Stephen Walther

ASP.NET MVC Framework

UNLEASHED

SAMS

ASP.NET MVC Framework Unleashed

ISBN-13: 978-0-672-32998-2

ISBN-10: 0-672-32998-0

Library of Congress Cataloging-in-Publication data

Walther, Stephen.

ASP.NET MVP framework unleashed / Stephen Walther.

p. cm.

ISBN 978-0-672-32998-2

1. Active server pages. 2. Microsoft .NET Framework. 3. Web site development. I. Title.

TK5105.8885.A26W3522 2010

006.7'882–dc22

2009021084

Printed in the United States of America

First Printing July 2009

Trademarks

Warning and Disclaimer

Bulk Sales

Editor-in-Chief
Karen Gettman

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Betsy Harris

Copy Editor
Apostrophe Editing Services

Indexer
Erika Millen

Proofreader
Keith Cline

Technical Editor
Rebecca Riordan

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Compositor
Jake McFarland

Introduction

ASP.NET MVC is Microsoft's newest technology for building web applications. Although ASP.NET MVC is new, there are already several large and successful websites that are built on the ASP.NET MVC framework including StackOverflow.com and parts of CodePlex.com.

ASP.NET MVC was created to appeal to several different audiences. If you are the type of developer who wants total control over every HTML tag and pixel that appears in a web page, the ASP.NET MVC framework will appeal to you.

ASP.NET MVC also enables you to expose intuitive URLs to the world. Exposing intuitive URLs is important for getting your website indexed by search engines. If you care about Search Engine Optimization, you will be happy with ASP.NET MVC.

The ASP.NET MVC framework enables you to build web applications that are easier to maintain and extend over time. The Model View Controller pattern encourages a clear separation of concerns. The framework encourages good software design patterns.

Finally, the ASP.NET MVC framework was designed from the ground up to support testability. In particular, the ASP.NET MVC framework enables you to practice test-driven development. You are not required to practice test-driven development when building an ASP.NET MVC application, but the ASP.NET MVC framework makes test-driven development possible.

How This Book Is Organized

The book is divided into two parts. The first part of the book describes the ASP.NET MVC framework feature-by-feature. For example, there are chapters devoted to the subject of controllers, caching, and validation.

The second part of this book contains a walkthrough of building a full ASP.NET MVC application: We build a simple blog application. We implement features such as data access and validation.

Because one of the primary benefits of the ASP.NET MVC framework is that it enables test-driven development, we build the blog application by using test-driven development. The blog application illustrates how you can overcome many challenges that you face when writing real-world applications with the ASP.NET MVC framework.

You can approach this book in two ways. Some readers might want to read through the first chapters of this book before reading the chapters on building the blog application. Other readers might want to read the walkthrough of building the blog application before reading anything else.

What You Should Know Before Reading This Book

I make few assumptions about your technical background. I assume that you know either the C# or the Visual Basic .NET programming language—all the code samples are included in both languages in the body of the book. I also assume that you know basic HTML.

ASP.NET MVC uses many advanced features of the C# and Visual Basic .NET language. The first appendix of this book, Appendix A, "C# and VB.NET Language Features," contains an overview of these new features. For example, if you are not familiar with anonymous types or LINQ to SQL, you should take a look at Appendix A.

The other two appendixes, Appendix B, "Using a Unit Testing Framework," and Appendix C, "Using a Mock Object Framework," are devoted to explaining how to use the main tools of test-driven development. In Appendix B, you learn how to use both the Visual Studio Unit Test framework and how to use the NUnit Unit Test framework. Appendix C is devoted to Mock Object Frameworks.

Throughout the book, when a line of code is too long for the printed page, a code-continuation arrow (➥) has been used to mark the continuation. For example:

```
ReallyLongClassName.ReallyLongMethodName("Here is a value",
➥"Here is another value")
```

What Software Do You Need?

You can download all the software that you need to build ASP.NET MVC applications by visiting the www.ASP.net/mvc website. You need to install three software components:

1. **Microsoft .NET Framework 3.5 Service Pack 1—**The Microsoft .NET framework includes the Microsoft ASP.NET framework.
2. **Microsoft ASP.NET MVC 1.0—**The actual ASP.NET MVC framework that runs on top of the ASP.NET framework.
3. **Microsoft Visual Web Developer 2008 Service Pack 1 or Microsoft Visual Studio 2008 Service Pack 1—**The development environment for creating ASP.NET applications. Also includes the option of installing Microsoft SQL Server Express.

