` setting or by code in `global.asax` for upgraded projects.

JSON Binding

MVC 3 included JavaScript Object Notation (JSON) binding support via the new `JsonValueProviderFactory`, enabling your action methods to accept and model-bind data in JSON format. This is especially useful in advanced Ajax scenarios like client templates and data binding that need to post data back to the server.

Dependency Resolution

MVC 3 introduced a new concept called a *dependency resolver*, which greatly simplified the use of dependency injection in your applications. This made it easier to decouple application components, making them more configurable and easier to test.

Support was added for the following scenarios:

- Controllers (registering and injecting controller factories, injecting controllers)
- Views (registering and injecting view engines, injecting dependencies into view pages)
- Action filters (locating and injecting filters)
- Model binders (registering and injecting)
- Model validation providers (registering and injecting)
- Model metadata providers (registering and injecting)
- Value providers (registering and injecting)

This is a big enough topic that we've devoted an entire new chapter (Chapter 12) to it.

Global Action Filters

MVC 2 action filters gave you hooks to execute code before or after an action method ran. They were implemented as custom attributes that could be applied to controller actions or to an entire controller. MVC 2 included some filters in the box, like the `Authorize` attribute.

MVC 3 extended this with global action filters, which apply to all action methods in your application. This is especially useful for application infrastructure concerns like error handling and logging.

MVC 4 Overview

The MVC 4 release is building on a pretty mature base and is able to focus on some more advanced scenarios. Some top features include:

- ASP.NET Web API
- Enhancements to default project templates
- Mobile project template using jQuery Mobile
- Display Modes
- Task support for Asynchronous Controllers
- Bundling and minification

The following sections provide an overview of these features. We'll be going into them in more detail throughout the book.

ASP.NET Web API

ASP.NET MVC was designed for creating websites. Throughout the platform are obvious design decisions that indicate the assumed usage: responding to requests from browsers and returning HTML.

However, ASP.NET MVC made it really easy to control the response down to the byte, and the MVC pattern was really useful in creating a service layer. ASP.NET developers found that they could use it to create web services that returned XML, JSON, or other non-HTML formats, and it was a lot easier than grappling with other service frameworks, such as Windows Communication Foundation (WCF), or writing raw HTTP handlers. It still had some quirks, as you were using a website framework to deliver services, but many found that it was better than the alternatives.

MVC 4 includes a better solution: ASP.NET Web API (referred to as *Web API*), a framework that offers the ASP.NET MVC development style but is tailored to writing HTTP services. This includes both modifying some ASP.NET MVC concepts to the HTTP service domain and supplying some new service-oriented features.

Here are some of the Web API features that are similar to MVC, just adapted for the HTTP service domain:

- **Routing:** ASP.NET Web API uses the same routing system for mapping URLs to controller actions. It contextualizes the routing to HTTP services by mapping HTTP verbs to actions by convention, which both makes the code easier to read and encourages following RESTful service design.
- **Model binding and validation:** Just as MVC simplifies the process of mapping input values (form fields, cookies, URL parameters, etc.) to model values, Web API automatically maps HTTP request values to models. The binding system is extensible and includes the same attribute-based validation that you use in MVC model binding.

- **Filters:** MVC uses filters (discussed in Chapter 15) to allow for adding behaviors to actions via attributes. For instance, adding an `[Authorize]` attribute to an MVC action will prohibit anonymous access, automatically redirecting to the login page. Web API also supports some of the standard MVC filters (like a service-optimized `[Authorize]` attribute) and custom filters.
- **Scaffolding:** You add new Web API controllers using the same dialog used to add an MVC controller (as described later this chapter). You have the option to use the Add Controller dialog to quickly scaffold a Web API controller based on an Entity Framework–based model type.
- **Easy unit testability:** Much like MVC, Web API is built around the concepts of dependency injection and avoiding the use of global state.

Web API also adds some new concepts and features specific to HTTP service development:

- **HTTP programming model**: The Web API development experience is optimized for working with HTTP requests and responses. There's a strongly typed HTTP object model, HTTP status codes and headers are easily accessible, etc.
- **Action dispatching based on HTTP verbs**: In MVC the dispatching of action methods is based on their names. In Web API methods can be automatically dispatched based on the HTTP verb. So, for example, a GET request would be automatically dispatched to a controller action named GetItem.
- **Content negotiation:** HTTP has long supported a system of content negotiation, in which browsers (and other HTTP clients) indicate their response format preferences, and the server responds with the highest preferred format that it can support. This means that your controller can supply XML, JSON, and other formats (you can add your own), responding to whichever the client most prefers. This allows you to add support for new formats without having to change any of your controller code.
- **Code-based configuration:** Service configuration can be complex. Unlike WCF's verbose and complex configuration file approach, Web API is configured entirely via code.

Although ASP.NET Web API is included with MVC 4, it can be used separately. In fact, it has no dependencies on ASP.NET at all, and can be self-hosted — that is, hosted outside of ASP.NET and IIS. This means you can run Web API in any .NET application, including a Windows Service or even a simple console application. For a more detailed look at ASP.NET Web API, see Chapter 11.

Enhancements to Default Project Templates

The visual design of the default template for MVC 1 projects had gone essentially unchanged through MVC 3. When you created a new MVC project and ran it, you got a white square on a blue background, as shown in Figure 1-2. (The blue doesn't show in this black and white book, but you get the idea.)

In MVC 4, both the HTML and CSS for the default template have been completely redesigned. A new MVC application appears as shown in Figure 1-3.

