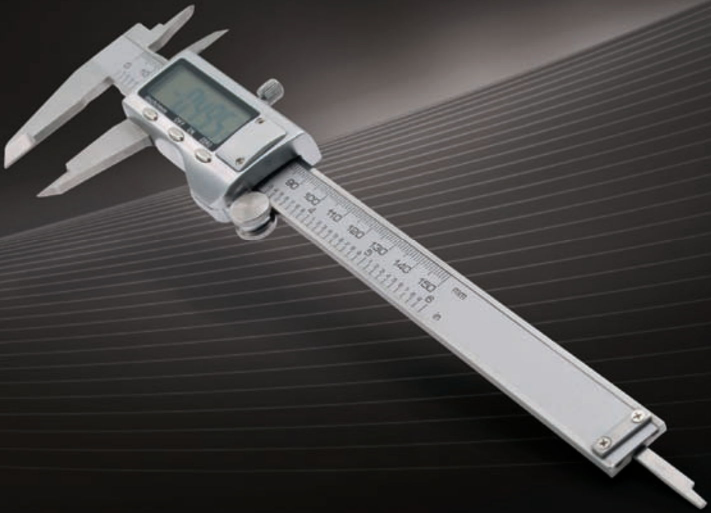
Microsoft®

Updated for ASP.NET MVC 3

Programming Microsoft® ASP.NET MVC

2 SECOND EDITION



Dino Esposito

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2011940367
ISBN: 978-0-7356-6284-1

Printed and bound in the United States of America.

First Printing

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Devon Musgrave
Copy Editor: Roger LeBlanc
Indexer: Christina Yeager
Editorial Production: Waypoint Press
Cover: Twist Creative • Seattle

To Silvia and my back for sustaining me.

Contents at a Glance

Table of Contents

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:

- One of Windows XP with Service Pack 3 (except Starter Edition), Windows Vista with Service Pack 2 (except Starter Edition), Windows 7, Windows Server 2003 with Service Pack 2, Windows Server 2003 R2, Windows Server 2008 with Service Pack 2, or Windows Server 2008 R2.
- 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:

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:

```
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:

```
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.

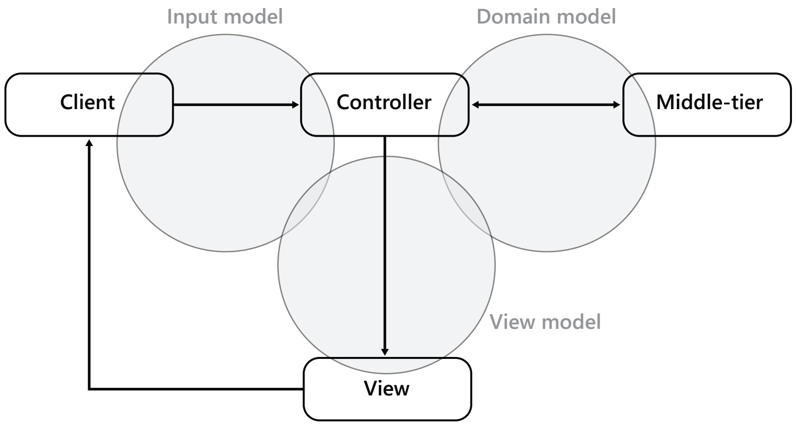


FIGURE 3-1 Types of models potentially involved in an ASP.NET MVC application.

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:

```
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:

```
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:

```
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:

```
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 3-1 Request collections for which a default value provider exists

Collection	Description
Form	Contains values posted from an HTML form, if any.
RouteData	Contains values excerpted from the URL route.
QueryString	Contains values specified as the URL's query string.
Files	A value is the entire content of an uploaded file, if any.