The Microsoft .NET framework, Microsoft ASP.NET MVC, and Microsoft Visual Web Developer are all free. You can build ASP.NET MVC applications without paying a single cent.

Instead of downloading and installing each of these software components one-by-one, you can take advantage of the Microsoft Web Platform Installer to manage the download and installation of all these components. You can launch the Microsoft Web Platform Installer from the www.ASP.net/mvc site.

Where Do You Download the Code Samples?

The code samples for the book are located on the book's product page, www.informit.com/title/9780672329982.

If You Like This Book

After you read this book, if you discover that this book helped you to understand and build ASP.NET MVC applications, please post a review of this book at the www.Amazon.com website.

To get the latest information on ASP.NET MVC, I encourage you to visit the official Microsoft ASP.NET MVC website at www.ASP.net/mvc. I also encourage you to subscribe to my blog at StephenWalther.com that contains ASP.NET MVC tips and tutorials. I also use my blog to post any errata that is discovered after the book is published.

CHAPTER 1

An Introduction to ASP.NET MVC

IN THIS CHAPTER

- A Story with a Moral
- What Is Good Software?
- What Is ASP.NET MVC?
- The Architecture of an ASP.NET MVC Application
- Understanding the Sample ASP.NET MVC Application

"There is nothing permanent except change."
Heraclitus

This chapter provides you with an overview and introduction to the Microsoft ASP.NET MVC framework. The goal of this chapter is to explain why you should build web applications using ASP.NET MVC.

Because the ASP.NET MVC framework was designed to enable you to write good software applications, the first part of this chapter is devoted to a discussion of the nature of good software. You learn about the software design principles and patterns that enable you to build software that is resilient to change.

Finally, we discuss the architecture of an ASP.NET MVC application and how this architecture enables you to write good software applications. We provide you with an overview of the different parts of an MVC application including models, views, and controllers and also introduce you to the sample application that you get when you create a new ASP.NET MVC project.

A Story with a Moral

I still remember the day that my manager came to my office and asked me to build the *Single Button Application*. He explained that he needed a simple call manager application to help interviewers dial phone numbers while conducting a health survey. The call manager application would load a list of phone numbers and dial each number one-by-one when you hit a button. What could be simpler?

I said, with great earnestness and confidence, that I would have the call manager application done that same afternoon. I closed my office door, put on my cowboy hat, turned up the music, and pounded out some code. By the end of the day, I had completed the application. My manager was happy, and I went home that night with the happy thought that I had done a good day of work.

The next morning, my manager appeared again at my office door. Worried, I asked if there was a problem with the call manager application. He reassured me that the application worked fine. In fact, he liked it so much that he wanted me to add another feature. He wanted the call manager application to display a survey form when a number is dialed. That way, survey answers could be stored in the database.

With heroic determination, I once again spent the day knocking out code. By the end of the day, I had finished updating the call manager and I proudly presented the finished application to my manager.

I won't continue this story, because anyone who builds software for a living knows how this story ends. The story never ends. When a software project is brought to life, it is almost impossible to kill it. A software application needs to be continuously fed with new features, bug fixes, and performance enhancements.

Being asked to change software that you have created is a compliment. Only useless software goes stagnant. When people care about software, when software is actively used, it undergoes constant change.

I no longer work at the company where I created the call manager application. (I am currently sitting in an office at Microsoft.) But I still have friends at the company and every once in a while I get a report on how the application has changed. Needless to say, it has turned into a massively complex application that supports different time zones, complicated calling rules, and advanced reporting with charts. It can no longer be described as the Single Button Application.

What Is Good Software?

I dropped out of graduate school at MIT to launch an Internet startup in the earliest days of the Web. At that time, building a website was difficult. This was before technologies such as Active Server Pages or ASP.NET existed. (We had only stone knives.) Saving the contents of an HTML form to a database table was a major accomplishment. Blinking text was the height of cool.

When I first started writing software, simply getting the software to do what I wanted was the goal. Adding as many features to a website in the shortest amount of time was the key to survival in the ferociously competitive startup world of the '90s. I used to sleep in my office under my desk.

