Programming Microsoft® ASP.NET MVC

SECOND EDITION

Your expert reference to the principles, internal mechanics, and techniques for ASP.NET MVC

Delve into the features, principles, and pillars of the ASP.NET MVC framework—and begin building your own MVC-based apps quickly. ASP.NET MVC forces developers to think in terms of distinct components—Model, View, Controller—that make it easier to manage application complexity, while enabling strict control over the markup. Web development expert Dino Esposito deftly illuminates the framework’s mechanics—and shares best ways to use this programming model versus Web Forms.

Discover how to:
- Exploit the separation of UI from code for more accurate design
- Process and develop views using HTML helper components
- Keep applications lean with good controller design
- Combine view model objects, template editors, and validators to build effective data entry pages
- Design views and controllers to be SEO-friendly and localization-aware
- Use AJAX to take full control over HTML
- Design for testability, extensibility, and security
- See when and how to customize ASP.NET MVC

About the Author
Dino Esposito is a well-known expert, trainer, and consultant on ASP.NET and mobile technologies. He has written several popular books, including Programming Microsoft ASP.NET 4, and is coauthor of Microsoft .NET: Architecting Applications for the Enterprise. He’s also a regular contributor to MSDN® Magazine and speaks at industry events such as DevConnections and Microsoft TechEd.
To Silvia and my back for sustaining me.
## Contents at a Glance

*Introduction*  

### Introduction

#### PART I  ASP.NET MVC FUNDAMENTALS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ASP.NET MVC Controllers</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>ASP.NET MVC Views</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>The Model-Binding Architecture</td>
<td>103</td>
</tr>
<tr>
<td>4</td>
<td>Input Forms</td>
<td>131</td>
</tr>
</tbody>
</table>

#### PART II  ASP.NET MVC SOFTWARE DESIGN

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Aspects of ASP.NET MVC Applications</td>
<td>189</td>
</tr>
<tr>
<td>6</td>
<td>Securing Your Application</td>
<td>227</td>
</tr>
<tr>
<td>7</td>
<td>Design Considerations for ASP.NET MVC Controllers</td>
<td>253</td>
</tr>
<tr>
<td>8</td>
<td>Customizing ASP.NET MVC Controllers</td>
<td>281</td>
</tr>
<tr>
<td>9</td>
<td>Testing and Testability in ASP.NET MVC</td>
<td>327</td>
</tr>
</tbody>
</table>

#### PART III  CLIENT-SIDE

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>More Effective JavaScript</td>
<td>373</td>
</tr>
</tbody>
</table>

*Index*  

415
# Table of Contents

<table>
<thead>
<tr>
<th>Introduction</th>
<th>xiii</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PART I</strong> ASP.NET MVC FUNDAMENTALS</td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 1</strong> ASP.NET MVC Controllers</td>
<td>3</td>
</tr>
<tr>
<td>Routing Incoming Requests</td>
<td>4</td>
</tr>
<tr>
<td>Simulating the ASP.NET MVC Runtime</td>
<td>4</td>
</tr>
<tr>
<td>The URL Routing HTTP Module</td>
<td>7</td>
</tr>
<tr>
<td>Application Routes</td>
<td>9</td>
</tr>
<tr>
<td>The Controller Class</td>
<td>15</td>
</tr>
<tr>
<td>Aspects of a Controller</td>
<td>15</td>
</tr>
<tr>
<td>Writing Controller Classes</td>
<td>17</td>
</tr>
<tr>
<td>Processing Input Data</td>
<td>21</td>
</tr>
<tr>
<td>Producing Action Results</td>
<td>25</td>
</tr>
<tr>
<td>Special Capabilities of Controllers</td>
<td>29</td>
</tr>
<tr>
<td>Grouping Controllers</td>
<td>29</td>
</tr>
<tr>
<td>Asynchronous Controllers</td>
<td>33</td>
</tr>
<tr>
<td><strong>Chapter 2</strong> ASP.NET MVC Views</td>
<td>41</td>
</tr>
<tr>
<td>Structure and Behavior of a View Engine</td>
<td>42</td>
</tr>
<tr>
<td>Mechanics of a View Engine</td>
<td>42</td>
</tr>
<tr>
<td>Definition of the View Template</td>
<td>47</td>
</tr>
</tbody>
</table>

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: microsoft.com/learning/booksurvey
Chapter 3  The Model-Binding Architecture

The Input Model ..................................................104
  Evolving from the Web Forms Input Processing .............104
  Input Processing in ASP.NET MVC ..........................105

Model Binding .....................................................107
  Model-Binding Infrastructure ................................107
  The Default Model Binder ......................................108
  Customizable Aspects of the Default Binder .................119

Advanced Model Binding ........................................120
  Custom Type Binders ..........................................121
  A Sample DateTime Model Binder ..............................124

Summary ..................................................................129
Chapter 4  Input Forms

General Patterns of Data Entry ................................................. 132
A Classic Select-Edit-Post Scenario ...................................... 132
Applying the Post-Redirect-Get Pattern ................................. 139
Ajax-Based Forms ................................................................. 143

Automating the Writing of Input Forms ................................. 153
  Predefined Display and Editor Templates ............................. 153
  Custom Templates for Model Data Types ............................ 163

Input Validation ................................................................. 167
  Using Data Annotations ..................................................... 168
  Advanced Data Annotations .............................................. 173
  Self-Validation ................................................................. 180

Summary ............................................................................... 185

PART II  ASP.NET MVC SOFTWARE DESIGN

Chapter 5  Aspects of ASP.NET MVC Applications

ASP.NET Intrinsic Objects ....................................................... 189
  SEO and HTTP Response ................................................ 190
  Managing the Session State ............................................. 193
  Caching Data ..................................................................... 194

Error Handling .................................................................... 200
  Handling Program Exceptions ....................................... 201
  Global Error Handling .................................................... 206
  Dealing with Missing Content ........................................ 209

Localization ......................................................................... 212
  Using Localizable Resources ......................................... 212
  Dealing with Localizable Applications ............................ 220

Summary .............................................................................. 226
Chapter 6  Securing Your Application  227
Security in ASP.NET MVC ......................................................... 227
Authentication and Authorization ................................. 228
Extending the Authorize Attribute .............................. 229
Implementing a Membership System ......................... 232
Defining a Membership Controller .............................. 232
The Remember-Me Feature and Ajax .......................... 237
External Authentication Services ............................... 240
The OpenID Protocol .............................................................. 240
Authenticating via Twitter ............................................... 246
Summary ............................................................... 251

Chapter 7  Design Considerations for ASP.NET MVC Controllers  253
Shaping Up Your Controller ........................................... 254
Choosing the Right Stereotype ........................................ 254
Fat-Free Controllers .......................................................... 257
Connecting the Presentation and Back End .................. 264
The iPODD Pattern .............................................................. 264
Injecting Data and Services in Layers .......................... 271
Gaining Control of the Controller Factory ................... 277
Summary ............................................................... 280

Chapter 8  Customizing ASP.NET MVC Controllers  281
The Extensibility Model of ASP.NET MVC ..................... 281
The Provider-Based Model ................................................. 282
The Service Locator Model ............................................... 286
Adding Aspects to Controllers ........................................290
  Action Filters .........................................................290
  Gallery of Action Filters ............................................293
  Special Filters ........................................................302
  Building a Dynamic Loader Filter .................................306
Action Result Types ....................................................312
  Built-in Action Result Types .......................................312
  Custom Result Types ................................................317
Summary .................................................................326

Chapter 9  Testing and Testability in ASP.NET MVC  327
Testability and Design ..................................................328
  Design for Testability ................................................328
  Loosen Up Your Design ..............................................330
Basics of Unit Testing ..................................................334
  Working with a Test Harness .......................................335
  Aspects of Testing ...................................................340
Testing Your ASP.NET MVC Code ....................................345
  Which Part of Your Code Should You Test?  .....................345
  Unit Testing ASP.NET MVC Code .................................348
  Dealing with Dependencies .......................................352
  Mocking the HTTP Context ......................................358
Summary .................................................................369
PART III CLIENT-SIDE

Chapter 10 More Effective JavaScript 373

Revisiting the JavaScript Language .............................. 374
  Language Basics ............................................... 374
  Object-Orientation in JavaScript ............................. 379

jQuery’s Executive Summary ................................. 383
  DOM Queries and Wrapped Sets ............................ 383
  Selectors .................................................... 385
  Events ....................................................... 390

Aspects of JavaScript Programming .......................... 392
  Unobtrusive Code ........................................... 392
  Reusable Packages and Dependencies ..................... 396
  Script and Resource Loading ................................ 399

ASP.NET MVC, Ajax and JavaScript ......................... 403
  The Ajax Service Layer ..................................... 404
  Ways to Write Ajax ASP.NET MVC Applications ............ 406

Summary ......................................................... 414

Index .............................................................. 415

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can continually improve our
books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey
Introduction

Get your facts first, and then you can distort them as much as you please.

—Mark Twain

Until late 2008, I was happy enough with Web Forms. I did recognize its weak points and could nicely work around them with discipline and systematic application of design principles. But a new thing called ASP.NET MVC was receiving enthusiastic reviews by a growing subset of the ASP.NET community. So I started to consider ASP.NET MVC and explore its architecture and potential while constantly trying to envision concrete business scenarios in which to employ it. I did this for about a year. Then I switched to ASP.NET MVC.

ASP.NET was devised in the late 1990s at a time when many companies in various industry sectors were rapidly discovering the Internet. For businesses, the Internet was a real breakthrough, making possible innovations in software infrastructure, marketing, distribution, and communication that were impractical or impossible before. Built on top of classic Active Server Pages (ASP), ASP.NET was the right technology at the right time, and it marked a turning point for the Web industry as a whole. For years, being a Web developer meant gaining a skill set centered on HTML and JavaScript and that was, therefore, radically different from the skills required for mainstream programming, which at the time was mostly based on C/C++, Java, and Delphi languages. ASP.NET combined the productivity of a visual and RAD environment with a component-based programming model. The primary goal of ASP.NET was to enable developers to build applications quickly and effectively without having to deal with low-level details such as HTTP, HTML, and JavaScript intricacies. That was exactly what the community loudly demanded in the late 1990s. And ASP.NET is what Microsoft delivered to address this request, exceeding expectations by a large extent.

Ten years later, today, ASP.NET is showing signs of age. The Web Forms paradigm still allows you to write functional applications, but it makes it harder and harder to stay in sync with new emerging standards, including both W3C recommendations and de facto industry standards. Today’s sites raise the bar of features high and demand
things like full accessibility, themeability, Ajax, and browser independence, not to mention support for new tags and features as those coming up with HTML 5 and the fast-growing mobile space.

Today, you can still use Web Forms in one way or another to create accessible sites that can be skinned with CSS, offer Ajax capabilities, and work nearly the same across a variety of browsers. Each of these features, however, is not natively supported and incorporated in ASP.NET Web Forms, and this contributes to making the resulting application more fragile and brittle. For this reason, a new foundation for Web development is needed. ASP.NET MVC is the natural follow-up for ASP.NET developers—even though Web Forms will still be there and improved version after version to the extent that it is possible.

This leads me to another thought. From what I can see, most people using Web Forms are maintaining applications written for ASP.NET 2.0 and topped with some Ajax extensions. Web Forms will continue to exist for legacy projects; I’m not really sure that for new projects that the small changes we had in ASP.NET 4 and those slated for ASP.NET 5.0 will really make a difference. The real big change is switching to ASP.NET MVC. Again, that’s just the natural follow up for ASP.NET developers.

**Who Should Read This Book**

This book is not for absolute beginners, but I do feel it is a book for everyone else, including current absolute beginners when they’re no longer beginners. The higher your level of competency and expertise is, the less you can expect to find here that adds value in your particular case. However, this book comes after a few years of real-world practice, so I’m sure it has a lot of solutions that may appeal also the experts. What I can say today is that there are aspects of the solutions presented in this book that go beyond ASP.NET MVC 4, at least judging from the publicly available roadmap.

If you do ASP.NET MVC, I’m confident that you will find something in this book that makes it worth the cost.

**Assumptions**

The ideal reader of this book fits the following profile to some degree. The reader has played a bit with ASP.NET MVC (the version doesn’t really matter) and is familiar with ASP.NET programming because of Web Forms development. The statement “Having
played a bit with ASP.NET MVC™ raises the bar a bit higher than ground level and specifically means the following:

- The reader understands the overall structure of an ASP.NET MVC project (for example, what controllers and views are for).
- The reader compiled a HelloWorld site and modified it a bit.
- The reader can securely tweak a web.config or global.asax file.

Anything beyond this level of familiarity is not a contra-indication for using this book. I built the book (and the courseware based on it) so that everyone beyond a basic level of knowledge can find some value in it. Rest assured that the value a seasoned architect can get out of it is different from the value the book has for an experienced developer.

In addition, the book also works for everybody who is familiar with the MVC pattern but not specifically with the ASP.NET platform. Clearly, readers with this background won’t find in this book a step-by-step guide to the ASP.NET infrastructure, but once they attain such knowledge from other resources (such as another recent book of mine published by Microsoft Press, Programming Microsoft ASP.NET 4), they can get the same value from reading this book as other readers.

**Who Should Not Read This Book**

The ideal reader of this book should not be looking for a step-by-step guide to ASP.NET MVC. The book’s aim is to explain the mechanics of the framework and effective ways to use it. It skims through basic steps. If you think you need a beginner’s guide, well, you probably will find this book a bit disappointing. You might not be able to see the logical flow of chapters and references and you could get lost quite soon. If you’re a beginner, I recommend you flip through the pages and purchase a copy only if you see something that will help you in a specific or immediate way (for example, material that helps you solve a problem you are currently experiencing). In this case, the book has helped you accomplish something significant.
System Requirements

You will need the following hardware and software to compile and run the code associated with this book:


- Visual Studio 2010, any edition (multiple downloads may be required if using Express Edition products).

- SQL Server 2008 Express Edition or higher (2008 or R2 release), with SQL Server Management Studio 2008 Express or higher (included with Visual Studio, Express Editions require separate download). For a couple of examples, you might need to install the Northwind database within SQL Server. The database is included in the package. After installing the Northwind database in SQL Server, you might also want to edit the connection string as required.

- Computer that has a 1.6 GHz or faster processor (2 GHz recommended).

- 1 GB (32 Bit) or 2 GB (64 Bit) RAM (Add 512 MB if running in a virtual machine or SQL Server Express Editions, more for advanced SQL Server editions).

- 3.5 GB of available hard disk space.