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:

```
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:

```
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:

```
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:

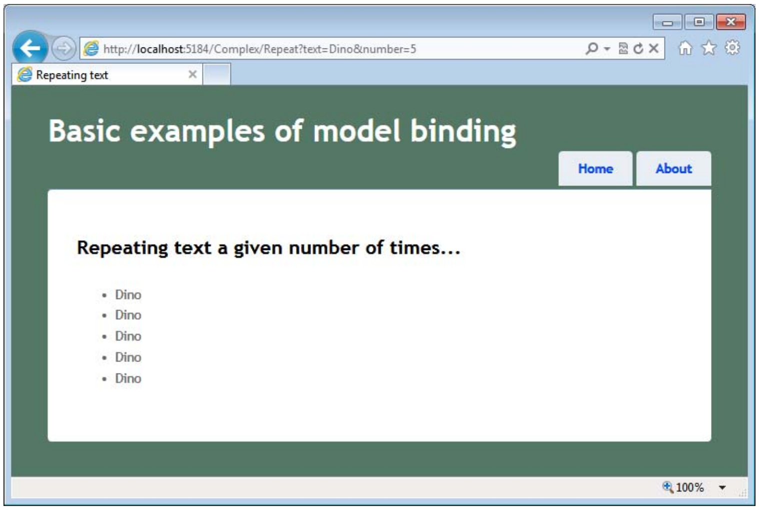
```
public class RepeatText
{
    public String Text { get; set; }
    public Int32 Number { get; set; }
}
```

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.


http://localhost:5184/Complex/Repeat?text=Dino&number=5
Repeating text
Basic examples of model binding
Home
About

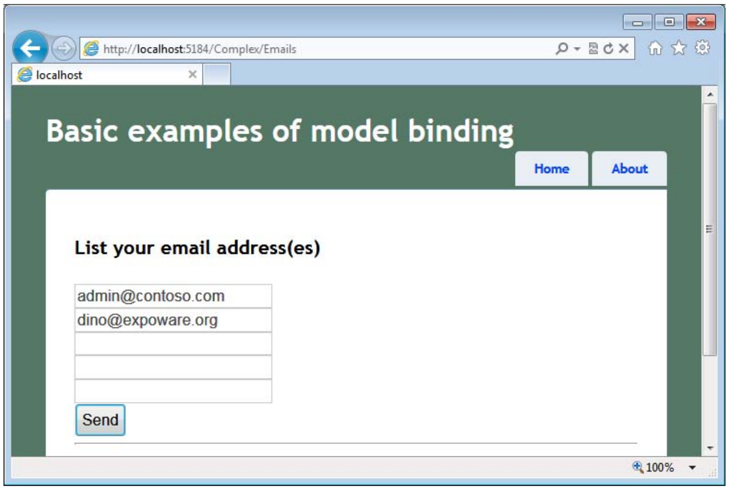
Repeating text a given number of times...

- Dino
- Dino
- Dino
- Dino
- Dino

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.


http://localhost:5184/Complex/Emails
localhost
Basic examples of model binding
Home
About

List your email address(es)

admin@contoso.com
dino@expoware.org

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:

```
TextBox1=admin@contoso.com&TextBox2=&TextBox3=&TextBox4=&TextBox5=
```

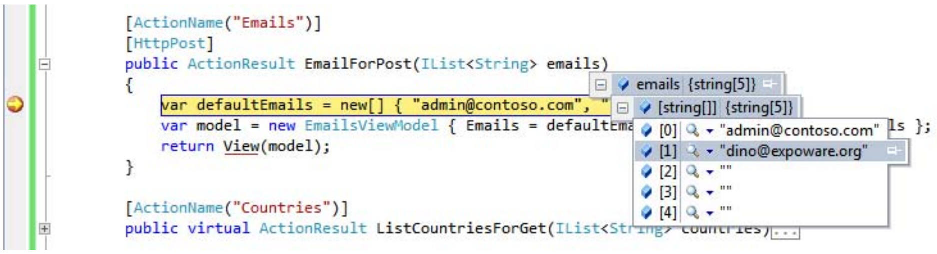
In classic ASP.NET, this is the only possible way of working because you can't just assign the same ID to multiple controls. However, if you manage the HTML yourself, nothing prevents you from assigning the five text boxes in the figure the same ID. The HTML DOM, in fact, fully supports this scenario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

