

# Programming Microsoft ASP.NET MVC Third Edition

Professiona

Dino Esposito



# Programming Microsoft ASP.NET MVC, Third Edition

**Dino Esposito** 

#### Copyright © 2014 Leonardo Esposito

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-8094-4

Second Printing: April 2014

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at *mspinput@microsoft.com*. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Microsoft and the trademarks listed at *http://www.microsoft.com/about/legal/en/us/IntellectualProperty/ Trademarks/EN-US.aspx* are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the author, O'Reilly Media, Inc., Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Developmental Editors: Russell Jones and Rachel Roumeliotis

Production Editor: Kristen Brown
Editorial Production: Dianne Russell, Octal Publishing, Inc.
Technical Reviewer: John Mueller
Copyeditor: Bob Russell, Octal Publishing, Inc.
Indexer: BIM Indexing Services
Cover Design: Twist Creative • Seattle and Joel Panchot
Cover Composition: Ellie Volckhausen
Illustrator: Rebecca Demarest

To Silvia, Francesco, Michela, and my back for sustaining me. —Dino

# Contents

	Introduction xiii
PART I	ASP.NET MVC FUNDAMENTALS
Chapter 1	ASP.NET MVC controllers 3
	Routing incoming requests       4         Simulating the ASP.NET MVC runtime       4         The URL routing HTTP module       7         Application routes       9
	The controller class       15         Aspects of a controller       16         Writing controller classes       17         Processing input data       22
	Producing action results
Chapter 2	ASP.NET MVC views 33
	The structure and behavior of a view engine
	HTML helpers
	The Razor view engine.    .54      Inside the view engine.    .54      Designing a sample view    .59
	Coding the view

Chapter 3	The model-binding architecture	75
	The input model	76
	Evolving from the Web Forms input processing	76
	Input processing in ASP.NET MVC.	78
	Model binding	79
	Model-binding infrastructure	79
	The default model binder	80
	Customizable aspects of the default binder	91
	Advanced model binding	93
	Custom type binders	93
	A sample <i>DateTime</i> model binder	96
	Summary	.101

# Chapter 4 Input forms

General patterns of data entry	
A classic Select-Edit-Post scenario	
Applying the Post-Redirect-Get pattern	
Automating the writing of input forms	
Predefined display and editor templates	
Custom templates for model data types	
nput validation	
Using data annotations	130
Advanced data annotations	135
Self-validation	
Summary	
r r	ieneral patterns of data entry A classic Select-Edit-Post scenario Applying the Post-Redirect-Get pattern utomating the writing of input forms Predefined display and editor templates Custom templates for model data types nput validation Using data annotations Advanced data annotations Self-validation

103

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

Chapter 5	Aspects of ASP.NET MVC applications	151
	ASP.NET intrinsic objects	
	HTTP response and SEO	
	Managing the session state	
	Caching data	

Error handling	
Handling program exceptions	
Global error handling169	
Dealing with missing content	
Localization	
Using localizable resources175	
Dealing with localizable applications	
Summary	

# Chapter 6 Securing your application

Security in ASP.NET MVC
Implementing a membership system.195Defining a membership controller.196The Remember-Me feature and Ajax.204
External authentication services
Summary

# Chapter 7 Design considerations for ASP.NET MVC controllers 225

Shaping up your controller
Choosing the right stereotype226
Fat-free controllers
Connecting the presentation and back end
The Layered Architecture pattern
Injecting data and services in layers
Gaining control of the controller factory
Summary

# 189

Chapter 8	Customizing ASP.NET MVC controllers	255
	The extensibility model of ASP.NET MVC	
	The provider-based model	
	The Service Locator pattern	
	Adding aspects to controllers	
	Action filters	
	Gallery of action filters	
	Special filters	
	Building a dynamic loader filter	
	Action result types	
	Built-in action result types	
	Custom result types	
	Summary	
Chapter 9	Testing and testability in ASP.NET MVC	301
	Testability and design	
	DfT	
	Loosen up your design	
	The basics of unit testing	
	Working with a test harness	
	Aspects of testing	
	Testing your ASP.NET MVC code	
	Which part of your code should you test?	
	Unit testing ASP.NET MVC code	
	Dealing with dependencies	
	Mocking the HTTP context	
	Summary	
Chapter 10	An executive guide to Web API	337
	The whys and wherefores of Web API	
	The need for a unified HTTP API	
	MVC controllers vs. Web API	

