Developing Real-World Web Applications with ASP.NET MVC

Programming

ASP.NET MVC 4

Free Sampler

Jess Chadwick, Todd Snyder & Hrusikesh Panda

O'REILLY®

Programming ASP.NET MVC 4
by Jess Chadwick, Todd Snyder, and Hrusikesh Panda

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Rachel Roumeliotis
Production Editor: Rachel Steely
Copyeditor: Rachel Head
Proofreader: Leslie Graham, nSight

Indexer: Lucie Haskins
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Rebecca Demarest

October 2012: First Edition.

Revision History for the First Edition:
2012-09-14 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449320317* for release details.

ISBN: 978-1-449-32031-7

[LSI]

1347629749

Table of Contents

CHAPTER 1

Fundamentals of ASP.NET MVC

Microsoft ASP.NET MVC is a web application development framework built on top of Microsoft's popular and mature .NET Framework. The ASP.NET MVC Framework leans heavily on proven developmental patterns and practices that place an emphasis on a loosely coupled application architecture and highly maintainable code.

In this chapter we'll take a look at the fundamentals of what makes ASP.NET MVC tick—from its proud lineage and the architectural concepts on which it is built, to the use of Microsoft Visual Studio 2011 to create a fully functioning ASP.NET MVC web application. Then we'll dive into the ASP.NET MVC web application project and see just what ASP.NET MVC gives you right from the start, including a working web page and built-in forms authentication to allow users to register and log in to your site.

By the end of the chapter, you'll have not only a working ASP.NET MVC web application, but also enough understanding of the fundamentals of ASP.NET MVC to begin building applications with it immediately. The rest of this book simply builds on these fundamentals, showing you how to make the most of the ASP.NET MVC Framework in any web application.

Microsoft's Web Development Platforms

Understanding the past can be a big help in appreciating the present; so, before we get into what ASP.NET MVC is and how it works, let's take a minute to see just where it came from.

Long ago, Microsoft saw the need for a Windows-based web development platform, and the company worked hard to produce a solution. Over the past two decades, Microsoft has given the development community several web development platforms.

Active Server Pages (ASP)

Microsoft's first answer to web development was Active Server Pages (ASP), a scripting language in which code and markup are authored together in a single file, with each

physical file corresponding to a page on the website. ASP's server-side scripting approach became widely popular and many websites grew out of it. Some of these sites continue to serve visitors today. After a while, though, developers wanted more. They asked for features such as improved code reuse, better separation of concerns, and easier application of object-oriented programming principles. In 2002, Microsoft offered ASP.NET as a solution to these concerns.

ASP.NET Web Forms

Like ASP, ASP.NET websites rely on a page-based approach where each page on the website is represented in the form of a physical file (called a Web Form) and is accessible using that file's name. Unlike a page using ASP, a Web Forms page provides some separation of code and markup by splitting the web content into two different files: one for the markup and one for the code. ASP.NET and the Web Forms approach served developers' needs for many years, and this continues to be the web development framework of choice for many .NET developers. Some .NET developers, however, consider the Web Forms approach too much of an abstraction from the underlying HTML, JavaScript, and CSS. Some developers just can't be pleased! Or can they?

ASP.NET MVC

Microsoft was quick to spot the growing need in the ASP.NET developer community for something different than the page-based Web Forms approach, and the company released the first version of ASP.NET MVC in 2008. Representing a total departure from the Web Forms approach, ASP.NET MVC abandons the page-based architecture completely, relying on the *Model-View-Controller* (MVC) architecture instead.

Unlike ASP.NET Web Forms, which was introduced as a replacement to its predecessor, ASP, ASP.NET MVC does not in any way *replace* the existing Web Forms Framework. Quite the contrary—both ASP.NET MVC and Web Forms applications are built on top of the common ASP.NET Framework, which provides a common web API that both frameworks leverage quite heavily.

The idea that ASP.NET MVC and Web Forms are just different ways of making an ASP.NET website is a common theme throughout this book; in fact, both Chapter 2 and Appendix A explore this concept in depth.

The Model-View-Controller Architecture

The Model-View-Controller pattern is an architectural pattern that encourages strict isolation between the individual parts of an application. This isolation is better known as *separation of concerns*, or, in more general terms, "loose coupling." Virtually all

aspects of MVC—and, consequently, the ASP.NET MVC Framework—are driven by this goal of keeping disparate parts of an application isolated from each other.

Architecting applications in a loosely coupled manner brings a number of both short- and long-term benefits:

Development
: Individual components do not directly depend on other components, which means that they can be more easily developed in isolation. Components can also be readily replaced or substituted, preventing complications in one component from affecting the development of other components with which it may interact.

Testability
: Loose coupling of components allows test implementations to stand in for "production" components. This makes it easier to, say, avoid making calls to a database, by replacing the component that makes database calls with one that simply returns static data. The ability for components to be easily swapped with mock representations greatly facilitates the testing process, which can drastically increase the reliability of the system over time.

Maintenance
: Isolated component logic means that changes are typically isolated to a small number of components—often just one. Since the risk of change generally correlates to the scope of the change, modifying fewer components is a good thing!

The MVC pattern splits an application into three layers: the model, the view, and the controller (see Figure 1-1). Each of these layers has a very specific job that it is responsible for and—most important—is not concerned with how the other layers do their jobs.

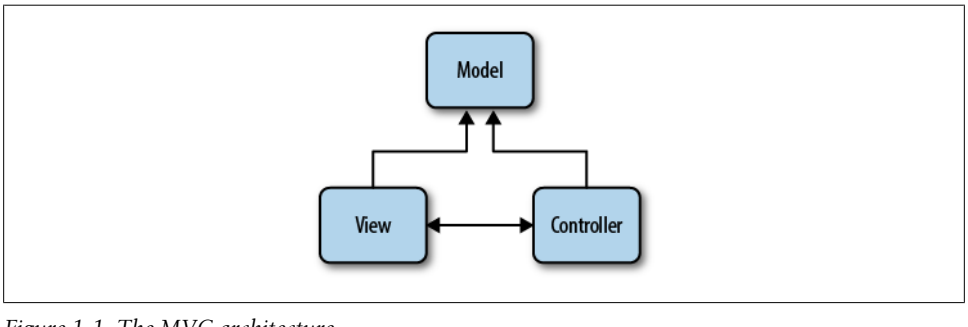

Figure 1-1. The MVC architecture

The Model

The *model* represents core business logic and data. Models encapsulate the properties and behavior of a domain entity and expose properties that describe the entity. For example, the `Auction` class represents the concept of an "auction" in the application

and may expose properties such as `Title` and `CurrentBid`, as well as exposing behavior in the form of methods such as `Bid()`.

The View

The *view* is responsible for transforming a model or models into a visual representation. In web applications, this most often means generating HTML to be rendered in the user's browser, although views can manifest in many forms. For instance, the same model might be visualized in HTML, PDF, XML, or perhaps even in a spreadsheet.

Following separation of concerns, views should concentrate only on *displaying* data and should not contain any business logic themselves—the business logic stays in the model, which should provide the view with everything it needs.

The Controller

The *controller*, as the name implies, controls the application logic and acts as the coordinator between the view and the model. Controllers receive input from users via the view, then work with the model to perform specific actions, passing the results back to the view.

What's New in ASP.NET MVC 4?

This book explores the ASP.NET MVC Framework in depth, showing how to make the most of the features and functionality it offers. Since we're now up to the fourth version of the framework, however, much of what the book covers is functionality that existed prior to this latest version. If you are already familiar with previous versions of the framework, you're probably eager to skip over what you already know and begin learning all about the new additions.

The list below gives a brief description of each of the features new to version 4 of ASP.NET MVC, along with references pointing you to the sections of the book that show these features in action:

Asynchronous controllers
: Internet Information Server (IIS) processes each request it receives on a new thread, so each new request ties up one of the finite number of threads available to IIS, even if that thread is sitting idle (for example, waiting for a response from a database query or web service). And, while recent updates in .NET Framework 4.0 and IIS 7 have drastically increased the default number of threads available to the IIS thread pool, it's still a good practice to avoid holding on to system resources for longer than you need to. Version 4 of the ASP.NET MVC Framework introduces *asynchronous controllers* to better handle these types of long-running requests in a more asynchronous fashion. Through the use of asynchronous controllers, you can tell the framework to free up the thread that is processing your request, letting it

perform other processing tasks while it waits for the various tasks in the request to finish. Once they finish, the framework picks up where it left off, and returns the same response as if the request had gone through a normal synchronous controller—except now you can handle many more requests at once! If you're interested in learning more about asynchronous controllers, see Chapter 11, which explains them in depth.

Display modes
: A growing number of devices are Internet-connected and ready to surf your site, and you need to be ready for them. Many times, the data displayed on these devices is the same as the data displayed on desktop devices, except the visual elements need to take into consideration the smaller form factor of mobile devices. ASP.NET MVC *display modes* provide an easy, convention-based approach for tailoring views and layouts to target different devices. Chapter 10 shows how to apply display modes to your site as part of a holistic approach to adding mobile device support to your sites.

Bundling and minification
: Even though it may seem like the only way to get on the Internet these days is through some sort of high-speed connection, that doesn't mean you can treat the client-side resources that your site depends on in a haphazard manner. In fact, when you consider how the overall download times are increasing, wasting even fractions of a second in download times can really add up and begin to have a very negative effect on the perceived performance of your site. Concepts such as script and stylesheet combining and minification may not be anything new, but with the .NET Framework 4.5 release, they are now a fundamental part of the framework. What's more, ASP.NET MVC embraces and extends the core .NET Framework functionality to make this tooling even more usable in your ASP.NET MVC applications. Chapter 13 helps you tackle all of these concepts and also shows you how to use the new tooling offered in the core ASP.NET and ASP.NET MVC Frameworks.

Web API
: Simple HTTP data services are rapidly becoming the primary way to supply data to the ever-increasing variety of applications, devices, and platforms. ASP.NET MVC has always provided the ability to return data in various formats, including JSON and XML; however, the *ASP.NET Web API* takes this interaction a step further, providing a more modern programming model that focuses on providing full-fledged data *services* rather than controller actions that happen to return data. In Chapter 6, you'll see how to really take advantage of AJAX on the client—and you'll use ASP.NET Web API services to do it!

Did You Know…?

ASP.NET MVC is open source! That's right—as of March 2012, the entire source code for the ASP.NET MVC, Web API, and Web Pages Frameworks is available to browse and download on CodePlex. What's more, developers are free to create their own forks and even submit patches to the core framework source code!

Introduction to EBuy

This book aims to show you not only the ins and outs of the ASP.NET MVC Framework, but also how to leverage the framework in real-world applications. The problem with such applications is that the very meaning of "real-world" indicates a certain level of complexity and uniqueness that can't be adequately represented in a single demo application.

Instead of attempting to demonstrate solutions to every problem you may face, we—the authors of this book—have assembled a list of the scenarios and issues that we have most frequently encountered and that we most frequently hear of others encountering. Though this list of scenarios may not include every scenario you'll face while developing your application, we believe it represents the majority of the real-world problems that most developers face over the course of creating their ASP.NET MVC applications.