```
@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:

```
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.



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

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

```
emails {string[5]}
  [string[]] {string[5]}
    [0] "admin@contoso.com"
    [1] "dino@expoware.org"
    [2] ""
    [3] ""
    [4] ""
```

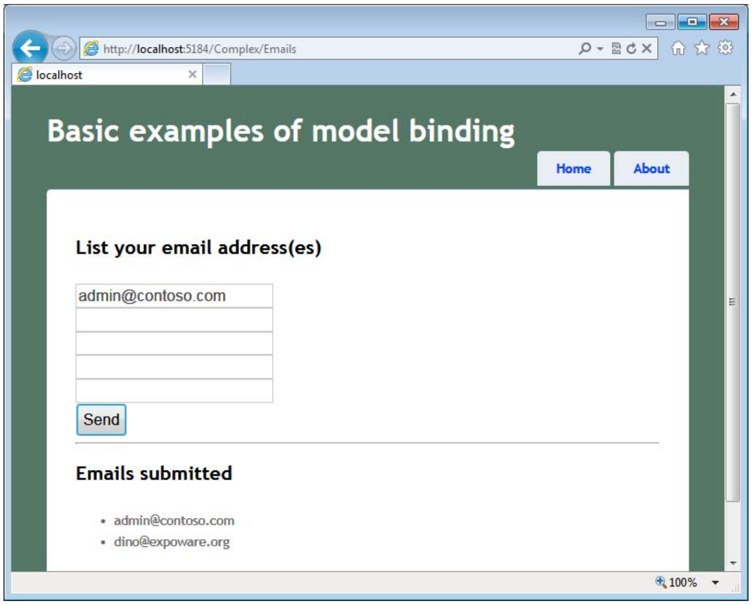
FIGURE 3-4 An array of strings has been posted.

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:

```
[ActionName("Emails")]
[HttpGet]
public ActionResult EmailForGet(IList<String> emails)
{
    // Input parameters
    var defaultEmails = new[] { "admin@contoso.com", "", "", "", "" };
    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>
```



Basic examples of model binding

List your email address(es)

admin@contoso.com

Emails submitted

- admin@contoso.com
- dino@expoware.org

FIGURE 3-5 The page rendered after a POST.

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:

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

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

```
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:

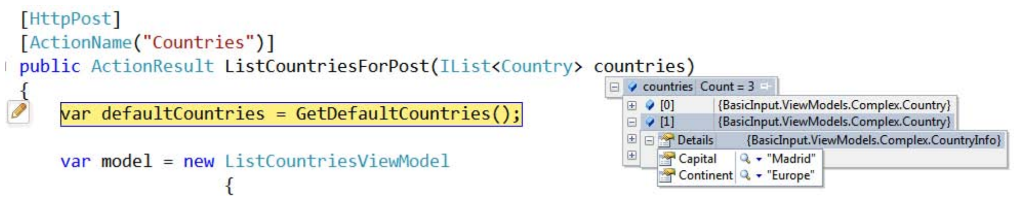
```
@using (Html.BeginForm())
{
    <h2>Select your favorite countries</h2>
    var index = 0;
    foreach (var country in Model.CountryList)
    {
        <fieldset>
        <div>
            <b>Name</b><br />
            <input type="text"
                   name="countries[@index].Name"
         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



```
[HttpPost]
[ActionName("Countries")]
public ActionResult ListCountriesForPost(IList<Country> countries)
{
    var defaultCountries = GetDefaultCountries();

    var model = new ListCountriesViewModel
                {
```

```
countries   Count = 3
  [0]       {BasicInput.ViewModels.Complex.Country}
  [1]       {BasicInput.ViewModels.Complex.Country}
    Details     {BasicInput.ViewModels.Complex.CountryInfo}
      Capital     "Madrid"
      Continent   "Europe"
```

FIGURE 3-6 Complex and nested types posted to the method.

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*:

```
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:

```
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:

```
<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:

```
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:

```
<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

Property	Description
Prefix	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.
Exclude	Gets or sets a comma-delimited list of property names for which binding is not allowed.
Include	Gets or sets a comma-delimited list of property names for which binding is per-mitted.