Code Samples

This book features a companion website that makes available to you all the code used in the book. This code is organized by chapter, and you can download it from the companion site at this address:

http://go.microsoft.com/FWLink/?Linkid=230567

Follow the instructions to download the Mvc3-SourceCode.zip file.
Errata & Book Support

We’ve made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

http://go.microsoft.com/fwlink/?LinkId=230565

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let’s keep the conversation going. We’re on Twitter: http://twitter.com/MicrosoftPress
Acknowledgments

The man who doesn’t read good books has no advantage over the man who can’t read them.
—Mark Twain

This is a book that I had no plans to write. It was Devon Musgrave who pushed me to update the previous edition, which was based on MVC 2. We looked at some Amazon reviews and we found out that there were some things in the previous edition that needed some fixing. Yes, feedback does help, and even though book reviews are not always crystal clear in their origin (there could be anybody behind a nickname), ideas expressed are always an asset.

So I looked over some of those reviews and critically reviewed the old book, chapter by chapter. And I found a few things to fix; not coincidentally, the same things I changed along the way in my ASP.NET MVC courseware. The fundamental change that hopefully makes this book far more valuable than the previous edition is that I managed to move the focus from the infrastructure to actual coding.

I wrote quite a few books that people found useful and helpful in their ability to understand the underlying machinery of a technology. This is not a winning point for a substantial part of the ASP.NET MVC audience. Most ASP.NET MVC developers have significant experience and excellent skills; they may not know ASP.NET MVC in detail, but they know a lot about Web programming and they’re quick learners. They need to ramp up on ASP.NET MVC and understand its intricacies and they don’t see the point of studying the underpinnings of the framework. So Devon guided me to refresh the book to give it a different slant. This book ended up as a complete rewrite; not simply a refresh. But now I’m really proud of this new baby. And I hope it addresses some of the nicknames (hopefully, real people) who reviewed and commented the MVC 2 book on Amazon a few months ago.

Marc Young took the responsibility of ensuring the technical quality of the book. And he pushed me hard on making the companion code a super-quality product, which is much better organized than in the past. (I admit I tend to be as lazy on companion code as I tend to be deep—and sometimes repetitive—on concepts.)

I have a joke about my English in every book. I write over and over again how bad my English is and how great Roger LeBlanc is in making it good. After a decade spent writing books in English I really think that it’s now good enough to keep Roger’s work to a minimum. And, in fact, in this book Roger played the wider role of managing editor.
Steve Sagman has been like a background task pushing notifications timely. I made most of the promised deadlines, but Steve has been flexible enough to adjust deadlines so that it seemed that I made all of them. Working with Steve is kind of relaxing; he never transmits pressure but he kicks in at the right time; which is probably the secret trick to not adding pressure.

Like millions of other Italian students, I spent many teenage hours trying to catch the spirit of the Divine Comedy. As you may know, the whole poem develops around a journey that Dante undertakes through the three realms of the dead guided by the Roman poet Virgilio. I too spent many hours of my past months trying to catch and express the gist of ASP.NET MVC. I began a journey through controllers, views, models and filters guided by a top-notch developer, trainer and friend—Hadi Hariri.

Loyal readers of my books may know about my (insane) passion for tennis. My wife Silvia told me once “OK, you like tennis so much, but is there any chance that you can make some money from it?” I never dared ask whether she meant “making money playing and winning tournaments” or “making money through software.” To be on the safe side, I decided to train and play a lot more while spending many hours helping out Giorgio Garcia and the entire team at Crionet and e-tennis.net to serve better Web and mobile services to tennis tournaments and their fans. I joined Crionet as the Chief Technical Officer and I’m really enjoying going out for tournaments and focusing on domain logic of a tennis game. It was really nice last June to make it to the Wimbledon’s Centre Court and claim it was for work and not for fun!

My son Francesco (13) is now officially a junior Windows Phone 7 developer with five applications already published to the marketplace. By the way, check out the nicest of his apps—ShillyShally, a truly professional tool for decision makers. He doesn’t do much Web programming now, but he’s pushing me hard for a mobile book—which is exactly one of my ongoing projects as I write these notes. If you do, or plan to do, mobile stay tuned or, better yet, get in touch.

Michela (10) is simply the perfect end user in this crazy technological world and a wonderful lover of German shepherds and baby tigers.
CHAPTER 3

The Model-Binding Architecture

*It does not matter how slowly you go, so long as you do not stop.*

—Confucius

By default, the Microsoft Visual Studio standard project template for ASP.NET MVC applications includes a Models folder. If you look around for some guidance on how to use it and information about its intended role, you quickly reach the conclusion that the Models folder exists to store model classes. Fine, but which model is it for? Or, more precisely, what’s the definition of a “model”?

In general, there are at least two distinct models—the domain model and the view model. The former describes the data you work with in the middle tier and is expected to provide a faithful representation of the domain entities. These entities are typically persisted by the data-access layer and consumed by services that implement business processes. This *domain model* (or *entity model* or even *data model*, if you like) pushes a vision of data that is, in general, distinct from the vision of data you find in the presentation layer. The view model just describes the data being worked on in the presentation.

Having said that, I agree with anyone who says that not every application needs a neat separation between the object models used in the presentation and business layers. Nonetheless, two distinct models logically exist, and coexist, in a typical layered web solution. You might decide that for your own purposes the two models nearly coincide, but you should always recognize the existence of two distinct models that operate in two distinct layers.

This chapter introduces a third type of model that, although hidden for years in the folds of the ASP.NET Web Forms runtime, stands on its own in ASP.NET MVC—the *input model*. The input model refers to the model through which posted data is exposed to controllers.

In Chapter 1, “ASP.NET MVC Controllers,” we discussed request routing and structure of controller methods. In Chapter 2, “ASP.NET MVC Views,” we discussed views as the primary result of action processing. We didn’t discuss thoroughly yet how a controller method gets input data.
The Input Model

In ASP.NET Web Forms, we had server controls, view state, and the overall page life cycle working in the background to serve developers input data that was ready to use. With ASP.NET Web Forms, developers had no need to worry about an input model. Server controls in ASP.NET Web Forms provide a faithful server-side representation of the client user interface. Developers just need to write C# code to read from input controls.

This approach doesn’t work well with the philosophy of ASP.NET MVC—which is more close to the metal with very thin abstraction over HTTP. Moreover, ASP.NET MVC makes a point of having highly testable controllers—which means that controllers should receive input data, not retrieve it. To pass input data to a controller, you need to package data in some way. This is precisely where the input model comes into play.

To better understand of the importance and power of the new ASP.NET MVC input model, let’s start from where ASP.NET Web Forms left us.

Evolving from the Web Forms Input Processing

An ASP.NET Web Forms application is based on pages, and each server page is based on server controls. The page has its own life cycle that spans from processing the raw request data to arranging the final response for the browser. The page life cycle is fed by raw request data such as HTTP headers, cookies, the URL, and the body, and it produces a raw HTTP response containing headers, cookies, the content type, and the body.

Inside the page life cycle there are a few steps in which HTTP raw data is massaged into more easily programmable containers—server controls. In ASP.NET Web Forms, these “programmable containers” are never perceived as being part of an input object model. Furthermore, many developers don’t realize the existence of such a model. In ASP.NET Web Forms, the input model is just based on server controls and the view state.

Role of Server Controls

Suppose you have a web page with a couple of TextBox controls to capture the user name and password. When the user posts the content of the form, there will likely be a piece of code to process the request as shown here:

```csharp
public void Button1_Click(Object sender, EventArgs e)
{
    // You're about to perform requested action using input data.
    CheckUserCredentials(TextBox1.Text, TextBox2.Text);
    ...
}
```

The overall idea behind the architecture of ASP.NET Web Forms is to keep the developer away from raw data. Any incoming request data is mapped to properties on server controls. When this is not possible, data is left parked in general-purpose containers such as QueryString or Form.
What would you expect from a method like the `Button1_Click` just shown? That method is the Web Forms counterpart of a controller action. Here’s how to refactor the previous code to use an explicit input model:

```csharp
public void Button1_Click(Object sender, EventArgs e)
{
    // You're actually filling in the input model of the page.
    var model = new UserCredentialsInputModel();
    model.UserName = TextBox1.Text;
    model.Password = TextBox2.Text;

    // You're about to perform the requested action using input data.
    CheckUserCredentials(model);
    ...;
}
```

The ASP.NET runtime environment breaks up raw HTTP request data into control properties, thus offering a control-centric approach to request processing.

Note that in the upcoming ASP.NET Web Forms 4.5, Microsoft is going to introduce some model binding capabilities just along the lines shown a moment ago. In particular, they suggest you call a method of yours from within `Button1_Click` (a go-without-saying practice) and give this method any signature you need. Parameters on this signature can be decorated with attributes to instruct the runtime to try to resolve those values from QueryString, Forms or other value providers.

### Role of the View State

Speaking in terms of a programming paradigm, a key distinguishing character between ASP.NET Web Forms and ASP.NET MVC is the view state. In Web Forms, the view state plays a central role and helps server controls to always be up to date. Because of the view state, as a developer you don’t need to care about segments of the user interface you don’t touch in a postback. Suppose you display a list of choices for the user to drill down into. When the request for details is made, in Web Forms all you need to do is display the details. The raw HTTP request, however, posted the list of choices as well as key information to find. The view state makes it unnecessary for you to deal with the list of choices.

The view state and server control build a thick abstraction layer on top of classic HTTP mechanics, and they make you think in terms of page sequences rather than successive requests. This is neither wrong nor right; it is just the paradigm behind Web Forms. In Web Forms, there’s no need for clearly defining an input model. If you do that, it’s only because you want to keep your code cleaner and more readable.

### Input Processing in ASP.NET MVC

In Chapter 1, you saw that a controller method can access input data through `Request` collections—such as `QueryString`, `Headers`, or `Form`—or value providers. Although it’s functional, this approach is not ideal from a readability and maintenance perspective. You need an ad hoc model that exposes data to controllers.
Role of Model Binders

The input model has one main trait. It models any data that comes your way through an HTTP request into manageable and expressive classes. As a developer, you’re largely responsible for designing these classes. What about mapping request data onto properties?

ASP.NET MVC provides an automatic binding layer that uses a built-in set of rules for mapping request data to properties from any value providers. The logic of the binding layer can be customized to a large extent, thus adding unprecedented heights of flexibility as far as the processing of input data is concerned.

Flavors of a Model

The ASP.NET MVC default project template offers just one Models folder, thus implicitly pushing the idea that “model” is just one thing—the model of the data the application is supposed to use. Generally speaking, this is a rather simplistic view, though it’s effective in very simple sites.

If you look deeper into things, you can recognize three different types of “models” in ASP.NET MVC, as illustrated in Figure 3-1.

![Figure 3-1](image)

The **input model** provides the representation of the data being posted to the controller. The **view model** provides the representation of the data being worked on in the view. Finally, the **domain model** is the representation of the domain-specific entities operating in the middle tier.

In this book, I’m not specifically covering the domain model because it results from the application of patterns and practices that would require a book of its own. I’ll briefly touch on these topics in Chapter 7, “Design Considerations for ASP.NET MVC Controllers.” You might want to check out the book I wrote with Andrea Saltarello called *Microsoft .NET: Architecting Applications for the Enterprise* (Microsoft Press, 2008) to read more about layered solutions. I covered the view model in Chapter 2 and will be discussing the input model in this chapter.

Note that the three models are not neatly separated, which Figure 3-1 shows to some extent. You might find overlap between the models. This means that classes in the domain model might be used
in the view, and classes posted from the client might be used in the view. The final structure and diagram of classes is up to you.

Model Binding

Model binding is the process of binding values posted over an HTTP request to the parameters used by the controller’s methods. Let’s find out more about the underlying infrastructure, mechanics, and components involved.

Model-Binding Infrastructure

The model-binding logic is encapsulated in a specific model-binder class. The binder works under the control of the action invoker and helps to figure out the parameters to pass to the selected controller method.

Analyzing the Method’s Signature

As you saw in Chapter 1, each and every request passed to ASP.NET MVC is resolved in terms of a controller name and an action name. Armed with these two pieces of data, the action invoker—a native component of the ASP.NET MVC runtime shell—kicks in to actually serve the request. First, the invoker expands the controller name to a class name and resolves the action name to a method name on the controller class. If something goes wrong, an exception is thrown.

Next, the invoker attempts to collect all values required to make the method call. In doing so, it looks at the method’s signature and attempts to find an input value for each parameter in the signature.

Getting the Binder for the Type

The action invoker knows the formal name and declared type of each parameter. (This information is obtained via reflection.) The action invoker also has access to the request context and to any data uploaded with the HTTP request—the query string, the form data, route parameters, cookies, headers, files, and so forth.

For each parameter, the invoker obtains a model-binder object. The model binder is a component that knows how to find values of a given type from the request context. The model binder applies its own algorithm—which includes the parameter name, parameter type, and request context available—and returns a value of the specified type. The details of the algorithm belong to the implementation of the model binder being used for the type.

ASP.NET MVC uses a built-in binder object that corresponds to the DefaultModelBinder class. The model binder is a class that implements the IModelBinder interface:

```csharp
public interface IModelBinder
{
    Object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext);
}
```
Let's first explore the capabilities of the default binder and then see what it takes to write custom binders for specific types later.

The Default Model Binder

The default model binder uses convention-based logic to match the names of posted values to parameter names in the controller’s method. The `DefaultModelBinder` class knows how to deal with primitive and complex types, as well as collections and dictionaries. In light of this, the default binder works just fine most of the time.

**Note** If the default binder supports primitive and complex types and the collections thereof, will you ever feel the need to use something other than the default binder? You will hardly ever feel the need to replace the default binder with another general-purpose binder. However, the default binder can’t apply your custom logic to massage request data into the properties of a given type. As you’ll see later, a custom binder is helpful when the values being posted with the request don’t exactly match the properties of the type you want the controller to use. In this case, a custom binder makes sense and helps keep the controller’s code lean and mean.

Binding Primitive Types

Admittedly, it sounds a bit magical at first, but there’s no actual wizardry behind model binding. The key fact about model binding is that it lets you focus exclusively on the data you want the controller method to receive. You completely ignore the details of how you retrieve that data, whether it comes from the query string or the route.

Let’s suppose you need a controller method to repeat a given string a given number of times. Here’s what you do:

```csharp
public class BindingController : Controller
{
    public ActionResult Repeat(String text, Int32 number)
    {
        var model = new RepeatViewModel {Number = number, Text = text};
        return View(model);
    }
}
```

Designed in this way, the controller is highly testable and completely decoupled from the ASP.NET runtime environment. There’s no need for you to access the `Request` object or the `Cookies` collection directly.

Where do the values for `text` and `number` come from? And which component is actually reading them into text and number parameters?
The actual values are read from the request context, and the default model-binder object does the trick. In particular, the default binder attempts to match formal parameter names (text and number in the example) to named values posted with the request. In other words, if the request carries a form field, a query string field, or a route parameter named text, the carried value is automatically bound to the text parameter. The mapping occurs successfully as long as the parameter type and actual value are compatible. If a conversion cannot be performed, an argument exception is thrown. The next URL works just fine:

http://server/binding/repeat?text=Dino&number=2

Conversely, the following URL causes an exception:

http://server/binding/repeat?text=Dino&number=true

The query string field text contains Dino, and the mapping to the String parameter text on the method Repeat takes place successfully. The query string field number, on the other hand, contains true, which can’t be successfully mapped to an Int32 parameter. The model binder returns a parameters dictionary where the entry for number contains null. Because the parameter type is Int32—that is, a non-nullable type—the invoker throws an argument exception.

