1

Getting Started

—by Jon Galloway

WHAT'S IN THIS CHAPTER?

- Understanding ASP.NET MVC
- An overview of ASP.NET MVC 5
- How to create MVC 5 applications
- How MVC applications are structured

This chapter gives you a quick introduction to ASP.NET MVC, explains how ASP.NET MVC 5 fits into the ASP.NET MVC release history, summarizes what's new in ASP.NET MVC 5, and shows you how to set up your development environment to build ASP.NET MVC 5 applications.

This is a Professional Series book about a version 5 web framework, so we keep the introductions short. We're not going to spend any time convincing you that you should learn ASP.NET MVC. We assume 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 lifecycle, 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, and so on. 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 that manipulates the data and enforces 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 (also known as the "DRY" principle)
> - Pluggability wherever possible
> - Try to be helpful, but if necessary, get out of the developer's way

The Road to MVC 5

In the five years since ASP.NET MVC 1 was released in March 2009, we've seen five major releases of ASP.NET MVC and several more interim releases. To understand ASP.NET MVC 5, 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.

> **DON'T PANIC!**
>
> We list some MVC-specific features in this section that might not all make sense to you if you're new to MVC. Don't worry! We explain some context behind the MVC 5 release, but if this doesn't all make sense, you can just skim or even skip until the "Creating an MVC 5 Application" section. We'll get you up to speed in the following chapters.

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 whether 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 March 13, 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 the client and server
- Strongly typed HTML helpers
- Improved Visual Studio tooling

It also had 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

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 talk about Razor in a lot more depth in Chapter 3.

MVC 4 Overview

The MVC 4 release built 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

Because MVC 4 is still a pretty recent release, we explain a few of these features in a little more detail here and describe 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 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 included 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, and so on) 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, and so on.
- **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, 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.

> **NOTE** *As described previously, MVC and Web API have a lot in common (model-controller patterns, routing, filters, etc.). Architectural reasons dictated that they would be separate frameworks which shared common models and paradigms in MVC 4 and 5. For example, MVC has maintained compatibility and a common codebase (e.g. the* `System.Web`*'s* `HttpContext`*) with ASP.NET, which didn't fit the long term goals of Web API.*
>
> *However, in May 2014 the ASP.NET team announced their plans to merge MVC, Web API and Web Pages in MVC 6. This next release is part of what is being called ASP.NET vNext, which is planned to run on a "cloud optimized" version of the .NET Framework. These framework changes provide a good*

opportunity to move MVC beyond `System.Web`*, which means it can more easily merge with Web API to form a next generation web stack. The goal is to support MVC 5 with minimal breaking changes. The.NET Web Development and Tools blog announcement post lists some of these plans as follows:*

- MVC, Web API, and Web Pages will be merged into one framework, called MVC 6. MVC 6 has no dependency on System.Web.
- ASP.NET vNext includes new cloud-optimized versions of MVC 6, SignalR 3, and Entity Framework 7.
- ASP.NET vNext will support true side-by-side deployment for all dependencies, including .NET for cloud. Nothing will be in the GAC.
- ASP.NET vNext is host-agnostic. You can host your app in IIS, or self-host in a custom process.
- Dependency injection is built into the framework.
- Web Forms, MVC 5, Web API 2, Web Pages 3, SignalR 2, EF 6 will be fully supported on ASP.NET vNext.
- .NET vNext (Cloud Optimized) will be a subset of the .NET vNext Framework, optimized for cloud and server workloads.
- MVC 6, SignalR 3, EF 7 will have some breaking changes:
  - New project system
  - New configuration system
  - MVC / Web API / Web Pages merge, using a common set of abstractions for HTTP, routing, action selection, filters, model binding, and so on
  - No System.Web, new lightweight HttpContext

For more information, see: `http://blogs.msdn.com/b/webdev/archive/2014/05/13/asp-net-vnext-the-future-of-net-on-the-server.aspx`.

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 5 automatically uses the mobile view when viewed in a mobile browser. Although the default determination of mobile browsers is based on user agent detection, you can customize this logic by registering your own custom device modes.

You find out more about Display modes in the mobile web discussion in Chapter 16.