Putting Web API to work	
Designing a RESTful interface	
Expected method behavior	
Using the Web API	
Designing an RPC-oriented Interface	
Security considerations	
Negotiating the response format	
The ASP.NET MVC approach	
How content negotiation works in Web API	
Summary	

# PART III MOBILE CLIENTS

#### **Chapter 11 Effective JavaScript**

Revisiting the JavaScript language
Language basics
Object-orientation in JavaScript
jQuery's executive summary
DOM queries and wrapped sets
Selectors
Events
Aspects of JavaScript programming
Unobtrusive code
Reusable packages and dependencies
Script and resource loading
Bundling and minification
Summary

#### Chapter 12 Making websites mobile-friendly

Technologies for enabling mobile on sites	99
HTML5 for the busy developer40	)0
RWD40	)7
jQuery Mobile's executive summary41	13
Twitter Bootstrap at a glance42	23

# 367

#### 399

Adding mobile capabilities to an existing site	0
Routing users to the correct site	;1
From mobile to devices	6
Summary	57

#### Chapter 13 Building sites for multiple devices 439

Understanding display modes in ASP.NET MVC	440
Separated mobile and desktop views	440
Rules for selecting the display mode	442
Adding custom display modes	443
Introducing the WURFL database	446
Structure of the repository	447
Essential WURFL capabilities	451
Using WURFL with ASP.NET MVC display modes	454
Configuring the WURFL framework	454
Detecting device capabilities	456
Using WURFL-based display modes	459
The WURFL cloud API	464
Why you should consider server-side solutions	466
Summary	467
Index	469
About the author	495

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

SP.NET was devised in the late 1990s at a time when many companies in various industry sectors were rapidly discovering the Internet. The primary goal of ASP.NET was to make it possible for 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 at that time. ASP.NET is what Microsoft delivered to address this request, exceeding expectations by a large extent.

Today, more than ten years later, ASP.NET is showing signs of age, and many started even questioning the real necessity of having a web framework at all. It's an amazing time, and several options exist. There are Web Forms and ASP.NET MVC applications, and then there are more JavaScript-intensive client applications (single-page applications) that just use a server-side back end for delivering the basic layout of the few pages they actually expose and for ad hoc services such as bundling.

Curiously, with the Web Forms paradigm, you can still write functional applications even though ASP.NET MVC addresses more closely the present needs of developers. The most common scenario of Web Forms is applications for which you focus on presenting data and use some third-party high-quality suite of controls for that. ASP.NET MVC is for everything else, including the scaffolding of client-side single-page applications.

The way web applications are changing proves that ASP.NET MVC probably failed to replace ASP.NET Web Forms in the heart of many developers, but it was the right choice and qualifies to be the ideal web platform for any application that needs a back end of some substance; in particular (as I see things), web applications that aim at being multidevice functional. And yes, that likely means all web applications in less than two years.

Switching to ASP.NET MVC is more than ever the natural follow-up for ASP.NET developers.

# Who should read this book

Over the years, quite a few people have read quite a few books and articles of mine. These readers are already aware that I'm not good at writing step-by-step, referencestyle books, in the similar manner that I'm unable to teach the same class twice, running topics in the same order and showing the same examples.

This book is not for absolute beginners; but I do feel it is a book for all the others, including those who are still fairly new to ASP.NET MVC. The higher your level of competency and expertise, the less you can expect to find here that adds value in your particular case. However, this book benefits from a few years of real-world practice; so I'm sure it has a lot of solutions that might also appeal to the experts, particularly with respect to mobile devices.

If you use ASP.NET MVC, I'm confident that you'll find something in this book that makes it worthwhile.

# Assumptions

This book expects that you have at least a minimal understanding of ASP.NET development.

# Who should not read this book

If you're looking for a step-by-step guide to ASP.NET MVC, this is not the ideal book for you.

# **Organization of this book**

This book is divided into three sections. Part I, "ASP.NET MVC fundamentals," provides a quick overview of the foundation of ASP.NET and its core components. Part II, "ASP.NET MVC software design," focuses on common aspects of web applications and specific design patterns and best practices. Finally, Part III, "Mobile clients," is about JavaScript and mobile interfaces.