**Dealing with Optional Values**

Note that an argument exception that occurs because invalid values are being passed is not detected at the controller level. The exception is fired before the execution flow reaches the controller. This means that you won’t be able to catch it with try/catch blocks.

If the default model binder can’t find a posted value that matches a required method parameter, it places a null value in the parameter dictionary returned to the action invoker. Again, if a value of null is not acceptable for the parameter type, an argument exception is thrown before the controller method is even called.

What if a method parameter has to be considered optional?

A possible approach entails changing the parameter type to a nullable type, as shown here:

```csharp
public ActionResult Repeat(String text, Nullable<Int32> number)
{
    var model = new RepeatViewModel {Number = number.GetValueOrDefault(), Text = text};
    return View(model);
}
```

Another approach consists of using a default value for the parameter:

```csharp
public ActionResult Repeat(String text, Int32 number=4)
{
    var model = new RepeatViewModel {Number = number, Text = text};
    return View(model);
}
```
Any decisions about the controller method’s signature are up to you. In general, you might want to use types that are very close to the real data being uploaded with the request. Using parameters of type `Object`, for example, will save you from argument exceptions, but it will make it hard to write clean code to process the input data.

The default binder can map all primitive types, such as `String`, integers, `Double`, `Decimal`, `Boolean`, `DateTime`, and related collections. To express a Boolean type in a URL, you resort to the `true` or `false` strings. These strings are parsed using .NET native Boolean parsing functions, which recognize `true` and `false` strings in a case-insensitive manner. If you use strings such as `yes/no` to mean a `Boolean`, the default binder won’t understand your intentions and places a null value in the parameter dictionary, which might cause an argument exception.

**Value Providers and Precedence**

The default model binder uses all the registered value providers to find a match between posted values and method parameters. By default, value providers cover the collections listed in Table 3-1.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>Contains values posted from an HTML form, if any.</td>
</tr>
<tr>
<td>RouteData</td>
<td>Contains values excerpted from the URL route.</td>
</tr>
<tr>
<td>QueryString</td>
<td>Contains values specified as the URL’s query string.</td>
</tr>
<tr>
<td>Files</td>
<td>A value is the entire content of an uploaded file, if any.</td>
</tr>
</tbody>
</table>

Table 3-1 lists request collections being considered in the exact order in which they are processed by the default binder. Suppose you have the following route:

```csharp
routes.MapRoute(
    "Test",
    "{controller}/{action}/test/{number}",
    new { controller = "Binding", action = "RepeatWithPrecedence", number = 5 }
);
```

As you can see, the route has a parameter named `number`. Now consider this URL:

```
/Binding/RepeatWithPrecedence/test/10?text=Dino&number=2
```

The request uploads two values that are good candidates to set the value of the `number` parameter in the `RepeatWithPrecedence` method. The first value is 10 and is the value of a route parameter named `number`. The second value is 2 and is the value of the QueryString element named `number`. The method itself provides a default value for the `number` parameter:

```csharp
public ActionResult RepeatWithPrecedence(String text, Int32 number=20)
{
    ...
}
```
Which value is actually picked up? As Table 3-1 suggests, the value that actually gets passed to the method is 10—the value read from the route data collection.

**Binding Complex Types**

There’s no limitation on the number of parameters you can list on a method’s signature. However, a container class is often better than a long list of individual parameters. For the default model binder, the result is nearly the same whether you list a sequence of parameters or just one parameter of a complex type. Both scenarios are fully supported. Here’s an example:

```csharp
public class ComplexController : Controller
{
    public ActionResult Repeat(RepeatText inputModel)
    {
        var model = new RepeatViewModel
        {
            Title = "Repeating text",
            Text = inputModel.Text,
            Number = inputModel.Number
        };
        return View(model);
    }
}
```

The controller method receives an object of type `RepeatText`. The class is a plain data-transfer object defined as follows:

```csharp
public class RepeatText
{
    public String Text { get; set; }
    public Int32 Number { get; set; }
}
```

As you can see, the class just contains members for the same values you passed as individual parameters in the previous example. The model binder works with this complex type as well as it did with single values.

For each public property in the declared type—`RepeatText` in this case—the model binder looks for posted values whose key names match the property name. The match is case insensitive. Here’s a sample URL that works with the `RepeatText` parameter type:

```
http://server/Complex/Repeat?text=Dino&number=5
```

Figure 3-2 shows the output the URL might generate.
FIGURE 3-2  Repeating text with values extracted from a complex type.

Binding Collections

What if the argument that a controller method expects is a collection? For example, can you bind the content of a posted form to an `IList<T>` parameter? The `DefaultModelBinder` class makes it possible, but doing so requires a bit of contrivance of your own. Have a look at Figure 3-3.

FIGURE 3-3  The page will post an array of strings.

When the user hits the Send button, the form submits its content. Specifically, it sends out the content of the various text boxes. If the text boxes have different IDs, the posted content takes the following form:

```plaintext
TextBox1=admin@contoso.com&TextBox2=&TextBox3=&TextBox4=&TextBox5=
```
In classic ASP.NET, this is the only possible way of working because you can’t just assign the same ID to multiple controls. However, if you manage the HTML yourself, nothing prevents you from assigning the five text boxes in the figure the same ID. The HTML DOM, in fact, fully supports this scenario (though it is not recommended). Therefore, the following markup is entirely legal in ASP.NET MVC and produces HTML that works on all browsers:

```csharp
@using (Html.BeginForm())
{
    <h2>List your email address(es)</h2>
    foreach(var email in Model.Emails)
    {
        <input type="text" name="emails" value="@email" />
        <br />
    }
    <input type="submit" value="Send" />
}
```

What’s the expected signature of a controller method that has to process the email addresses typed in the form? Here it is:

```csharp
public ActionResult Emails(IList<String> emails)
{
    ...
}
```

Figure 3-4 shows that an array of strings is correctly passed to the method thanks to the default binder class.

As you’ll see in greater detail in the next chapter, when you work with HTML forms you likely need to have a pair of methods—one to handle the display of the view (the verb GET), and one to handle the scenario in which data is posted to the view. The `HttpPost` and `HttpGet` attributes allow you to mark which scenario a given method is handling for the same action name. Here’s the full implementation of the example, which uses two distinct methods to handle GET and POST scenarios:

```csharp
[ActionName("Emails")]
[HttpPost]
public ActionResult EmailForPost(IList<String> emails)
{
    var defaultEmails = new [] { "admin@contoso.com", "", "", "", "" };
    var model = new EmailsViewModel { Emails = defaultEmails };
    return View(model);
}
```

```csharp
[ActionName("Countries")]
public virtual ActionResult ListCountriesForGet(IList<String> countries)
{
    ...
}
```

**FIGURE 3-4** An array of strings has been posted.
if (emails == null) 
    emails = defaultEmails;
if (emails.Count == 0) 
    emails = defaultEmails;
var model = new EmailsViewModel {Emails = emails};
return View(model);
}

[ActionName("Emails")]
[HttpPost]
public ActionResult EmailForPost(IList<String> emails) 
{
    var defaultEmails = new[] { "admin@contoso.com", "", "", "", "" };
    var model = new EmailsViewModel { Emails = defaultEmails, RegisteredEmails = emails };
    return View(model);
}

Here’s the full Razor markup for the view you see rendered in Figure 3-5:

@using BasicInput.ViewModels.Complex;
@model EmailsViewModel
@section title{
    @Model.Title
}

@using (Html.BeginForm())
{
    <h2>List your email address(es)</h2>
    foreach(var email in Model.Emails)
    {
        <input type="text" name="emails" value="@email" />
        <br />
    }
    <input type="submit" value="Send" />
}

<hr />
<h2>Emails submitted</h2>
<ul>
    @foreach (var email in Model.RegisteredEmails)
    {
        if (String.IsNullOrWhiteSpace(email))
        {
            continue;
        }
        <li>@email</li>
    }
</ul>
In the end, to ensure that a collection of values is passed to a controller method, you need to ensure that elements with the same ID are emitted to the response stream. The ID, then, has to match to the controller method’s signature according to the normal rules of the binder.

**Note** As you might have figured out already, the default model binder does a lot of work for you. However, it requires that you use fixed IDs in the HTML forms you create. That’s the way in which the component works, and it’s probably the only way to make it work in a rather generic way for a variety of classes.

**Binding Collections of Complex Types**

The default binder can also handle situations in which the collection contains complex types, even nested:

```csharp
[ActionName("Countries")] [HttpPost] public ActionResult ListCountriesForPost(IList<Country> countries) {
    ...
}
```

As an example, consider the following definition for type *Country*:

```csharp
public class Country {
    public Country()
```
{  
    Details = new CountryInfo();
}

public String Name { get; set; }
public CountryInfo Details { get; set; }

public class CountryInfo
{
    public String Capital { get; set; }
    public String Continent { get; set; }
}

For model binding to occur successfully, all you really need to do is use a progressive index on the IDs in the markup. The resulting pattern is prefix[index].Property, where prefix matches the name of the formal parameter in the controller method's signature:

@using (Html.BeginForm())
{
    <h2>Select your favorite countries</h2>
    var index = 0;
    foreach (var country in Model.CountryList)
    {
        <fieldset>
            <div>
                <b>Name</b><br />
                <input type="text"
                    name="countries[@index].Name"
                    value="@country.Name" />
                <b>Capital</b><br />
                <input type="text"
                    name="countries[@index].Details.Capital"
                    value="@country.Details.Capital" />
                <b>Continent</b><br />
                @{
                    var id = String.Format("countries[{0}].Details.Continent", index++);
                }
                @Html.TextBox(id, country.Details.Continent)
            </div>
        </fieldset>
    }
    <input type="submit" value="Send" />
}

The index is numeric, 0-based, and progressive. In this example, I'm building user interface blocks for each specified default country. If you have a fixed number of user interface blocks to render, you can use static indexes:

<input type="text"
    name="countries[0].Name"
    value="@country.Name" />

<input type="text"
    name="countries[1].Name"
    value="@country.Name" />
Note that holes in the series (for example, 0 and then 2) stop the parsing, and all you get back is the sequence of data types from 0 to the hole.

The posting of data works fine as well. The POST method on the controller class will just receive the same hierarchy of data, as Figure 3-6 shows.

![FIGURE 3-6 Complex and nested types posted to the method.](image)

Rest assured that if you’re having trouble mapping posted values to your expected hierarchy of types, it might be wise to consider a custom model binder.

### Binding Content from Uploaded Files

Table 3-1 indicates that uploaded files can also be subject to model binding. The default binder does the binding by matching the name of the input file element used to upload with the name of a parameter. The parameter (or the property on a parameter type), however, must be declared of type `HttpPostedFileBase`:

```csharp
public class UserData
{
    public String Name { get; set; }
    public String Email { get; set; }
    public HttpPostedFileBase Picture { get; set; }
}
```

The following code shows a possible implementation of a controller action that saves the uploaded file somewhere on the server machine:

```csharp
public ActionResult Add(UserData inputModel)
{
    var destinationFolder = Server.MapPath("/Users");
    var postedFile = inputModel.Picture;
    if (postedFile.ContentLength > 0)
    {
        var fileName = Path.GetFileName(postedFile.FileName);
        var path = Path.Combine(destinationFolder, fileName);
        postedFile.SaveAs(path);
    }

    return View();
}
```
By default, any ASP.NET request can't be longer than 4 MB. This amount should include any uploads, headers, body, and whatever is being transmitted. The value is configurable at various levels. You do that through the `maxRequestLength` entry in the `httpRuntime` section of the `web.config` file:

```xml
<system.web>
  <httpRuntime maxRequestLength="6000" />
</system.web>
```

Obviously, the larger a request is, the more room you potentially leave for hackers to `prepare` attacks on your site. Note also that in a hosting scenario your application-level settings might be ignored if the hoster has set a different limit at the domain level and locked down the `maxRequestLength` property at lower levels.

What about multiple file uploads? As long as the overall size of all uploads is compatible with the current maximum length of a request, you are allowed to upload multiple files within a single request. However, consider that web browsers just don't know how to upload multiple files. All a web browser can do is upload a single file, and only if you reference it through an input element of type file. To upload multiple files, you can resort to some client-side ad hoc component or place multiple INPUT elements in the form. If multiple INPUT elements are used, and properly named, a class like the one shown here will bind them all:

```csharp
public class UserData
{
    public String Name { get; set; }
    public String Email { get; set; }
    public HttpPostedFileBase Picture { get; set; }
    public IList<HttpPostedFileBase> AlternatePictures { get; set; }
}
```

The class represents the data posted for a new user with a default picture and a list of alternate pictures. Here is the markup for the alternate pictures. Here is the markup for the alternate pictures:

```html
<input type="file" id="AlternatePictures[0]" name="AlternatePictures[0]" />
<input type="file" id="AlternatePictures[1]" name="AlternatePictures[1]" />
```

### Note
Creating files on the web server is not usually an operation that can be accomplished by relying on the default permission set. Any ASP.NET application runs under the account of the worker process serving the application pool the application belongs to. Under normal circumstances, this account is NETWORK SERVICE, and it isn’t granted the permission to create new files. This means that to save files you must change the account behind the ASP.NET application or elevate the privileges of the default account.

For years, the identity of the application pool has been a fixed identity—the aforementioned NETWORKSERVICE account, which is a relatively low-privileged, built-in identity in Microsoft Windows. Originally welcomed as an excellent security measure, the practice of using a single account for a potentially high number of concurrently running services in the end created more problems than it helped to solve.
In a nutshell, services running under the same account could tamper with each other. For this reason, in Microsoft Internet Information Services 7.5, by default worker processes run under unique identities that are automatically and transparently created for each newly created application pool. The underlying technology is known as Virtual Accounts and is currently supported by Windows Server 2008 R2 and Windows 7. For more information, have a look at http://technet.microsoft.com/en-us/library/dd548356(WS.10).aspx.

Customizable Aspects of the Default Binder

Automatic binding stems from a convention-over-configuration approach. Conventions, though, sometimes harbor bad surprises. If, for some reason, you lose control over the posted data (for example, in the case of data that has been tampered with), it can result in undesired binding—any posted key/value pair will, in fact, be bound. For this reason, you might want to consider using the Bind attribute to customize some aspects of the binding process.

The Bind Attribute

The Bind attribute comes with three properties, which are described in Table 3-2.