Bundling and Minification

ASP.NET MVC 4 (and later) supports the same bundling and minification framework 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, enabling 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 5 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, because the references are made to script and CSS bundles instead of 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")
```

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

Open-Source Release

ASP.NET MVC has been under an open-source license since the initial release, but it was just open-source code instead of 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 (also known as *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 shipped with the MVC 5 release. External

code submissions are reviewed and tested by the ASP.NET team, and when released Microsoft will support them just as they have any of the previous ASP.NET MVC releases.

Even if you're not planning to contribute any source code, the public repository makes a huge difference in visibility. Although in the past you needed to wait for interim releases to see what the team was working on, you can now 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.

ASP.NET MVC 5 OVERVIEW

MVC 5 was released along with Visual Studio 2013 in October 2013. The main focus of this release was on a "One ASP.NET" initiative (described in the following sections) and core enhancements across the ASP.NET frameworks. Some of the top features include:

- One ASP.NET
- New Web Project Experience
- ASP.NET Identity
- Bootstrap templates
- Attribute Routing
- ASP.NET scaffolding
- Authentication filters
- Filter overrides

One ASP.NET

Options are nice. Web applications vary quite a bit, and web tools and platforms are not "one size fits all."

On the other hand, some choices can be paralyzing. We don't like having to choose one thing if it means giving up something else. This applies doubly to choices at the beginning of a project: I'm just getting started; I have no idea what this project will require a year down the line!

In previous versions of MVC, you were faced with a choice every time you created a project. You had to choose between an MVC application, Web Forms application, or some other project type. After you had made your decision, you were essentially trapped. You could kind of add Web Forms to an MVC application, but adding MVC to a Web Forms application was difficult. MVC applications had a special project type GUID hidden in their `csproj` file, and that was just one of the mysterious changes you had to make when attempting to add MVC to Web Forms applications.

In MVC 5, that all goes away, because just one ASP.NET project type exists. When you create a new web application in Visual Studio 2013, there's no difficult choice, just a Web application. This isn't just supported when you first create an ASP.NET project; you can add in support for other frameworks as you develop, because the tooling and features are delivered as NuGet packages. For

example, if you change your mind later on, you can use ASP.NET Scaffolding to add MVC to any existing ASP.NET application.

New Web Project Experience