We're not kidding, we actually wrote a list—and it's in the back of this book! Appendix D has a cross-referenced list of all the features and scenarios we cover and the chapter(s) in which we cover them.

In order to cover the scenarios on this list, we came up with a web application that combines them all into as close to a real-world application as we could get, while still limiting the scope to something everyone understands: an online auction site.

Introducing EBuy, the online auction site powered by ASP.NET MVC! From a high level, the goals of the site are pretty straightforward: allow users to list items they wish to sell, and bid on items they wish to buy. As you take a deeper look, however, you'll begin to see that the application is a bit more complex than it sounds, requiring not only everything ASP.NET MVC has to offer, but also integration with other technologies.

EBuy is not just a bunch of code that we ship along with the book, though. Each chapter of the book not only introduces more features and functionality, but uses them to build the EBuy application—from new project to deployed application, preferably while you follow along and write the code, too!

OK, we'll admit that EBuy is *also* "just a bunch of code." In fact, you can download EBuy in its entirety from the book's website: *http://www.programmingaspnetmvc.com*.

Now, let's stop talking about an application that doesn't exist yet and start building it!

Installing ASP.NET MVC

In order to begin developing ASP.NET MVC applications, you'll need to download and install the ASP.NET MVC 4 Framework. This is as easy as visiting the ASP.NET MVC website and clicking the Install button.

This launches the Web Platform Installer, a free tool that simplifies the installation of many web tools and applications. Follow the Web Platform Installer wizard to download and install ASP.NET MVC 4 and its dependencies to your machine.

Note that in order to install and use ASP.NET MVC 4, you must have at least PowerShell 2.0 and Visual Studio 2010 Service Pack 1 or Visual Web Developer Express 2010 Service Pack 1. Luckily, if you do not already have them installed, the Web Platform Installer should figure it out and proceed to download and install the latest versions of PowerShell and Visual Studio for you!

If you are currently using the previous version of ASP.NET MVC and would like to both create ASP.NET MVC 4 applications and continue working with ASP.NET MVC 3 applications, fear not—ASP.NET MVC can be installed and run side by side with ASP.NET MVC 3 installations.

Once you've gotten everything installed, it's time to proceed to the next step: creating your first ASP.NET MVC 4 application.

Creating an ASP.NET MVC Application

The ASP.NET MVC 4 installer adds a new Visual Studio project type named *ASP.NET MVC 4 Web Application*. This is your entry point to the world of ASP.NET MVC and is what you'll use to create the new EBuy web application project that you'll build on as you progress through this book.

To create a new project, select the Visual C# version of the ASP.NET MVC 4 Web Application template and enter Ebuy.Website into the Name field (see Figure 1-2).

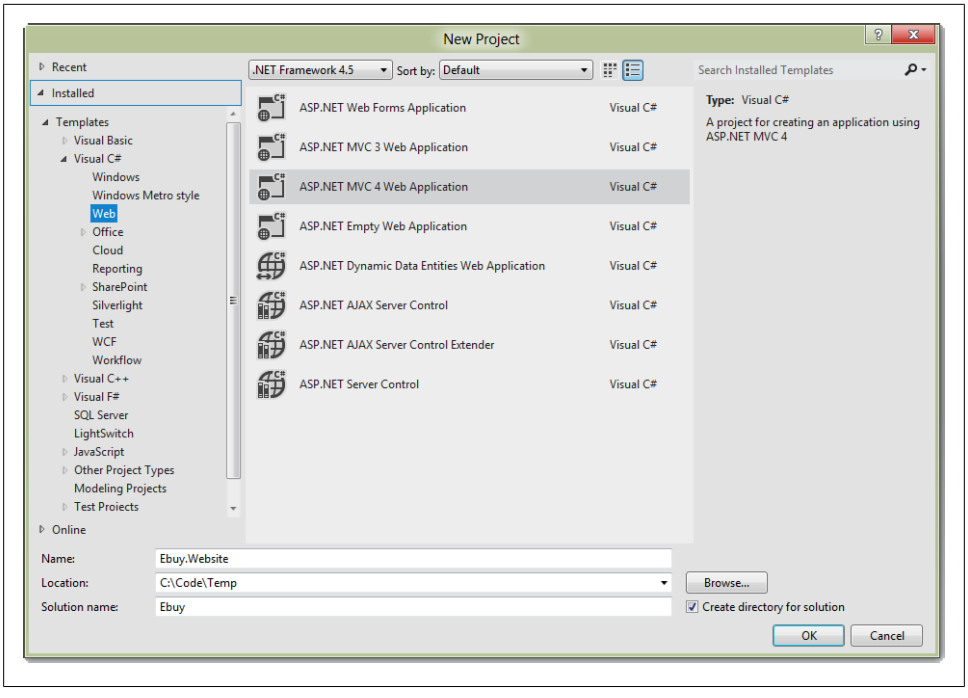

Figure 1-2. Creating the EBuy project

When you click OK to continue, you'll be presented with another dialog with more options (see Figure 1-3).

This dialog lets you customize the ASP.NET MVC 4 application that Visual Studio is going to generate for you by letting you specify what kind of ASP.NET MVC site you want to create.

Project Templates

To begin, ASP.NET MVC 4 offers several project templates, each of which targets a different scenario:

Empty
: The *Empty* template creates a bare-bones ASP.NET MVC 4 application with the appropriate folder structure that includes references to the ASP.NET MVC assemblies as well as some JavaScript libraries that you'll probably use along the way. The template also includes a default view layout and generates a *Global.asax* file that includes the standard configuration code that most ASP.NET MVC applications will need.

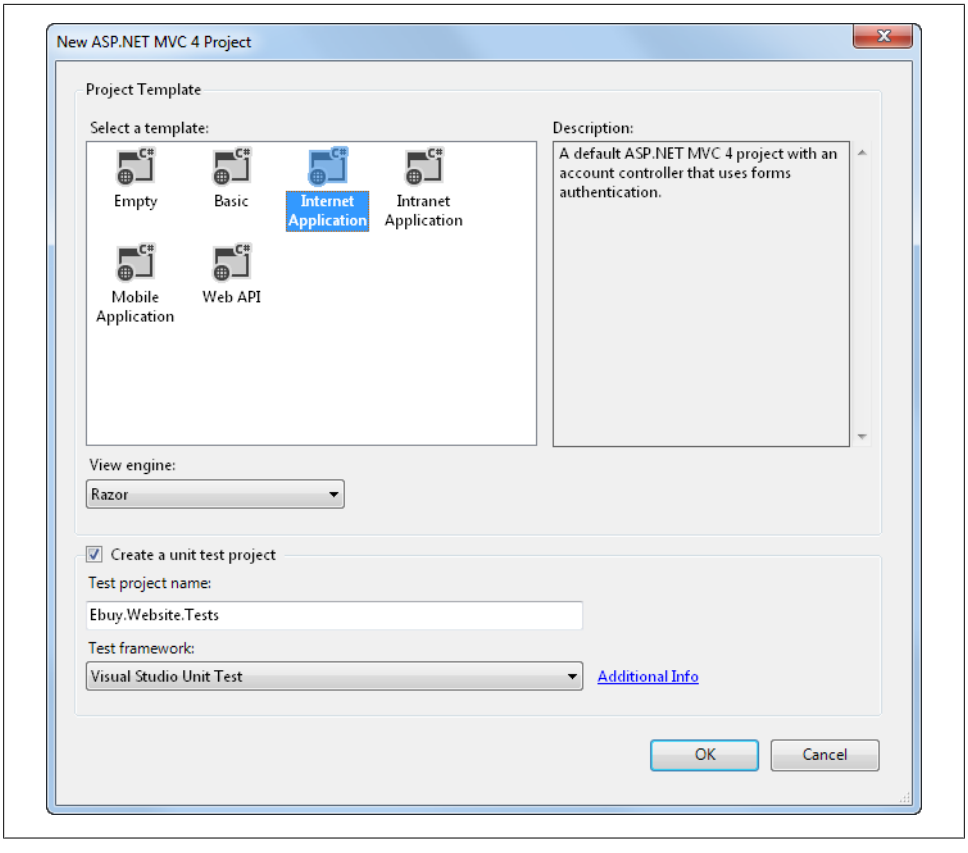

Figure 1-3. Customizing the EBuy project

Basic
: The *Basic* template creates a folder structure that follows ASP.NET MVC 4 conventions and includes references to the ASP.NET MVC assemblies. This template represents the bare minimum that you'll need to begin creating an ASP.NET MVC 4 project, but no more—you'll have to do all the work from here!

Internet Application
: The *Internet Application* template picks up where the Empty template leaves off, extending the Empty template to include a simple default controller (`HomeController`), an `AccountController` with all the logic required for users to register and log in to the website, and default views for both of these controllers.

Intranet Application
: The *Intranet Application* template is much like the Internet Application template, except that it is preconfigured to use Windows-based authentication, which is desirable in intranet scenarios.

Mobile Application
: The *Mobile Application* template is another variation of the Internet Application template. This template, however, is optimized for mobile devices and includes the jQuery Mobile JavaScript framework and views that apply the HTML that works best with jQuery Mobile.

Web API
: The *Web API* template is yet another variation of the Internet Application template that includes a preconfigured Web API controller. Web API is the new lightweight, RESTful HTTP web services framework that integrates quite nicely with ASP.NET MVC. Web API is a great choice for quickly and easily creating data services that your AJAX-enabled applications can easily consume. Chapter 6 covers this new API in great detail.

The New ASP.NET MVC Project dialog also lets you select a *view engine*, or syntax that your views will be written in. We'll be using the new Razor syntax to build the EBuy reference application, so you can leave the default value ("Razor") selected. Rest assured that you can change the view engine your application uses at any time—this option exists only to inform the wizard of the kind of views it should *generate* for you, not to lock the application into a specific view engine forever.

Finally, choose whether or not you'd like the wizard to generate a unit test project for this solution. Once again, you don't have to worry about this decision too much—as with any other Visual Studio solution, you are able to add a unit test project to an ASP.NET MVC web application anytime you'd like.

When you're happy with the options you've selected, click OK to have the wizard generate your new project!

NuGet Package Management

If you pay attention to the status bar as Visual Studio creates your new web application project, you may notice messages (such as "Installing package AspNetMvc...") referring to the fact that the project template is utilizing the *NuGet Package Manager* to install and manage the assembly references in your application. The concept of using a package manager to manage application dependencies—especially as part of the new project template phase—is quite powerful, and also new to ASP.NET MVC 4 project types.

Introduced as part of the ASP.NET MVC 3 installer, NuGet offers an alternative workflow for managing application dependencies. Though it is not actually part of the ASP.NET MVC Framework, NuGet is doing much of the work behind the scenes to make your projects possible.

A NuGet package may contain a mixture of assemblies, content, and even tools to aid in development. In the course of installing a package, NuGet will add the assemblies to the target project's References list, copy any content into the application's folder structure, and register any tools in the current path so that they can be executed from the Package Manager Console.