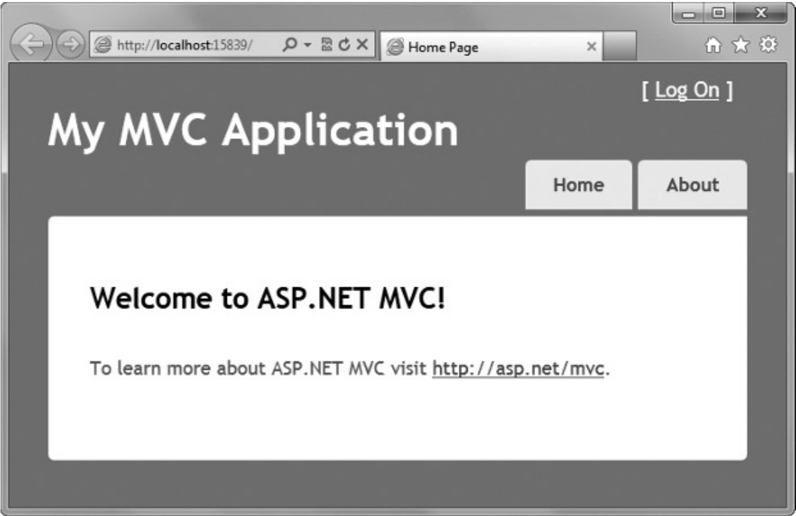

FIGURE 1-2

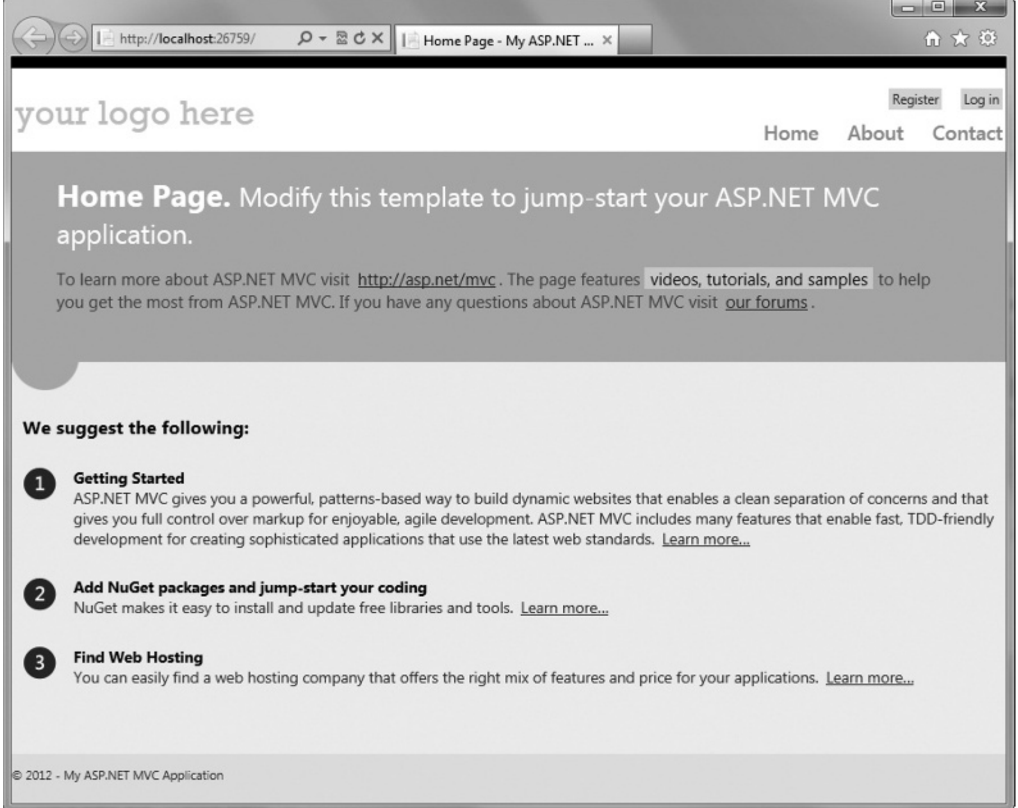

FIGURE 1-3

In addition to a more modern design (or, some would say, any thought to design at all), the new template also features support for mobile browsers through adaptive layout. Adaptive layout is a technique for designing fluid web layouts that respond to differing screen dimensions through CSS media queries. When the site is viewed at lower than 850px width (such as on a phone or tablet), the CSS automatically reconfigures to optimize for the small form factor, as shown in the mobile emulator in Figure 1-4.

While your sites deserve their own custom design, it's nice that the underlying HTML and CSS in an MVC 4 project are set up using modern markup and CSS that responds well to the growing mobile browser viewership.

Mobile Project Template Using jQuery Mobile

If you're going to be creating sites that will only be viewed in mobile browsers, you can make use of the new Mobile Project template. This template preconfigures your site to use the popular jQuery Mobile library, which provides styles that look good and work well on mobile devices, as shown in Figure 1-5. jQuery Mobile is touch optimized, supports Ajax navigation, and uses progressive enhancement to support mobile device features.

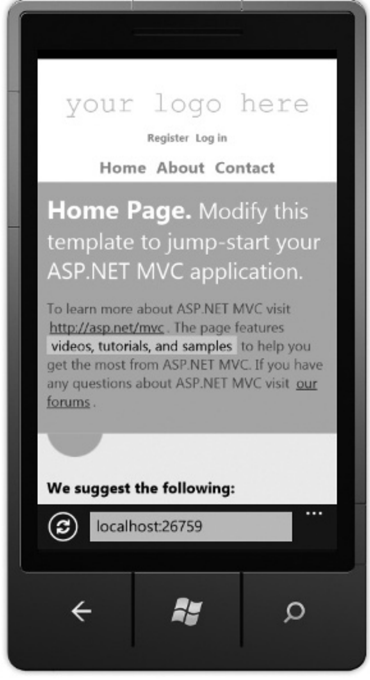

FIGURE 1-4

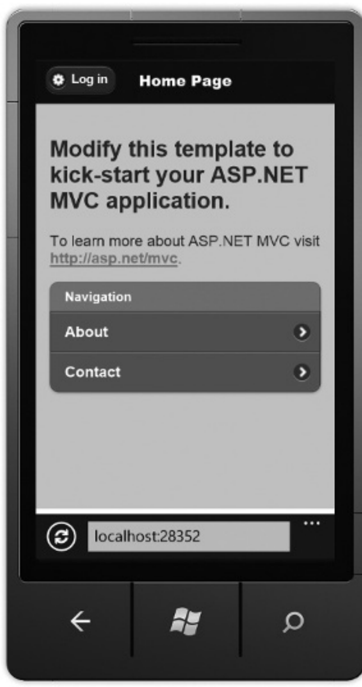

FIGURE 1-5

Display Modes

Display modes use a convention-based approach to allow selecting different views based on the browser making the request. The default view engine first looks for views with names ending with `.Mobile.cshtml` when the browser's user agent indicates a known mobile device. For example, if you have a generic view titled `Index.cshtml` and a mobile view titled `Index.Mobile.cshtml`, MVC 4 will automatically use the mobile view when viewed in a mobile browser.