As part of the new One ASP.NET experience, the dialogs for creating a new MVC application in Visual Studio 2013 have been merged and simplified. You find out more about the new dialogs later in this chapter, in the section titled "Creating an MVC 5 Application."

ASP.NET Identity

The membership and authentication systems in MVC 5 have been completely rewritten as part of the new ASP.NET Identity system. This new system moves beyond some outdated constraints of the previous ASP.NET Membership system, while adding some sophistication and configurability to the Simple Membership system that shipped with MVC 4.

Here are some of the top new features in ASP.NET Identity:

- **One ASP.NET Identity system:** In support of the One ASP.NET focus we discussed earlier, the new ASP.NET Identity was designed to work across the ASP.NET family (MVC, Web Forms, Web Pages, Web API, SignalR, and hybrid applications using any combination).
- **Control over user profile data:** Although it's a frequently used application for storing additional, custom information about your users, the ASP.NET Membership system made doing it very difficult. ASP.NET Identity makes storing additional user information (for example, account numbers, social media information, and contact address) as easily as adding properties to the model class that represents the user.
- **Control over persistence:** By default, all user information is stored using Entity Framework Code First. This gives you both the simplicity and control you're used to with Entity Framework Code First. However, you can plug in any other persistence mechanism you want, including other ORMs, databases, your own custom web services, and so on.
- **Testability:** The ASP.NET Identity API was designed using interfaces. These allow you to write unit tests for your user-related application code.
- **Claims Based:** Although ASP.NET Identity continues to offer support for user roles, it also supports claims-based authentication. Claims are a lot more expressive than roles, so this gives you a lot more power and flexibility. Whereas role membership is a simple Boolean value (a user either is or isn't in the Administrator role), a user claim can carry rich information, such as a user's membership level or identity specifics.
- **Login providers:** Rather than just focusing on username / password authentication, ASP.NET Identity understands that users often are authenticated through social providers (for example, Microsoft Account, Facebook, or Twitter) and Windows Azure Active Directory.

- **NuGet distribution:** ASP.NET Identity is installed in your applications as a NuGet package. This means you can install it separately, as well as upgrade to newer releases with the simplicity of updating a single NuGet package.

We'll discuss ASP.NET Identity in more detail in Chapter 7.

Bootstrap 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 1-1. (The blue doesn't show in this black and white book, but you get the idea.)

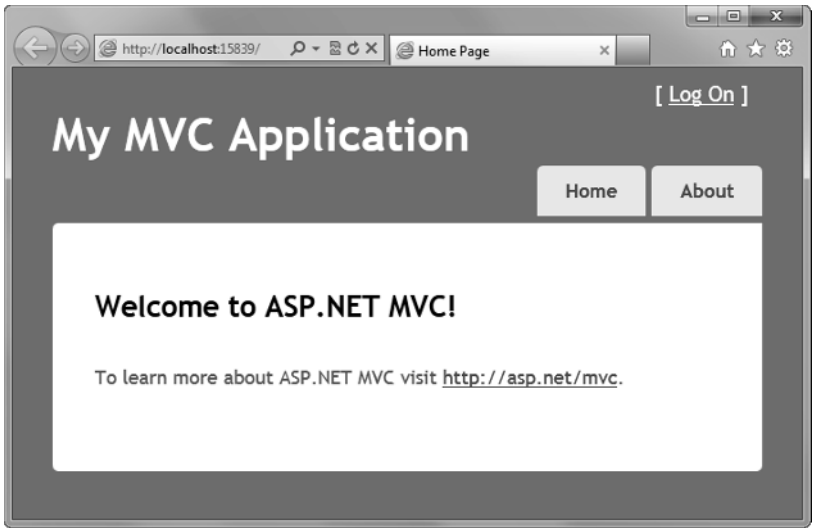

FIGURE 1-1