# System requirements

You preferably have the following software installed in order to run the examples presented in this book:

- One of the following operating systems: Windows 8/8.1, Windows 7, Windows Vista with Service Pack 2 (except Starter Edition), Windows XP with Service Pack 3 (except Starter Edition), Windows Server 2008 with Service Pack 2, Windows Server 2003 R2
- Microsoft Visual Studio 2013, any edition (multiple downloads might be required if you're using Express Edition products)
- Microsoft SQL Server 2012 Express Edition or higher, with SQL Server Management Studio 2012 Express or higher (included with Visual Studio; Express Editions require a separate download)

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2013 and SQL Server 2012 products.

# **Code samples**

Most of the chapters in this book include exercises with which you can interactively try out new material learned in the main text. You can download all sample projects, in both their pre-exercise and post-exercise formats, from the following page:

http://aka.ms/programASP-NET\_MVC/files

Follow the instructions to download the asp-net-mvc-examples.zip file.

# Installing the code samples

Perform the following steps to install the code samples on your computer so that you can use them with the exercises in this book.

- 1. Unzip the asp-net-mvc-examples.zip file that you downloaded from the book's website (name a specific directory along with directions to create it, if necessary).
- **2.** If prompted, review the displayed end-user license agreement. If you accept the terms, select the Accept option, and then click Next.

-			-	
-	_	_	-	
-	_	_	-	
-	_	_	-	
-	_	_	-	

**Note** If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the asp-net-mvc-examples.zip file.

# Using the code samples

The folder created by the Setup.exe program contains one subfolder for each chapter. In turn, each chapter might contain additional subfolders. All examples are organized in a single Visual Studio 2013 solution. You open the solution file in Visual Studio 2013 and navigate through the examples.

# 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:

http://aka.ms/programASP-NET\_MVC/errata

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 aforementioned addresses.

# 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://aka.ms/tellpress

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

# 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'll 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"?

I like to say that "model" is the most misunderstood idea in the history of software. As a concept, it needs to be expanded a bit to make sense in modern software. When the Model-View-Controller (MVC) pattern was introduced, software engineering was in its infancy, and applications were much simpler than today. Nobody really felt the need to break up the concept of model into smaller pieces. Such smaller pieces, however, existed.

In general, I find *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 entities and relationships that populate the business domain. These entities are typically persisted by the data-access layer and consumed by services that implement business processes. This domain model pushes a vision of data that is, in general, distinct and likely different from the vision of data you find in the presentation layer. The view model just describes the data that is being worked on in the presentation layer. A good example might be the canonical *Order* entity. There might be use-cases in which the application needs to present a collection of orders to users but not all properties are required. For example, you might need ID, date, and total, and likely a distinct container class—a data-transfer object (DTO).

Having said that, I agree with anyone who points out that not every application needs a neat separation between the object models used in the presentation and business layers. 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 and subsequently received by the application. The input model defines the DTOs the application uses to receive data from input forms.

**Note** Yet another flavor of model not mentioned here is the *data model* or the (mostly relational) model used to persist data.

# The input model

Chapter 1, "ASP.NET MVC controllers," discusses request routing and the overall structure of controller methods. Chapter 2, "ASP.NET MVC views," explores views as the primary result of action processing. However, neither chapter thoroughly discusses how in ASP.NET MVC a controller method gets input data.

In ASP.NET Web Forms, we had server controls, view state, and the overall page life cycle working in the background to serve 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 provided a faithful server-side representation of the client user interface. Developers just needed to write C# code to read from input controls.

ASP.NET MVC makes a point of having controllers 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 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. In ASP.NET Web Forms, the input model is just based on server controls and the view state.

#### Role of server controls

Suppose that you have a webpage with a couple of *TextBox* controls to capture a user name and password. When the user posts the content of the form, there is likely a piece of code to process the request similar to what is shown in the following code:

```
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 such as 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.

#### Role of the view state

Speaking in terms of a programming paradigm, a key distinguishing characteristic between ASP.NET Web Forms and ASP.NET MVC is the view state. In Web Forms, the view state 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 that you don't touch in a postback. Suppose that you display a list of choices into which the user can drill down. 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

Chapter 1, explains 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.

#### The role of model binders

ASP.NET MVC provides an automatic binding layer that uses a built-in set of rules for mapping raw request data from any value providers to properties of input model classes. As a developer, you are largely responsible for the design of input model classes. 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.

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

Chapter 1 points out that 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.

Suppose that you need a controller method to repeat a particular 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 in which 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

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 must 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.

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 Request collections for which a default value provider exists

Table 3-1 lists request collections being considered in the exact order in which they are processed by the default binder. Suppose that 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 is actually passed to the method is 10, which is the value read from the route data collection.

#### Binding complex types

There's no limitation regarding 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=ASP.NET%20MVC&number=5
```

Figure 3-2 shows the output that 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.

