Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional ASP.NET MVC 5

Foreword by Scott Hanselman

Jon Galloway, Brad Wilson, K. Scott Allen, David Matson

Table of Contents

List of Illustrations

List of Tables

Chapter 1
Getting Started

—by Jon Galloway

What's In This Chapter?

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

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

This is a Professional Series book about a version 5 web framework, so we keep the introductions short. We're not going to spend any time convincing you that you should learn ASP.NET MVC. We assume that you've bought this book for that reason, and that the best proof of software frameworks and patterns is in showing how they're used in real-world scenarios.

A QUICK INTRODUCTION TO ASP.NET MVC

ASP.NET MVC is a framework for building web applications that applies the general Model-View-Controller pattern to the ASP.NET framework. Let's break that down by first

looking at how ASP.NET MVC and the ASP.NET framework are related.

How ASP.NET MVC Fits in with ASP.NET

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

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

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

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

The MVC Pattern

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

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

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

MVC as a User Interface Pattern

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

MVC as Applied to Web Frameworks

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

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

Note

It's important to keep in mind that MVC is a high-level architectural pattern, and its application varies depending on use. ASP.NET MVC is contextualized both to the problem domain (a stateless web environment) and the host system (ASP.NET).

Occasionally I talk to developers who have used the MVC pattern in very different environments, and they get confused, frustrated, or both (confustrated?) because they assume that ASP.NET MVC works the exact same way it worked in their mainframe account processing system 15 years ago. It doesn't, and that's a good thing—ASP.NET MVC is focused on providing a great web development framework using the MVC pattern and running on the .NET platform, and that contextualization is part of what makes it great.

ASP.NET MVC relies on many of the same core strategies that the other MVC platforms use, plus it offers the benefits of compiled and managed code and exploits newer .NET language features, such as lambdas and dynamic and anonymous types. At its heart, though, ASP.NET applies the fundamental tenets found in most MVC-based web frameworks:

- Convention over configuration
- Don't repeat yourself (also known as the “DRY” principle)
- Pluggability wherever possible
- Try to be helpful, but if necessary, get out of the developer's way

The Road to MVC 5

In the five years since ASP.NET MVC 1 was released in March 2009, we've seen five major releases of ASP.NET MVC and several more interim releases. To understand ASP.NET MVC 5, it's important to understand how we got here. This section describes the contents and background of each of the three major ASP.NET MVC releases.

Don't Panic!

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

ASP.NET MVC 1 Overview

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

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

and he and ScottGu bounced prototypes, code, and ideas back and forth.

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

ASP.NET MVC 2 Overview

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

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

It also had lots of API enhancements and “pro” features, based on feedback from developers building a variety of applications on ASP.NET MVC 1, such as:

- Support for partitioning large applications into *areas*
- Asynchronous controllers support

- Support for rendering subsections of a page/site using `Html.RenderAction`
- Lots of new helper functions, utilities, and API enhancements

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

ASP.NET MVC 3 Overview

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

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

Razor is the first major update to rendering HTML since ASP.NET 1 shipped almost a decade ago. The default view engine used in MVC 1 and 2 was commonly called the Web Forms view engine, because it uses the same ASPX/ASCX/MASTER files and syntax used in Web Forms. It works, but it was designed to support editing controls in

a graphical editor, and that legacy shows. An example of this syntax in a Web Forms page is shown here:

```
<%@ Page Language="C#"
  MasterPageFile="~/Views/Shared/Site.Master" Inherits=

"System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowseVi
ewModel>"
%>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent"
runat="server">
   Browse Albums
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent"
runat="server">
   <div class="genre">
     <h3><em><%: Model.Genre.Name %></em> Albums</h3>
     <ul id="album-list">
       <% foreach (var album in Model.Albums) { %>
       <li>
         <a href="<%: Url.Action("Details", new { id =
album.AlbumId }) %>">
           <img alt="<%: album.Title %>" src="<%:
album.AlbumArtUrl %>" />
           <span><%: album.Title %></span>
         </a>
       </li>
       <% } %>
     </ul>
   </div>
</asp:Content>
```

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

```
@model MvcMusicStore.Models.Genre
@{ViewBag.Title = "Browse Albums";}
<div class="genre">
   <h3><em>@Model.Name</em> Albums</h3>
   <ul id="album-list">
       @foreach (var album in Model.Albums)
       {
```

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

The Razor syntax is easier to type, and easier to read. Razor doesn't have the XML-like heavy syntax of the Web Forms view engine. We talk about Razor in a lot more depth in Chapter 3.

MVC 4 Overview

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

- ASP.NET Web API
- Enhancements to default project templates
- Mobile project template using jQuery Mobile
- Display modes
- Task support for asynchronous controllers
- Bundling and minification

Because MVC 4 is still a pretty recent release, we explain a few of these features in a little more detail here and describe them in more detail throughout the book.

ASP.NET Web API

ASP.NET MVC was designed for creating websites. Throughout the platform are obvious design decisions that

indicate the assumed usage: responding to requests from browsers and returning HTML.

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

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

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

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

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

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

- **HTTP programming model:** The Web API development experience is optimized for working with HTTP requests and responses. There's a strongly typed HTTP object model, HTTP status codes and headers are easily accessible, and so on.
- **Action dispatching based on HTTP verbs:** In MVC the dispatching of action methods is based on their names. In Web API, methods can be automatically dispatched based on the HTTP verb. So, for example, a GET request would be automatically dispatched to a controller action named GetItem.
- **Content negotiation:** HTTP has long supported a system of content negotiation, in which browsers (and other HTTP clients) indicate their response format

preferences, and the server responds with the highest preferred format that it can support. This means that your controller can supply XML, JSON, and other formats (you can add your own), responding to whichever the client most prefers. This allows you to add support for new formats without having to change any of your controller code.

- **Code-based configuration:** Service configuration can be complex. Unlike WCF's verbose and complex configuration file approach, Web API is configured entirely via code.

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

Note

As described previously, MVC and Web API have a lot in common (model-controller patterns, routing, filters, etc.). Architectural reasons dictated that they would be separate frameworks which shared common models and paradigms in MVC 4 and 5. For example, MVC has maintained compatibility and a common codebase (e.g. the `System.Web`'s `HttpContext`) *with ASP.NET, which didn't fit the long term goals of Web API.*

However, in May 2014 the ASP.NET team announced their plans to merge MVC, Web API and Web Pages in MVC 6. This next release is part of what is being called ASP.NET vNext, which is planned to run on a "cloud optimized" version of the .NET Framework. These framework changes provide a good

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

- MVC, Web API, and Web Pages will be merged into one framework, called MVC 6. MVC 6 has no dependency on System.Web.
- ASP.NET vNext includes new cloud-optimized versions of MVC 6, SignalR 3, and Entity Framework 7.
- ASP.NET vNext will support true side-by-side deployment for all dependencies, including .NET for cloud. Nothing will be in the GAC.