In MVC 4, both the HTML and CSS for the default templates were redesigned to look somewhat presentable out of the box. They also work well in different screen resolutions. However, the HTML and CSS in the MVC 4 default templates were all custom, which wasn't ideal. Visual design updates were tied to the MVC product release cycle, and you couldn't easily share design templates with the broader web development community.

In MVC 5, the project templates moved to run on the popular Bootstrap framework. Bootstrap was first created by a developer and a designer at Twitter, who later split off to focus on Bootstrap completely. The default design for MVC 5 actually looks like something you might deploy to production, as shown in Figure 1-2.

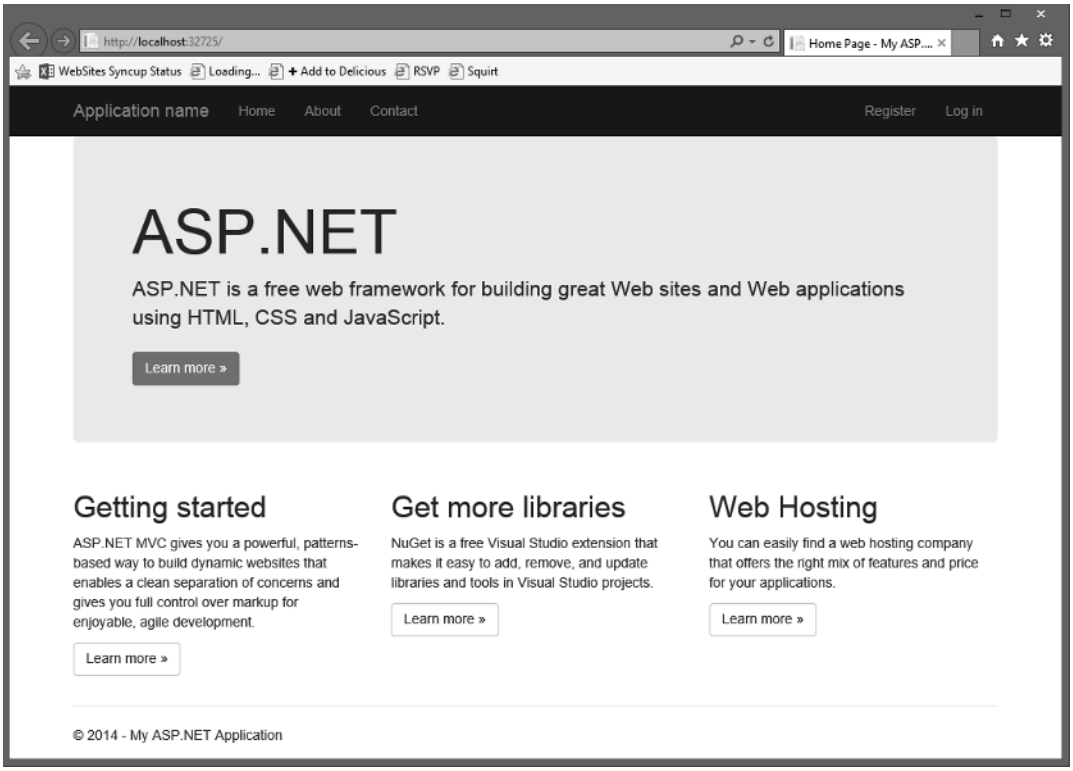

FIGURE 1-2

What's even nicer is that, because the Bootstrap framework has broad acceptance across the web developer community, a large variety of Bootstrap themes (both free and paid) are available from sites like `http://wrapbootstrap.com` and `http://bootswatch.com`. For example, Figure 1-3 shows a default MVC 5 application using the free Slate theme from Bootswatch.

Chapter 16 covers Bootstrap in more detail, when you look at optimizing your MVC applications for mobile web browsers.

Attribute Routing

Attribute Routing is a new option for specifying routes by placing annotations on your controller classes or action methods. It was made possible due to an open source contribution from the popular AttributeRouting project (`http://attributerouting.net`).

Chapter 9 describes Attribute Routing in detail.

ASP.NET Scaffolding

Scaffolding is the process of generating boilerplate code based on your model classes. MVC has had scaffolding since version 1, but it was limited to MVC projects. The new ASP.NET scaffolding

system works in any ASP.NET application. Additionally, it includes support for building powerful custom scaffolders, complete with custom dialogs and a comprehensive scaffolding API.

Chapters 3 and 4 describe scaffolding basics, and Chapter 16 explains two ways you can extend the scaffolding system.

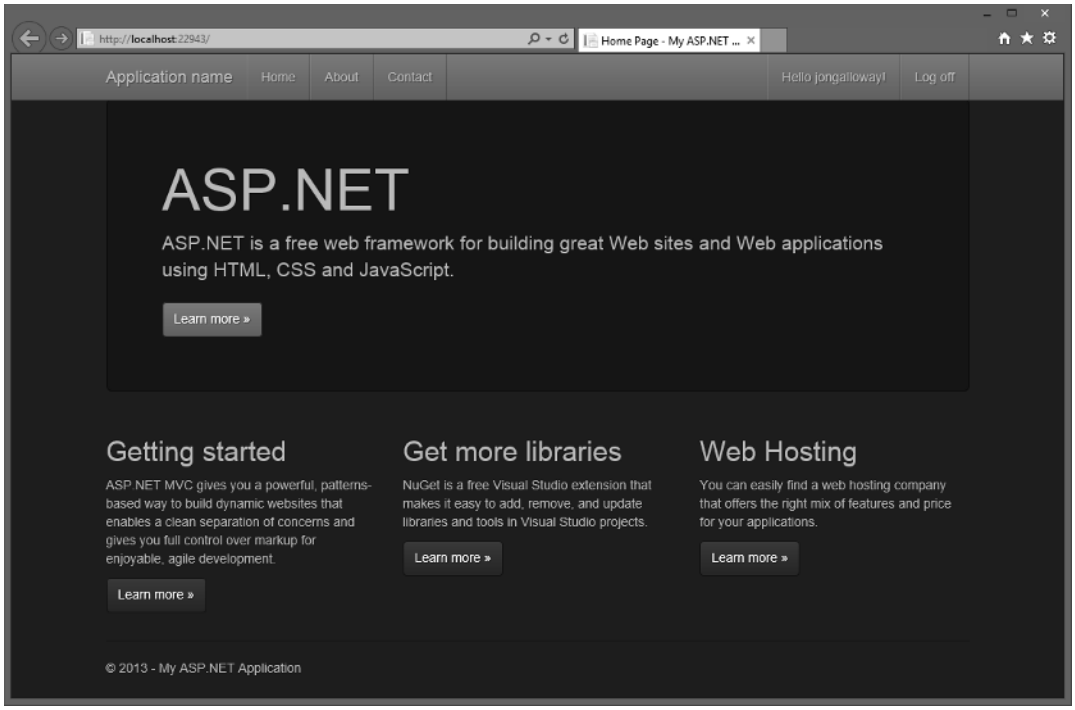

FIGURE 1-3

Authentication Filters

MVC has long supported a feature called authorization filters, which allow you to restrict access to a controller or action based on role membership or other custom logic. However, as discussed in Chapter 7, there's an important distinction between *authentication* (determining who a user is) and *authorization* (what an authenticated user is allowed to do). The newly added authentication filters execute before the authorize filter, allowing you to access the user claims that ASP.NET Identity provides and to run your own custom authentication logic.

Chapter 15 covers authentication filters in detail.

Filter Overrides

Filters are an advanced MVC feature that allow the developer to participate in the action and result execution pipeline. Filter overrides mean that you can exclude a controller or actions from executing a global filter.

Chapter 15 describes filters in detail, including filter overrides.

INSTALLING MVC 5 AND CREATING APPLICATIONS

The best way to learn about how MVC 5 works is to get started by building an application, so let's do that.

Software Requirements for ASP.NET MVC 5