-	×
+ Http://localhost:3195/complex/emails	• ≥¢ û ☆ ŵ
🧟 localhost 🛛 🗙	
List your email address(es)	
admin@contoso.com	
Send	
Emails submitted	
	€ 100% ▼

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

When the user clicks 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:

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> email)
{
    ...
}
```

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



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

As is dicussed in greater detail in Chapter 4, "Input forms," 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 make it possible for 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> email)
{
    var defaultEmails = new[] { "admin@contoso.com", "", "", "", "" };
   var model = new EmailsViewModel { Emails = defaultEmails, RegisteredEmails = email };
    return View(model);
}
```

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

@model BindingFun.ViewModels.Complex.EmailsViewModel

```
<h2>List your email address(es)</h2>
@using (Html.BeginForm())
{
    foreach(var email in Model.Emails)
    ł
       <input type="text" name="email" value="@email" />
       <br />
   }
   <input type="submit" value="Send" />
}
<hr />
<h2>Emails submitted</h2>
<u1>
@foreach (var email in Model.RegisteredEmails)
{
   if (String.IsNullOrWhiteSpace(email))
    {
       continue;
    }
   @email
}
```

	- • ×
+ ttp://localhost:3195/complex/emails	- BC 🏠 🛱
@localhost ×	
List your email address(es)	
admin@contoso.com	
Send	
Emails submitted	
admin@contoso.com	
foo@contoso.com	
	€,100% ▼



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, must match to the controller method's signature according to the normal rules of the binder.

In case of collections, the required match between names forces you to violate basic naming conventions. In the view, you have input fields and would like to call them, for instance, *email* using the singular. When you name the parameter in the controller, because you're getting a collection, you would like to name it, for instance, *emails*. Instead, you're forced to use either *email* or *emails* all the way through. The workaround comes in a moment when we move on to consider customizable aspects of model binders.

#### Binding collections of complex types

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

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

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())
ł
    <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="country[@index].Details.Capital"
                   value="@country.Details.Capital" /><br />
            <b>Continent</b><br />
            @{
                var id = String.Format("country[{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="country[0].Name"
value="@country.Name" />
<input type="text"
name="country[1].Name"
value="@country.Name" />
```

Be aware 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.

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 computer:

```
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. You can configure the value 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. Also keep in mind that in a hosting scenario your application-level settings might be ignored if the host has set a different limit at the domain level and locked down the *maxRequest*-*Length* 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 can 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 you use multiple <input> elements that are 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]" />
```

# ASP.NET application account

Creating files on the web server is not usually an operation that can be accomplished relying on the default permission set. Any ASP.NET application runs under the account of the worker process serving the application pool to which the application belongs. 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 NETWORK SERVICE account, which is a relatively low-privileged, built-in identity in Microsoft Windows. Originally welcomed as an excellent security measure, in the end the practice of using a single account for a potentially high number of concurrently running services created more problems than it helped to solve.

In a nutshell, services running under the same account could tamper with one another. 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 and newer versions. For more information, have a look at *http://technet.microsoft.com/library/dd548356.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.

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 permitted.

TARI F 3.2	Properties	for the	RindAttrihute	class
TADLE 3-2	roperties	ior the	DITIUALLIDULE	Class

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, such as shown here:

```
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 doing so makes it possible for you to better define the set of properties to bind. For instance, if both attributes refer to the same property, *Exclude* will win.

#### Aliasing parameters by 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. Using the *Prefix* attribute, you can 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 = "email")]IList<String> emails)
{
    ...
}
```

For the *emails* parameter to be successfully filled, you need to have posted fields whose name is *email*, not *emails*. The *Prefix* attribute makes particular sense on POST methods and fixes the aforementioned issue with naming conventions and collections of parameters. Finally, note that if a prefix is specified, it becomes mandatory; subsequently, fields whose names are not prefixed are not bound.

**Note** Yes, the name chosen for the attribute—*Prefix*—is not really explanatory of the scenarios it addresses. Everybody agrees that *Alias* would have been a much better name. But, now it's too late to change it!

# 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. For example, if all you need to do is to keep the default behavior and simply force the binder to use a non-default 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:

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 employed 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 address in Chapter 4.

#### Implementing a model binder from scratch

The IModelBinder interface is defined as follows:

}

Following is 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 that 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.

Keep in mind 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:

Local binders always take precedence over globally defined binders.

As you can glean 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();
```

However, modifying the default binder can have a considerable impact on the behavior of the application and should therefore be a very thoughtful choice.