During my startup phase, I would define good software like this:

Good software is software that works as you intended.

If I was feeling particularly ambitious, I would worry about performance. And maybe, just maybe, if I had extra time, I would add a comment or two to my code. But really, at the end of the day, my criterion for success was simply that the software worked.

For the past 8 years, I've provided training and consulting to large companies and organizations such as Boeing, NASA, Lockheed Martin, and the National Science Foundation. Large organizations are not startups. In a large organization, the focus is not on building software applications as fast as possible; the focus is on building software applications that can be easily maintained over time.

Over the years, my definition of good software has shifted substantially. As I have been faced with the scary prospect of maintaining my own monsters, I've changed my definition of good software to this:

Good software is software that works as you intended and that is easy to change.

There are many reasons that software changes over time. Michael Feathers, in his excellent book *Working Effectively with Legacy Code*, offers the following reasons:

1. You might need to add a new feature to existing software.
2. You might need to fix a bug in existing software.
3. You might need to optimize existing software.
4. You might need to improve the design of existing software.

For example, you might need to add a new feature to an application. The call manager application started as a Single Button Application. However, each day, more and more features were added to the application.

You also need to change software when you discover a bug in the software. For instance, in the case of the call manager, we discovered that it did not calculate daylight savings time correctly. (It was waking some people up in the morning!) We rushed to change the broken code.

You also might need to modify a software application to make the application run faster. At one point, the call manager application took as long as 12 seconds to dial a new phone number. The business rules were getting complex. We had to rewrite the code to get the phone number retrieval time down to the millisecond range.

Finally, you might need to modify software to improve its design. In other words, you might need to take badly written code and convert it into good code. You might need to make your code more resilient to change.

Avoiding Code Smells

Unless you are careful, a software application quickly becomes difficult to change. We all have had the experience of inheriting an application that someone else has written and being asked to modify it. Think of the fear that strikes your heart just before you make your first change.

In the game of Pick-Up Sticks, you must remove stick after stick from a pile of sticks without disturbing the other sticks. The slightest mistake and the whole pile of sticks might scatter.

Modifying an existing software application is similar to the game of Pick-Up Sticks. You bump the wrong piece of code and you introduce a bug.

Bad software is software that is difficult to change. Robert and Micah Martin describe the markers of bad software as *code smells*. The following code smells indicate that software is badly written:

- **Rigidity**—Rigid software is software that requires a cascade of changes when you make a change in one place.
- **Fragility**—Fragile software is software that breaks in multiple places when you make a change.
- **Needless complexity**—Needlessly complex software is software that is overdesigned to handle any possible change.
- **Needless repetition**—Needlessly repetitious software contains duplicate code.
- **Opacity**—Opaque software is difficult to understand.

NOTE

These code smells are described by Micah and Robert Martin in their book *Agile Principles, Patterns, and Practices in C#* on page 104. This book is strongly recommended!

Notice that these code smells are all related to change. Each of these code smells is a barrier to change.

Software Design Principles

Software does not need to be badly written. A software application can be designed from the beginning to survive change.

The best strategy for making software easy to change is to make the components of the application *loosely coupled*. In a loosely coupled application, you can make a change to one component of an application without making changes to other parts.

Over the years, several principles have emerged for writing good software. These principles enable you to reduce the dependencies between different parts of an application. These software principles have been collected together in the work of Robert Martin (AKA Uncle Bob).

Robert Martin did not invent all the principles; however, he was the first one to gather the principles into a single list. Here is his list of software design principles:

- **SRP**—Single Responsibility Principle
- **OCP**—Open Closed Principle

- **LSP**—Liskov Substitution Principle
- **ISP**—Interface Segregation Principle
- **DIP**—Dependency Inversion Principle

This collection of principles is collectively known by the acronym SOLID. (Yes, SOLID is an acronym of acronyms.)

For example, according to the Single Responsibility Principle, a class should have one, and only one, reason to change. Here's a concrete example of how this principle is applied: If you know that you might need to modify your application's validation logic separately from its data access logic, then you should not mix validation and data access logic in the same class.

NOTE

There are other lists of software design principles. For example, the *Head First Design Patterns* book has a nice list. You should also visit the C2.com website.