MVC 5 requires .NET 4.5. As such, it runs on the following Windows client operating systems:

- Windows Vista SP2
- Windows 7
- Windows 8

It runs on the following server operating systems:

- Windows Server 2008 R2
- Windows Server 2012

Installing ASP.NET MVC 5

After ensuring you've met the basic software requirements, it's time to install ASP.NET MVC 5 on your development and production machines. Fortunately, that's pretty simple.

> **SIDE-BY-SIDE INSTALLATION WITH PREVIOUS VERSIONS OF MVC**
>
> MVC 5 installs side-by-side with previous versions of MVC, so you can install and start using MVC 5 right away. You'll still be able to create and update existing applications running on previous versions.

Installing the MVC 5 Development Components

The developer tooling for ASP.NET MVC 5 supports Visual Studio 2012 and Visual Studio 2013, including the free Express versions of both products.

MVC 5 is included with Visual Studio 2013, so there's nothing to install. If you're using Visual Studio 2012, you can install MVC 5 support using this installer: `http://www.microsoft.com/en-us/download/41532`. Note that all screenshots in this book show Visual Studio 2013 rather than Visual Studio 2012.

Server Installation

MVC 5 is completely bin deployed, meaning that all necessary assemblies are included in the bin directory of your application. As long as you have .NET 4.5 on your server, you're set.

Creating an ASP.NET MVC 5 Application

You can create a new MVC 5 application using either Visual Studio 2013 or Visual Studio 2013 Express for Web 2013. The experience in both IDEs is very similar; because this is a *Professional Series* book we focus on Visual Studio development, mentioning Visual Web Developer only when there are significant differences.

MVC MUSIC STORE

We loosely base some of our samples on the MVC Music Store tutorial. This tutorial is available online at `http://mvcmusicstore.codeplex.com` and includes an e-book tutorial covering the basics of building an MVC application. We go quite a bit further than the basics in this book, but having a common base is nice if you need more information on the introductory topics.

To create a new MVC project:

1. Choose File ➪ New Project, as shown in Figure 1-4.

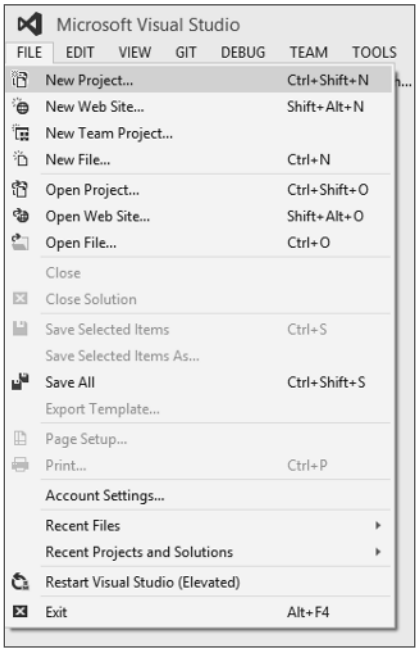

FIGURE 1-4

2. In the Installed Templates section in the left column of the New Project dialog, shown in Figure 1-5, select the Visual C# ➪ Web templates list. A list of web application types appears in the center column.

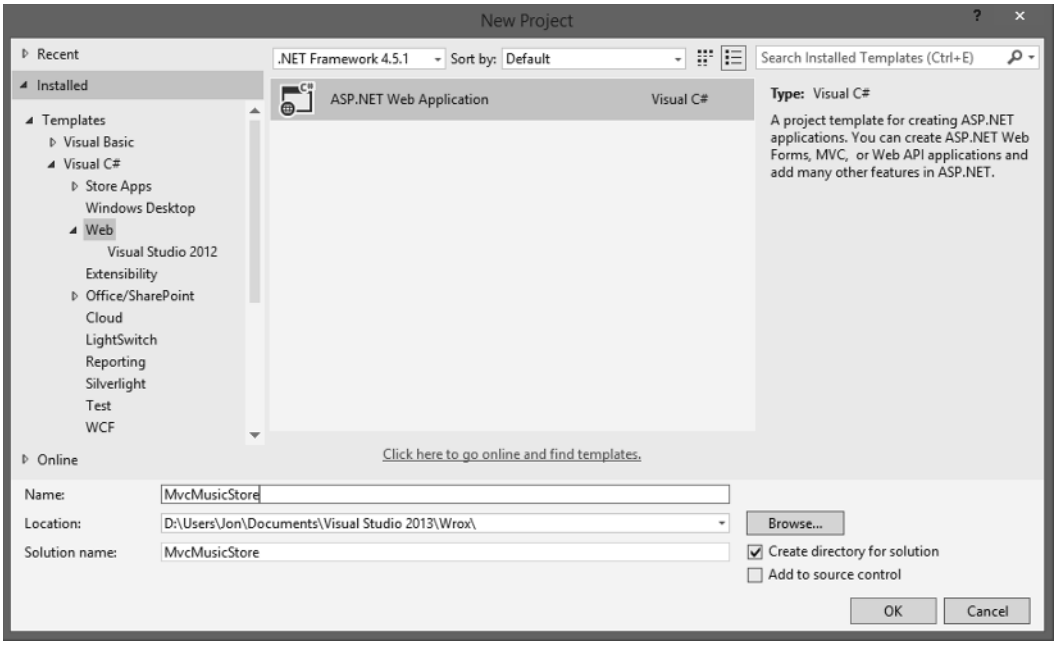

FIGURE 1-5

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

> **ONE ASP.NET PROJECT TEMPLATE**
>
> Note that there isn't an MVC project type; there's just an ASP.NET Web Application. Whereas previous versions of Visual Studio and ASP.NET used a different project type for MVC, in Visual Studio 2013 they've been united into one common project type.

The New ASP.NET Project Dialog

After you create a new MVC 5 application, the New ASP.NET Project dialog appears, as shown in Figure 1-6. This presents common options for all ASP.NET applications:

- Select a template
- Add framework-specific folders and core references