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:

```
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:

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

You can use *Include* and *Exclude* in the same attribute if dong 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:

```
[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.

Sometimes, 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:

```
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:

```
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 *OnModelUpdated* 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:

```
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*:

```
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:

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

Local association requires the following syntax:

```
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:

```
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:

```
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:

```
@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:

```
@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.

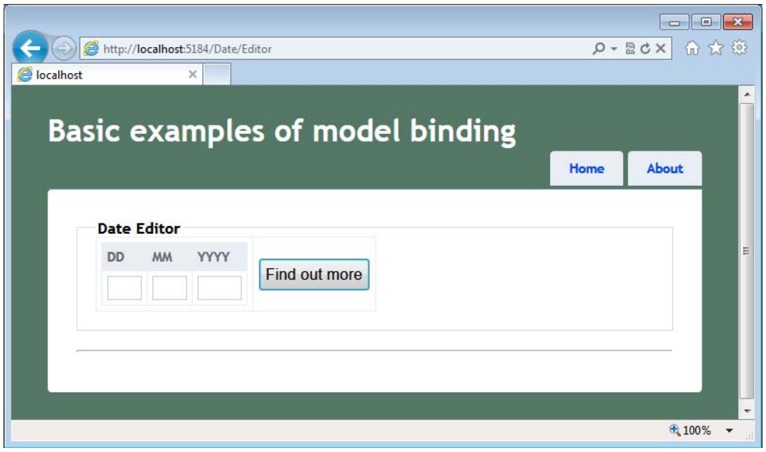

http://localhost:5184/Date/Editor
localhost
Basic examples of model binding
Home
About
Date Editor
DD
MM
YYYY
Find out more
100%

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:

```
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:

```
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:

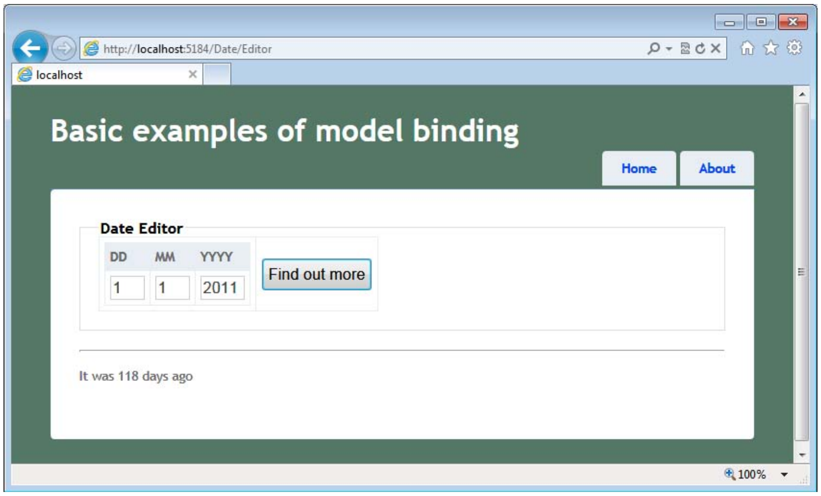
```
public class DateModelBinder : IModelBinder
{
    public Object BindModel(ControllerContext controllerContext, ModelBindingContext
bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException("bindingContext");
        }

        // 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:

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

Figure 3-8 shows the final page in action.


http://localhost:5184/Date/Editor
localhost
Basic examples of model binding
Home
About
Date Editor
DD
MM
YYYY
1
1
2011
Find out more
100%

It was 118 days ago

FIGURE 3-8 Working with dates using a custom type binder.

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.

Index