Additionally, you can register your own custom device modes that will be based on your own custom criteria — all in just one code statement. For example, to register a WinPhone device mode that would serve views ending with `.WinPhone.cshtml` to Windows Phone devices, you'd use the following code in the `Application_Start` method of your `Global.asax`:

```
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("WinPhone")
{
    ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
        ("Windows Phone OS", StringComparison.OrdinalIgnoreCase) >= 0)
});
```

Bundling and Minification

ASP.NET 4 supports the same bundling and minification framework that is included in ASP.NET 4.5. This system reduces requests to your site by combining several individual script references into a single request. It also "minifies" the requests through a number of techniques, such as shortening variable names and removing whitespace and comments. This system works on CSS as well, bundling CSS requests into a single request and compressing the size of the CSS request to produce equivalent rules using a minimum of bytes, including advanced techniques like semantic analysis to collapse CSS selectors.

The bundling system is highly configurable, allowing you to create custom bundles that contain specific scripts and reference them with a single URL. You can see some examples by referring to default bundles listed in `/App_Start/BundleConfig.cs` in a new MVC 4 application using the Internet template.

One nice byproduct of using bundling and minification is that you can remove file references from your view code. This means that you can add or upgrade script libraries and CSS files that have different filenames without having to update your views or layout, since the references are made to script and CSS bundles rather than individual files. For example, the MVC Internet application template includes a jQuery bundle that is not tied to the version number:

```
bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                        "~/Scripts/jquery-{version}.js"));
```

This is then referenced in the site layout (`_Layout.cshtml`) by the bundle URL, as follows:

```
@Scripts.Render("~/bundles/jquery")
```

Since these references aren't tied to a jQuery version number, updating the jQuery library (either manually or via NuGet) will be picked up automatically by the bundling and minification system without requiring any code changes.

Included Open Source Libraries

MVC project templates have long been including top open source libraries like jQuery and Modernizr. As of MVC 3, these were included via NuGet, which makes it even simpler to upgrade them and manage dependencies. MVC 4 project templates include a few new libraries:

- **Json.NET:** Json.NET is a .NET library for manipulating information in JavaScript Object Notation (JSON). It was included in MVC 4 as part of Web API to support serializing data to JSON format, allowing for data contracts, anonymous types, dynamic types, Dates, TimeSpans, object reference preservation, indenting, camel casing, and many other useful serialization features. However, you can make use of Json.NET's additional features, including LINQ to JSON and automatic conversion from JSON to XML.

- **DotNetOpenAuth:** MVC uses DotNetOpenAuth to support OpenID- and OAuth-based logins using third-party identity providers. The Account Controller is set up to make it easy to add support for Facebook, Microsoft, Google, and Twitter; however, since these logins are built on top of OpenID and OAuth, you can easily plug in additional providers. While you can use the DotNetOpenAuth classes directly, MVC 4 also provides an OAuthWebSecurity (in the `Microsoft.Web.WebPages.OAuth` namespace) to simplify common usage.

Miscellaneous Features

MVC 4 includes a lot of features not listed previously. The full list is in the release notes, available at `http://www.asp.net/whitepapers/mvc4-release-notes`. Some of the most interesting ones that don't fit into any of the preceding themes are listed here.

- **Configuration logic moved to App_Start:** New features are nice, but the additional logic for features that are configured via code was really starting to clutter up the `Global.asax Application_Start` method. These configurations have been moved to static classes in the `App_Start` directory.
  - **AuthConfig.cs**: Used to configure security settings, including sites for OAuth login.
  - **BundleConfig.cs**: Used to register bundles used by the bundling and minification system. Several bundles are added by default, including jQuery, jQueryUI, jQuery validation, Modernizr, and default CSS references.
  - **FilterConfig.cs**: Unsurprisingly, this is used to register global MVC filters. The only filter registered by default is the HandleErrorAttribute, but this is a great place to put other filter registrations.
  - **RouteConfig.cs**: Holds the granddaddy of the MVC config statements, Route configuration. Routing is discussed in detail in Chapter 9.
  - **WebApiConfig.cs**: Used to register Web API routes, as well as set any additional Web API configuration settings.

- **Empty MVC project template:** MVC 4 has included an Empty project template since MVC 2, but it wasn't really empty; it still included the folder structure, a CSS file, and more than a dozen JavaScript files. Due to popular request, that template has been renamed Basic, and the new Empty project template really is empty.

- **Add Controller anywhere:** Previously, the Visual Studio Add Controller menu item only displayed when you right-clicked on the Controllers folder. However, the use of the Controllers folder was purely for organization. (MVC will recognize any class that implements the IController interface as a Controller, regardless of its location in your application.) The MVC 4 Visual Studio tooling has been modified to display the Add Controller menu item for any folder in your MVC project. This allows you to organize your controllers however you would like, perhaps separating them into logical groups or separating MVC and Web API controllers.

NOT APPEARING IN THIS FILM: SINGLE PAGE APPLICATION AND RECIPES

The MVC 4 Beta included a few previews of interesting, experimental features that are not included in the release version of MVC 4. Both are planned to ship later as out-of-band releases.

Single Page Application

Single Page Application (SPA) is a new project template for building single page applications that focus mainly on client-side interactions using JavaScript and Web APIs. This kind of web application can be very interactive and efficient — think of Microsoft Outlook Web Access, Gmail, etc. — but is also significantly harder to build. The Single Page Application template included:

- A set of JavaScript libraries to interact with local and cached data
- Additional Web API components for unit of work and DAL support
- An MVC project template with scaffolding support to tie the pieces together

This preview generated a lot of interest and feedback. Unfortunately, the team determined that they wouldn't be able to complete it in time to ship with MVC 4, and it was removed from the MVC 4 RC. It's still in development and planned for an out-of-band release following the MVC 4 release.

Recipes

Recipes make it easy to update Visual Studio tooling via NuGet packages. The team initially worked to allow extending the MVC tooling (e.g., the Add Area, Add Controller, and Add View dialogs). Phil demonstrated a View Mobilizer sample Recipe that created mobile versions of existing views with a simple checkbox dialog.

However, the team realized that Recipes had a lot more potential than just extending MVC tooling. A wide variety of NuGet packages could benefit from custom Visual Studio tooling to provide simplified configuration, automation, designers, etc. Based on this, Recipes was removed after the MVC 4 Beta but will be included in a future NuGet release.

Open Source Release

ASP.NET MVC has been under an open source license since the initial release, but it was just open source code rather than a full open source project. You could read the code; you could modify code; you could even distribute your modifications; but you couldn't contribute your code back to the official MVC code repository.