- Add unit tests

- Configure authentication
- Windows Azure (Visual Studio 2013.2 and later)

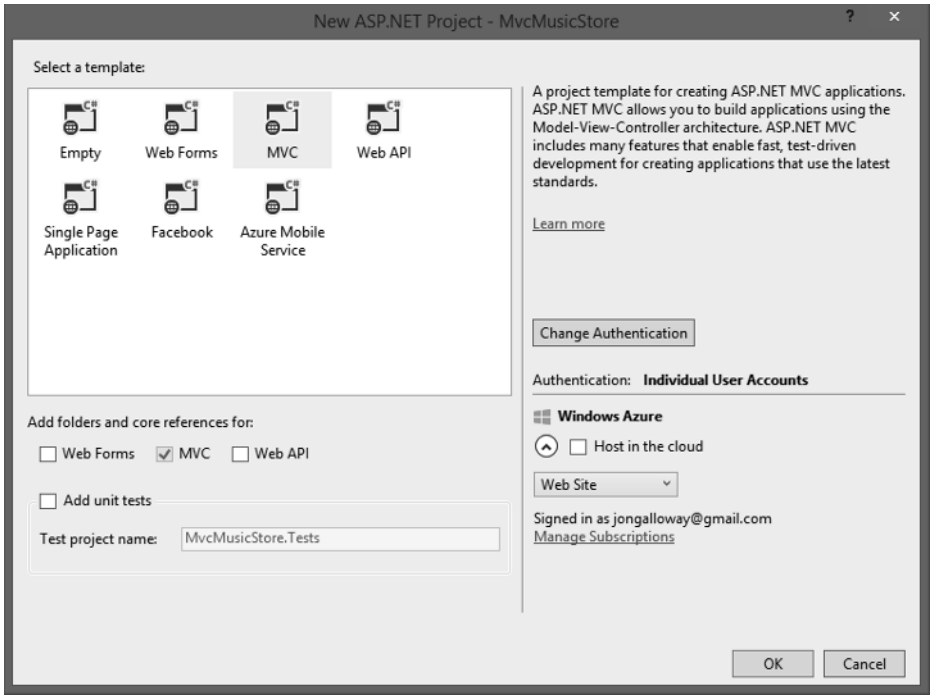

FIGURE 1-6

The first two selections (Select a Template and Add Folders and Core References For) work together. The template selects the starting point, but then you can use the framework checkboxes to add support for Web Forms, MVC, and Web API. This means you can select an MVC template and add in Web Forms support, or select an Empty template and add in support for any of the frameworks. That capability extends beyond new project creation; you can add in support for any of the frameworks at any time, because the framework folders and core references are added via NuGet packages.

Remember the discussion in the earlier "One ASP.NET" section: Template and core reference selections are options, not hard choices. They'll help you get started, but they won't lock you in.

Selecting an Application Template

Because you can use the Add Folders and Core References For option on any project, why do you need anything more than an Empty template? Well, the application templates give you a little more of a start by setting up some common things (as described in the list that follows) for a "mostly

MVC," "mostly Web API," or "mostly Web Forms" application. This section reviews those templates now. Remember, though, they're just conveniences in Visual Studio 2013 rather than requirements; you could start with an Empty template and add in MVC support two weeks later by adding the NuGet packages.

- **MVC:** Let's start with this template, because it's the one you'll use the most. The MVC template sets up a standard home controller with a few views, configures the site layout, and includes an MVC-specific Project_Readme.html page. The next section digs into this in a lot more detail.
- **Empty:** As you would expect, the empty template sets you up with an empty project skeleton. You get a `web.config` (with some default website configuration settings) and a few assembly references you'll need to get started, but that's it. There's no code, no JavaScript includes or CSS, not even a static HTML file. You can't run an empty project until you put something in it. The empty template is for people who want to start completely from scratch.
- **Web Forms:** The Web Forms template sets you up for ASP.NET Web Forms development.

> **NOTE** *You can learn more about Web Forms development in the Wrox book titled* Professional ASP.NET 4.5 in C# and VB *if you're interested. However, it's listed here because you can create a project using the Web Forms template and still add in support for MVC.*

- **Web API:** This creates an application with both MVC and Web API support. The MVC support is included partly to display the API Help pages, which document the public API signature. You can read more about Web API in Chapter 11.
- **Single Page Application:** The Single Page Application template sets you up for an application that's primarily driven via JavaScript requests to Web API services rather than the traditional web page request / response cycle. The initial HTML is served via an MVC Home Controller, but the rest of the server-side interactions are handled by a Web API controller. This template uses the Knockout.js library to help manage interactions in the browser. Chapter 12 covers single-page applications, although the focus is on the Angular.js library rather than Knockout.js.