Software Design Patterns

Software design patterns represent strategies for applying software design principles. In other words, a software design principle is a good idea and a software design pattern is the tool that you use to implement the good idea. (It's the hammer.)

The idea behind software design patterns was originally promoted by the book *Design Patterns: Elements of Reusable Object-Oriented Software*. (This book is known as the Gang of Four book.) This book has inspired many other books that describe software design patterns.

The *Head First Design Pattern* book provides a more user-friendly introduction to the design patterns from the Gang of Four book. The *Head First Design* book devotes chapters to 14 patterns with names like Observer, Façade, Singleton, and Adaptor.

Another influential book on software design patterns is Martin Fowler's book *Patterns of Enterprise Application Architecture*. This book has a companion website that lists the patterns from the book: www.martinfowler.com/eaaCatalog.

Software design patterns provide you with patterns for making your code more resilient to change. For example, in many places in this book, we take advantage of a software design pattern named the Repository pattern. Eric Evans, in his book *Domain-Driven Design*, describes the Repository pattern like this:

"A REPOSITORY represents all objects of a certain type as a conceptual set (usually emulated). It acts like a collection, except with more elaborate querying capability. Objects of the appropriate type are added and removed, and the machinery behind the REPOSITORY inserts them or deletes them from the database" (see page 151).

According to Evans, one of the major benefits of the Repository pattern is that it enables you to "decouple application and domain design from persistence technology, multiple

database strategies, or even multiple data sources." In other words, the Repository pattern enables you to shield your application from changes in how you perform database access.

For example, when we write our blog application at the end of this book, we take advantage of the Repository pattern to isolate our blog application from a particular persistence technology. The blog application will be designed in such a way that we could switch between different data access technologies such as LINQ to SQL, the Entity Framework, or even NHibernate.

Writing Unit Tests for Your Code

By taking advantage of software design principles and patterns, you can build software that is more resilient to change. Software design patterns are architectural patterns. They focus on the gross architecture of your application.

If you want to make your applications more change proof on a more granular level, then you can build unit tests for your application. A unit test enables you to verify whether a particular method in your application works as you intend it to work.

There are many benefits that result from writing unit tests for your code:

1. Building tests for your code provides you with a safety net for change.
2. Building tests for your code forces you to write loosely coupled code.
3. Building tests for your code forces you to take a user perspective on the code.

First, unit tests provide you with a safety net for change. This is a point that Michael Feathers emphasizes again and again in his book *Working Effectively with Legacy Code*. In fact, he defines legacy code as "simply code without tests" (see xvi).

When your application code is covered by unit tests, you can modify the code without the fear that the modifications will break the functionality of your code. Unit tests make your code safe to refactor. If you can refactor, then you can modify your code using software design patterns and thus produce better code that is more resilient to change.

NOTE

Refactoring is the process of modifying code without changing the functionality of the code.

Second, writing unit tests for your code forces you to write code in a particular way. Testable code tends to be loosely coupled code. A unit test performs a test on a unit of code in isolation. To build your application so that it is testable, you need to build the application in such a way that it has isolatable components.

One class is loosely coupled to a second class when you can change the first class without changing the second class. Test-driven development often forces you to write loosely coupled code. Loosely coupled code is resistant to change.

Finally, writing unit tests forces you to take a user's perspective on the code. When writing a unit test, you take on the same perspective as a developer who will use your code in the

future. Because writing tests forces you to think about how a developer (perhaps, your future self) will use your code, the code tends to be better designed.

Test-Driven Development

In the previous section, we discussed the importance of building unit tests for your code. Test-driven development is a software design methodology that makes unit tests central to the process of writing software applications. When you practice test-driven development, you write tests first and then write code against the tests.

More precisely, when practicing test-driven development, you complete three steps when creating code (Red/Green/Refactor):

1. Write a unit test that fails (Red).
2. Write code that passes the unit test (Green).
3. Refactor your code (Refactor).

First, you write the unit test. The unit test should express your intention for how you expect your code to behave. When you first create the unit test, the unit test should fail. The test should fail because you have not yet written any application code that satisfies the test.

Next, you write just enough code for the unit test to pass. The goal is to write the code in the laziest, sloppiest, and fastest possible way. You should not waste time thinking about the architecture of your application. Instead, you should focus on writing the minimal amount of code necessary to satisfy the intention expressed by the unit test.