That changed with the ASP.NET Web Stack open source announcement in May 2012. This announcement marked the transition of ASP.NET MVC, ASP.NET Web Pages (including the Razor view engine), and ASP.NET Web API from open source licensed code to fully open source projects. All code changes and issue tracking for these projects is done in public code repositories, and these projects are allowed to accept community code contributions (aka *pull requests*) if the team agrees that the changes make sense.

Even in the short time since the project has been opened, several bug fixes and feature enhancements have already been accepted into the official source and will ship with the MVC 4 release. External code submissions are reviewed and tested by the ASP.NET team, and when released will be supported by Microsoft just as any previous ASP.NET MVC releases have been.

Even if you're not planning to contribute any source code, the public repository makes a huge difference in visibility. While in the past you needed to wait for interim releases to see what the team was working on, you can view source check-ins as they happen (at `http://aspnetwebstack.codeplex.com/SourceControl/list/changesets`) and even run nightly releases of the code to test out new features as they're written.

CREATING AN MVC 4 APPLICATION

The best way to learn about how MVC 4 works is to get started with building an application, so let's do that.

Software Requirements for ASP.NET MVC 4

MVC 4 runs on the following Windows client operating systems:

- Windows XP
- Windows Vista
- Windows 7
- Windows 8

It runs on the following server operating systems:

- Windows Server 2003
- Windows Server 2008
- Windows Server 2008 R2

MVC 4 development tooling is included with Visual Studio 2012 and can be installed on Visual Studio 2010 SP1/Visual Web Developer 2010 Express SP1.

Installing ASP.NET MVC 4

After ensuring you've met the basic software requirements, it's time to install ASP.NET MVC 4 on your development and production machines. Fortunately, that's pretty simple.

> **SIDE-BY-SIDE INSTALLATION WITH PREVIOUS VERSIONS OF MVC**
>
> MVC 4 installs side-by-side with previous versions of MVC, so you can install and start using MVC 4 right away. You'll still be able to create and update existing MVC 1, 2, and 3 applications, as before.

Installing the MVC 4 Development Components

The developer tooling for ASP.NET MVC 4 supports Visual Studio 2010 and Visual Studio 2012, including the free Express versions of both products.

MVC 4 is included with Visual Studio 2012, so there's nothing to install. If you're using Visual Studio 2010, you can install MVC 4 using either the Web Platform Installer (`http://www.microsoft.com/web/gallery/install.aspx?appid=MVC4VS2010`) or the executable installer package (available at `http://go.microsoft.com/fwlink/?LinkID=243392`). I generally prefer to use the Web Platform Installer (often called the *WebPI*, which makes me picture it with a magnificent Tom Selleck moustache for some reason) because it downloads and installs only the components you don't already have; the executable installer is able to run offline so it includes everything you might need, just in case.

Installing MVC 4 on a Server

The installers detect if they're running on a computer without a supported development environment and just install the server portion. Assuming your server has Internet access, WebPI is a lighter weight install, because there's no need to install any of the developer tooling.

When you install MVC 4 on a server, the MVC runtime assemblies are installed in the Global Assembly Cache (GAC), meaning they are available to any website running on that server. Alternatively, you can just include the necessary assemblies in your application without requiring that MVC 4 install on the server at all. In the past, this process, called *bin deployment*, required some extra work. Prior to the MVC 3 Tools Update, you either had to manually set assemblies to Copy Local in Visual Studio or use the Include Deployable Assemblies dialog. Starting with MVC 4, all assemblies are included via NuGet references. As such, all necessary assemblies are automatically added to the bin directory, and any MVC 4 application is bin-deployable. For this reason, the Include Deployable Assemblies dialog has been removed from Visual Studio 2012.

Creating an ASP.NET MVC 4 Application

After installing MVC 4, you'll have some new options in Visual Studio 2010 and Visual Web Developer 2010. The experience in both IDEs is very similar; because this is a Professional Series book we'll be focusing on Visual Studio development, mentioning Visual Web Developer only when there are significant differences.

MVC MUSIC STORE

We'll be loosely basing some of our samples on the MVC Music Store tutorial. This tutorial is available online at `http://mvcmusicstore.codeplex.com` and includes a 150-page e-book covering the basics of building an MVC 4 application. We'll be going quite a bit further in this book, but it's nice to have a common base if you need more information on the introductory topics.

To create a new MVC project:

1. Begin by choosing File ➪ New Project, as shown in Figure 1-6.

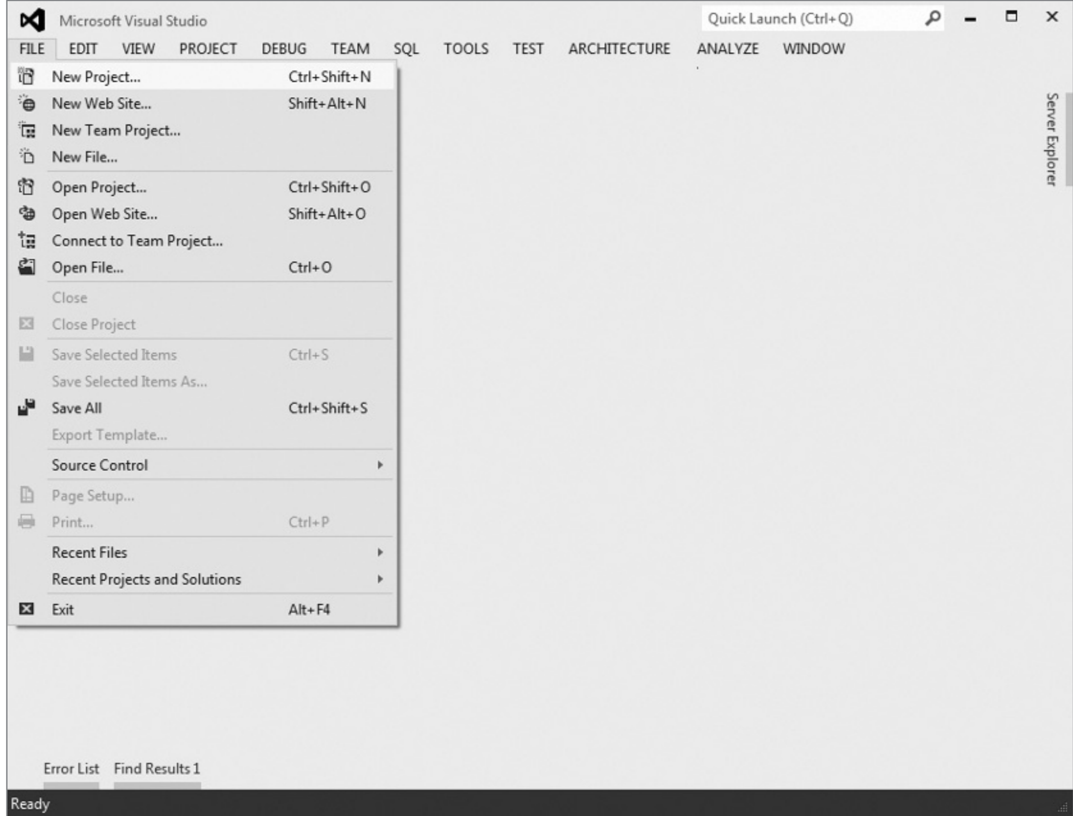

FIGURE 1-6

2. In the Installed Templates section on the left column of the New Project dialog, shown in Figure 1-7, select the Visual C# ➪ Web templates list. This displays a list of web application types in the center column.

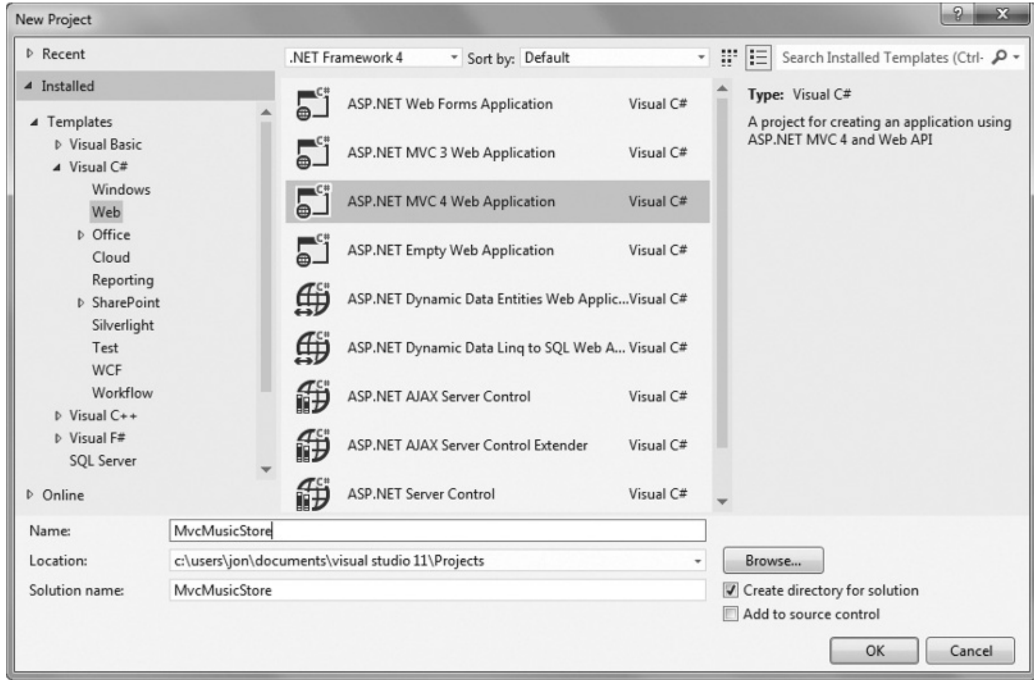

FIGURE 1-7

3. Select ASP.NET MVC 4 Web Application, name your application **MvcMusicStore**, and click OK.

The New ASP.NET MVC 4 Dialog

After creating a new MVC 4 application, you'll be presented with an intermediate dialog with some MVC-specific options for how the project should be created, as shown in Figure 1-8. The options you select from this dialog can set up a lot of the infrastructure for your application, from account management to view engines to testing.

Application Templates

First, you have the option to select from several preinstalled project templates.

- **The Internet Application template:** This contains the beginnings of an MVC web application — enough so that you can run the application immediately after creating it and see a few pages. You'll do that in just a minute. This template also includes some basic account management functions which run against the ASP.NET Membership system (as discussed in Chapter 7).
- **The Intranet Application template:** The Intranet Application template was added as part of the ASP.NET MVC 3 Tools Update. It is similar to the Internet Application template,

but the account management functions run against Windows accounts rather than the ASP .NET Membership system.

- **The Basic template:** This template is pretty minimal. It still has the basic folders, CSS, and MVC application infrastructure in place, but no more. Running an application created using the Empty template just gives you an error message — you need to work just to get to square one. Why include it, then? The Basic template is intended for experienced MVC developers who want to set up and configure things exactly how they want them.
- **The Empty template:** The Basic template used to be called the Empty template, but developers complained that it wasn't quite empty enough. With MVC 4, the previous Empty template was renamed Basic, and the new Empty template is about as empty as you can get. It has the assemblies and basic folder structure in place, but that's about it.
- **The Mobile Application template:** As described earlier in this chapter, the Mobile Application template is preconfigured with jQuery Mobile to jump-start creating a mobile-only website. It includes mobile visual themes, a touch-optimized UI, and support for Ajax navigation.
- **The Web API template:** ASP.NET Web API is a framework for creating HTTP services (and is discussed in detail in Chapter 11). The Web API template is similar to the Internet Application template but is streamlined for Web API development. For instance, there is no user account management functionality, as Web API account management is often significantly different from standard MVC account management. Web API functionality is also available in the other MVC project templates, and even in non-MVC project types.