- **Facebook:** This template makes it easier to build a Facebook "Canvas" application, a web application that appears hosted inside of the Facebook website. This template is beyond the scope of this book, but you can read more about it in this tutorial: `http://go.microsoft.com/fwlink/?LinkId=301873`.

> **NOTE** Changes to the Facebook API have caused authorization redirection issues with this template at the time of this writing, as detailed in this CodePlex issue: `https://aspnetwebstack.codeplex.com/workitem/1666`. The fix will likely require updating or replacing the `Microsoft.AspNet.Mvc.Facebook` NuGet package. Consult the bug reference above for status and fix information.

- **Azure Mobile Service:** If you have Visual Studio 2013 Update 2 (also known as 2013.2) installed, you'll see this additional option. Because Azure Mobile Services now support Web API services, this template makes it easy to create a Web API intended for Azure Mobile Services. You can read more about it in this tutorial: `http://msdn.microsoft.com/en-us/library/windows/apps/xaml/dn629482.aspx`.

Testing

All the built-in project templates have an option to create a unit test project with sample unit tests.

RECOMMENDATION: CHECK THE BOX

I hope you get in the habit of checking that Add Unit Tests box for *every* project you create.

I'm not going to try to sell you the Unit Testing religion—not just yet. We talk about unit testing throughout the book, especially in Chapter 14, 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 value exists 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 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.

Configuring Authentication

You can choose the authentication method by clicking the Change Authentication button, which then opens the Change Authentication dialog, as shown in Figure 1-7.

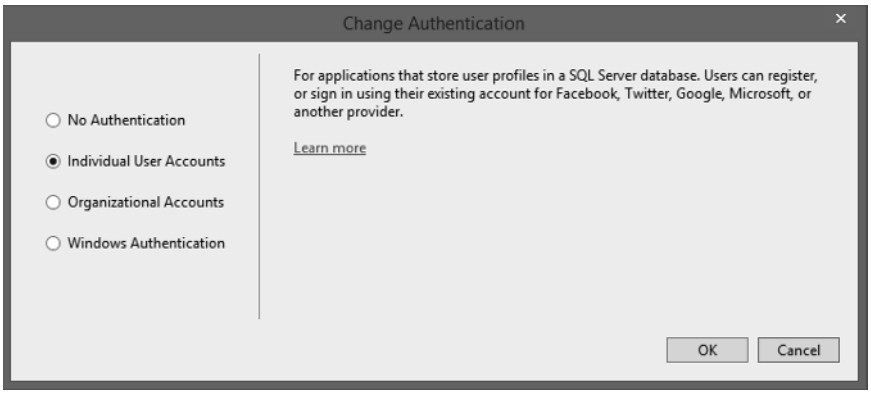

FIGURE 1-7

There are four options:

- **No Authentication:** Used for an application that requires no authentication, such as a public website with no administration section.
- **Individual User Accounts:** Used for applications that store user profiles locally, such as in a SQL Server database. This includes support for username / password accounts as well as social authentication providers.
- **Organizational Accounts:** Used for accounts that authenticate via some form of Active Directory (including Azure Active Directory and Office 365).
- **Windows Authentication:** Used for intranet applications.

This book most often uses Individual User Accounts. Chapter 7 offers a discussion of some of the additional options. You can click the Learn More link for each option in the Change Authentication dialog for the official documentation.

Configuring Windows Azure Resources