**TABLE 3-2** Properties for the `BindAttribute` class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix</td>
<td>String property. It indicates the prefix that must be found in the name of the posted value for the binder to resolve it. The default value is the empty string.</td>
</tr>
<tr>
<td>Exclude</td>
<td>Gets or sets a comma-delimited list of property names for which binding is not allowed.</td>
</tr>
<tr>
<td>Include</td>
<td>Gets or sets a comma-delimited list of property names for which binding is permitted.</td>
</tr>
</tbody>
</table>

You apply the Bind attribute to parameters on a method signature.

Creating Whitelists of Properties

As mentioned, automatic model binding is potentially dangerous when you have complex types. In such cases, in fact, the default binder attempts to populate all public properties on the complex types for which it finds a match in the posted values. This might end up filling the server type with unexpected data, especially in the case of request tampering. To avoid that, you can use the Include property on the Bind attribute to create a whitelist of acceptable properties:

```csharp
public ActionResult RepeatOnlyText([Bind(Include = "text")]RepeatText inputModel) {
    ...
}
```
The binding on the RepeatText type will be limited to the listed properties (in the example, only Text). Any other property is not bound and takes whatever default value the implementation of RepeatText assigned to it. Multiple properties are separated by a comma.

Creating Blacklists of Properties

The Exclude attribute employs the opposite logic: it lists properties that must be excluded from binding. All properties except those explicitly listed will be bound:

```csharp
public ActionResult RepeatOnlyText([Bind(Exclude = "number")]RepeatText inputModel)
{
    ...
}
```

You can use Include and Exclude in the same attribute if doing so allows you to better define the set of properties to bind. If, for instance, both attributes refer to the same property, Exclude will win.

Using a Prefix

The default model binder forces you to give your request parameters (for example, form and query string fields) given names that match formal parameters on target action methods. The Prefix attribute allows you to change this convention. By setting the Prefix attribute, you instruct the model binder to match request parameters against the prefix rather than against the formal parameter name. All in all, alias would have been a much better name for this attribute. Consider the following example:

```csharp
[HttpPost]
[ActionName("Emails")]
public ActionResult EmailForPost([Bind(Prefix = "foo")]IList<String> emails)
{
    ...
}
```

For the emails parameter to be successfully filled, you need to have posted a field whose name is foo, not emails. The Prefix attribute makes particular sense on POST methods.

Finally, note that if a prefix is specified, it becomes mandatory and fields whose name is not prefixed are not bound.

Advanced Model Binding

So far, we’ve examined the behavior of the default model binder. The default binder does excellent work, but it is a general-purpose tool designed to work with most possible types in a way that is not specific to any of them. The Bind attribute gives you some more control over the binding process, but there are some reasonable limitations to its abilities. If you want to achieve total control over the binding process, all you do is create a custom binder for a specific type.
Custom Type Binders

There’s just one primary reason you should be willing to create a custom binder: the default binder is limited to taking into account only a one-to-one correspondence between posted values and properties on the model.

Sometime, though, the target model has a different granularity than the one expressed by form fields. The canonical example is when you employ multiple input fields to let users enter content for a single property—for example, distinct input fields for day, month, and year that then map to a single DateTime value.

Customizing the Default Binder

To create a custom binder from scratch, you implement the IModelBinder interface. Implementing the interface is recommended if you need total control over the binding process. If, say, all you need to do is keep the default behavior and simply force the binder to use a nondefault constructor for a given type, inheriting from DefaultModelBinder is the best approach. Here’s the schema to follow:

```csharp
public RepeatTextModelBinder : DefaultModelBinder
{
    protected override object CreateModel( ControllerContext controllerContext,
                ModelBindingContext bindingContext,
                Type modelType)
    {
        ...
        return new RepeatText( ... );
    }
}
```

Another common scenario for simply overriding the default binder is when all you want is the ability to validate against a specific type. In this case, you override OnModelUpdated and insert your own validation logic, as shown here:

```csharp
protected override void OnModelUpdated(ControllerContext controllerContext,
                ModelBindingContext bindingContext)
{
    var obj = bindingContext.Model as RepeatText;
    if (obj == null)
        return;

    // Apply validation logic here for the whole model
    if (String.IsNullOrEmpty(obj.Text))
    {
        bindingContext.ModelState.AddModelError("Text", ...);
    }
    ...
}
```

You override OnModelError if you prefer to keep in a single place all validations for any properties. You resort to OnPropertyValidating if you prefer to validate properties individually.
Important  When binding occurs on a complex type, the default binder simply copies matching values into properties. You can’t do much to refuse some values if they put the instance of the complex type in an invalid state.

A custom binder could integrate some logic to check the values being assigned to properties and signal an error to the controller method or degrade gracefully by replacing the invalid value with a default one. Although it’s possible to use this approach, it’s not commonly used because there are more powerful options in ASP.NET MVC that you can use to deal with data validation across an input form. And that is exactly the topic I’ll address in the next chapter.

Implementing a Model Binder from Scratch

The `IModelBinder` interface is defined as follows:

```csharp
public interface IModelBinder
{
    object BindModel(ControllerContext controllerContext,
                      ModelBindingContext bindingContext);
}
```

Here’s the skeleton of a custom binder that directly implements the `IModelBinder` interface. The model binder is written for a specific type—in this case, `MyComplexType`:

```csharp
public class MyComplexTypeModelBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
                              ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
            throw new ArgumentNullException("bindingContext");

        // Create the model instance (using the ctor you like best)
        var obj = new MyComplexType();

        // Set properties reading values from registered value providers
        obj.SomeProperty = FromPostedData<string>(bindingContext, "SomeProperty");
        ...
        return obj;
    }

    // Helper routine
    private T FromPostedData<T>(ModelBindingContext context, String key)
    {
        // Get the value from any of the input collections
        ValueProviderResult result;
        context.ValueProvider.TryGetValue(key, out result);

        // Set the state of the model property resulting from value
        context.ModelState.SetModelValue(key, result);
    }
}
```
// Return the value converted (if possible) to the target type
return (T) result.ConvertTo(typeof(T));
}

The structure of BindModel is straightforward. You first create a new instance of the type of interest. In doing so, you can use the constructor (or factory) you like best and perform any sort of custom initialization that is required by the context. Next, you simply populate properties of the freshly created instance with values read or inferred from posted data. In the preceding code snippet, I assume you simply replicate the behavior of the default provider and read values from registered value providers based on a property name match. Obviously, this is just the place where you might want to add your own logic to interpret and massage what’s being posted by the request.

Note that when writing a model binder, you are in no way restricted to getting information for the model only from the posted data—which represents only the most common scenario. You can grab information from anywhere—for example, from the ASP.NET cache and session state—parse it, and store it in the model.

Note ASP.NET MVC comes with two built-in binders beyond the default one. These additional binders are automatically selected for use when posted data is a Base64 stream (ByteArrayModelBinder type) and when the content of a file is being uploaded (HttpPostedFileBaseModelBinder type).

Registering a Custom Binder
You can associate a model binder with its target type globally or locally. In the former case, any occurrence of model binding for the type will be resolved through the registered custom binder. In the latter case, you apply the binding to just one occurrence of one parameter in a controller method. Global association takes place in the global.asax file as follows:

```csharp
void Application_Start()
{
...
    ModelBinders.Binders[typeof(MyComplexTypeModelBinder)] =
        new MyCustomTypeModelBinder();
}
```

Local association requires the following syntax:

```csharp
public ActionResult RepeatText(
    [ModelBinder(typeof(MyComplexTypeModelBinder))] MyComplexType info)
{
...
}
```

Local binders always take precedence over globally defined binders.
As you can tell clearly from the preceding code within Application_Start, you can have multiple binders registered. You can also override the default binder if required:

```csharp
ModelBinders.Binders.DefaultBinder = new MyNewDefaultBinder();
```

Modifying the default binder, however, can have a large impact on the behavior of the application and should therefore be a very thoughtful choice.

### A Sample DateTime Model Binder

In input forms, it is quite common to have users enter a date. You can sometimes use a jQuery UI to let users pick dates from a graphical calendar. The selection is translated to a string and saved to a text box. When the form posts back, the date string is uploaded and the default binder attempts to parse it to a DateTime object.

In other situations, you might decide to split the date into three distinct text boxes—for day, month, and year. These pieces are uploaded as distinct values in the request. The result is that the default binder can manage them only separately—the burden of creating a valid DateTime object out of day, month, and year values is up to the controller. With a custom default binder, you can take this code out of the controller and still enjoy the pleasure of having the following signature for a controller method:

```csharp
public ActionResult MakeReservation(DateTime theDate)
```

Let's see how to arrange a more realistic example of a model binder.

### The Displayed Data

The sample view we consider next shows three text boxes for the items that make up a date and a submit button. You enter a date, and the system calculates how many days have elapsed since or how many days you have to wait for the specified day to arrive. Here's the Razor markup:

```csharp
@model DateEditorResponseViewModel
@section title{
    @Model.Title
}

@using (Html.BeginForm())
{
    <fieldset>
        <legend>Date Editor</legend>
        <div style="margin:20">
            <table>
                <tr>
                    <td>@DateHelpers.InputDate("theDate", Model.DefaultDate)</td>
                    <td><input type="submit" value="Find out more" /></td>
                </tr>
            </table>
        </div>
    </fieldset>
}
<hr />
@DateHelpers.Distance(Model.TimeToToday)
```
As you can see, I'm using a couple of custom helpers to better encapsulate the rendering of some view code. Here's how you render the date elements:

```csharp
@helper InputDate(String name, DateTime? theDate)
{
    String day = "", month = "", year = "";
    if(theDate.HasValue)
    {
        day = theDate.Value.Day.ToString();
        month = theDate.Value.Month.ToString();
        year = theDate.Value.Year.ToString();
    }
    <table cellpadding="0">
        <thead>
            <th>DD</th>
            <th>MM</th>
            <th>YYYY</th>
        </thead>
        <tr>
            <td><input type="text" name="@(name + ".day")" value="@day" style="width:30px" /></td>
            <td><input type="text" name="@(name + ".month")" value="@month" style="width:30px"></td>
            <td><input type="text" name="@(name + ".year")" value="@year" style="width:40px" /></td>
        </tr>
    </table>
}
```

Figure 3-7 shows the output.

![Figure 3-7](http://localhost5884/Date/Editor)

**FIGURE 3-7** A sample view that splits date input text into day-month-year elements.
The Controller Method

The view in Figure 3-7 is served and processed by the following controller methods:

```csharp
public class DateController : Controller
{
    [HttpGet]
    [ActionName("Editor")]
    public ActionResult EditorForGet()
    {
        var model = new DateEditorViewModel();
        return View(model);
    }

    [HttpPost]
    [ActionName("Editor")]
    public ActionResult EditorForPost(DateTime theDate)
    {
        var model = new DateEditorViewModel();
        if (theDate != default(DateTime))
        {
            model.DefaultDate = theDate;
            model.TimeToToday = DateTime.Today.Subtract(theDate);
        }
        return View(model);
    }
}
```

After the date is posted back, the controller action calculates the difference with the current day and stores the results back in the view model using a `TimeSpan` object. Here's the view model object:

```csharp
public class DateEditorViewModel : ViewModelBase
{
    public DateEditorViewModel()
    {
        DefaultDate = null;
        TimeToToday = null;
    }

    public DateTime? DefaultDate { get; set; }
    public TimeSpan? TimeToToday { get; set; }
}
```

What remains to be examined is the code that performs the trick of transforming three distinct values uploaded independently into one `DateTime` object.

Creating the `DateTime` Binder

The structure of the `DateTimeModelBinder` object is not much different from the skeleton I described earlier. It is just tailor-made for the `DateTime` type:

```csharp
public class DateModelBinder : IModelBinder
{
    public Object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)
    {
```

---

126 PART I ASP.NET MVC Fundamentals
if (bindingContext == null)
{
    throw new ArgumentNullException("bindingContext");
}

// This will return a DateTime object
var theDate = default(DateTime);

// Try to read from posted data. xxx.Day|xxx.Month|xxx.Year is assumed.
var day = FromPostedData<int>(bindingContext, "Day");
var month = FromPostedData<int>(bindingContext, "Month");
var year = FromPostedData<int>(bindingContext, "Year");

return CreateDateOrDefault(year, month, day, theDate);

// Helper routines
private static T FromPostedData<T>(ModelBindingContext context, String id)
{
    if (String.IsNullOrEmpty(id))
        return default(T);

    // Get the value from any of the input collections
    var key = String.Format("{0}.{1}", context.ModelName, id);
    var result = context.ValueProvider.GetValue(key);
    if (result == null && context.FallbackToEmptyPrefix)
    {
        // Try without prefix
        result = context.ValueProvider.GetValue(id);
        if (result == null)
            return default(T);
    }

    // Set the state of the model property resulting from value
    context.ModelState.SetModelValue(id, result);

    // Return the value converted (if possible) to the target type
    T valueToReturn = default(T);
    try
    {
        valueToReturn = (T)result.ConvertTo(typeof(T));
    }
    catch
    {
    }

    return valueToReturn;
}

private DateTime CreateDateOrDefault(Int32 year, Int32 month, Int32 day, DateTime? defaultDate)
{
    var theDate = defaultDate ?? default(DateTime);
    try
    {
        theDate = new DateTime(year, month, day);
    }
}
catch (ArgumentOutOfRangeException e)
{
}

return theDate;
}

The binder makes some assumptions about the naming convention of the three input elements. In particular, it requires that those elements be named *day*, *month*, and *year*—possibly prefixed by the model name. It is the support for the prefix that makes it possible to have multiple date input boxes in the same view without conflicts.

With this custom binder available, all you need to do is register it either globally or locally. Here’s how to make it work with just a specific controller method:

```csharp
[HttpPost]
[ActionName("Editor")]
public ActionResult EditorForPost([ModelBinder(typeof(DateModelBinder))] DateTime theDate)
{
    ...
}
```

Figure 3-8 shows the final page in action.

![Figure 3-8 Working with dates using a custom type binder.](image-url)
Summary

In ASP.NET MVC as well as in ASP.NET Web Forms, posted data arrives within an HTTP packet and is mapped to various collections on the Request object. To offer a nice service to developers, ASP.NET then attempts to expose that content in a more usable way.

In ASP.NET Web Forms, the content is parsed and passed on to server controls; in ASP.NET MVC, on the other hand, it is bound to parameters of the selected controller’s method. The process of binding posted values to parameters is known as model binding and occurs through a registered model-binder class. Model binders provide you with complete control over the deserialization of form-posted values into simple and complex types.

In functional terms, the use of the default binder is transparent to developers—no action is required on your end—and it keeps the controller code clean. By using model binders, including custom binders, you also keep your controller’s code free of dependencies on ASP.NET intrinsic objects, and thus make it cleaner and more testable.