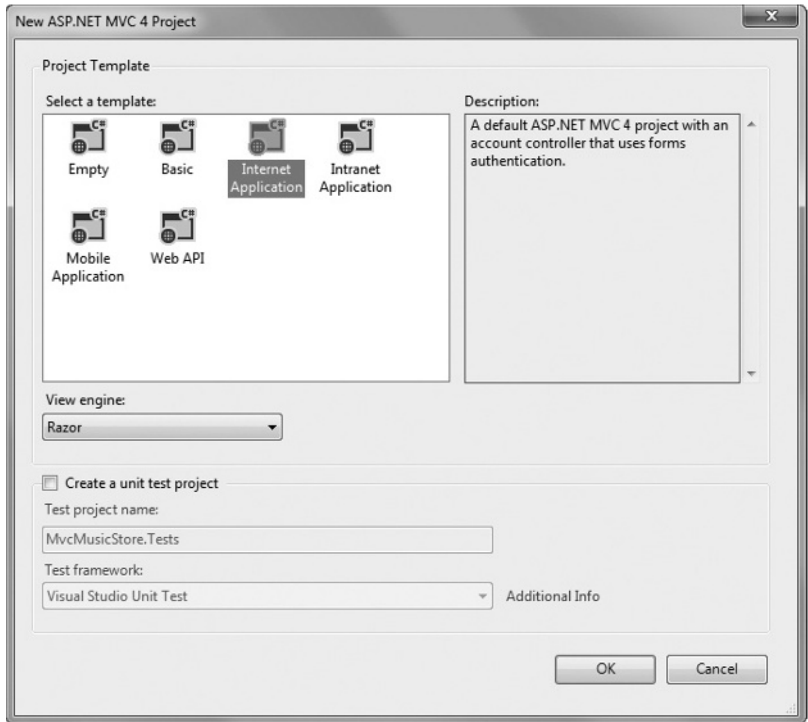

FIGURE 1-8

View Engines

The next option on the New ASP.NET MVC 4 Project dialog is a View Engine drop-down. View engines offer different templating languages used to generate the HTML markup in your MVC application. Prior to MVC 3, the only built-in option was the ASPX, or Web Forms, view engine. That option is still available, as shown in Figure 1-9.

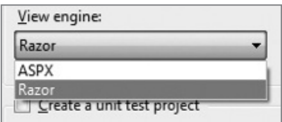

FIGURE 1-9

However, MVC 3 added a new option here: the Razor view engine. We'll be looking at that in a lot more detail, especially in Chapter 3.

Testing

All the built-in project templates have an option to create a unit test project with sample unit tests, as shown in Figure 1-10.

FIGURE 1-10

Leaving the Create a Unit Test Project checkbox unselected means that your project will be created without any unit tests, so there's nothing else to do.

RECOMMENDATION: CHECK THE BOX

I'm hoping you'll get in the habit of checking that Create a Unit Test Project box for *every* project you create.

I'm not going to try to sell you the Unit Testing religion — not just yet. We'll be talking about unit testing throughout the book, especially in Chapter 12, which covers unit testing and testable patterns, but we're not going to try to ram it down your throat.

Most developers I talk to are convinced that there is value in unit testing. Those who aren't using unit tests would like to, but they're worried that it's just too hard. They don't know where to get started, they're worried that they'll get it wrong, and they are just kind of paralyzed. I know just how they feel; I was there.

So, here's my sales pitch: Just check the box. You don't have to know anything to do it; you don't need an ALT.NET tattoo or a certification. We'll cover some unit testing in this book to get you started, but the best way to get started with unit testing is to just check the box, so that later you can start writing a few tests without having to set anything up.

After checking the Create a Unit Test Project box, you'll have a few more choices:

- The first is simple: You can change the name of your unit test project to anything you want.
- The second option allows you to select a test framework, as shown in Figure 1-11.

You may have noticed that there's only one test framework option shown, which doesn't seem to make a whole lot of sense. The reason there's a drop-down is that unit testing frameworks can register with the dialog, so if you've installed other unit testing frameworks (like xUnit, NUnit, MbUnit, and so on) you'll see them in that drop-down list as well.

FIGURE 1-11

NOTE *The Visual Studio Unit Test Framework is available only with Visual Studio 2012 Professional and higher versions. If you are using Visual Studio 2012 Standard Edition or Express, you will need to download and install the NUnit, MbUnit, or xUnit extensions for ASP.NET MVC in order for this dialog to be shown.*

REGISTERING UNIT TESTING FRAMEWORKS WITH THE UNIT TESTING FRAMEWORK DROP-DOWN

Ever wondered what's involved in registering a testing framework with the MVC New Project dialog?

The process is described in detail on MSDN (`http://msdn.microsoft.com/en-us/library/dd381614.aspx`). There are two main steps:

1. Create and install a template project for the new MVC Test Project.

continues

(continued)

2. Register the test project type by adding a few registry entries under `HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\10.0_Config\MVC4\TestProjectTemplates`.

Of course, both of these things can be included in the installation process for a unit testing framework, but you can customize them if you'd like without a huge amount of effort.

Review your settings on the New ASP.NET MVC 4 Project dialog to make sure they match Figure 1-12, and then click OK.

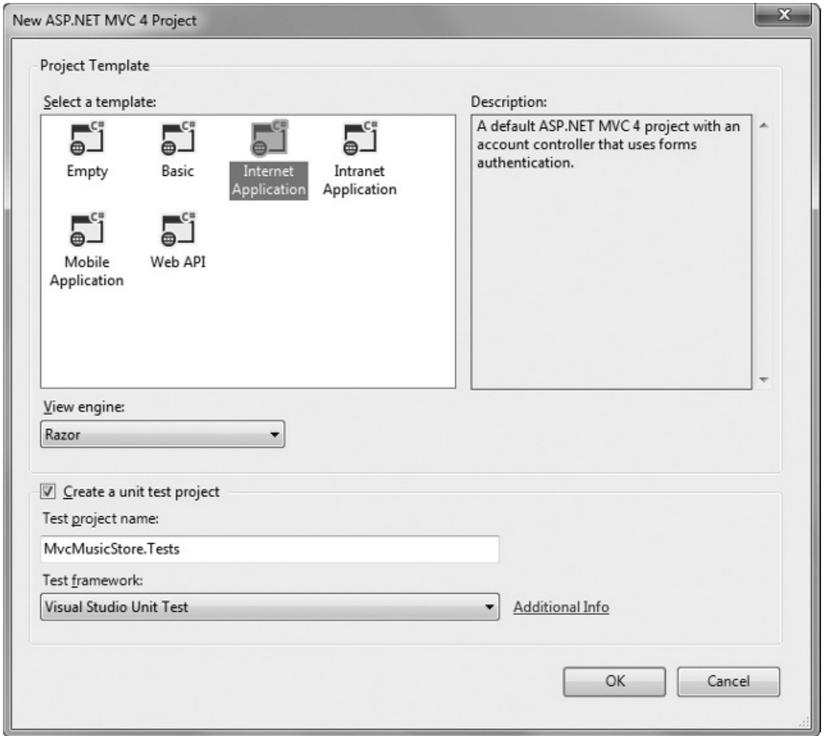

FIGURE 1-12

This creates a solution for you with two projects — one for the web application and one for the unit tests, as shown in Figure 1-13.