However, the most important aspect of NuGet packages—indeed, the primary reason NuGet was created to begin with—has to do with *dependency management*. .NET applications are not monolithic, single-assembly applications—most assemblies rely on references to other assemblies in order to do their job. What's more, assemblies generally depend on specific *versions* (or, at least, a minimum version) of other assemblies.

In a nutshell, NuGet calculates the potentially complex relationships between all of the assemblies that an application depends on, then makes sure that you have all of the assemblies you need—and the correct versions of those assemblies.

Your gateway to NuGet's power is the NuGet Package Manager. You can access the NuGet Package Manager in two ways:

The graphical user interface
: The NuGet Package Manager has a graphical user interface (GUI) that makes it easy to search for, install, update, and uninstall packages for a project. You can access the graphical Package Manager interface by right-clicking the website project in the Solution Explorer and selecting the "Manage NuGet Packages..." option.

The Console mode
: The Library Package Manager Console is a Visual Studio window containing an integrated PowerShell prompt specially configured for Library Package Manager access. If you do not see the Package Manager Console window already open in Visual Studio, you can access it via the Tools > Library Package Manager > Package Manager Console menu option. To install a package from the Package Manager Console window, simply type the command `Install-Package _Package Name_`. For example, to install the Entity Framework package, execute the `Install-Package EntityFramework` command. The Package Manager Console will proceed to download the *EntityFramework* package and install it into your project. After the "Install-Package" step has completed, the Entity Framework assemblies will be visible in the project's References list.

Convention over Configuration

To make website development easier and help developers be more productive, ASP.NET MVC relies on the concept of *convention over configuration* whenever possible. This means that, instead of relying on explicit configuration settings, ASP.NET MVC simply assumes that developers will follow certain conventions as they build their applications.

The ASP.NET MVC project folder structure (Figure 1-4) is a great example of the framework's use of convention over configuration. There are three special folders in the project that correspond to the elements of the MVC pattern: the *Controllers*, *Models*, and *Views* folders. It's pretty clear at a glance what each of these folders contains.

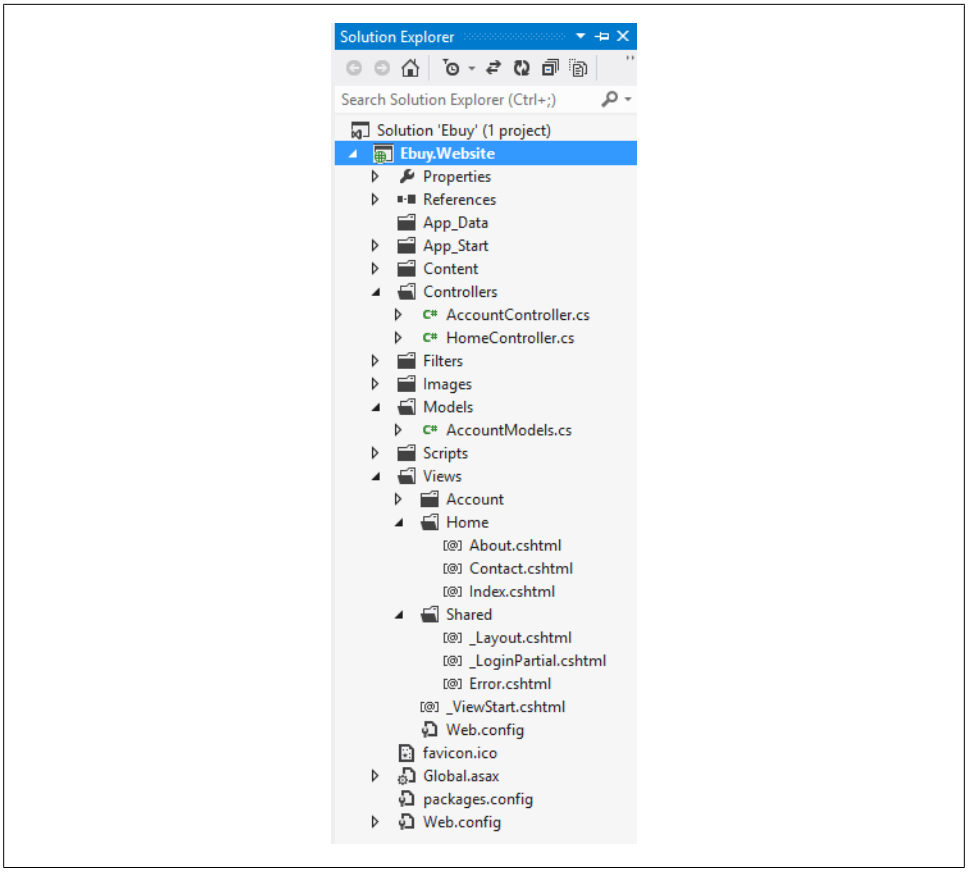

Figure 1-4. The ASP.NET MVC project folder structure

When you look at the contents of these folders, you'll find even more conventions at work. For example, not only does the *Controllers* folder contain all of the application's controller classes, but the controller classes all follow the convention of ending their names with the *Controller* suffix. The framework uses this convention to register the application's controllers when it starts up and associate controllers with their corresponding routes.

Next, take a look at the *Views* folder. Beyond the obvious convention dictating that the application's views should live under this folder, it is split into subfolders: a *Shared* folder, and an optional folder to contain the views for each controller. This convention helps save developers from providing explicit locations of the views they'd like to display to users. Instead, developers can just provide the name of a view—say, "Index"—and the framework will try its best to find the view within the *Views* folder, first in the controller-specific folder and then, failing that, in the *Shared* views folder.

At first glance, the concept of convention over configuration may seem trivial. However, these seemingly small or meaningless optimizations can really add up to significant time savings, improved code readability, and increased developer productivity.

Running the Application

Once your project is created, feel free to hit F5 to execute your ASP.NET MVC website and watch it render in your browser.

Congratulations, you've just created your first ASP.NET MVC 4 application!

After you've calmed down from the immense excitement you experience as a result of making words show up in a web browser, you might be left wondering, "What just happened? *How did it do that*?"

Figure 1-5 shows, from a high level, how ASP.NET MVC processes a request.

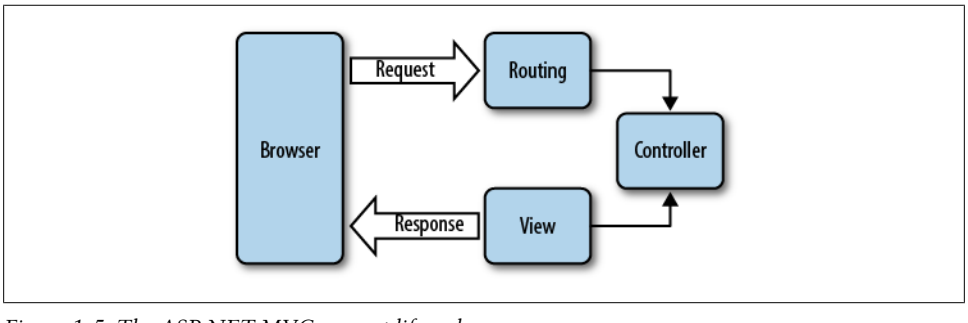

Figure 1-5. The ASP.NET MVC request lifecycle

Though we'll spend the rest of this book diving deeper and deeper into the components of that diagram, the next few sections start out by explaining those fundamental building blocks of ASP.NET MVC.

Routing

All ASP.NET MVC traffic starts out like any other website traffic: with a request to a URL. This means that, despite the fact that it is not mentioned anywhere in the name, the *ASP.NET Routing* framework is at the core of every ASP.NET MVC request.

In simple terms, ASP.NET routing is just a pattern-matching system. At startup, the application registers one or more patterns with the framework's *route table* to tell the routing system what to do with any requests that match those patterns. When the routing engine receives a request at runtime, it matches that request's URL against the URL patterns registered with it (Figure 1-6).

When the routing engine finds a matching pattern in its route table, it forwards the request to the appropriate handler for that request.

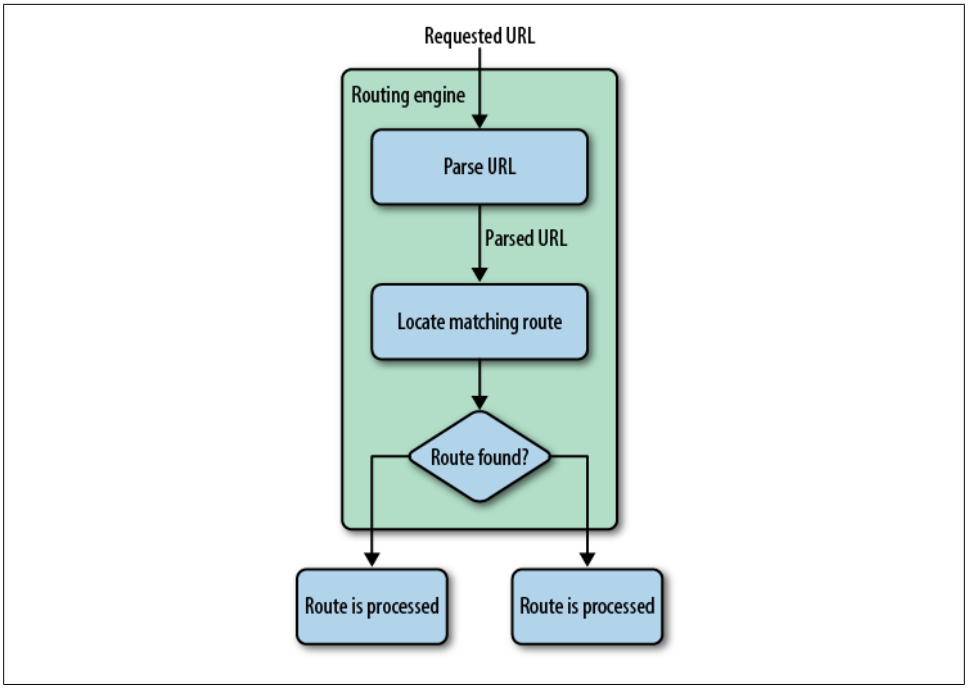

Figure 1-6. ASP.NET routing

Otherwise, when the request's URL does not match any of the registered route patterns, the routing engine indicates that it could not figure out how to handle the request by returning a 404 HTTP status code.

Configuring Routes

ASP.NET MVC routes are responsible for determining which controller method (otherwise known as a *controller action*) to execute for a given URL. They consist of the following properties:

Unique name
: A name may be used as a specific reference to a given route

URL pattern
: A simple pattern syntax that parses matching URLs into meaningful segments

Defaults
: An optional set of default values for the segments defined in the URL pattern

Constraints
: A set of constraints to apply against the URL pattern to more narrowly define the URLs that it matches

The default ASP.NET MVC project templates add a generic route that uses the following URL convention to break the URL for a given request into three named segments, wrapped with brackets ({}): "controller", "action", and "id":

```
{controller}/{action}/{id}
```

This route pattern is registered via a call to the `MapRoute()` extension method that runs during application startup (located in *App_Start/RouteConfig.cs*):

```
routes.MapRoute(
    "Default", // Route name
    "{controller}/{action}/{id}", // URL with parameters
    new { controller = "Home", action = "Index",
        id = UrlParameter.Optional } // Parameter defaults
);
```

In addition to providing a name and URL pattern, this route also defines a set of default parameters to be used in the event that the URL fits the route pattern, but doesn't actually provide values for every segment.