Finally, after you write enough code, you can step back and consider the overall architecture of your application. In this step, you rewrite (refactor) your code by taking advantage of software design patterns—such as the Repository pattern—so that your code is more maintainable. You can fearlessly rewrite your code in this step because your code is covered by unit tests.

There are many benefits that result from practicing test-driven development. First, test-driven development forces you to focus on code that actually needs to be written. Because you are constantly focused on just writing enough code to pass a particular test, you are prevented from wandering into the weeds and writing massive amounts of code that you will never use.

Second, a "test first" design methodology forces you to write code from the perspective of how your code will be used. In other words, when practicing test-driven development, you constant write your tests from a user perspective. Therefore, test-driven development can result in cleaner and more understandable APIs.

Finally, test-driven development forces you to write unit tests as part of the normal process of writing an application. As a project deadline approaches, testing is typically the first thing that goes out the window. When practicing test-driven development, on the other hand, you are more likely to be virtuous about writing unit tests because test-driven development makes unit tests central to the process of building an application.

Short-Term Pain, Long-Term Gain

Building software designed for change requires more upfront effort. Implementing software design principles and patterns takes thought and effort. Writing tests takes time. However, the idea is that the initial effort required to build software the right way will pay huge dividends in the future.

There are two ways to be a developer. You can be a cowboy or you can be a craftsman. A cowboy jumps right in and starts coding. A cowboy can build a software application quickly. The problem with being a cowboy is that software must be maintained over time.

A craftsman is patient. A craftsman builds software carefully by hand. A craftsman is careful to build unit tests that cover all the code in an application. It takes longer for a craftsman to create an application. However, after the application is created, it is easier to fix bugs in the application and add new features to the application.

Most software developers start their programming careers as cowboys. At some point, however, you must hang up your saddle and start building software that can stand the test of time.

What Is ASP.NET MVC?

The Microsoft ASP.NET MVC framework is Microsoft's newest framework for building web applications. The ASP.NET MVC framework was designed from the ground up to make it easier to build good software in the sense of good software discussed in this chapter.

The ASP.NET MVC framework was created to support *pattern-based* software development. In other words, the framework was designed to make it easier to implement software design principles and patterns when building web applications.

Furthermore, the ASP.NET MVC framework was designed to its core to support unit tests. Web applications written with the ASP.NET MVC framework are highly testable.

Because ASP.NET MVC applications are highly testable, this makes the ASP.NET MVC framework a great framework to use when practicing test-driven development.

ASP.NET MVC Is Part of the ASP.NET Framework

Microsoft's framework for building software applications—any type of application including desktop, web, and console applications—is called the *.NET framework*. The .NET framework consists of a vast set of classes, tens of thousands of classes, which you can use when building any type of software application. For example, the .NET framework includes classes for working with the file system, accessing a database, using regular expressions, and generating images.

The ASP.NET framework is one part of the .NET framework. The ASP.NET framework is Microsoft's framework for building web applications. It contains a set of classes that were created specifically to support building web applications. For example, the ASP.NET framework includes classes for implementing web page caching, authentication, and authorization.

Microsoft has two frameworks for building web applications built on top of the ASP.NET framework: ASP.NET Web Forms and ASP.NET MVC (see Figure 1.1).

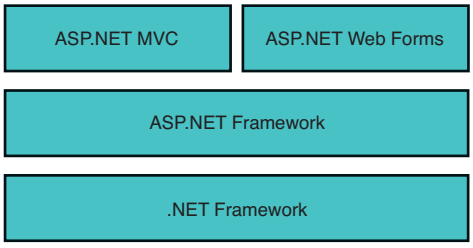

FIGURE 1.1 The ASP.NET frameworks

ASP.NET MVC is an alternative to, but not a replacement for, ASP.NET Web Forms. Some developers find the style of programming represented by ASP.NET Web Forms more compelling, and some developers find ASP.NET MVC more compelling. Microsoft continues to make heavy investments in both technologies.

NOTE

This book is devoted to the topic of ASP.NET MVC. If you want to learn about ASP.NET Web Forms, buy my book *ASP.NET Unleashed*.