The use of model binders is strictly related to posting and input forms. In the next chapter, we’ll discuss aspects of input forms, input modeling, and data validation.
Symbols and Numbers

/ (forward slash) character, 10
  route processing and, 13
@ expressions, 73–75
  for passing data to view, 78
  templated delegates, 86–88
~ (tilde) operator, 66, 216

A

about view, 82
about.aspx file, 48–49
Accept-Encoding header, 295–297, 401
acceptance tests, 348
AcceptVerbs attribute, 19, 303–304
access to action methods, restricting, 228–229
access tokens, 250
AccountController class, reworking, 232–233
action filters, 203, 281–282
  action method selectors, 303–304
  action name selectors, 302–303
  Ajax calls, restricting to, 305
  built-in, 292–293
  classification of, 291–292
  Controller class filter methods, 292
dynamically added, 307–312
embedded and external filters, 290–291
filter interfaces, 292
fluent code, 307–308
global, 206, 293
HandleError attribute, 205
interception points, 307
interfaces, 290, 292
for localization, 219
precedence, 293
response compression, 295–298
response header, 294–295
submit buttons, restricting to, 305–306
testing, 365–368
view model, filling, 301–302
view selector, 298–300
action helpers, 21, 98
action invokers, 18–19, 45–46
customizing, 308–309
default action invoker, 202–203
registering, 285, 309–310
replaceability of, 282
serving requests, 107
unit-test-specific, 367–368
action links, 152–153
Ajax helpers, 406–408
action method selectors, 303–304
action methods, 20–21. See also controller methods
  asynchronous, 36–40
  attaching behavior to, 291. See also action filters
  authorization, 229
code in, 263
coding, 254
ideal coding, 263–264
input data processing, 20–25
localization logic, 219
RDD stereotypes for, 255–257
Request objects, input data from, 22
restricting access, 228–229
results, 25–29
results generation, 21
roles of, 254–255
signatures of, 21
task performance, 21
action name selectors, 302–303
action names, 17
  binding to methods, 18–19
  validating, 302–303
action parameter, 18
action result types, 312–325
  binary data, 322–324
  built-in, 312–317
  ContentResult objects, 317
  custom types, 317–325
  EmptyResult objects, 317
  HttpStatusCodeResult class, 313–314
  JavaScriptResult class, 314–316
  JSON data, 316–317
  JSONP responses, 318–320
  PDF files, 324–325
  processing, 45
  syndication feed responses, 320–322
action selectors, 302
  action method selectors, 303–304
  action name selectors, 302–303
ActionFilterAttribute class, 298
  deriving from, 293
ActionLink method, 54–55, 152–153
ActionMethodSelectorAttribute class,
  IsValidForRequest method overrides, 305
ActionName attribute, 303
ActionResult class, 25, 27
  result types, 26
ActionResult objects, 21, 312
  ViewName property, 349
actions
  binding to methods, 19–20
  HTTP verbs, expressing with, 10, 19
ad hoc method selectors, 306
AddHeader filter, 294–295
  overriding, 368
address bar, URLs in, 139–140
aggregate roots, 269
Agile manifesto, 266
Ajax, 3
  ASP.NET MVC, JavaScript, and, 403–414
  jQuery model, 151–152
  JSON data, downloading via, 318–320
  Remember-me feature and, 237–239
Ajax applications, 373
  development, 403
  JavaScript code, organizing, 409–410
  JSON data providers, 408–409
  service layer, 403–405
Ajax calls, restrictions on, 305, 318
Ajax forms, 143–153
  callbacks, order of, 146
  controller methods, 147–148
  creating, 144–145
  hyperlinks, 152–153
  infrastructure, 145–147
  page title, updating, 148–150
  unobtrusive JavaScript model, 150–152
  validation, 177–178
Ajax helpers, 406–408
  sample, 61
Ajax requests, custom headers, 149
Ajax.BeginForm helper, 144
AjaxContext class members, 146
AjaxOptions class, 144
  members, 152
  onSuccess property, 149
  properties of, 144–146
anonymous functions, 374, 378–379, 394
AppFabric Caching services, 195, 198
application layer, in iPODD architecture, 266
application pools, worker processes, 119
application routes, 9–15. See also routes
  defining, 11–12
  physical files, requests for, 14
  processing, 12–13
  route handlers, 13–14
  routing, preventing, 14–15
applications. See also Ajax applications; ASP.NET
MVC; Web Forms
  coexistence of MVC and Web Forms, 3
  localizable, 220–226
  session state management, 193–194
  three-level architecture, 264
architecture
  iPODD pattern, 265–271
  three-level architecture, 264
argument exceptions from invalid values, 108–109
arrange, act, assert layout, 337–338
array of strings, posting, 112–113
Art of Unit Testing, The (Osherove), 348
ArtOfTest, 101
ASP.NET
  intrinsic objects, accessing, 71
  requests, serving, 4
  stateful programming model, 131
  URL routing module, 8
  virtual path providers, 64–65
ASP.NET 4 auto-encoding feature, 59
ASP.NET MVC
asynchronous actions, 35–36
binders, built-in, 123
custom controllers, 7, 15–40
custom components, 3
error handling, 200–212
error messages, displaying, 137–139
extensibility model, 281–289
HTML helper methods, 51
input processing, 105–106
intrinsic objects, 189–200
JavaScript, Ajax, and, 403–414
localization, 212–226
REST orientation, 10
runtime, simulating, 4–7
security, 227–231
server controls, 70–71
Server Locator pattern, 287–288
testing code, 345–368
transitioning to, 3–4
validation provider infrastructure, 168–169
view, recreating and repopulating, 135
workflow, 41
ASP.NET MVC 3, 4
dependency resolvers, 285
sample code, 232
view engines, 42–43
ASP.NET Web Forms. See Web Forms
ASPX pages, 27–28
ASPX view engine, 28. See also view engines
cache, 63
location formats, 63
precedence of, 48
server controls, 50
visual layout representation, 47
assertions per test, 343
async points, 34–35
Async suffix, 37
AsyncController class, 33
custom controllers derived from, 36
asynchronous action methods, 36–37
attributes, 39
candidates for, 39–40
coding, 38
invokers for, 38
naming, 37
asynchronous actions, 34–36
asynchronous controllers, 33–40
asynchronous methods, testing, 354–358
asynchronous pages, 33–34
AsyncManager class
  OutstandingOperations member, 38
Parameters dictionary, 38
AsyncTimeout attribute, 39, 292
ATOM 1.0 feeds, 320–322
attribute filters in jQuery, 388–389
attributes, for exception handling, 263–264
Authenticated method, 244–245
authentication, 228–229
configuring, 228
Forms authentication, 228–229
OpenID, 240–246
via Twitter, 246–251
Windows authentication, 231
authentication cookies
adding data, 245
creating, 243–244, 249
long-lasting, 237
authentication services, 240–251
IIS, 227
authorization, 228–229
for action methods, 229
output caching and, 229
Authorize attribute, 228–229
authorization capabilities, 229
extending, 229–231
output caching and, 229
Authorize filter, 292
AuthorizedOnly attribute, extending, 238–239
auto-adapting applications, 221
auto-encoding feature, 59
AutoFac, 275
AutoMapper, 269
automated input forms, 153–167
automated tests, 327. See also testability; testing framework for, 335
automatic parameter resolution, 21

B
back end, connecting with presentation layer, 264–279
back-office systems, 153
BeginForm helper, 52–53
BeginRouteForm helper, 52–53
behaviors
reusing, 294. See also action filters
separating from content, presentation, 394
separating from response generation, 3
binary data, returning, 322–324
Bind attribute, 119, 134–135
Exclude property, 120
Include property, 119–120
Prefix attribute, 120
bind/unbind functions, 390–391
BindAttribute class, 119
binding and unbinding, 390–391
live binding, 391
binding collections, 112–115
of complex types, 115–117
binding layer, 105–106
Boolean values, display template for, 157–158
browser caching, 401–402
Browser Template pattern, 406
browsers
closures and prototypes and, 382
go-location capabilities, 225
HTTP compression, 295–298
nonblocking scripts download, 400–401
synchronous download, 399–400
view selectors, 298–300
BuildManagerViewEngine class, 64
business-layer validation, 180
ByteArrayModelBinder type, 123
calls
attaching functions, 390–391
mapping between ASP.NET MVC and jQuery, 152
unobtrusive JavaScript and, 151–152
canonical URL format, 192
Cascading Style Sheets (CSS), 393–394
Castle Windsor, 275
catch-all parameters, 14–15
catch-all routes, 210–211
definition, 211
catch blocks, 202
CDN, 145
centralized validation, 181–182
CheckIfUserIsAuthenticated method, 239
child actions, 99
input data from, 24
marking, 200
child filters in jQuery, 387
ChildActionOnly attribute, 99, 292
class-level validation, 181–182
ClassCleanup attribute, 337
classes. See also controller classes
dependency inversion, 271–272
partial classes, 343
Service Locator based, 286–287
test fixtures, 336–337
testing in isolation, 340–341
ClassInitialize attribute, 337
client-side features, 373. See also JavaScript; jQuery
client-side validation, 167–185
culture based, 178–179
custom attributes, 182–183
closures, 380–381
code. See also design, software; design principles;
software
“Don’t Repeat Yourself (DRY)” principle, 86
fluent code, 307–308
packaging, 397–399
readability, 95
code blocks, 66–68
Code Contracts, 185, 263–264
code coverage of testing, 344–345
code injection, 274
code nuggets, 73–75
special expressions, 74–75
code reuse, HTML helpers and, 70
collections, binding, 112–117
comments, in Razor expressions, 75
Compare attribute, 175
complex types
   acceptable properties, 119–120
   binding, 111–112, 122
   binding collections of, 115–117
Compress attribute, 296
   writing, 297
compression
   of responses, 295–298
   of static resources, 401
compression filters, 295–298
configuration files, validation attributes, 180
confirmation messages, for reposting input forms, 139
consumer key/consumer secret pair, 246, 249
containers, 21
   classes, 111
   global containers, 395–397
   IoC frameworks, 275
   passing data to view with, 90
content
   separating from presentation, behavior, 394
   URLs, synchronization with, 139–140
content delivery networks (CDNs), 401–402
Content-Encoding header, 295–297
content files, referencing, 216
content filters in jQuery, 388
Content folder, grouping resources under, 216
ContentResult class, 26
ContentResult objects, 29, 317
context objects, lifetime of, 271
control of software, 329
Controller class, 290
   filter methods, 292
   ModelState dictionary, 137
   RouteData property, 22
   ValueProvider property, 23
controller classes
   Authorize attribute, adding, 228–229
   creating, 254
   global action filters, 293
   HandleError attribute, 205–206
   handling exceptions directly, 201–202
   method overrides in, 290–291
   OnException method, overriding, 202–204
   response headers, 290
   worker services, injecting, 276–277
controller factory, 288
   controlling, 277–279
   custom, 277–278
registering, 277, 285
   replaceability of, 282
   Unity based, 278–279
controller methods. See also action methods
   Ajax aware, 147–148
   input parameters, 22
   renaming, 303
   render actions, 194
   testing, 365–368
   view component selection, 9
controller names, 17–18
Controller RDD stereotype, 255–256
controllers, 7, 15–29, 41
   action methods, 21–25, 291. See also action filters
   adding aspects, 290–312
   aspects of, 15–16
   asynchronous, 33–40
   caching, injecting, 195–197
   capabilities of, 29–40
   collections, passing to, 112–115
   coordinators, 133
   custom base class for, 309–310
   data exposed to, 103
   design considerations, 253–264
   granularity, 15
   grouping, 29–33
   as infrastructure, 263
   input data to, 104
   intrinsic objects, accessing, 71
   JavaScript, 409–410
   membership controllers, 232–237
   method signature, 107
   number of, 15
   in presentation layer, 16
   presentation layer and back end, connecting, 264–279
   render actions, 98–99
   role of, 253
   session state, accessing, 194
   stateless components, 15
   testability, 16
   testing, 99
   view, rebuilding, 301
   view, separation from, 16, 194
   and view engines, interaction, 42
   worker services and, 259–262
   writing, 17–21
Convention-over-Configuration pattern, 96, 119
cookies

mocking, 361
persistent authentication cookies, 237
value provider and, 24
Coordinator stereotype, 133, 255–257, 264
coordinators, 133
coupling
loosely coupled systems, 286
low coupling principle, 330
testability and, 333
tight coupling, 330–331
CPU-bound operations, 39
CreateAuthCookie method, 249
CreateMetadata method, overriding, 161–162
CreateRequest method, 243
cross-property annotations, 173–175
CRUD (Create, Read, Update, Delete) applications, 256, 347
.cshtml file, for declarative helpers, 83
CSS, 393–394
CSS-based selectors, 385
culture
auto-adapting, 221
changing programmatically, 222–225
detecting, 215
getting and setting, 221–222
neutral, 221
culture-based client-side validation, 178–179
culture-driven view engines, 283–284
Culture property, 221
culture script files, 179
CultureAttribute class, 224
CultureAttribute filter, 298
CurrentCulture property, 222
CurrentUICulture property, 222
custom components, registering, 285
custom errors, enabling, 205
custom headers, for Ajax requests, 149
custom HTML helpers, 59–61
custom model binders, 121–124
custom view engines, 96–98
CustomerModelBase class, 301
CustomValidation attribute, 174–175
vs. custom validation attributes, 175–176
Cutting Edge column in MSDN Magazine, 276