For instance, Table 1-1 contains a list of URLs that match this route pattern, along with corresponding values that the routing framework will provide for each of them.

Table 1-1. Values provided for URLs that match our route pattern

URL	Controller	Action	ID
/auctions/auction/1234	`AuctionsController`	`Auction`	`1234`
/auctions/recent	`AuctionsController`	`Recent`	
/auctions	`AuctionsController`	`Index`	
/	`HomeController`	`Index`	

The first URL (*/auctions/auction/1234*) in the table is a perfect match because it satisfies every segment of the route pattern, but as you continue down the list and remove segments from the end of the URL, you begin to see defaults filling in for values that are not provided by the URL.

This is a very important example of how ASP.NET MVC leverages the concept of convention over configuration: when the application starts up, ASP.NET MVC discovers all of the application's controllers by searching through the available assemblies for classes that implement the `System.Web.Mvc.IController` interface (or derive from a class that implements this interface, such as `System.Web.Mvc.Controller`) *and* whose class names end with the suffix *Controller*. When the routing framework uses this list to figure out which controllers it has access to, it chops off the *Controller* suffix from all of the controller class names. So, whenever you need to refer to a controller, you do so by its shortened name, e.g., `AuctionsController` is referred to as *Auctions*, and `HomeController` becomes *Home*.

What's more, the controller and action values in a route are not case-sensitive. This means that each of these requests—*/Auctions/Recent*, */auctions/Recent*, */auctions/recent*, or even */aucTionS/rEceNt*—will successfully resolve to the `Recent` action in the `AuctionsController`.

URL route patterns are relative to the application root, so they do not need to start with a forward slash (/) or a virtual path designator (~/). Route patterns that include these characters are invalid and will cause the routing system to throw an exception.

As you may have noticed, URL routes can contain a wealth of information that the routing engine is able to extract. In order to process an ASP.NET MVC request, however, the routing engine must be able to determine two crucial pieces of information: the *controller* and the *action*. The routing engine can then pass these values to the ASP.NET MVC runtime to create and execute the specified action of the appropriate controller.

Controllers

In the context of the MVC architectural pattern, a *controller* responds to user input (e.g., a user clicking a Save button) and collaborates between the model, view, and (quite often) data access layers. In an ASP.NET MVC application, controllers are classes that contain methods that are called by the routing framework to process a request.

To see an example of an ASP.NET MVC controller, take a look at the `HomeController` class found in *Controllers/HomeController.cs*:

```
using System.Web.Mvc;

namespace Ebuy.Website.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Your app description page.";

            return View();
        }

        public ActionResult About()
        {
            ViewBag.Message = "Your quintessential app description page.";

            return View();
        }

        public ActionResult Contact()
```

```
        {
            ViewBag.Message = "Your quintessential contact page.";

            return View();
        }
    }
}
```

Controller Actions

As you can see, controller classes themselves aren't very special; that is, they don't look much different from any other .NET class. In fact, it's the *methods* in controller classes—referred to as *controller actions*—that do all the heavy lifting that's involved in processing requests.

You'll often hear the terms *controller* and *controller action* used somewhat interchangeably, even throughout this book. This is because the MVC pattern makes no differentiation between the two. However, the ASP.NET MVC Framework is mostly concerned with controller actions since they contain the actual logic to process the request.

For instance, the `HomeController` class we just looked at contains three actions: `Index`, `About`, and `Contact`. Thus, given the default route pattern `{controller}/{action}/{id}`, when a request is made to the URL */Home/About*, the routing framework determines that it is the `About()` method of the `HomeController` class that should process the request. The ASP.NET MVC Framework then creates a new instance of the `HomeController` class and executes its `About()` method.

In this case, the `About()` method is pretty simple: it passes data to the view via the `ViewBag` property (more on that later), and then tells the ASP.NET MVC Framework to display the view named "About" by calling the `View()` method, which returns an `ActionResult` of type `ViewResult`.

Action Results

It is very important to note that it is the controller's job to tell the ASP.NET MVC Framework *what* it should do next, but not *how* to do it. This communication occurs through the use of +ActionResult+s, the return values which every controller action is expected to provide.

For example, when a controller decides to show a view, it tells the ASP.NET MVC Framework to show the view by returning a `ViewResult`. It does not render the view itself. This loose coupling is another great example of separation of concerns in action (*what* to do versus *how* it should be done).

Despite the fact that every controller action needs to return an `ActionResult`, you will rarely be creating them manually. Instead, you'll usually rely on the helper methods that the `System.Web.Mvc.Controller` base class provides, such as:

`Content()`
: Returns a `ContentResult` that renders arbitrary text, e.g., "Hello, world!"

`File()`
: Returns a `FileResult` that renders the contents of a file, e.g., a PDF.

`HttpNotFound()`
: Returns an `HttpNotFoundResult` that renders a 404 HTTP status code response.

`JavaScript()`*:: Returns a* `JavaScriptResult`
: that renders JavaScript, e.g., "function hello() { alert(*Hello, World!*); }".

`Json()`
: Returns a `JsonResult` that serializes an object and renders it in JavaScript Object Notation (JSON) format, e.g., "{ "Message": *Hello, World!* }".

`PartialView()`
: Returns a `PartialViewResult` that renders only the content of a view (i.e., a view without its layout).

`Redirect()`
: Returns a `RedirectResult` that renders a 302 (temporary) status code to redirect the user to a given URL, e.g., "302 *http://www.ebuy.com/auctions/recent*". This method has a sibling, `RedirectPermanent()`, that also returns a `RedirectResult`, but uses HTTP status code 301 to indicate a permanent redirect rather than a temporary one.

`RedirectToAction()` *and* `RedirectToRoute()`
: Act just like the `Redirect()` helper, only the framework dynamically determines the external URL by querying the routing engine. Like the `Redirect()` helper, these two helpers also have permanent redirect variants: `RedirectToActionPermanent()` and `RedirectToRoutePermanent()`.

`View()`
: Returns a `ViewResult` that renders a view.

As you can tell from this list, the framework provides an action result for just about any situation you need to support, and, if it doesn't, you are free to create your own!

Though all controller actions are required to provide an `ActionResult` that indicates the next steps that should be taken to process the request, not all controller actions need to specify `ActionResult` as their return type. Controller actions can specify any return type that derives from `ActionResult`, or even any other type.

When the ASP.NET MVC Framework comes across a controller action that returns a non-`ActionResult` type, it automatically wraps the value in a `ContentResult` and renders the value as raw content.

Action Parameters

Controller actions are—when it comes down to it—just like any other method. In fact, a controller action can even specify parameters that ASP.NET MVC populates, using information from the request, when it executes. This functionality is called *model binding*, and it is one of ASP.NET MVC's most powerful and useful features.

Before diving into how model binding works, first take a step back and consider an example of the "traditional" way of interacting with request values:

```
public ActionResult Create()
{
   var auction = new Auction() {
      Title = Request["title"],
      CurrentPrice = Decimal.Parse(Request["currentPrice"]),
      StartTime = DateTime.Parse(Request["startTime"]),
      EndTime = DateTime.Parse(Request["endTime"]),
   };
   // ...
}
```

The controller action in this particular example creates and populates the properties of a new `Auction` object with values taken straight from the request. Since some of `Auction`'s properties are defined as various primitive, non-`string` types, the action also needs to parse each of those corresponding request values into the proper type.

This example may seem simple and straightforward, but it's actually quite frail: if any of the parsing attempts fails, the entire action will fail. Switching to the various `TryParse()` methods may help avoid most exceptions, but applying these methods also means additional code.

The side effect of this approach is that every action is very explicit. The downside to writing such explicit code is that it puts the burden on you, the developer, to perform all the work and to remember to perform this work every time it is required. A larger amount of code also tends to obscure the real goal: in this example, adding a new `Auction` to the system.

Model binding basics

Not only does model binding avoid all of this explicit code, it is also very easy to apply. So easy, in fact, that you don't even need to think about it.

For example, here's the same controller action as before, this time using model-bound method parameters:

```
public ActionResult Create(
      string title, decimal currentPrice,
      DateTime startTime, DateTime endTime
   )
{
   var auction = new Auction() {
      Title = title,
      CurrentPrice = currentPrice,
      StartTime = startTime,
      EndTime = endTime,
   };
   // ...
}
```

Now, instead of retrieving the values from the `Request` explicitly, the action declares them as parameters. When the ASP.NET MVC framework executes this method, it attempts to populate the action's parameters using the same values from the request that the previous example showed. Note that—even though we're not accessing the `Request` dictionary directly—the parameter names are still very important, because they still correspond to values from in the `Request`.

The `Request` object isn't the only place the ASP.NET MVC model binder gets its values from, however. Out of the box, the framework looks in several places, such as route data, query string parameters, form post values, and even serialized JSON objects. For example, the following snippet retrieves the `id` value from the URL simply by declaring a parameter with the same name:

Example 1-1. Retrieving the id from a URL (e.g. /auctions/auction/123)

```
public ActionResult Auction(long id)
{
    var context = new EBuyContext();
    var auction = context.Auctions.FirstOrDefault(x => x.Id == id);
    return View("Auction", auction);
}
```

Where and how the ASP.NET MVC model binder finds these values is actually quite configurable and even extensible. See Chapter 8 for an in-depth discussion of ASP.NET MVC model binding.

As these examples demonstrate, model binding lets ASP.NET MVC handle much of the mundane, boilerplate code so the logic within the action can concentrate on providing business value. The code that is left is much more meaningful, not to mention more readable.

Model binding complex objects

Applying the model binding approach even to simple, primitive types can make a pretty big impact in making your code more expressive. In the real world, though, things are much more complex—only the most basic scenarios rely on just a couple of parameters. Luckily, ASP.NET MVC supports binding to complex types as well as to primitive types.

This example takes one more pass at the `Create` action, this time skipping the middleman primitive types and binding directly to an `Auction` instance:

```
public ActionResult Create(Auction auction)
{
   // ...
}
```

The action shown here is equivalent to what you saw in the previous example. That's right—ASP.NET MVC's complex model binding just eliminated *all* of the boilerplate code required to create and populate a new `Auction` instance! This example shows the true power of model binding.

Action Filters

Action filters provide a simple yet powerful technique to modify or enhance the ASP.NET MVC pipeline by "injecting" logic at certain points, helping to address "cross-cutting concerns" that apply to many (or all) components of an application. Application logging is a classic example of a cross-cutting concern in that it is equally applicable to any component in an application, regardless of what that component's primary responsibility may be.