The Origins of MVC

The ASP.NET MVC framework is new; however, the MVC software design pattern itself has a long history. The MVC pattern was invented by Trygve Reenskaug while he was a visiting scientist at the Smalltalk group at the famed Xerox Palo Alto Research Center. He wrote his first paper on MVC in 1978. He originally called it the Thing Model View Editor pattern, but he quickly changed the name of the pattern to the Model View Controller pattern.

NOTE

Trygve Reenskaug, the inventor of the MVC pattern, currently works as a professor of informatics at the University of Oslo in Norway.

The MVC pattern was first implemented as part of the Smalltalk-80 class library. It was originally used as an architectural pattern for creating graphical user interfaces (GUIs).

The meaning of MVC shifted radically when the pattern was adapted to work with web applications. In the context of web applications, the MVC pattern is sometimes referred to as the Model 2 MVC pattern.

The MVC pattern has proven to be very successful. Today, the MVC pattern is used by several popular web application frameworks including Ruby on Rails, Merb, and Django. The MVC pattern is also popular in the Java world. In the Java world, MVC is used in the Struts, Spring, and Tapestry frameworks.

The first major MVC framework for ASP.NET was the open source MonoRail project (see CastleProject.org). There continues to be an active developer community around this project.

The Microsoft ASP.NET MVC framework was originally created by Scott Guthrie on an airplane trip to Austin, Texas, to speak at the first Alt.NET conference in October 2007. (Scott Guthrie was one of the creators of ASP.NET.) Scott Guthrie's talk generated so much excitement that the ASP.NET MVC framework became an official Microsoft product. ASP.NET MVC 1.0 was released in the first part of 2009.

The Architecture of an ASP.NET MVC Application

An MVC application, a Model View Controller application, is divided into the following three parts:

- **Model—**An MVC model contains all of an application's logic that is not contained in a view or controller. The model includes all of an application's validation logic, business logic, and data access logic. The MVC model contains model classes that model objects in the application's domain.
- **View—**An MVC view contains HTML markup and view logic.
- **Controller—**An MVC controller contains control-flow logic. An MVC controller interacts with MVC models and views to control the flow of application execution.

Enforcing this separation of concerns among models, views, and controllers has proven to be a useful way of structuring a web application.

First, sharply separating views from the remainder of a web application enables you to redesign the appearance of your application without touching any of the core logic. A web page designer (the person who wears the black beret) can modify the views independently of the software engineers who build the business and data access logic. People with different skills and roles can modify different parts of the application without stepping on each other's toes.

Furthermore, separating the views from the remainder of your application logic enables you to easily change the view technology in the future. One fine day, you might decide to re-implement the views in your application using Silverlight instead of HTML. If you entangle your view logic with the rest of your application logic, migrating to a new view technology will be difficult.

Separating controller logic from the remainder of your application logic has also proven to be a useful pattern for building web applications. You often need to modify the way that a user interacts with your application. You don't want to touch your view logic or model logic when modifying the flow of execution of your application.

Understanding the Sample ASP.NET MVC Application

A good way to get a firmer grasp on the three logical parts of an MVC application is to take a look at the sample application that is created automatically when you create a new ASP.NET MVC project with Visual Studio.

NOTE

We discuss installing ASP.NET MVC in the Introduction.

Follow these steps:

1. Launch Visual Studio.
2. Select the menu option File, New Project.
3. In the New Project dialog, select your favorite programming language (C# or VB.NET) and select the ASP.NET MVC Web Application template. Give your project the name `MyFirstMvcApp` and click the OK button (see Figure 1.2).

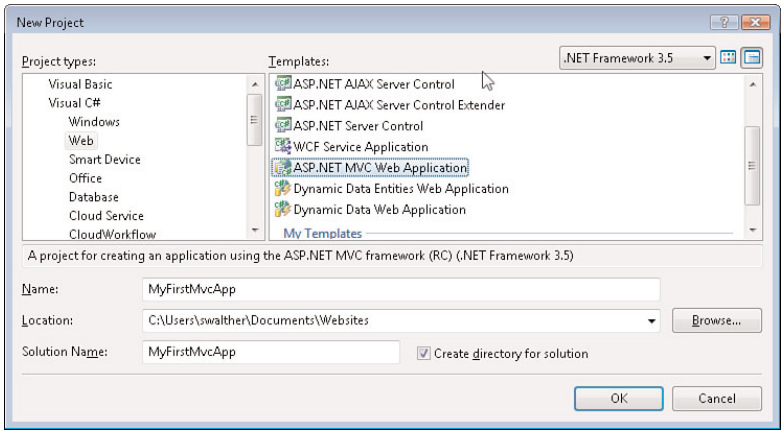

FIGURE 1.2 Creating a new ASP.NET MVC project

Immediately after you click the OK button to create a new ASP.NET MVC project, you see the Create Unit Test Project dialog in Figure 1.3. Leave the default option selected—**Yes, Create a Unit Test Project**—and click the OK button.