data
bringing to client, 408–409
caching, 194–200
displaying or editing, 154
drop-down lists, populating, 133
data access, testing, 353–354
data access layer
in iPODD architecture, 266
testing, 348
data annotations, 154–155
advanced, 173–180
cross-property annotations, 173–175
for read-only members, 161
Remote attribute, 179–180
self-validation and, 181
specifying, 180
using, 168–173
for validation, 169
validation attributes, custom, 175–176
vs. VAB, 183–184
data-driven tests, 339
data entry, general patterns, 132–153
data layer, in iPODD architecture, 269–271
data tables
building with code blocks, 66–68
building with HTML helpers, 68–69
data-transfer objects (DTOs), 269
data types
custom display and edit templates for, 158–160
default display and edit templates for, 156–158
edit and display modes, 156
validating, 160
data validation. See validation
DataAnnotationsModelMetadataProvider class, 161
DataAnnotationsModelValidatorProvider class, 168–169
databases, storing localizable resources in, 225–226
DataSet design, 268
DataSource attribute, 339
DataType attribute, 160
date objects, 261
rendering, 125
DateTime model binder, 124–128
controller method, 126
DateTimeModelBinder object, 126–128
DDD methodology, 274
debugging vs. testing, 328
decision coverage of testing, 344
declarative HTML helpers, 83–84
limitations of, 84–86
DefaultActionInvoker class, 282
DefaultControllerFactory class, 282
overrideable methods, 278
DefaultModelBinder class, 107–108, 112–113
inheriting from, 121
DefaultViewLocationCache class, 63
defer attribute, 400
delegates, templated, 86–88
delimiters, 10
dependencies
abstracting, 345
grouping, 274
resolving, 275–276
resolving with Service Locator, 287
testing and, 352–358
Dependency Injection (DI), 257, 264, 274–275, 332
Inversion of Control frameworks, 275–276
vs. Service Locator pattern, 286–287
Dependency Inversion Principle (DIP), 257, 271–272, 286, 332
Dependency Injection pattern, 274–275
Service Locator pattern, 272–273
dependency resolvers
defining, 288–289
registering, 289
services, locating with, 162
design, software
collaboration, 266
considerations, 253–264
interface-based programming, 330–332
iPODD pattern, 265–271
testability and, 328–334
Design for Testability (DfT), 328–330
design principles
code comments, 258
Dependency Inversion Principle, 257, 271–275, 286
Design for Testability, 328–330
“Do not put it in the controller” principle, 346
“Don’t Repeat Yourself (DRY)” principle, 86
Interface Segregation principle, 236
low coupling, 330–334
principle of separation of concerns (SoC), 16
short methods, 258
for worker service classes, 258–262
YAGNI principle (You Aren’t Gonna’ Need It), 196
desktop browsers, 298
development of software, SOLID acronym, 332
DfT, 328–330
DI. See Dependency Injection (DI)
dictionary values
registering, 285
replaceability of, 282
DIP. See Dependency Inversion Principle (DIP)
display, separating from request processing, 40
Display helpers, 56–57, 154–156, 158–160
display templates
custom, 158–160
for data types, 156–158
naming, 159–160
nested models, 166–167
in Razor, 163
tabular, 163–165
DisplayForModel, 58
distributed caching, 198–199
DLR, 91
DNOA library, 241, 247, 249–250
Domain-Driven Design (DDD) methodology, 274
domain entities, 103
domain layer
class visibility, 269
in iPODD architecture, 268–269
testing, 347
domain models, 103, 106
definition of, 268
Entity Framework model, 132
domain services, 268–269
domains, transient vs. persistent, 269
“Don’t Repeat Yourself (DRY)” principle, 86
DotNetOpenAuth (DNOA) library, 241, 247
token manager objects, 249–250
double encoding, 59–60
drop-down lists, populating, 133–134
Dynamic Language Runtime (DLR), 91
dynamic objects, 57, 91–92, 95–96
for internal data, 95–96
E
each function, 384
ECMAScript 5, 374
Edit actions
POST and GET actions, splitting, 140–141
requests for, 134
Edit-and-Post pattern, 132
editable fields, validation helpers, 137
Editor helpers

*Editor helpers*, 56–58, 154–156
- customizing, 158–160

**editor templates**
- custom, 158–160
- for data types, 156–158
- naming, 159–160
- in Razor, 163
- tabular, 164–165

*EditorForModel*, 58

**ELMAH**, 208

**embedded files**, referencing, 217–219
- Embedded Resource build action, 213–214, 218
- embedded resources, 213–214

*EmptyResult* class, 26

*EmptyResult* objects, 317

**encapsulation**, closures, 380–381

**encoding values**, 297

*Engines* member, 43

**Enterprise Library Validation Application Block**, 180

**Entity Framework model**, 132, 268

**entity model**
- creating, 268
- definition of, 268

**enumerated types**, validating, 171–172

*Error controller*, 210

*Error event*, 206–207

**error handling**, 200–212
- with attributes, 263–264
- catch-all routes, 210–211
- email messages for, 207

Error Logging Modules And Handlers, 208

**global error handling**, 206–209
- from *global.asax* file, 206–207
- with HTTP modules, 207–208

*IIS error handling*, bypassing, 211–212

**landing page**, 207

**missing content**, 209–212

**program exceptions**, 201–206

**route exceptions**, 209

Error Logging Modules And Handlers (ELMAH), 208

**error messages**
- capturing, 175
- displaying, 137–139
- for input validation errors, 172–173

**error pages**, IIS level, 211–212

**error view**, switching to, 204

**event handling**, 70

*Exception* class, exception classes deriving from, 202

exception classes, deriving from *Exception*, 202

**exception handling**, 202

**exception types, built-in**, 202

**exceptions**
- alerting administrator about, 207
- handling directly, 201–202
- handling with attributes, 263–264
- model binding, 208
- route exceptions, 209

*Execute method*, 77

*ExecuteResult method*, 46, 313

*ExecuteResult* objects, 323

**expando objects**, 95–96

**ExpandoObject** type, 96

**extensibility models**, 281–289, 327
- provider-based model, 282–286
- Service Locator model, 286–289

*TempData* storage, alternate, 284–285

**view engine**, culture-driven, 283–284

**extensibility points**, 281
- replaceable components, 282–283

**extension methods**, 88

F

*F5 keypresses*, 139–140

**factories**, 274

**factory classes**
- globally accessible, 285
- registering, 285

**fake objects**, 341–342, 352
- humble objects, 358

feedback, passing to view, 141–142

Felker, Donn, blog, 25

file content, replying with, 26

*FileContentResult* class, 26

*FileContentResult* objects, 323

*FilePathResult* class, 26

*FilePathResult* objects, 323–324

*FileResult* class, 323

**files**
- creating on web server, 118
- localizable, 216–217

*FileStreamResult* class, 26

*FileStreamResult* objects, 323

*Filter class*, 311

*filter function*, 389

filter methods in *Controller* class, 292

filter providers, 310–312
FilterAttribute class  
  inheriting from, 293  
Order property, 293
FilterInfo class, 307 
filters in JQuery, 387–389 
FilterScope enumeration, 311 
finalizer methods, 34, 36  
coding, 38
find function, 389
FindPartialView method, 44
FindView method, 44
Flickr, 240 
fluent code, 307–308
form data, input data from, 24
<form> tag, 53
formatted text  
  displaying, 88  
  templated delegates and, 88–89
forms. See also Web Forms  
  Ajax-based, 143–153  
  posting changes, 136–137
Forms authentication, 228–229
friendly names, 243–245
FriendlyIdentifierForDisplay property, 243
front-end test tools, 99–101
function coverage of testing, 344
functions, in JavaScript, 378–379

G

global.asax file  
  action filters, registering, 310  
  Application_Start, 206  
  custom view engine entry, 97–98  
  error handling from, 206–207  
  routes registered in, 11
GlobalFilters collection, 293
  globalization, validation and, 178–179
Google, 240
  OpenID URL, 241
Google Gears, 225
GZIP compression, 401

H

HandleError attribute, 205–206, 292–293
helper classes, 94  
  proliferation, 95
@helper keyword, 84
HiddenInput attribute, 167
hiding user interface elements, 231
hoisting, 377
HTML  
  form tag, 53  
  composing, 27–28
HTML 5, 373
  new tags, 394
HTML content, replying with, 26
HTML elements, rendering, 54
HTML encoding, 86
HTML forms, rendering, 52–53
HTML helpers, 50–61, 68–70, 131
  action links, 54–55
  Ajax helpers, 61, 406
  custom helpers, 59–61
  declarative, 83–86
  for default text, 60
  editable forms, building, 153–154
  HTML elements, rendering, 54
  HTML forms, rendering, 52–53
  HtmlHelper class, 55–56
  metadata use, 154
  MvcHtmlString wrapper objects, 59–60
  overloads, 52
  partial views, rendering, 55
  stock set of methods, 51
  structure of, 59
  templated HTML helpers, 56–58
  templates, acceptance of, 87
  validation helpers, 137
HTML layouts, matching to view names

HTML layouts, matching to view names, 47
HTML-level code reuse, 51
HTML markup, returning, 27–28
HTML Message pattern, 143, 406, 408
Html property, 52
HTML responses
  generation of, 41
  view engines and, 42
Html.ActionLink helper, 18
HtmlHelper class, 51–55–56
  native methods, 55–56
HtmlHelper type, 85
Html.Pager helper, 69
Html.Raw helper method, 75
Html.SimpleGrid helper, 69
HTTP codes
  custom pages or routes for, 209
  HTTP 301 code, 190–191
  HTTP 302 code, 190
  HTTP 401 code, 231
  HTTP 403 code, 313–314
  HTTP 404 code, 8, 14, 209, 211, 313–314
HTTP compression, 295–298
HTTP context, mocking, 350, 358–368
HTTP endpoints, calling, 408
HTTP GETs, 19
HTTP handlers
  asynchronous, 33
  behavior, defining, 5–6
  invoking, 6–7
  URLs, binding to, 4
HTTP modules, global error handling with, 207–208
HTTP POSTs, 19
HTTP verbs
  accepting, 304
  actions, expressing with, 10, 19
HttpApplication objects Error event, 206–207
HttpContext objects, 190
  mocking, 359
HttpContextBase class Cache property, 363–364
HttpGet attribute, 113
HttpMethodOverride method, 53
 HttpNotFoundResult class, 26
HttpPost attribute, 113
HttpPostFileBase type, 117
HttpPostFileBaseModelBinder type, 123
HttpRequest objects, 190
HttpResponse class RedirectPermanent method, 191
HttpResponse objects, 190–193
HttpSessionState class, 193–194
HttpResponse objects, 190–193
HttpSessionState class, 190
HttpStatusCodeResult class, 313–314
HttpUnauthorizedResult class, 26
HttpVerbs enum type, 19
Humble Object pattern, 355–358
hyperlinks in Ajax forms, 152–153
I/O-bound operations, 39–40
I/O completion ports, 35
IActionFilter interface, 290, 292
IAsyncResult objects, 34–35
IAuthorizationFilter interface, 292
ICacheService interface, 196–197
ICacheService objects, 196
IClientValidatable interface, 178, 182–183
IClientValidatable interface, 178, 182–183
IControllerFactory interface, 277
ID selectors, 385
Idiomatic Presentation, Orchestration, Domain and Data pattern. See iPODD (Idiomatic Presentation, Orchestration, Domain and Data) pattern
IExceptionFilter interface, 292
Ignore attribute, 338
IHttpString interface, 60
IHttpHandler interface, 5
IIS
  authentication services, 227
  error handling, bypassing, 211–212
  integrated mode, 227
  worker process identities, 119
images, optimizing, 402–403
immediate functions, 398
IModelBinder class, 282
IModelBinder interface, 107, 121
  definition of, 122
IModelBinderProvider class, 282
inbound links, case sensitivity, 192
inheritance, 310
  with prototypes, 382
@inherits declaration, 78
initialization code, 391–392
injectable dependencies, 393–394. See also unobtrusive JavaScript
injection
  caching service, 195–198
  code injection, 274
injection (continued)
of dependencies, 264. See also Dependency Injection (DI)
of malicious code, 227
inline code, 66
inline expressions, 66
input data
controller access to, 105
editing, 134–136
invalid, 138
manual retrieval of, 21–25
from multiple sources, 23
passing to processing layer, 133
processing, 21–25
from Request object, 22
from routes, 22–23
saving, 136–139
sources of, 24
INPUT elements, multiple, 118
input forms
Ajax-based, 143–153
automating writing of, 153–167
data entry patterns, 132–153
Post-Redirect-Get pattern, 139–143
repost confirmation message, 139
script files for, 178
Select-Edit-Post pattern, 132–139
validation, 54, 167–185
input model, 103–106
model binders, 105–106
input parameters, source of, 20
input processing
ASP.NET MVC, 105–106
Web Forms, 104–105
input validation, 167–185
for Ajax forms, 177–178
centralized validation, 181–182
client-side validation, 177–178
Code Contracts, 185
cross-property annotations, 173–175
culture-based client-side validation, 178–179
data annotations, 168–180
dynamic, 183–185
enumerated types, 171–172
error messages, 172–173
levels of, 167–168
model classes, decorating, 170–171
remote, 179–180
self-validation, 180–185
validation attributes, custom, 175–176
validation provider infrastructure, 168–169
integration tests, 333
interception points for action filters, 307
interface-based programming, 330
Interface Segregation principle, 236, 332
interfaces for action filters, 292
internal components, replacement of, 281–289
InternalsVisibleTo attribute, 343
Internet Information Services. See IIS
intrinsic objects, 189–200
Cache object, 194–200
HttpResponse object, 190–193
Session object, 193–194
Inversion of Control (IoC) frameworks, 275–276, 281
configuration data, 279
dependency resolvers in, 288–289
InvokeActionMethodWithFilters method, 307
overriding, 307–308
invokers for asynchronous methods, 38
IoC. See Inversion of Control (IoC) frameworks
iPODD (Idiomatic Presentation, Orchestration, Domain and Data) pattern, 265–271
data layer, 269–271
domain layer, 268–269
orchestration layer, 267
presentation layer, 266–267
IResultFilter interface, 292
IRouteHandler interface, 13
IsValidForRequest method, overriding, 305
@item, 87
Item indexer, 22
ITempDataProvider class, 282
iTextSharp library, 324
IValidatableObject interface, 181
IValueProvider class, 282
IView interface, 46
IViewEngine class, 282
IViewEngine interface, 42
members, 44
methods, 44

J

JavaScript
in Ajax applications, organizing, 409–410
anonymous functions, 374
ASP.NET MVC, Ajax, and, 403–414
basics, 374–379
closures, 380–381
JavaScript (continued)
custom objects, 382
functions, 378–379
global self-contained functions, 410
hoisting, 377
for HTML 5, 373
immediate functions, 398
jQuery library, 383–392
libraries, 298
local and global variables, 375–376
Module pattern, 398–399
Namespace pattern, 397–398
null vs. undefined, 375
object orientation, 379–382
objects, 377–378
properties, grouping in containers, 395–397
prototypes, 381–382
reusable packages and dependencies, 396–399
script and resource loading, 399–403
static analysis of, 376
type system, 374–375
unobtrusive, 150–152
unobtrusive code, 392–396
updaters and controllers, 409–413
uses, 374
var keyword, 376
JavaScript Patterns (Stefanov), 398
JavaScriptResult class, 26–27, 314–316
JavaScriptSerializer class, 316
JetBrains ReShaper, 92
jQuery, 151–152, 383–392
bind/unbind functions, 390–391
constraints of, 396
DOM queries, 383–385
DOM readiness, 391–392
events, 390–392
filters, 387–389
getJSON function, 320
global namespace, pollution of, 409–410
globalization plug-in, 179
HTTP endpoints, calling, 408
jQuery function, 383–384
library structure, 383
live binding, 391
page readiness, 391–392
plug-ins, 394–396, 409
queries, running, 384
selectors, 385–390
Template plug-in, 412
wrapped sets, 383–385
jQuery function, 383–384
jquery.unobtrusive-ajax.js file, 151, 153
JSLint, 376
JSON-based solutions, 413–414
JSON calls, 316
JSON content, returning, 28–29
JSON data
Ajax, downloading via, 318–320
returning, 316–317
JSON data providers, 408–409
Json method, 21
JSONP protocol (JSON with Padding), 318–320
JsonpResult class, 318–319
JsonResult class, 26, 316
L
LABjs library, 400
lambda expressions, 57
Language attribute, 67
languages, switching on the fly, 224
layer, definition of, 264
layered code, 16
layout pages, 79
data, sharing with, 88–89
default content, 81
nesting, 82–83
sections, 80–81
templated delegates and, 87–88
Layout property, 79
layout templates
code, flowing into, 80
requesting, 82
legacy URLs, 193
lengthy operations, 33–36
asynchronous methods for, 39–40
links, route-based, 17–18
Liskov’s Substitution Principle, 332
live binding in jQuery, 391
local binders, 123, 128
local variables, 375–376
localizable applications, 220–226
localizable resources, 212–220
global file for, 214–215
localizable files, 216–217
localizable text, 214–216
localizable views, 219–220
storing in database, 225–226
localization, 212–226
   auto-adapting applications, 221
culture, changing programmatically, 222–225
custom view engine, 283–284
global filters for, 222–223
localizable applications, 220–226
localizable resources, 212–220
multilingual applications, 221–222
   with resource files, 172–173
testing, 349–351
localized views, 283
location formats, 63
locators, calling, 273
login and logout, 232–233
LogOff method, 233
LogOn method, 233
LogOnViewModel class, 234
loosely coupled systems, 286
low coupling principle, 330