Action filter logic is primarily introduced by applying an `ActionFilterAttribute` to a controller action in order to affect how that action executes, as is the case in the following example that protects a controller action from unauthorized access by applying the `AuthorizeAttribute`:

```
[Authorize]
public ActionResult Profile()
{
    // Retrieve profile information for current user
    return View();
}
```

The ASP.NET MVC Framework includes quite a few action filters that target common scenarios. You'll see these action filters in use throughout this book, helping accomplish a variety of tasks in a clean, loosely coupled way.

Action filters are a great way to apply custom logic throughout your site. Keep in mind that you are free to create your own action filters by extending the `ActionFilterAttribute` base class or any of the ASP.NET MVC action filters.

Views

In the ASP.NET MVC Framework, controller actions that wish to display HTML to the user return an instance of `ViewResult`, a type of `ActionResult` that knows how to render content to the response. When it comes time to render the view, the ASP.NET MVC Framework will look for the view using the name provided by the controller.

Take the `Index` action in the `HomeController`:

```
public ActionResult Index()
{
    ViewBag.Message = "Your app description page.";
    return View();
}
```

This action takes advantage of the `View()` helper method to create a `ViewResult`. Calling `View()` without any parameters, as in this example, instructs ASP.NET MVC to find a view with the same name as the current controller action. In this instance, ASP.NET MVC will look for a view named "Index", but where will it look?

Locating Views

ASP.NET MVC relies on the convention that keeps all the application's views underneath the *Views* folder in the root of the website. More specifically, ASP.NET MVC expects views to live within folders named after the controller to which they relate.

Thus, when the framework wants to show the view for the `Index` action in the `HomeController`, it is going to look in the */Views/Home* folder for a file named *Index*. The screenshot in Figure 1-7 shows that the project template was nice enough to include an *Index.cshtml* view for us.

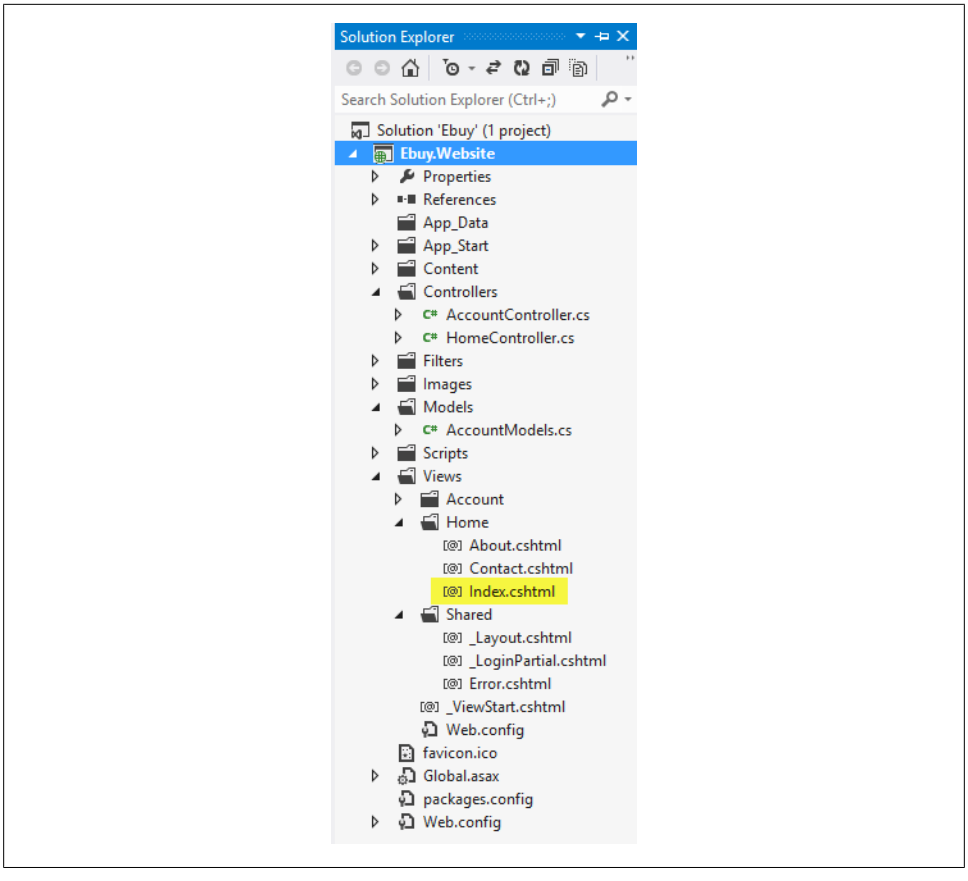

Figure 1-7. Locating the Index view

When it does not find a view that matches the name of the view it is looking for in the controller's Views folder, ASP.NET MVC continues looking in the common */Views/Shared* folder.

The */Views/Shared* folder is a great place to keep views that are shared across multiple controllers.

Now that you've found the view that the action requested, open it up and take a look at what's inside: HTML markup and code. But, it's not just *any* HTML markup and code—it's *Razor*!

Hello, Razor!

Razor is a syntax that allows you to combine code and content in a fluid and expressive manner. Though it introduces a few symbols and keywords, Razor is not a new language. Instead, Razor lets you write code using languages you probably already know, such as C# or Visual Basic .NET.

Razor's learning curve is very short, because it lets you work with your existing skills rather than requiring you to learn an entirely new language. Therefore, if you know how to write HTML and .NET code using C# or Visual Basic .NET, you can easily write markup such as the following:

```
<div>This page rendered at @DateTime.Now</div>
```

Which produces the following output:

```
<div>This page rendered at 12/7/1941 7:38:00 AM</div>
```

This example begins with a standard HTML tag (the `<div>` tag), followed by a bit of "hardcoded" text, then a bit of dynamic text rendered as the result of referencing a .NET property (`System.DateTime.Now`), followed by the closing (`</div>`) tag.

Razor's intelligent parser allows developers to be more expressive with their logic and make easier transitions between code and markup. Though Razor's syntax might be different from other markup syntaxes (such as the Web Forms syntax), it's ultimately working toward the same goal: rendering HTML.

To illustrate this point, take a look at the following snippets that show examples of common scenarios implemented in both Razor markup and Web Forms markup.

Here is an `if/else` statement using Web Forms syntax:

```
<% if(User.IsAuthenticated) { %>
    <span>Hello, <%: User.Username %>!</span>
<% } %>
<% else { %>
    <span>Please <%: Html.ActionLink("log in") %></span>
<% } %>
```

and using Razor syntax:

```
@if(User.IsAuthenticated) {
    <span>Hello, @User.Username!</span>
} else {
    <span>Please @Html.ActionLink("log in")</span>
}
```

And here is a `foreach` loop using Web Forms syntax:

```
<ul>
<% foreach( var auction in auctions) { %>
    <li><a href="<%: auction.Href %>"><%: auction.Title %></a></li>
<% } %>
</ul>
```

and using Razor syntax:

```
<ul>
@foreach( var auction in auctions) {
    <li><a href="@auction.Href">@auction.Title</a></li>
}
</ul>
```

Though they use a different syntax, the two snippets for each of the examples render the same HTML.

Differentiating Code and Markup

Razor provides two ways to differentiate code from markup: code nuggets and code blocks.

Code nuggets

Code nuggets are simple expressions that are evaluated and rendered inline. They can be mixed with text and look like this:

```
Not Logged In: @Html.ActionLink("Login", "Login")
```

The expression begins immediately after the `@` symbol, and Razor is smart enough to know that the closing parenthesis indicates the end of this particular statement.

The previous example will render this output:

```
Not Logged In: <a href="/Login">Login</a>
```

Notice that code nuggets must always return markup for the view to render. If you write a code nugget that evaluates to a `void` value, you will receive an error when the view executes.

Code blocks

A *code block* is a section of the view that contains strictly code rather than a combination of markup and code. Razor defines code blocks as any section of a Razor template wrapped in `@{ }` characters. The `@{` characters mark the beginning of the block, followed by any number of lines of fully formed code. The `}` character closes the code block.

Keep in mind that the code within a code block is not like code in a code nugget. It is regular code that must follow the rules of the current language. For example, each line of code written in C# must include a semicolon (`;`) at the end, just as if it lived within a class in a *.cs* file.

Here is an example of a typical code block:

```
@{
 LayoutPage = "~/Views/Shared/_Layout.cshtml";
 View.Title = "Auction " + Model.Title;
}
```

Code blocks do not render anything to the view. Instead, they allow you to write arbitrary code that requires no return value.

Also, variables defined within code blocks may be used by code nuggets in the same scope. That is, variables defined within the scope of a `foreach` loop or similar container will be accessible only within that container, while variables that are defined at the top level of a view (not in any kind of container) will be accessible to any other code blocks or code nuggets in that same view.

To better clarify this, take a look at a view with a few variables defined at different scopes:

```
@{
    // The title and bids variables are
    // available to the entire view
    var title = Model.Title;
    var bids = Model.Bids;
}

<h1>@title<h1>
<div class="items">
<!-- Loop through the objects in the bids variable -->
@foreach(var bid in bids) {
    <!-- The bid variable is only available within the foreach loop -->
    <div class="bid">
        <span class="bidder">@bid.Username</span>
        <span class="amount">@bid.Amount</span>
    </div>
}

<!-- This will throw an error: the bid variable does not exist at this scope! -->
<div>Last Bid Amount: @bid.Amount</div>
</div>
```

Code blocks are a means to execute code within a template and do not render anything to the view. In direct contrast to the way that code nuggets must provide a return value for the view to render, the view will completely ignore values that a code block returns.

Layouts

Razor offers the ability to maintain a consistent look and feel throughout your entire website through *layouts*. With layouts, a single view acts as a template for all other views to use, defining the site-wide page layout and style.

A layout template typically includes the primary markup (scripts, CSS stylesheets, and structural HTML elements such as navigation and content containers), specifying locations within the markup in which views can define content. Each view in the site then refers to this layout, including only the content within the locations the layout has indicated.