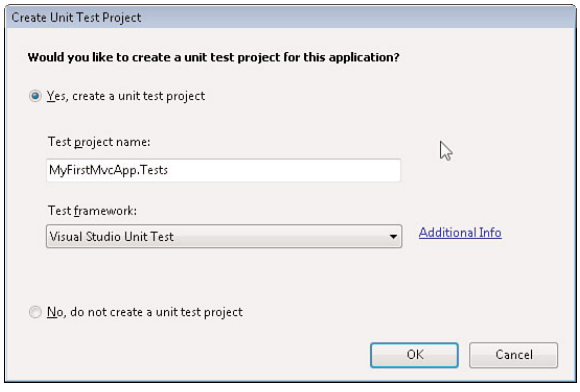

FIGURE 1.3 Creating a unit test project

Your computer hard drive will churn for a few seconds while Visual Studio creates the default files for a new ASP.NET MVC project. After all the files are created, the Solution Explorer window should contain the files in Figure 1.4.

The Solution Explorer window in Figure 1.4 contains two separate projects: the ASP.NET MVC project and the Test project. The Test project contains all the unit tests for your application.

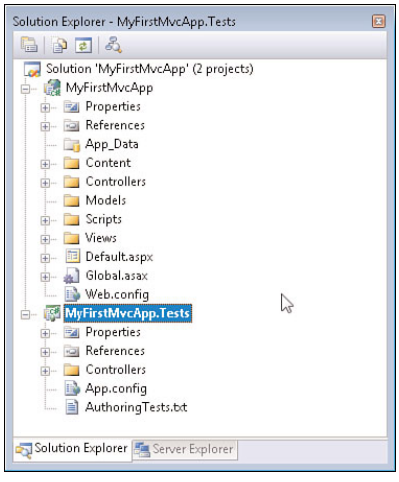

FIGURE 1.4 Files in a new ASP.NET MVC project

ASP.NET MVC Folder Conventions

The ASP.NET MVC framework emphasizes convention over configuration. There are standard locations for each type of file in an ASP.NET MVC project. The ASP.NET MVC application project contains the following folders:

- **App_Data—**Contains database files. For example, the App_Data folder might contain a local instance of a SQL Server Express database.
- **Content—**Contains static content such as images and Cascading Style Sheet files.
- **Controllers—**Contains ASP.NET MVC controller classes.
- **Models—**Contains ASP.NET MVC model classes.
- **Scripts—**Contains JavaScript files including the ASP.NET AJAX Library and jQuery.
- **Views—**Contains ASP.NET MVC views.

When building an ASP.NET MVC application, you should place controllers only in the Controllers folder, JavaScript scripts only in the Scripts folder, ASP.NET MVC views only in the Views folder, and so on. By following these conventions, your application is more easily maintained, and it can be more easily understood by others.

Running the Sample ASP.NET MVC Application

When you create a new ASP.NET MVC application, you get a simple sample application. You can run this sample application by selecting the menu option Debug, Start Debugging (or press the F5 key).

NOTE

When running an ASP.NET MVC application, make sure that the ASP.NET MVC project and not the Test project is selected in the Solution Explorer window.

The first time that you run a new ASP.NET MVC application in Visual Studio, you receive a dialog asking if you want to enable debugging. Simply click the OK button.

When you run the application, your browser opens with the page in Figure 1.5.

You can use the tabs that appear at the top of the page to navigate to either the Home or the About page. You also can click the Login link to register or log in to the application. And, that is all you can do with the application.

This sample application is implemented with one ASP.NET MVC controller and two ASP.NET MVC views. The sample application does not contain any business or data access logic, so it does not contain any ASP.NET MVC model classes.