Visual Studio 2013.2 adds an additional "Host in the cloud" option to configure Azure resources for your project right from the File ➪ New Project dialog. For more information about using this option, see this tutorial: `http://azure.microsoft.com/en-us/documentation/articles/web-sites-dotnet-get-started/`. For this chapter, we'll run against the local development server, so ensure this checkbox is unchecked.

Review your settings on the New ASP.NET MVC 5 Project dialog to make sure they match Figure 1-8, and then click OK.

This creates a solution for you with two projects—one for the web application and one for the unit tests, as shown in Figure 1-9.

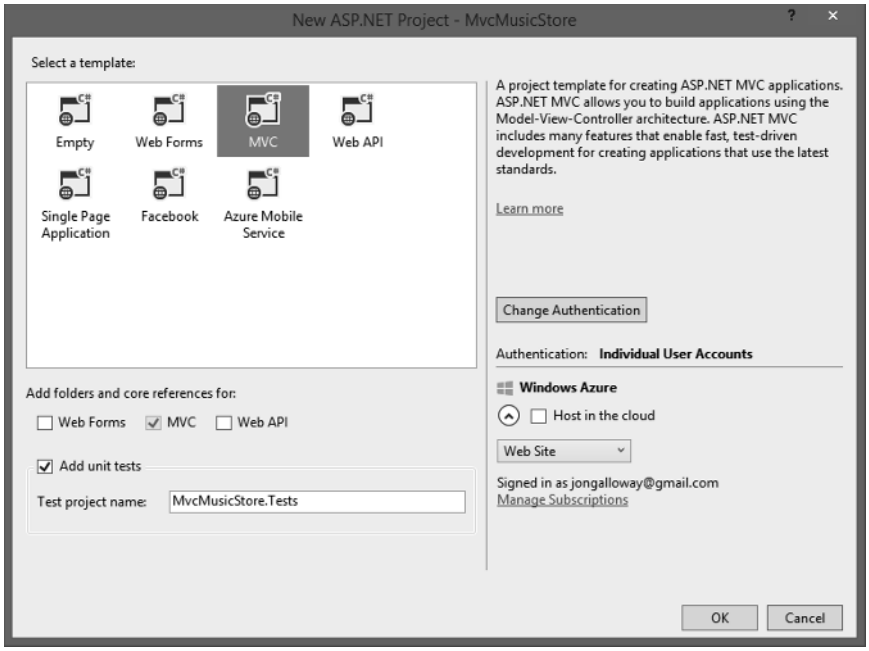

FIGURE 1-8

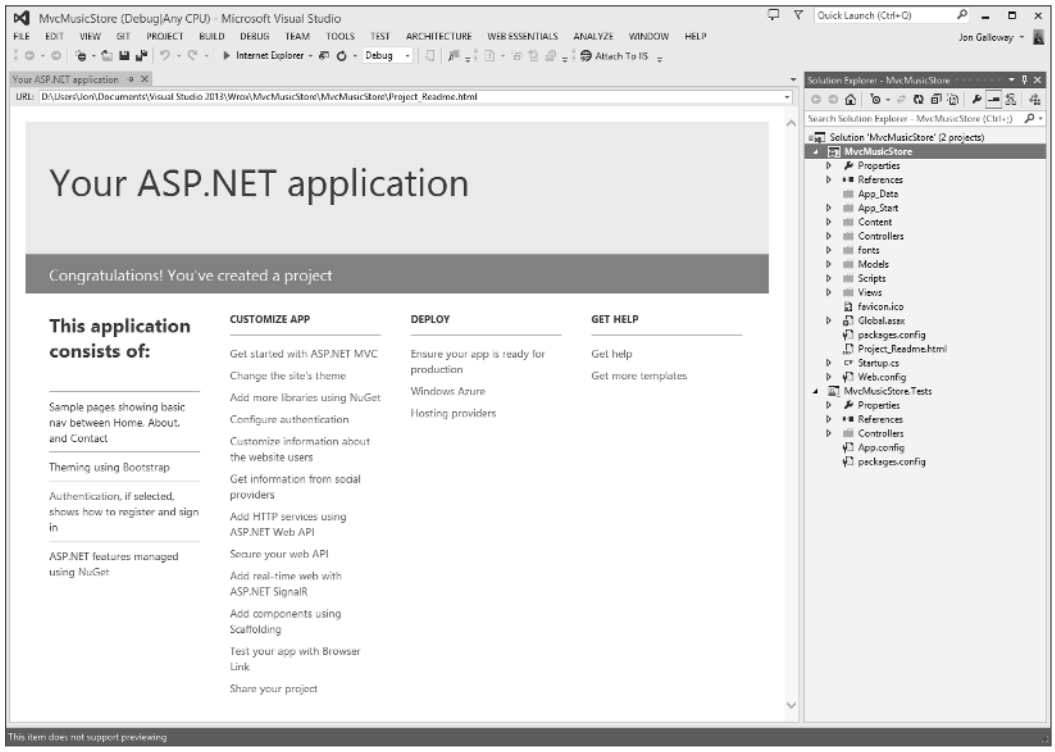

FIGURE 1-9

New MVC projects include a `Project_Readme.html` file in the root of the application. This file is automatically displayed when your project is created, as shown in Figure 1-9. It is completely self-contained—all styles are included via HTML style tags, so when you're done with it you can just delete the one file. This `Project_Readme.html` file is customized for each application template and contains a lot of useful links to help you get started.

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 1-10. ASP.NET MVC projects created with the Internet application template have eight top-level directories, shown in Table 1-1.

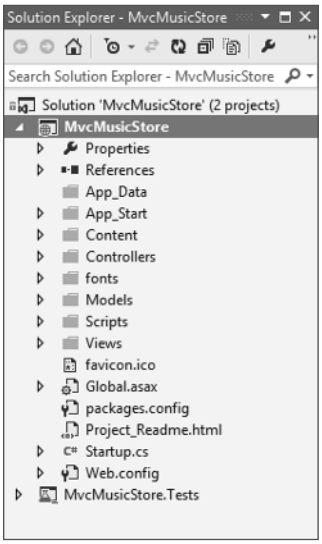

FIGURE 1-10

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`)

DIRECTORY	PURPOSE
/fonts	The Bootstrap template system includes some custom web fonts, which are placed in this directory
/Content	Where you put CSS, images, and other site content, other than scripts
/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 (see Figure 1-11)—HomeController and AccountController—by default to the project.

FIGURE 1-11

- 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 1-12).

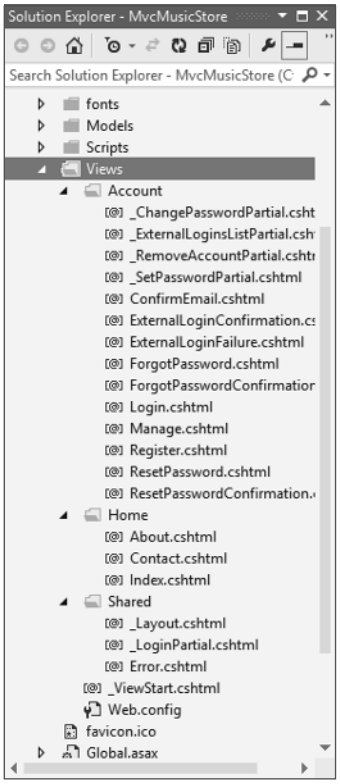

FIGURE 1-12

- The `/Content` and `/Scripts` directories, you'll find the CSS files that is used to style all HTML on the site, as well as JavaScript libraries that can enable jQuery support within the application (see Figure 1-13).
- The `MvcMusicStore.Tests` project, you'll find a class that contains unit tests for your `HomeController` classes (see Figure 1-14).

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).

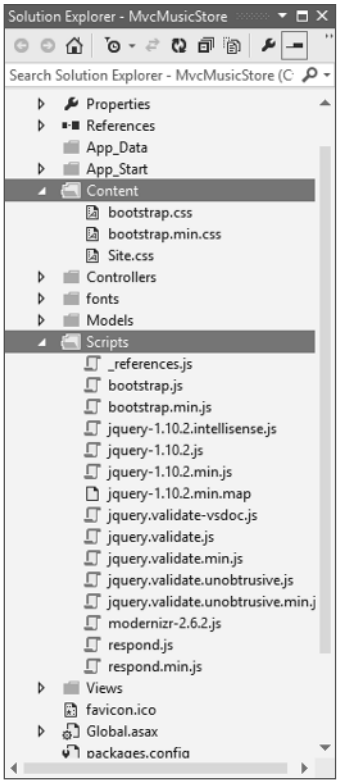

FIGURE 1-13

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."

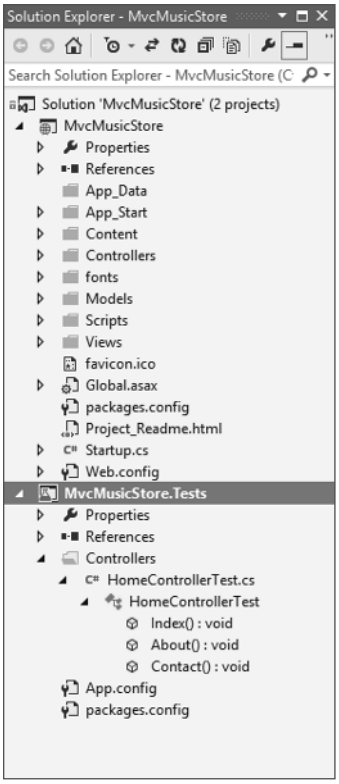

FIGURE 1-14

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 four previous releases, examining in more depth the features and focus of ASP.NET MVC 5. With the background established, you set up your development environment and began creating a sample MVC 5 application. You finished up by looking at the structure and components of an MVC 5 application. You'll be looking at all those components in more detail in the following chapters, starting with controllers in Chapter 2.

REMINDER FOR ADVANCED READERS

As mentioned in the introduction, the first six chapters of this book are intended to provide a firm foundation in the fundamentals of ASP.NET MVC. If you already have a pretty good grasp of how ASP.NET MVC works, you might want to skip ahead to Chapter 7.