THE MVC APPLICATION STRUCTURE

When you create a new ASP.NET MVC application with Visual Studio, it automatically adds several files and directories to the project, as shown in Figure 1-14. ASP.NET MVC projects created with the Internet application template have eight top-level directories, shown in Table 1-1.

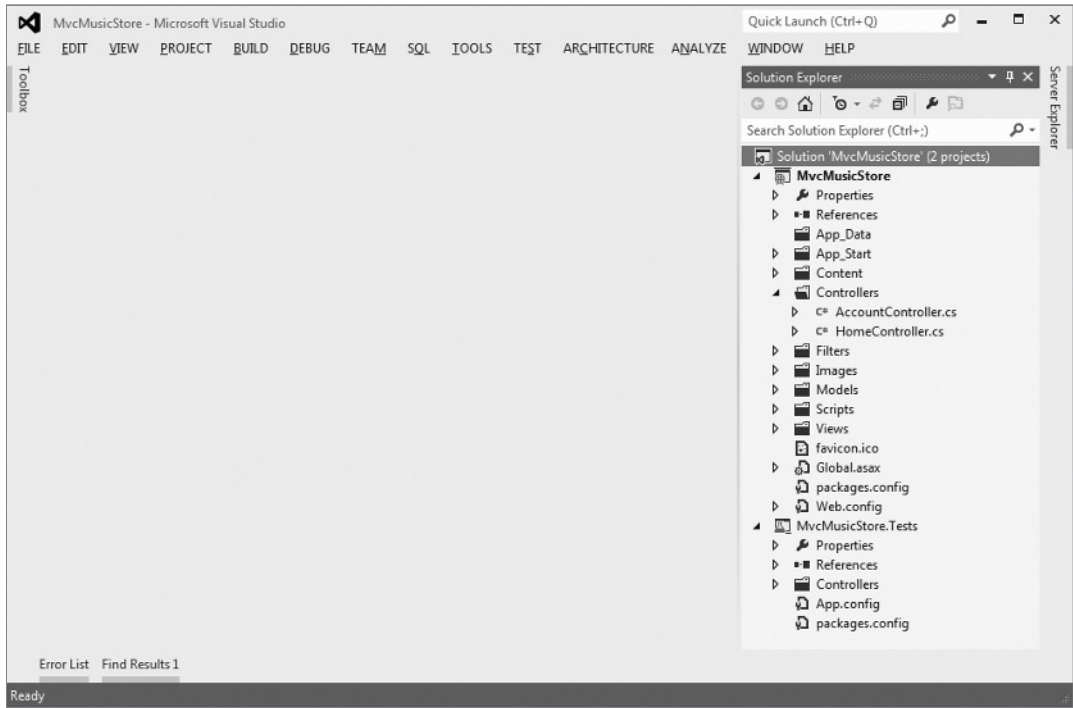

FIGURE 1-13

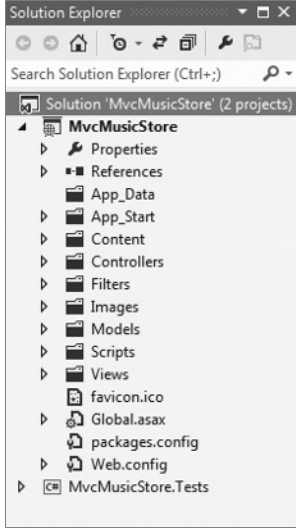

FIGURE 1-14

TABLE 1-1: Default Top-Level Directories

DIRECTORY	PURPOSE
`/Controllers`	Where you put Controller classes that handle URL requests
`/Models`	Where you put classes that represent and manipulate data and business objects
`/Views`	Where you put UI template files that are responsible for rendering output, such as HTML
`/Scripts`	Where you put JavaScript library files and scripts (`.js`)
`/Images`	Where you put images used in your site
`/Content`	Where you put CSS and other site content, other than scripts and images
`/Filters`	Where you put filter code. Filters are an advanced feature, discussed in Chapter 14.
`/App_Data`	Where you store data files you want to read/write
`/App_Start`	Where you put configuration code for features like Routing, Bundling, and Web API

WHAT IF I DON'T LIKE THAT DIRECTORY STRUCTURE?

ASP.NET MVC does not require this structure. In fact, developers working on large applications will typically partition the application across multiple projects to make it more manageable (for example, data model classes often go in a separate class library project from the web application). The default project structure, however, does provide a nice default directory convention that you can use to keep your application concerns clean.

Note the following about these files and directories. When you expand:

- The `/Controllers` directory, you'll find that Visual Studio added two `Controller` classes (Figure 1-15) — `HomeController` and `AccountController` — by default to the project.
- The `/Views` directory, you'll find that three subdirectories — `/Account`, `/Home`, and `/Shared` — as well as several template files within them, were also added to the project by default (Figure 1-16).
- The `/Content` and `/Scripts` directories, you'll find a `Site.css` file that is used to style all HTML on the site, as well as JavaScript libraries that can enable jQuery support within the application (Figure 1-17).
- The `MvcMusicStore.Tests` project, you'll find two classes that contain unit tests for your `Controller` classes (Figure 1-18).