# A sample DateTime model binder

With input forms, it is quite common to have users enter a date. You can sometimes use a jQuery user interface to let users pick dates from a graphical calendar. If you use HTML5 markup on recent browsers, the calendar is automatically provided. 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, one each 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 as well as 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())
ł
<fieldset>
   <legend>Date Editor</legend>
   <div>
       @DateHelpers.InputDate("theDate", Model.DefaultDate)
       <input type="submit" value="Find out more" />
       </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();
   }
   <thead>
          DD
          MM
          YYYY
      </thead>
      <input type="number" name="@(name + ".day")"
                   value="@day" style="width:30px" />
          <input type="number" name="@(name + ".month")"
                   value="@month" style="width:30px">
          <input type="number" name="@(name + ".year")"
                   value="@year" style="width:40px" />
      }
```

Figure 3-7 shows the output.

			x
+ Description Alternation	- 20 G	$\Rightarrow$	ξţ;
🧟 localhost 🛛 🗙			
Date Editor			
DD MM YYYY Find out more			
	۹ 10	0% 🔻	, d

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

#### The controller methods

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 EditorViewModel();
        return View(model);
    }
    [HttpPost]
    [ActionName("Editor")]
    public ActionResult EditorForPost(DateTime theDate)
    {
        var model = new EditorViewModel();
        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 by using a *TimeSpan* object. Here's the view model object:

```
public class EditorViewModel : ViewModelBase
{
    public EditorViewModel()
    {
        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
Ł
    public Object BindModel(ControllerContext controllerContext, ModelBindingContext
bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException("bindingContext");
        3
        // 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)
```

```
{
        // Trv 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);
    trv
    {
        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:3195/date/editor	- ≧0 合 ☆ 袋
@ localhost ×	
Date Editor	
DD     MM     YYYY       31     12     2013   Find out more	
It will be in 9 days	
	⊕ 100% ▾

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 Chapter 4, I discuss aspects of input forms, input modeling, and data validation.

# Index

# Symbols

51degrees.mobi, 447 200 response code (HTTP), 347 202 response code (HTTP), 348 204 response code (HTTP), 347 301 response code (HTTP), 152-153, 155 302 response code (HTTP), 8, 113, 152, 172 400 response code (HTTP), 163 403 response code (HTTP), 172, 286 404 response code (HTTP), 8, 172-173 500 response code (HTTP), 163, 345 \$ function, 377 , (comma), 381 {} curly brackets, 10 . (dot), 380 / (forward slash), 10, 13 + operator, 380 == operator, 369 === operator, 369 > operator, 380 | (pipe symbol), 370 @ symbol, 44, 55-56, 58 ~ (tilde), 177, 381, 455

# A

AAA (arrange, act, assert), 311 acceptance tests, 322 Accept-Encoding header, 269, 270, 392 Accept header, 360 AcceptVerbs attribute, 19, 348, 353, 355 access tokens social authorization, 221–223 Web API, 357 Account controller, 218 Accuracy mode, 459 action attributes, 353 ActionFilterAttribute class, 267, 272 action filters built-in, 266 custom adding response header, 267-268 compressing response, 268-271 view selector, 271-275 defined, 255, 263 dynamic loader filter adding action filter using fluent code, 280 customizing action invoker, 280-282 enabling via filter provider, 282-285 interception points, 279-280 registering custom invoker, 282 embedded, 263-264 external, 263-264 global, 266-267 implementing as attribute, 264 types of, 265-266 Action HTML helper, 72 action invoker defined. 79 replaceable components, 256, 258 view engine and, 37-38 ActionLink HTML helper, 18, 43, 47, 53 action links, 46-47 action methods controller classes, 20-22 example of proper, 236 keeping lean, 231-236 restricting, 190-191 routing to REST, 344-346 RPC, 353