The controller is located in the Controllers folder:

(C#)

\Controllers\HomeController.cs

(VB)

\Controllers\HomeController.vb

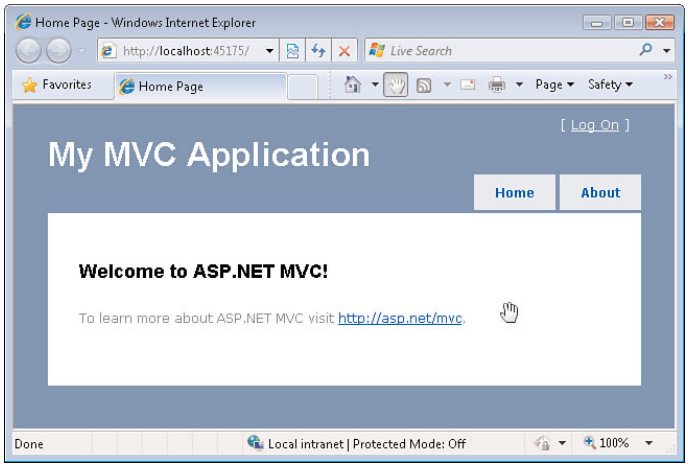

FIGURE 1.5 The sample application

If you open the HomeController in the Code Editor window, you see the file in Listing 1.1.

LISTING 1.1 Controllers\HomeController.cs (C#)

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

namespace MyFirstMvcApp.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Message"] = "Welcome to ASP.NET MVC!";

            return View();
        }

        public ActionResult About()
        {
            return View();
        }
```

```
    }
}
```

LISTING 1.1 Controllers\HomeController.vb (VB)

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index() As ActionResult
        ViewData("Message") = "Welcome to ASP.NET MVC!"

        Return View()
    End Function

    Function About() As ActionResult
        Return View()
    End Function
End Class
```

The file in Listing 1.1 contains a class with two methods named `Index()` and `About()`. Methods exposed by a controller are called actions. Both the `Index()` and `About()` actions return a view.

When you first run the sample application, the `Index()` action is invoked and this action returns the Index view. If you click the About tab, the `About()` action is invoked and this action returns the About view.

The two views can be found in the Views folder at the following location:

\Views\Home\About.aspx

\Views\Home\Index.aspx

The content of the Index view is contained in Listing 1.2.

LISTING 1.2 Views\Home\Index.aspx (C#)

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


<asp:Content ID="indexTitle" ContentPlaceHolderID="TitleContent" runat="server">
    Home Page
</asp:Content>

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

```
    <h2><%= Html.Encode(ViewData["Message"]) %></h2>
    <p>
        To learn more about ASP.NET MVC visit <a href="http://asp.net/mvc"
➥title="ASP.NET MVC Website">http://asp.net/mvc</a>.
    </p>
</asp:Content>
```

LISTING 1.2 Views\Home\Index.aspx (VB)

```
<%@ Page Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
➥Inherits="System.Web.Mvc.ViewPage" %>


<asp:Content ID="indexTitle" ContentPlaceHolderID="TitleContent" runat="server">
    Home Page
</asp:Content>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">
    <h2><%= Html.Encode(ViewData("Message")) %></h2>
    <p>
        To learn more about ASP.NET MVC visit <a href="http://asp.net/mvc"
➥title="ASP.NET MVC Website">http://asp.net/mvc</a>.
    </p>
</asp:Content>
```

Notice that a view consists mostly of standard HTML content. For example, the view contains standard <h2> and <p> tags. A view generates a page that is sent to the browser.

Summary

The goal of this chapter was to provide you with an overview of the ASP.NET MVC framework. The first part of this chapter was devoted to a discussion of a definition of good software. You were provided with a brief introduction to software design principles and patterns and the importance of unit tests. You learned how software design principles and patterns and unit tests enable you to create software that is resilient to change.

Next, you were provided with an introduction to the Model View Controller software design pattern. You learned about the history and benefits of this pattern. You learned how the ASP.NET MVC framework implements the Model View Controller pattern and how ASP.NET MVC enables you to perform pattern-based software development.

Finally, we explored the sample ASP.NET MVC application that is created when you create a new ASP.NET MVC project. We took our first look at an ASP.NET MVC controller and an ASP.NET MVC view.

Index

Symbols

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q-R

S

T

U

V

W

X-Y-Z