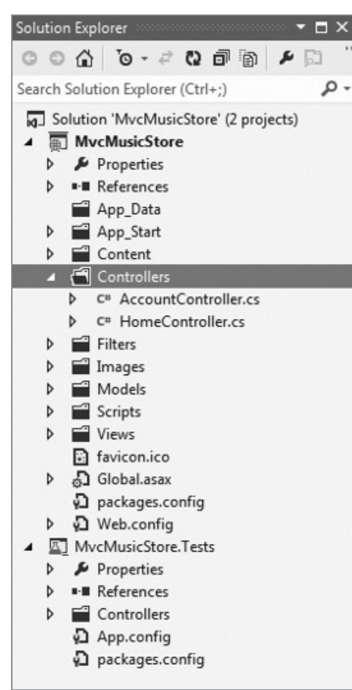

FIGURE 1-15

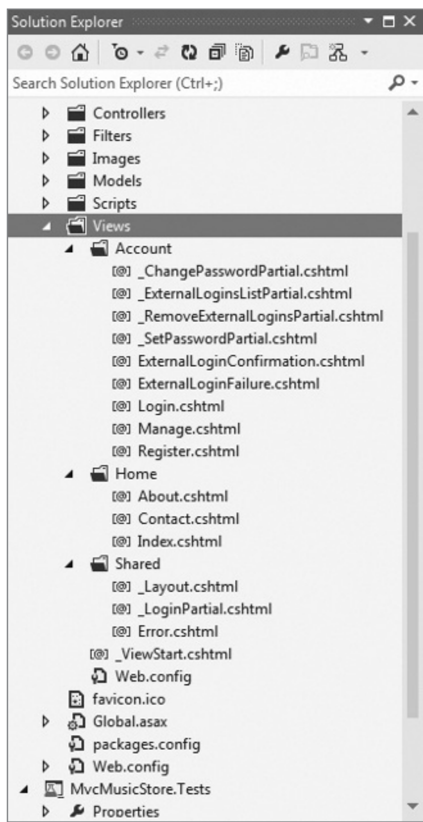

FIGURE 1-16

These default files, added by Visual Studio, provide you with a basic structure for a working application, complete with homepage, about page, account login/logout/registration pages, and an unhandled error page (all wired up and working out of the box).

ASP.NET MVC and Conventions

ASP.NET MVC applications, by default, rely heavily on conventions. This allows developers to avoid having to configure and specify things that can be inferred based on convention.

For instance, MVC uses a convention-based directory-naming structure when resolving View templates, and this convention allows you to omit the location path when referencing views from within a `Controller` class. By default, ASP.NET MVC looks for the View template file within the `\Views\[ControllerName]\` directory underneath the application.

MVC is designed around some sensible convention-based defaults that can be overridden as needed. This concept is commonly referred to as "convention over configuration."

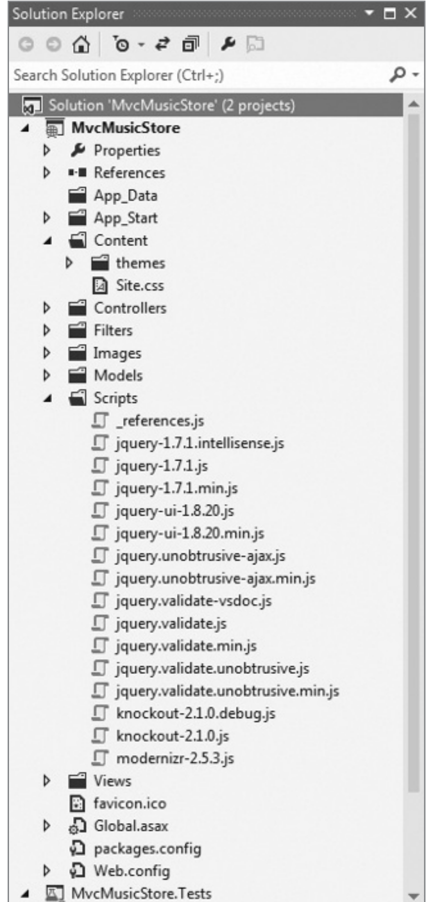

FIGURE 1-17

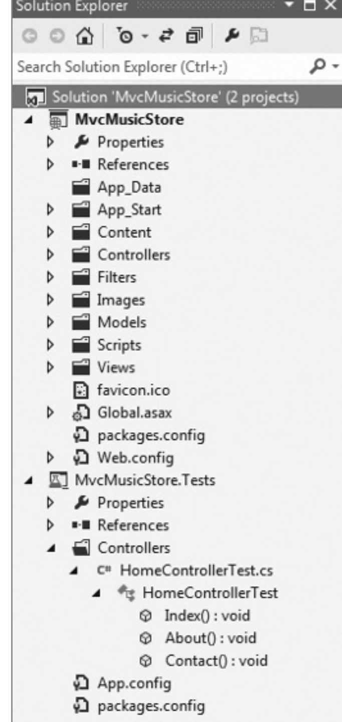

FIGURE 1-18

Convention over Configuration

The *convention over configuration* concept was made popular by Ruby on Rails a few years back, and essentially means:

> We know, by now, how to build a web application. Let's roll that experience into the framework so we don't have to configure absolutely everything again.

You can see this concept at work in ASP.NET MVC by taking a look at the three core directories that make the application work:

- Controllers
- Models
- Views

You don't have to set these folder names in the `web.config` file — they are just expected to be there by convention. This saves you the work of having to edit an XML file like your `web.config`, for

example, in order to explicitly tell the MVC engine, "You can find my views in the Views directory" — it already knows. It's a *convention*.

This isn't meant to be magical. Well, actually, it is; it's just not meant to be *black magic* — the kind of magic where you may not get the outcome you expected (and moreover can actually harm you).

ASP.NET MVC's conventions are pretty straightforward. This is what is expected of your application's structure:

- Each controller's class name ends with *Controller*: `ProductController`, `HomeController`, and so on, and lives in the `Controllers` directory.
- There is a single `Views` directory for all the views of your application.
- Views that controllers use live in a subdirectory of the `Views` main directory and are named according to the controller name (minus the *Controller* suffix). For example, the views for the `ProductController` discussed earlier would live in `/Views/Product`.

All reusable UI elements live in a similar structure, but in a `Shared` directory in the `Views` folder. You'll hear more about views in Chapter 3.

Conventions Simplify Communication

You write code to communicate. You're speaking to two very different audiences:

- You need to clearly and unambiguously communicate instructions to the computer for execution.
- You want developers to be able to navigate and read your code for later maintenance, debugging, and enhancement.

We've already discussed how convention over configuration helps you to efficiently communicate your intent to MVC. Convention also helps you to clearly communicate with other developers (including your future self). Rather than having to describe every facet of how your applications are structured over and over, following common conventions allows MVC developers worldwide to share a common baseline for all our applications. One of the advantages of software design patterns in general is the way they establish a standard language. Because ASP.NET MVC applies the MVC pattern along with some opinionated conventions, MVC developers can very easily understand code — even in large applications — that they didn't write (or don't remember writing).

SUMMARY

We've covered a lot of ground in this chapter. We began with an introduction to ASP.NET MVC, showing how the ASP.NET web framework and the MVC software pattern combine to provide a powerful system for building web applications. We looked at how ASP.NET MVC has matured through three previous releases, examining in more depth the features and focus of ASP.NET MVC 4. With the background established, you set up your development environment and began creating a sample MVC 4 application. You finished up by looking at the structure and components of an MVC 4 application. You'll be looking at all those components in more detail in the following chapters, starting with controllers in Chapter 2.