Take a look at a basic Razor layout file (*_Layout.cshtml*):

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>@View.Title</title>
    </head>
    <body>
        <div class="header">
            @RenderSection("Header")
        </div>

        @RenderBody()

        <div class="footer">
            @RenderSection("Footer")
        </div>
    </body>
</html>
```

The layout file contains the main HTML content, defining the HTML structure for the entire site. The layout relies on variables (such as `@View.Title`) and helper functions like `@RenderSection([Section Name])` and `@RenderBody()` to interact with individual views.

Once a Razor layout is defined, views reference the layout and supply content for the sections defined within the layout.

The following is a basic content page that refers to the previously defined *_Layout.cshtml* file:

```
@{ Layout = "~/_Layout.cshtml"; }

@section Header {
    <h1>EBuy Online Auction Site<h1>
}

@section Footer {
    Copyright @DateTime.Now.Year
}

<div class="main">
    This is the main content.
</div>
```

Razor layouts and the content views that depend on them are assembled together like puzzle pieces, each one defining one or more portions of the entire page. When all the pieces get assembled, the result is a complete web page.

Partial Views

While layouts offer a helpful way to reuse portions of markup and maintain a consistent look and feel throughout multiple pages in your site, some scenarios may require a more focused approach.

The most common scenario is needing to display the same high-level information in multiple locations in a site, but only on a few specific pages and in different places on each of those pages.

For instance, the Ebuy auction site may render a list of compact auction details—showing only the auction's title, current price, and perhaps a thumbnail of the item—in multiple places in the site such as the search results page as well as a list of featured auctions on the site's homepage.

ASP.NET MVC supports these kinds of scenarios through partial views.

Partial views are views that contain targeted markup designed to be rendered as part of a larger view. The following snippet demonstrates a partial view to display the compact auction details mentioned in the scenario above:

```
@model Auction

<div class="auction">
    <a href="@Model.Url">
         <img src="@Model.ImageUrl" />
     </a>
    <h4><a href="@Model.Url">@Model.Title</a></h4>
    <p>Current Price: @Model.CurrentPrice</p>
</div>
```

To render this snippet as a partial view, simply save it as its own standalone view file (e.g. */Views/Shared/Auction.cshtml*) and use one of ASP.NET MVC's HTML Helpers —`Html.Partial()`—to render the view as part of another view.

To see this in action, take a look at the following snippet, which iterates over a collection of auction objects and uses the partial view above to render the HTML for each auction:

```
@model IEnumerable<Auction>

<h2>Search Results</h2>

@foreach(var auction in Model) {
     @Html.Partial("Auction", auction)
}
```

Notice that the first parameter to the `Html.Partial()` helper method is a string containing the name of the view without its extension.

This is because the `Html.Partial()` helper method is just a simple layer on top of ASP.NET MVC's powerful view engine, which renders the view very similar to what occurs after a controller action calls the `View()` method to return a view action result: the engine uses the view name to locate and execute the appropriate view.

In this way, partial views are developed and executed almost exactly like any other kind of view. The only difference is that they are designed to be rendered as part of a larger view.

The second parameter (`auction` in the example above) accepts the partial view's model, just like the model parameter in the `View(_View Name_, _[Model]_)` controller helper method. This second model parameter is optional; when it's not specified, it defaults to the model in the view from which the `Html.Partial()` helper was called. For instance, if the second `auction` parameter were omitted in the example above, ASP.NET MVC would pass the view's `Model` property (of type `IEnumerable<Auction>`) in its place.

The examples above show how partial views can provide reusable sections of markup that can help reduce duplication and complexity in your views.

Though useful, this is only one way to take advantage of partial views—"Partial Rendering" on page 111 shows how to take advantage of partial views to provide a simple and effective way to enhance your site with AJAX.

Displaying Data

The MVC architecture depends on the model, view, and controller all remaining separate and distinct, while still working together to accomplish a common goal. In this relationship, it is the controller's job to be the "traffic cop," coordinating various parts of the system to execute the application's logic. This processing typically results in some kind of data that needs to be relayed to the user. Alas, it is not the controller's job to display things to the user—that is what views are for! The question then becomes, how does the controller communicate this information to the view?

ASP.NET MVC offers two ways to communicate data across model-view-controller boundaries: `ViewData` and `TempData`. These objects are dictionaries available as properties in both controllers and views. Thus, passing data from a controller to a view can be as simple as setting a value in the controller, as in this snippet from *HomeController.cs*:

```
public ActionResult About()
{
    ViewData["Username"] = User.Identity.Username;

    ViewData["CompanyName"] = "EBuy: The ASP.NET MVC Demo Site";
    ViewData["CompanyDescription"] =
        "EBuy is the world leader in ASP.NET MVC demoing!";

    return View("About");
}
```

and referencing the value in the view, as in this portion of the *About.cshtml* file:

```
<h1>@ViewData["CompanyName"]</h1>
<div>@ViewData["CompanyDescription"]</div>
```

Cleaner access to ViewData values via ViewBag

ASP.NET MVC controllers and views that expose the `ViewData` property also expose a similar property named `ViewBag`. The `ViewBag` property is simply a wrapper around the `ViewData` that exposes the `ViewData` dictionary as a `dynamic` object.

For example, any references to values in the `ViewData` dictionary in the preceding snippets can be replaced with references to `dynamic` properties on the `ViewBag` object, as in:

```
public ActionResult About()
{
    ViewBag.Username = User.Identity.Username;

    ViewBag.CompanyName = "EBuy: The ASP.NET MVC Demo Site";
    ViewBag.CompanyDescription = "EBuy is the world leader in ASP.NET MVC demoing!";

    return View("About");
}
```

and:

```
<h1>@ViewBag.CompanyName</h1>
<div>@ViewBag.CompanyDescription</div>
```

View models

In addition to its basic dictionary behavior, the `ViewData` object also offers a `Model` property, which represents the primary object that is the target of the request. Though the `ViewData.Model` property is conceptually no different from `ViewData["Model"]`, it promotes the model to a first-class citizen and recognizes it as more important than the other data that might be in the request.

For example, the previous two snippets showed that the `CompanyName` and `CompanyDescription` dictionary values are clearly related to each other and represent a great opportunity to wrap together in a model.

Take a look at *CompanyInfo.cs*:

```
public class CompanyInfo
{
    public string Name { get; set; }
    public string Description { get; set; }
}
```

the `About` action in *HomeController.cs*:

```
public ActionResult About()
{
    ViewBag.Username = User.Identity.Username;

    var company = new CompanyInfo {
        Name = "EBuy: The ASP.NET MVC Demo Site",
```

```
            Description = "EBuy is the world leader in ASP.NET MVC demoing!",
        };

        return View("About", company);
    }
```

and this snippet from *About.cshtml*:

```
@{ var company = (CompanyInfo)ViewData.Model; }

<h1>@company.Name</h1>
<div>@company.Description</div>
```

In these snippets, the references to the `CompanyName` and `CompanyDescription` dictionary values have been merged into an instance of a new class named `CompanyInfo` (`company`). The updated *HomeController.cs* snippet also shows an overload of the `View()` helper method in action. This overload continues to accept the name of the desired view as the first parameter. The second parameter, however, represents the object that will be assigned to the `ViewData.Model` property.

Now, instead of setting the dictionary values directly, `company` is passed as the `model` parameter to the `View()` helper method and the view (*About.cshtml*) can get a local reference to the `company` object and access its values.

Strongly typed views

By default, the `Model` property available within Razor views is `dynamic`, which means that you are able to access its values without needing to know its exact type.

However, given the static nature of the C# language and Visual Studio's excellent IntelliSense support for Razor views, it is often beneficial to specify the type of the page's model explicitly.

Luckily, Razor makes this pretty easy—simply use the `@model` keyword to indicate the model's type name:

```
@model Auction

<h1>@Model.Name</h1>
<div>@Model.Description</div>
```

This example modifies the previous *Auction.cshtml* example, avoiding the need to add an intermediary variable to cast the `ViewData.Model` into. Instead, the first line uses the `@model` keyword to indicate that the model's type is `CompanyInfo`, making all references to `ViewData.Model` strongly typed and directly accessible.

HTML and URL Helpers

The primary goal of most web requests is to deliver HTML to the user, and as such, ASP.NET MVC goes out of its way to help you create HTML. In addition to the Razor markup language, ASP.NET MVC also offers many helpers to generate HTML simply

and effectively. The two most important of these helpers are the `HtmlHelper` and `UrlHelper` classes, exposed in controllers and views as the `Html` and `Url` properties, respectively.

Here are some examples of the two helpers in action:

```
<img src='@Url.Content("~/Content/images/header.jpg")' />
@Html.ActionLink("Homepage", "Index", "Home")
```

The rendered markup looks like this:

```
<img src='/vdir/Content/images/header.jpg' />
<a href="/vdir/Home/Index">Homepage</a>
```

For the most part, the `HtmlHelper` and `UrlHelper` types don't have many methods of their own and are merely shims that the framework attaches behaviors to via extension methods. This makes them an important extensibility point, and you'll see references to the two types throughout this book.

Though there are far too many methods to list in this section, the one thing to take away at this point is: the `HtmlHelper` class helps you generate HTML markup and the `UrlHelper` class helps you generate URLs. Keep this in mind, and turn to these helpers anytime you need to generate URLs or HTML.

Models

Now that we've covered controllers and views, it's time to complete the definition of MVC by discussing *models*, which are usually considered the most important part of the MVC architecture. If they are so important, why are they the last to be explained? Well, the model layer is notoriously difficult to explain because it is the layer that contains all of the business logic for the application—and that logic is different for every application.

From a more technical standpoint, the model typically consists of normal classes that expose data in the form of properties and logic in the form of methods. These classes come in all shapes and sizes, but the most common example is the "data model" or "domain model," whose primary job is to manage data.

For example, take a look at the following snippet, which shows the `Auction` class—the model that will drive the entire EBuy reference application:

```
public class Auction
{
    public long Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public decimal StartPrice { get; set; }
    public decimal CurrentPrice { get; set; }
    public DateTime StartTime { get; set; }
    public DateTime EndTime { get; set; }
}
```

Though we will add various functionality such as validation and behavior to the `Auction` class throughout this book, this snippet is still very representative of a model in that it defines the data that makes up an "auction."

And, just as we will build on the `Auction` class throughout the book, be on the lookout for more kinds of classes (such as services and helpers) that all work together to make up the "Model" in "MVC."

Putting It All Together

So far we've described all the parts that make up an ASP.NET MVC application, but the discussion has focused on the code that Visual Studio generates for us as part of the project template. In other words, we haven't actually *made* anything yet. So let's change that!

This section will focus on how to implement a feature from scratch, creating everything you need to accomplish an example scenario: displaying an auction. As a recap, every ASP.NET MVC request requires at least three things: a route, a controller action, and a view (and, optionally, a model).

The Route

To figure out the routing pattern that you'd like to use for a given feature, you must first determine what you'd like your URL for that feature to look like. In this example we are going to choose a relatively standard URL of *Auctions/Details/[Auction ID]*; for example, *http://www.ebuy.biz/Auctions/Details/1234*.

What a nice surprise—the default route configuration already supports this URL!

The Controller

Next, we'll need to create a controller to host the actions that will process the request.

Since controllers are merely classes that implement the ASP.NET MVC controller interface, you *could* manually add a new class to the *Controllers* folder that derives from `System.Web.Mvc.Controller` and begin adding controller actions to that class. However, Visual Studio offers a bit of tooling to take most of the work out of creating new controllers: simply right-click on the *Controllers* folder and choose the `Add > Controller...` menu option, which will pop up the Add Controller dialog shown in Figure 1-8.

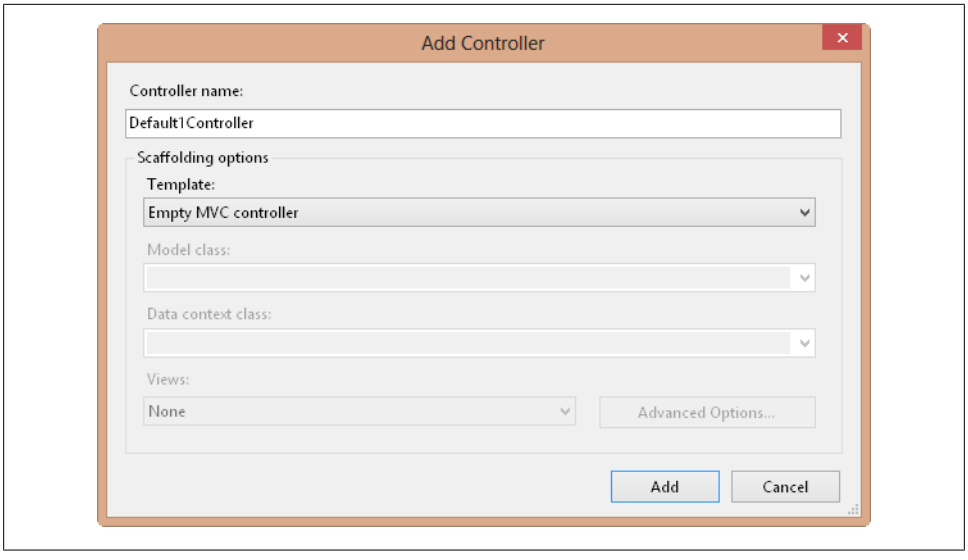

Figure 1-8. Adding a controller to an ASP.NET MVC application

The Add Controller dialog begins by asking for the name of the new controller class (in this case, we'll call it `AuctionsController`) and follows up by asking which scaffolding template you'd like to use, along with providing a set of options to give you a little bit of control over how ASP.NET MVC is going to generate the new controller class.

Controller templates

The Add Controller dialog offers several different controller templates that can help you get started developing your controllers more quickly:

Empty MVC controller
: The default template ("Empty MVC controller") is the simplest one. It doesn't offer any customization options because, well, it's too simple to have any options! It merely creates a new controller with the given name that has a single generated action named `Index`.

MVC controller with read/write actions and views, using Entity Framework
: The "MVC controller with read/write actions and views, using Entity Framework" template is just as impressive as it sounds. This template starts with the same output as the "MVC controller with empty read/write actions" template (see below) and then kicks it up a notch, generating code to help you access objects stored in an Entity Framework context and even generating *Create*, *Edit*, *Details*, and *Delete* views for those objects! This template is a great kick-start when your project uses Entity Framework to access your data, and in some cases the code it generates may be all that you need to support the Read, Edit, Update, and Delete operations for that data.

MVC controller with empty read/write actions
: The next option—"MVC controller with empty read/write actions"—generates the same code as the "Empty MVC controller" template, but adds a few more actions that you'll most likely need in order to expose standard "data access" operations: `Details`, `Create`, `Edit`, and `Delete`.

API controller templates
: The final three templates—"Empty API controller," "API controller with empty read/write actions," and "API controller with read/write actions and views, using Entity Framework"—are the Web API counterparts to the MVC controller templates of the same names. We will cover these templates in more detail when we discuss ASP.NET MVC's Web API functionality in Chapter 6.

An interesting thing to notice about the controller code that Visual Studio generates is that the `Index` and `Details` actions each have only one method, while the `Create`, `Edit`, and `Delete` actions each have two overloads—one decorated with an `HttpPost Attribute`, and one without.

This is because `Create`, `Edit`, and `Delete` all involve two requests in order to complete: the first request returns the view that the user can interact with to create the second request, which actually performs the desired action (creating, editing, or deleting data). This is a very common interaction on the Web, and you'll see several examples of it throughout this book.

Unfortunately, we have not yet reached the point in the book where we are able to use Entity Framework, so for now you can choose the "MVC controller with empty read/write actions" option and click Add to have Visual Studio generate the next controller class.

After Visual Studio is done creating the `AuctionsController`, find the `Details` action and update it so it creates a new instance of the `Auction` model shown earlier and passes that instance to the view using the `View(object model)` method.

Yes, this is a silly example. Normally, you'd retrieve this information from somewhere such as a database—and we will show you how to do just that in Chapter 4—but for this example, we are using hardcoded values:

```
public ActionResult Details(long id = 0)
{
    var auction = new Ebuy.Website.Models.Auction {
            Id = id,
            Title = "Brand new Widget 2.0",
            Description = "This is a brand new version 2.0 Widget!",
            StartPrice = 1.00m,
            CurrentPrice = 13.40m,
            StartTime = DateTime.Parse("6-15-2012 12:34 PM"),
            EndTime = DateTime.Parse("6-23-2012 12:34 PM"),
        };
```

```
        return View(auction);
    }
```

The View

With a `Details` controller action in place and providing data to a view, it's time to create that view.

As with the controller class in the previous section, you are free to manually add new views (and folders to store them in) directly to the *Views* folder; however, if you're the type who prefers a bit more automation, Visual Studio offers yet another wizard to do the work of creating the views—and the folders they live in—for you.

To add a view using the Visual Studio wizard, simply right-click anywhere within the code of the action in a controller and choose the Add View option, which will display the Add View wizard (Figure 1-9). This is actually quite similar to the Add Controller dialog you just used to generate the `AuctionsController`.

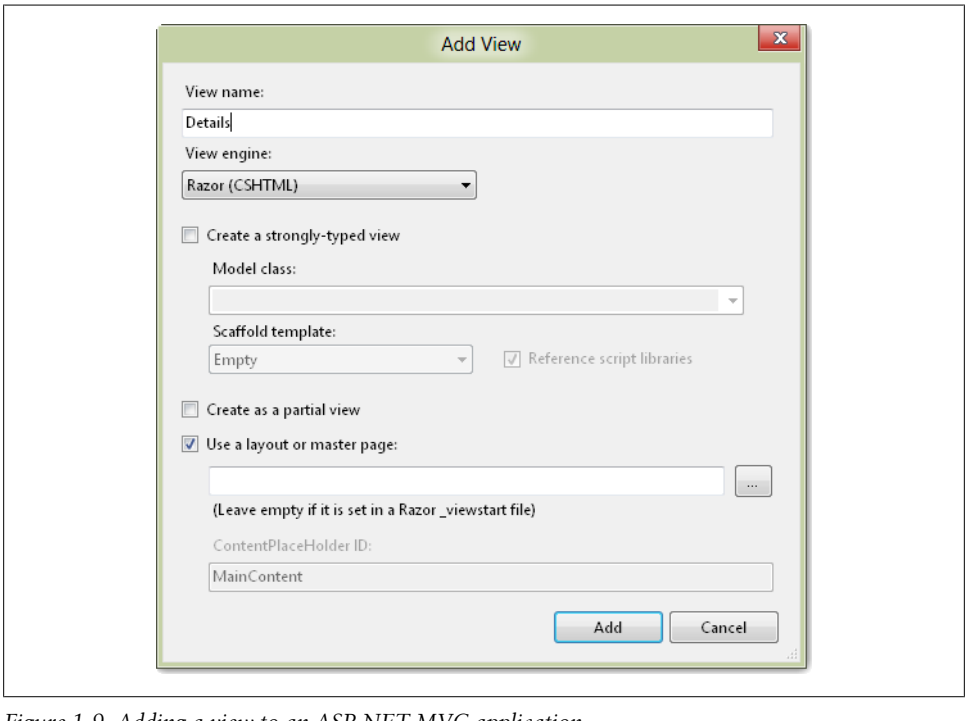

Figure 1-9. Adding a view to an ASP.NET MVC application

The Add View dialog starts off by asking what you'd like to call the new view, defaulting to the name of the controller action from which you triggered the dialog (e.g., *Details* when called from the `Details` action). Then, the dialog allows you to choose the syntax

(aka "View Engine") that you'd like to use when authoring the view. This value defaults to the syntax you chose when you created the web project, but (as promised earlier) you are free to switch between syntaxes if it suits you, perhaps using Razor for some views and the "ASPX" Web Forms syntax for others.

As in the Add Controller dialog, the rest of the options in the Add View wizard have to do with the code and markup that Visual Studio is going to generate when it creates the new view. For example, you can choose to have a strongly typed view model (as discussed in "Strongly typed views" on page 33) by selecting the model type from the list of classes in your project, or typing the type name in yourself. If you choose a strongly typed view, the wizard also lets you choose a template (e.g., *Edit*, *Create*, *Delete*), which analyzes the model type and generates the appropriate form fields for that type.

This is a great way to get up and running quickly and can save you quite a bit of typing, so let's take advantage of it by checking the "Create a strongly typed view" checkbox, choosing our Auction model from the "Model class" drop-down list, and selecting the Details scaffold template.

Visual Studio will only include in the "Model class" drop-down classes that it has been able to compile successfully, so if you do not see the `Auction` class you created earlier, try to compile the solution and then open the Add View dialog again.

Finally, you'll need to tell Visual Studio whether this view is a partial view or should refer to a layout. When you're using the ASPX Web Forms syntax to author your pages and you choose the "Create as a partial view" option, Visual Studio will create it as a User Control (*.ascx*) rather than a full page (*.aspx*). When using the Razor syntax, however, the "Create as a partial view" option has very little effect—Visual Studio creates the same type of file (*.cshtml* or *.vbhtml*) for both partial views and full pages. In the case of Razor syntax, the only effect this checkbox has is on the markup that gets generated inside of the new view.

For the purposes of this demo, you can leave the defaults alone: "Create as a partial view" should remain unchecked, while "Use a layout or master page" should be checked, with the layout text box left empty (see Figure 1-10).

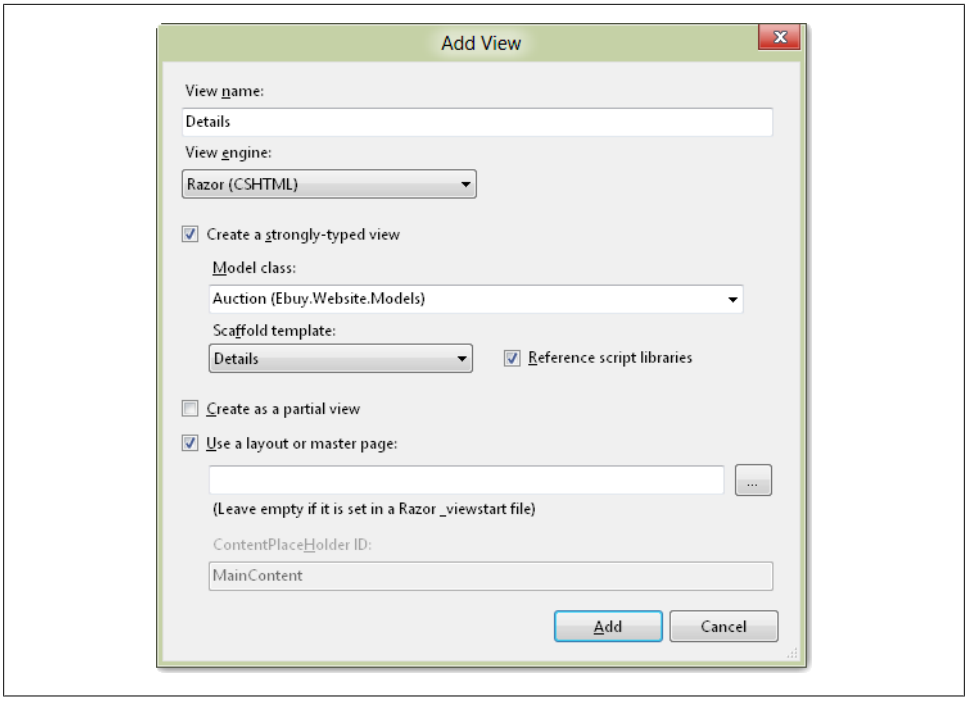

Figure 1-10. Customizing your view

When you're ready, click the Add button to have Visual Studio add the new view to your project. After it's finished, you will see that Visual Studio has analyzed the `Auction` model and generated the required HTML markup—complete with references to HTML helpers such as `Html.DisplayFor`—to display all of the `Auction` fields.

At this point, you should be able to run your site, navigate to the controller action (e.g., */auctions/details/1234*), and see the details of the `Auction` object rendered in your browser, as shown in Figure 1-11.

It sure isn't pretty, but remember, the HTML that Visual Studio generates is just a starting point to help you save time. Once it's generated, you can feel free to change it however you like.

Congratulations—you have just created your first controller action and view from scratch!

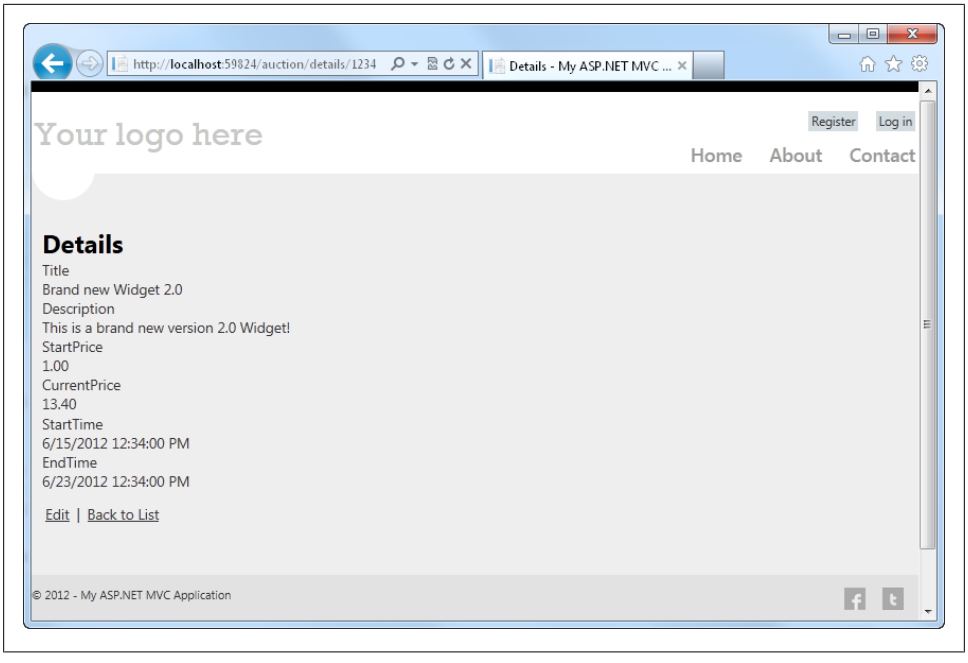

Figure 1-11. The new view rendered in a browser

Authentication

So far we've covered just about everything you need to know in order to create an ASP.NET MVC application, but there is one more very important concept that you should know about before you continue with the rest of the book: how to protect your site by requiring users to authenticate themselves before they can access certain controller actions.

You may have noticed that the Internet Application template generates an `AccountController`—along with some views to support it—which provides a full implementation of forms authentication right out of the box. This is an acknowledgment that security and authentication are crucial to just about every web application and that, at some point, you'll probably want to lock down some or all of your site to restrict access to specific users (or groups of users) and prevent unauthorized access by all other visitors. So, since you'll most likely need it anyway, why shouldn't Visual Studio generate the controller and views to get you started?

The traditional tactic used to lock down ASP.NET applications is to apply authorization settings to specific pages or directories via *web.config* configuration settings. Unfortunately, this approach does not work in ASP.NET MVC applications because ASP.NET MVC applications rely on routing to controller actions, not to physical pages.

Instead, the ASP.NET MVC Framework provides the `AuthorizeAttribute`, which can be applied to individual controller actions—or even entire controllers—to restrict access only to authenticated users or, alternatively, to specific users and user roles.

Take a look at the `Profile` action on the `UsersController` that we'll create later in the book, which displays the profile information for the current user:

```
public class UsersController
{
    public ActionResult Profile()
    {
        var user = _repository.GetUserByUsername(User.Identity.Name);
        return View("Profile", user);
    }
}
```

Clearly, this action will fail if the user is not logged in. Applying the `AuthorizeAttribute` to this controller action causes any requests made to this action by users who are not authenticated to be rejected:

```
public class UsersController
{
    [Authorize]
    public ActionResult Profile()
    {
        var user = _repository.GetUserByUsername(User.Identity.Name);
        return View("Profile", user);
    }
}
```

If you'd like to be even more specific about the users who can access the controller action, the `AuthorizeAttribute` exposes the `Users` property, which accepts a comma-delimited whitelist of acceptable usernames, as well as the `Roles` property, which accepts a list of allowed roles.

Now, when nonauthenticated users attempt to access this URL, they will instead be redirected to the login URL: the `Login` action on the `AccountController`.

The AccountController

In order to help you get a jump-start on your application, the ASP.NET MVC Internet Application project template includes the `AccountController`, which contains controller actions that leverage the ASP.NET Membership Providers.

The `AccountController` provides quite a bit of functionality out of the box, along with the views to support each controller action. This means that your brand new ASP.NET

MVC application already contains the following fully implemented features, without any coding on your part:

- Login
- Logoff
- New user registration
- Change password

Thus, when you apply the `AuthorizeAttribute` to any of your controller actions, users are redirected to the existing login page that the project template creates for you (see Figure 1-12).

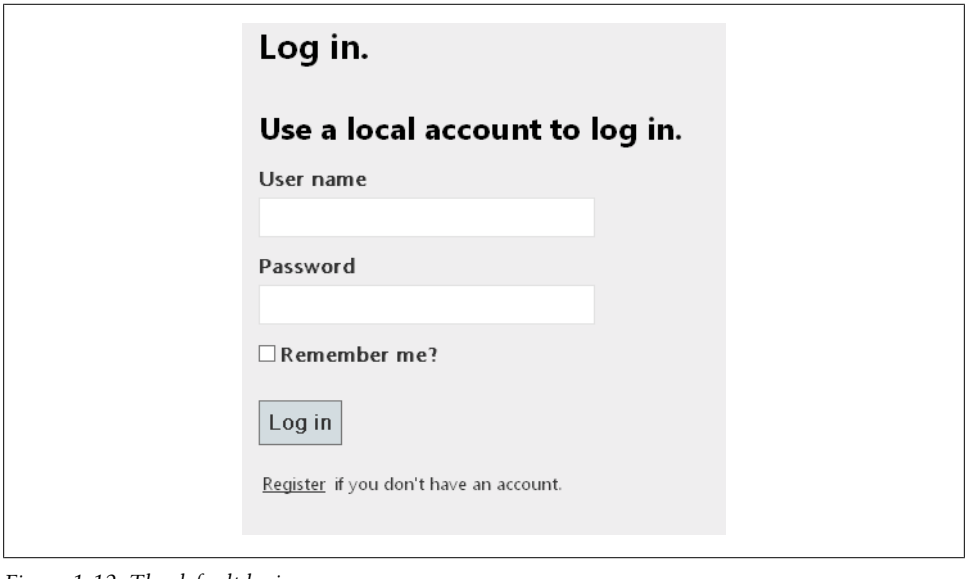

Figure 1-12. The default login page

And, when users need to create a new account in order to log in, they can click the Register link to view the prebuilt Registration page (Figure 1-13).

Plus, if you don't like the out-of-the-box views, they are easily customizable to meet your needs.

As this section shows, not only does the ASP.NET MVC Framework make it very easy to protect controller actions, but the default project template implements just about everything users will need to authenticate themselves on your site!

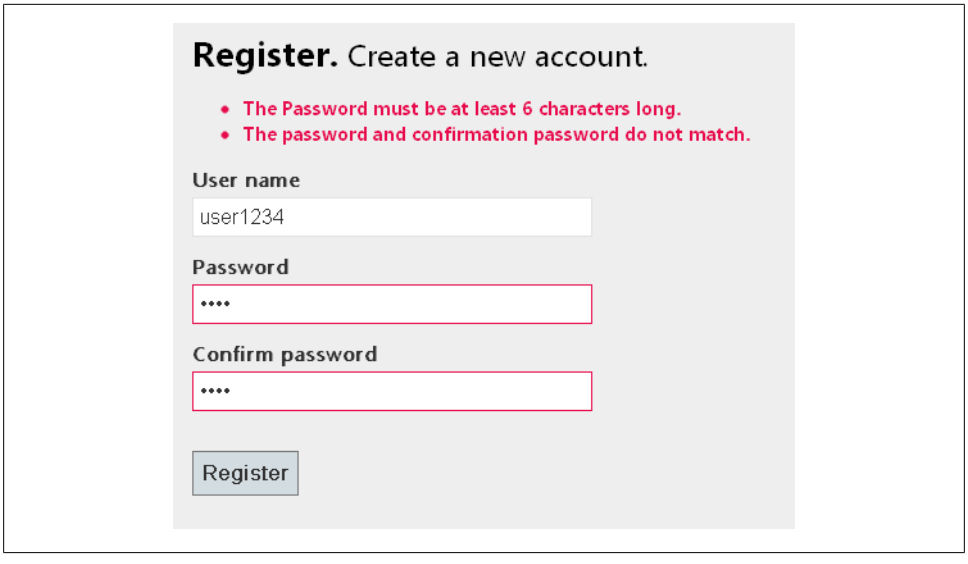

Figure 1-13. The default registration page

Summary

ASP.NET MVC leverages the time-tested Model-View-Controller architecture pattern to provide a website development framework that encourages loosely coupled architecture and many other popular object-oriented programming patterns and practices.

The ASP.NET MVC Framework gets you on the ground running right from the start with helpful project templates and a "convention over configuration" approach that cuts down on the amount of configuration required to create and maintain your application, freeing up more of your time so you can be more productive in getting your application completed and out the door.

This chapter introduced you to the fundamental concepts and basic skills that you need in order to get up and running building ASP.NET MVC 4 applications. The rest of this book will expand on this foundation, showing more features that the ASP.NET MVC Framework has to offer to help you build robust, maintainable web applications using the MVC architectural pattern.

So, what are you waiting for? Keep reading and learn everything you need to know to build the greatest web applications you've ever written!