M
magic strings, 91
maintainability of software, 327, 330
malicious code, preventing injection, 227
MapRoute method, 11, 36
markup
   control over, 3
   HTML encoding, 88
   wrapping in HTML tags, 74
Martin, Robert, 271
@Master directive, 65
master views, 50
   defining in ASPX, 65–66
   defining in Razor, 78–79
McKean, Alan, 254
mediators in testing, 355–358
membership API, integrating with, 234–235
Membership class, 234
membership controllers, defining, 232–237
membership providers, creating, 234–235
membership system
   implementing, 232–239
   membership controller, defining, 232–237
   Role API, 236–237
   SimpleMembership API, 236
   user credentials, validating, 233–234
Memcached, 195
Memo class, 170–171

MemoryCache class, 198
MemoryCache objects, 198
metadata
   consumption of, 154–155
   data annotations, 154
   processing, 57
   reading, 158
metadata providers, 154
   for read-only members, 161–162
   registering, 162
method responses
   caching, 199–200
   compressing, 295–298
method selectors, 303–304, 306
method signature, controller, 107
methods
   Ajax calls, restricting to, 305
   duplicate names, 20
   HTTP verbs, associating, 19
   validating, 303–304
Microsoft content delivery network (CDN), 145
Microsoft Enterprise Library, 183–184
Microsoft IntelliSense, 382
Microsoft Internet Information Services. See IIS
Microsoft Moles, 334, 364
Microsoft .NET: Architecting Applications for the
   Enterprise (Esposito and Saltarello), 106, 168, 332
Microsoft Office, creating PDFs with, 325
Microsoft Office automation, 325
Microsoft Passport, 240
Microsoft Visual Studio. See Visual Studio
Microsoft Visual Studio Team Foundation Server
   custom policies, 330
   MicrosoftAjax.js file, 145, 153, 177
   MicrosoftMvcAjax.js file, 145, 153, 177
minification, 401
missing content, handling, 209–212
missing resources, 26
mobile browsers, 298
mock objects, 341–342, 352
   dynamically generated, 361
mocking
   Cache object, 363–365
   HTTP context, 350, 358–368
   HttpContext object, 359
   Request object, 359–360
   Response object, 360–361
   Session object, 362–363
mocking frameworks, 353–354
model binder, default

model binder, default, 108–119
  binding collections, 112–117
  complex types, 111–112
  customizable aspects, 119–120
  customizing, 121–122, 124
  fixed IDs, 115
  optional values, 109–110
  prefixes, 120
  primitive types, 108–109
  validation, 168
  value providers used, 110
model-binder classes, 107
model binder provider
  registering, 285
  replaceability of, 282
model binders, 22
  implementing, 122–123
  registering, 123–124, 285
  replaceability of, 282
  role of, 105–106
  sample binder, 124–128
model binding, 107–120
  advanced model binding, 120–128
  `Bind` attribute, 119
  binder, getting, 107–108
  binding collections, 112–117
  complex types, 111–112
  custom binders, 121–124
  default model binder, 108–120
  exceptions, intercepting, 208
  infrastructure, 107
  method signature, analyzing, 107
  nullness, checking for, 139
  optional values, 109–110
  primitive types, 108–109
  progressive and static indexes, 116–117
  properties, excluding, 120
  properties, including, 119–120
  sample custom binder, 124–128
  uploaded files, 117–119
model binding layer, 20
model metadata
  registering, 285
  replaceability of, 282
`Model` property, 78–79
model validator
  registering, 285
  replaceability of, 282
Model-View-Controller pattern, 3

`ModelMetadataProvider` class, 282
`ModelMetadataProvider` type, 162
models, types of, 106
`Models` folder, 103, 106
`ModelState` dictionary, 137–139
  editing, 172–173
  feedback, adding, 142
  invalid values in, 169
`ModelValidatorProvider` class, 282
Module pattern, 398–399
Moq, 342
`MSDN Magazine`, 276
MSTest, 335–336, 344
  `TestContext` variable, 339
multilingual applications, 221–222
multiple file uploads, 118
multiserving applications, 298
`Mvc Futures`, 284–285
`MvcHtmlString` wrapper objects, 59–60
myOpenID, 242

N

name/value pairs, 377
`namespace` function, 397–398
Namespace pattern, 397–398
namespaces
  data annotations in, 154
  importing, 80
NCache, 195
nested layouts, 82–83
nested models, 166–167
.NET objects, serializing, 316
NETWORK SERVICE account, 118
`new` keyword, 379–380
NHibernate, 268
Ninject, 275
NMock2, 342
`NoAsyncTimeout` attribute, 39
`NonAction` attribute, 19, 303–304
nonpublic members, testing, 343–344
nonvirtual methods, testing, 334
`nth-child` filter, 388
`null` values, 375
nullness, checking for, 24, 139
NUnit, 335–336
O

OAuth
  applications, registering, 246–247
  functionality of, 250
  vs. OpenID, 246, 251
Object Design: Roles, Responsibilities, and Collaborations (Wirfs-Brock and McKean), 254
object model, definition of, 268
object orientation
  closures, 380–381
  in JavaScript, 379–382
  prototypes, 381–382
  testability and, 334
Object/Relational Mapper (O/RM) tools, 268
object stereotypes, 254–255
Object type, 380
Object.aspx template, 158
objects
  editing, 57–58
  in JavaScript, 377–378
Office automation, 325
OnActionExecuted method, 290, 292
OnActionExecuting method, 224, 290, 292
OnAuthorization method, 292
  overriding, 230
OnBegin callbacks, 152
OnComplete callbacks, 152
onEditCompleted JavaScript function, 149
OnException method, 292
  implementing, 204
  overriding, 202–204
onload event, 391–392
OnModelUpdated method, overriding, 121
OnPropertyValidating method, 121–122
OnResultExcepted method, 292
OnResultExcepting method, 292
OnResultExecuting method, 298
OnSuccess callback, 152
Open/Closed Principle, 152
OpenID, 240–246
  OpenID URL, 241–242
  sample application for connecting to providers, 241–242
  vs. OAuth, 246, 251
OpenID providers, 240
OpenIdRelyingParty type, 243
OperationCounter class, 38

O

operations
  lengthy operations, 33–36
  termination of, 35
  optimization, 401–402
  orchestration, chunky vs. chatty, 267
  orchestration layer, 267
  testing, 347
Order property, 293
Osherove, Roy, 348
output caching
  authorization and, 229
  partial, 200
OutputCache attribute, 199–200, 293
  properties of, 199
OutstandingOperations member, 38

P

page life cycle, 104–105
page-loading performance, 399–400
page titles, updating in Ajax forms, 148–150
PageData dictionary, sharing with layouts and views, 88–89
paging data, 69–70
parameters
  catch-all parameters, 14–15
  prefixes, 120
Parameters dictionary, 38
partial classes, 343
Partial helper, 55, 136
Partial method, extending, 219–220
partial output caching, 200
partial rendering, 143, 147–148. See also Ajax forms
  partial views
    for property types, 57
    rendering, 55
    resolving, 58, 284
PartialViewResult class, 26
Passport authentication, 227
passwords
  storing, 235
  validating, 235–236
path coverage of testing, 344
patterns
  Browser Template pattern, 406
  of data entry, 132–153
  Dependency Injection pattern, 274–275
  HTML Message pattern, 406, 408
  Humble Object pattern, 355–358

429
patterns (continued)
iPODD pattern, 265–271
Module pattern, 398–399
Namespace pattern, 397–398
Post-Redirect-Get pattern, 139–143
Select-Edit-Post pattern, 132–139
Service Locator pattern, 272–273
unobtrusive JavaScript, 150–152

PDF files
creating, 324–325
returning, 324–325
per-status code views, 209
performance
closures and prototypes and, 382
databases and, 226
permanent redirection, 190–191
persistent cookies, 237
persistent entities, 269
Pex add-in, 345
physical files
request handling, 14
requests for, 14
placeholders in URLs, 10
plain-old CLR (POCO) classes, 268
positional filters in jQuery, 387
POST actions
and GET actions, splitting, 140–141
updating via, 141
Post-Redirect-Get pattern, 139–143
POST actions, updating via, 141
POST and GET actions, splitting, 140–141
POST requests, terminating, 140
redirects, saving data across, 141–143
POST requests, termination of, 140
postbacks
server controls and, 70
state, persisting, 131
URL for, 137
posted files, input data from, 24
posting changes, 136–137
PostResolveRequestCache event, 9
precedence of request collections, 110
preconditions, determining with Code Contracts, 263–264
prefixes, 120, 128
presentation, separating from content, behavior, 394
presentation layer
and back end, connecting, 264–279
controller in, 40
idiomatic, 266
in iPODD architecture, 266–267
validation, 168–183
primitive types, 374
binding, 108–109
principle of separation of concerns (SoC), 16
PrivateObject class, 344
productivity, 327
program exceptions, handling, 201–206
progressive indexes, 116
properties
remote validation, 179–180
whitelists and blacklists of, 119–120
prototype property, 377
prototypes, 381–382
provider-based extensibility model, 282–286
extensibility points, 282–283

Q
queries, selectors, 385–390
query strings, input data from, 24

R
Rapid Application Development (RAD), 327
raw HTTP request data
modeling of, 105–106
Web Forms processing, 104–105
Razor view engine, 28, 72–89. See also view engines
@ expressions, 73–75
C#-based vs. Visual Basic-based views, 75–76
code nuggets, 73–75
custom base class, 77–78
custom templates, 163
DateTime binder markup, 124
declarative HTML helpers, 83–86
default content placeholders, 81
layout pages, 79
precedence of, 48
sample view, 78–86
search locations, 72–73
view object, 76–77
WebMatrix part, 47
RDD, 254
concepts and terms, 255
Controller stereotype, 255–256
Coordinator stereotype, 133, 255–257
stereotypes, 255
read-only templates, 161–162
Readability of software, 330
ReadOnly attribute, 154, 161–162
ready event, 392
readyState property, 392
redirection
  permanent, 190–191
  saving data across, 141–143
  testing, 351
RedirectPermanent method, 191
RedirectResult class, 26, 351
RedirectResult type Permanent, 191
RedirectToRouteResult class, 26, 351
refactoring, 70
reflection, 58
Register actions, 41
register.aspx pages, 41
RegisterCacheService method, 198
RegisterRoutes method, 11, 351
RelyingParty property, 243
Remember-me feature, Ajax and, 237–239
Remote attribute, 179–180
remote validation, 179–180
render action methods, 68
render actions, 98–99, 194, 301
RenderAction helper, 98–99
RenderBody method, 80
RenderPage method, 81
RenderPartial helper, 55, 136
RenderSection method, 80–81
  overloading, 87–88
replaceable components, 283–289
  registering, 285
replacement of internal components, 281–289
repositories, injecting worker services, 276–277
repository classes, 269–270
  context objects, lifetime of, 271
  methods in, 270
  structure of, 270
Representational State Transfer (REST), 9–10
Request collections, 105
request collections, precedence, 110
Request objects
  input data from, 22
  mocking, 359–360
request processing
  asynchronous, 33–37
  separating from display, 40
  sequence of, 45
  synchronous, 36–37
  by URL routing HTTP module, 7–9
request tampering, 119
RequestContext class, 13
Request.Params dictionary, 22
requests
  asynchronous, 33–36
  culture, setting, 222–223
  execution, 255
  handling via routing, 14
  length of, 118
  for physical files, 14
  resolving, 17, 107
  results of, 3
  routing, 4–15
  user placement of, 3
RequireHttps filter, 293
RequireJS library, 400
requirement churn, 327
ReSharper (JetBrains), 92, 335
resource files, multiple, 215
resource manager culture detection, 215
ResourceManager class, 217
resources
  Content folder, grouping under, 216
  culture-specific, 215
  definition of, 213
  embedded, 213–214, 217–219
  for error messages, 172–173
  global vs. local, 212
  identification of, 4
  localizable, 212–220
  localizable text, 214–216
  missing, 26
  public and internal, 213
  storing in database, 225–226
Resources folder, multiple RESX files, 215
response generation, separating from behavior, 3
response headers
  for Ajax requests, 149
  custom, 290–291, 294–295
Response objects
  mocking, 360–361
  TrySkipIisCustomErrors property, 211
response stream
  customizing, 190
  generating, 46
responses
  binary data, 322–324
  compressing, 295–298
  JavaScript, 314–316
  JSON data, 316–317
responses (continued)
- JSONP responses, 319
- PDF files, 324–325
- status code and description, 313
- syndication feed responses, 320–322

Responsibility-Driven Design (RDD). See RDD

REST, 9–10

result filters, 298

results, action, 25–29
- HTML markup, 27–28
- JSON content, 28–29
- mechanics of, 27
- predefined types, 26

RESX files
- multiple, 215
- role of, 213–214

reusable packages, 396–399

Rhino Mocks, 342

Role API, 236–237

role providers, custom, 236–237

RoleProvider class, 236

roles
- definition of, 236
- validating, 237

round brackets in Razor expressions, 75

Route class, 55

route exceptions, handling, 209

route handlers, 13–14
- custom, 14, 17–18

RouteData collection, 17

RouteData objects, 13

RouteData.Values dictionary, 23

RouteExistingFiles property, 14

RouteLink helper, 54

routes, 10
- attributes of, 12
- catch-all routes, 210–211
- constraints, 12
- default, 10, 12
- devising for SEO, 191–192
- input data from, 22–24
- placeholders in, 10
- processing, 12–13
- processing order, 12
- registering, 11, 17–18
- testing, 351–352
- URLs, matching to, 12–13

RouteTable.Routes collection, 11

routing
- preventing, 14–15
- subdomains awareness, 55
- RSS 2.0 feeds, 320–322
- run-time environment of ASP.NET MVC, 3

S

Saltarello, Andrea, 106, 168, 332

Save As dialog box, 323

ScaffoldColumn attribute, 167

scalability, asynchronous operations and, 33–34

ScaleOut, 195

script functions, programmatically invoking, 316

Script property, 314

<script> tags, 316
- defer attribute, 400
  for JSONP calls, 320

scripts
- compressing, 401
- loading, 399–401

sealed classes, testing, 334

search engine optimization (SEO). See SEO

security, 227–231
- authentication services, 240–251
- definition of, 227
- membership system, implementing, 232–239
- per-status codes and, 209

Select-Edit-Post pattern, 132–139
- drop-down lists, presenting, 133–134
- Edit actions requests, 134
- input data, editing, 134–136
- input data, passing to processing layer, 133
- input data, saving, 136–139
- storage layer, saving changes to, 132
- view, recreating and repopulating, 135
- view model objects, obtaining, 133

selectors, 385–390
- compound, 386–387
- CSS-based selectors, 385
- filters, 387–389
- ID selectors, 385
- tag-based selectors, 386

Selenium, 101

self-validation, 180–185

SEO, 190–193
- per-status code views and, 209
- redirection and, 190–191
- routes and URLs, devising, 191–192
- trailing slash and, 192–193
separation of concerns (SoC), 16, 257
layers for, 271
server applications, Office automation in, 325
server controls, 49–51, 70–71
input validation and, 131
role of, 104–105
stateful programming and, 131
server-side validation, 179
servers
files, creating on, 118
files, saving uploaded, 117
service layer, Ajax, 403–405
Service Locator (SL) pattern, 272–273
in ASP.NET MVC, 287–288
dependency resolver, defining, 288–289
vs. Dependency Injection, 286–287
service locators, 162
registering, 287
Service-Oriented Architecture (SOA), 267
session class, faking, 362
Session object, 193–194
feedback, saving to, 141–143
mocking, 362–363
session state management, 193–194
SetControllerFactory method, 277
Shared Cache, 195
SimpleMembership API, 236
simplicity of code, 329, 340
Single Responsibility Principle, 332
single sign-on schemes, 246
singly registered components, 287–288
SOA, 267
SoC, 16, 257, 271
software. See also design, software; design principles
control, 329
interface-based programming, 330–332
maintainability, 330
readability of, 330
simplicity of, 329
SOLID development principles, 236, 257, 271, 332
testing and testability, 327–369
units of, 334
visibility of behavior, 329
SOLID acronym, 236, 257, 271, 332
solutions, layering in, 40
SortEncodings method, 297
Spring.NET, 275
sprites, 402–403
StackOverflow, 284
stateful programming models, 131
statement coverage of testing, 344
static indexes, 116–117
static resources
download time, minimizing, 401–402
script loading, 399–401
status codes. See also HTTP codes
custom pages or routes for, 209
encapsulating, 314
Stefanov, Stoyan, 398
stereotypes, object, 254–255
StopRoutingHandler class, 14
storage, types of, 266
storage layer, saving changes to, 132
string members, null, 139
string parameters, null, 139
strings, arrays of, 112–113
strongly typed view models, 78, 92–94
StructureMap, 275
subdomains, awareness of’, 55
submit buttons, restricting methods to specific,
305–306
SwitchToErrorView method, 204
syndication feed responses, 320–322
SyndicationResult class, 320
System.Runtime.Caching assembly, 198
System.ServiceModel.Syndication namespace, 320
System.ServiceModel.Web assembly, 320
System.Web.WebPages assembly, 76

T

table-based layouts, 163–164
tag-based selectors, 386
TagBuilder class, 59
TempData
registering, 285
replaceability of, 282
TempData dictionary, 141–143
alternate storage, 284–285
templated delegates, 86–88
and formatted text, 88–89
and layout pages, 87–88
sharing, 89
templated HTML helpers, 56–58
overloads, 56
TemplateInfo property, 157
test doubles, 341–342
for dependencies in testing, 352–356
test environment, choosing, 335–336
test fixtures, 336–337

test harnesses, 330, 334–339
  arrange, act, assert layout, 337–338

test environment, choosing, 335–336

test fixtures, 336–337

test methods, 334
  arrange, act, assert layout, 337–338

testability
  control, 329
  coupling and, 330–331, 333
  definition, 328
  object orientation and, 334
  relativity of, 332–333
  simplicity, 329
  software design and, 328–334
  visibility, 329

TestClass attribute, 337

TestCleanup attribute, 337

TestDriven.NET, 335

testing
  acceptance tests, 348
  action filters, 365–368
  assertions per test, 343
  asynchronous methods, 354–358
  data access layer, 348
  data access operations, 353–354
  data-driven tests, 339
  definition, 328
  dependencies, dealing with, 352–358
  domain layer, 347
  front-end test tools, 99–101
  HTTP context, mocking, 358–368
  Humble Object pattern, 355–358
  ignoring tests, 338–339
  importance of, 327
  inconclusive tests, 338–339
  integration tests, 333
  localization, 349–351
  mediators in, 355–358
  for nullness, 139
  orchestration layer, 347
  redirects, 351–352
  test environment, choosing, 335–336
  test fixtures, 336–337
  unit testing, 334–345
  unit testing ASP.NET MVC code, 348–352
  views, 99–101
  vs. debugging, 328
  which part to test, 345–348

TestInitialize attribute, 337

TestMethod attribute, 337

text
  HTML encoding, 59–60
  localizable, 214–216

ThreadAbortException, 203

threads in asynchronous operations, 34–36

three-level architecture, 264

tier, definition of, 264

tight coupling, 330–331

timeout values, for asynchronous operations, 39

TimeSpan objects, 126

titles, page, updating in Ajax forms, 148–150

token managers, 249–250

tooltips, adding, 70–71

transient entities, 269

trigger methods, 34–36
  attributes on, 39
  coding, 38

try/catch blocks, exception handling with, 201–202

TrySkipIsisCustomErrors property, 211

Twitter user authentication, 246–251

Typemock, 334, 342

Typemock Isolator, 364

U

UILanguage property, 221

UIHint annotation, 159
  for read-only templates, 161–162

unauthorized access, 26

undefined variables, 375

Uniform Resource Identifiers (URIs), 4

Uniform Resource Locators (URLs). See URLs

Uniform Resource Names (URNs), 4

unit testing, 334–345. See also testability; testing
  of ASP.NET MVC code, 348–352
  assertions, 343
  code coverage, 344–345
  fakes and mocks, 341–342
  focus of, 331
  granularity of, 329
  HTTP context, mocking, 358–368
  inner member testing, 343–344
  limited scope, 340
  reliability of, 344
  test environment, choosing, 335–336
  test harnesses, 335–338
  testing in isolation, 340–341
unit testing (continued)
  as white-box testing, 347
  writing tests, 348
units of code, 334
Unity, 275–276, 279
  custom controller factory in, 278–279
  dependency resolver, 288–289
unobtrusive JavaScript, 147, 150–152, 392–396. See also JavaScript
  Ajax callbacks, order of, 146
  for client-side validation, 178
  jQuery plug-ins, 394–396
  rules of, 393–394
Update method, 137
update operations, 141
updaters, JavaScript, 409–413
uploaded files, binding, 117–119
URLs, 4
  URL parameters, 10
  URL patterns, 10
    in routes, 12
  URL rewriting, 7–8
  URL routing, 8–10
    physical files requests, 14
    preventing for defined URLs, 14–15
    route handlers, 13–14
    routes, defining, 11–12
    routes, processing, 12–13
URL routing HTTP module, 7–9
  internal structure, 9
  role of, 8–9
URL template, 156–157
Url.Content method, 66
  content files, referencing, 216
  extending, 216–217
UrlHelper class, 54
URLs, 4
  absolute paths of, 10
  canonical URL format, 192
  case sensitivity, 192
  custom, 10, 17
  devising for SEO, 191–192
  HTTP handlers, binding to, 4
  invalid, 209
  legacy URLs, 193
  page-agnostic, 6–7
  parsing, 5–6
  parts of, 4
  for posts, generating, 137
  recognized, 4–5, 7
routes, matching to, 12–13
routing, preventing, 14–15
synchronization with displayed content, 139–140
syntax, defining, 4–5
tilde () operator, 66, 216
trailing slash, 192–193
uniqueness, 191–192
of update operations, 141
URNs, 4
use-cases, presentation layer, 269
user authentication via Twitter, 246–251
user credentials, validating, 233–234
user interface
  drop-down lists, presenting, 133–134
  hiding elements, 231
  updating, 409
UserData property, 245
users
  creating, 232
  friendly names of, 243
  identifying, 241–245
  logging on and off, 232–233
  requests, placing, 3
  roles, 236
@using directives in Razor files, 80

V
VAB, 183–184
ValidateAntiForgeryToken filter, 293
ValidateInput filter, 293
validation, 58
  of action names, 302–303
  centralized, 181–182
  client-side validation, 177–178
  data type validation, 160
  input validation, 167–185
  of methods, 303–304
  of passwords, 235–236
  of roles, 237
  on server, 179
  against specific types, 121
  for update operations, 137
  of user credentials, 233–234
Validation Application Block (VAB), 183–184
validation attributes, custom, 175–176
validation messages, 54
validation providers, 168–169
ValidationAttribute class, 169
ValidationMessage helper

ValidationMessage helper, 137
value provider factories, 285–286
value providers, 105, 285–286
custom, 25
list of, 24
request collections covered, 110
ValueProvider dictionary, 24–25
ValueProviderResult type, 24
values, editing, 57–58
var keyword, 376
variables
dynamic variables, 95–96
hoisting, 377
local and global, 375–376
values of, 375
VaryByParam attribute, 199
view class, custom, 77–78
view context, 90
view engines, 41. See also ASPX view engine; Razor view engine
anatomy of, 44–45
calling, 45–46
class hierarchy, 64
and controllers, interaction, 42
culture-driven, 283–284
custom, 96–98
default conventions and folders, 47–49
detecting, 43–44
HTML helpers, 50–61
HTML markup, composing, 28
location format properties, 98
markup language, 47
mechanics of, 42–47
Razor view engine, 72–89
registering, 285
replaceability of, 282
role of, 46
structure and behavior of, 42–50
Web Forms view engine, 62–71
View method, 45–46
view name, 46
view-model classes
defining, 92–93
modeling view in, 94
packaging, 96
retrieving, 93–94
reusing, 94
view model objects, obtaining, 133
view models, 21, 56, 103, 106
for Ajax forms, 148
filling, 301–302
messages related to, 137–138
view names
HTML markup, translating into, 47
resolving, 47–49
view objects, 44, 46–47
Razor view object, 76–77
view selectors, 298–300
view state
role of, 105
stateful programming and, 131
view subsystem, action results and, 3
view templates
defining, 47–50
generation of, 3
locating, 47–48
locations, 28, 62
markup languages, 48
master view, 50
purpose, 49–50
readability, 72
search locations, 63
sections, 80–81
syntaxes, 28, 48–49
ViewData content, retrieving, 90–91
ViewBag dictionary, 50
in code nuggets, 78
passing data to view with, 90–92
ViewBag property, 57
ViewData dictionary, 50
in code nuggets, 78
content, retrieving, 90–91
passing data to view with, 90–91
ViewEngineResult objects, 44–45
ViewEngines class, 43
ViewModels folder, 96
ViewName property, 349
ViewPage class, 52
ViewResult class, 26, 46
ViewResult objects, 28, 47
views
Ajax, adding, 143
code blocks, 66–68
coding, 90–101
controller methods, calling from within, 98
controllers, separation from, 16, 71, 194
views (continued)
designing in ASPX view engine, 65–71
designing in Razor view engine, 78–86
feedback, passing to, 141–143
global information around, 301–302
hierarchical, 163
localizable, 219–220
locations cache, 63–64
master views, 65–66
modeling, 90–96
names, 46
partial, 55, 147–148
passing data to, 66–67, 71, 78, 88–92
reading and maintaining, 72
recreating and repopulating, 135
rendering, 46–47
rendering logic, 68–69
resolving, 58, 284
strongly typed models, 92–94
testing, 99–101, 348–349
view-model classes, 92–94
Views folder, 28, 47–48
_ViewStart.cshtml file, 79
_ViewStartPage class, 88–89
ViewUserControl class, 55
Virtual Accounts, 119
virtual file systems, 65
virtual path providers, 64–65
VirtualPathProvider class, 64–65
VirtualPathProviderViewEngine class, 64
visibility of software behavior, 329
Visual Basic, in Razor templates, 74–76
Visual Studio
code-coverage tools, 344
controller class, creating, 254
debugging tools, 327
IntelliSense, and dynamic types, 92
Moles add-in, 334, 364
Pex add-in, 345
test fixtures, 336–337
Web browsers. See browsers
Web Forms
code-behind class, 40
compression, 295
HTML, composing, 27–28
input processing, 104–105
location-aware physical resources, 4
master/detail view, 301
obsolescence of, 3
page life cycle, 104–105
page-processing phase, 40
pages, logic and view in, 41
resource folders, 212
server controls, 49
submit buttons, 305
switching views, 300
view, recreating and repopulating, 135
view engine, 62–71, 81
view state, 105
web pages
initialization code, 391–392
script loading, 399–401
sprites, 402–403
static files, 401–402
Web UI testing tools, 101
web.config file
<authentication> section, 228
client-side validation entry, 177
for custom view engines, 98
<customErrors> section, 200
<dynamicFilters> section, 309
globalization section, 221–222
httpRuntime section, 118
WebFormViewEngine class, 62–63
properties, 62
WebMatrix, 84
WebPage class, 52
WebSecurity class, 236
WebViewPage class, 76
IsSectionDefined method, 81
RenderBody method, 80
Wilson, Brad, 158
Windows 7 Virtual Accounts, 119
Windows authentication, 228, 231
Windows Server 2008 R2 Virtual Accounts, 119
Wirfs-Brock, Rebecca, 254
worker process identities, 119
worker service classes, 258–262
implementing, 260–262
W WatiN, 100–101
W3C API, 225
web applications. See also applications
session state management, 193–194
virtual file system for, 65
worker services

controller classes, injecting into, 276–277
orchestration and, 267
repositories, injecting, 276–277
WorkerServices folder, 260
wrapped sets, 384
chaining operations, 389–390
enumerating content, 384–385
find or filter function, 389

XMLHttpRequest objects, 237
xUnit.net, 335–336
Assert.Throws method, 339
test classes, 337
xxxAsync method, 35–37
xxxCompleted method, 37
xxxFor helpers, 52

Y

YAGNI principle (You Aren’t Gonna Need It), 196
Yahoo!, 240
OpenID URL, 241