#### ActionMethodSelectorAttribute class

ActionMethodSelectorAttribute class, 277 ActionName attribute, 19, 275, 278 ActionNameSelectorAttribute class, 275 ActionResult class, 22, 26, 285-286, 290, 340 action results built-in returning custom status code, 285-287 returning HTML, 28-29 returning JavaScript, 287–288 returning JSON, 29-31, 288-290 returning primitive types, 290 custom returning binary data, 295-297 returning JSONP, 290-292 returning PDF files, 297-299 returning syndication feed, 293-295 mechanics of, 27-28 types of, 26-28 Actions and HTTP verbs, 19-20 ActionScript, 368 action selectors action method selectors, 276-277 action name selectors, 274-275 restricting method to Ajax calls only, 277-278 restricting method to button, 277-279 addMethod function, 144 adjacent operator, 380 advertised browser capability, 458 advertised\_device\_os capability, 458 advertised\_device\_os\_version capability, 458 Ajax (Asynchronous JavaScript and XML) form submission using, 104, 116 in jQM, 416 JavaScript history, 367 Remember-Me feature and reproducing problem, 204-205 solving, 205-207 restricting method to calls using, 277-278 WURFL capability group, 448 Ajax.BeginForm, 139 Ajax helper, 51, 53 Ajax property, 58 aliasing parameters, 92-93 AllowAnonymous attribute, 191, 356 Android, 399 AngularJS, 367 :animated filter, 381

annotations form display templates, 117-120 input validation client-side validation, 139-140 cross-property validation, 135-137 culture-based validation, 140-141 custom validation attributes, 137-139 decorating model class, 132-133 enumerated types, 133-134 error messages, 134-135 validating properties remotely, 141-142 validation provider, 130-131 anonymous functions, 372 anonymous users authorization and, 191 vs. not authorized, 193 AntiForgeryToken method, 48 ApiController class, 341-344, 361 Appcelerator Titanium, 367 Apple Safari, 402 ApplicationConfigurer class, 455 ApplicationDbContext class, 203 ApplicationException class, 164 application layer, 240-241 application routes defining, 11-12 processing, 12–13 Application\_Start event handler, 11, 36, 251 ApplicationUser class, 202, 203 architectural style, 342 AreaMasterLocationFormats property, 54-55 AreaPartialViewLocationFormats property, 54-55 AreaViewLocationFormats property, 54–55 ArgumentException, 165, 312 ArgumentNullException, 165 arrange, act, assert (AAA), 311 Array object, 368 <article> elements, 401 aspect-ratio property, 409 AspNetCacheService class, 161 ASP.NET MVC backward compatibility, 4 version 4, 34, 61 .aspx files, 39 ASPX pages and security, 189 ASPX view engine, 28, 34, 38, 40, 42 assemblies for resources, 177 referencing embedded files, 178 visibility of internal members, 318

assemblyinfo.cs file, 179, 317 assertions, 317 Assert. Throws method, 313 async attribute, 391 asynchronous calls to Web API, 351-352 Asynchronous JavaScript and XML. See Ajax asvnc kevword, 30, 351 AsyncTimeout filter, 266 Atom Syndication Format, 293 at symbol, 44, 55-56, 58 AttemptedValue property, 25 AttributeEncode method, 48 attribute filters, jQuery, 383 attribute routing defined, 15 enabling, 354-355 overview, 353-354 audio in HTML5, 406-407 AuthController class, 196 authentication authentication filters, 194-195 configuring, 190 membership system identity system, 201-204 integrating with roles, 200-201 Membership API, 198–199 overview, 195 SimpleMembership API, 200-201 validating user credentials, 196-198 methods for, 189 OpenID protocol vs. OAuth, 214-216 overview, 208 using, 209-214 Remember-Me feature and Ajax reproducing problem, 204-205 solving, 205-207 social networks access tokens, 221-223 enabling social authentication, 217-218 membership system, 220-221 registering application with Twitter, 215-216 starting process, 218-219 Twitter response, 219 Twitter, 221 AuthenticationResult class, 219, 222 <authentication> section, 190 authorization action methods restrictions, 190-191 allowing anonymous callers, 191

anonymous vs. not authorized, 193 hiding user interface elements, 192 output caching and, 192 authorization filters, 265 Authorize action filter, 286 Authorize attribute, 190–192, 355–356 AuthorizeAttribute class, 356 AuthorizeAttribute class, 356 AuthorizedOnly attribute, 206 Authorize filter, 266 AuthorizeRequest action filter, 73 Autofac, 249 AutoMapper, 243 AVI codec, 407 await keyword, 30, 351

# В

background-position property, 393 basic authentication, 357-358 basic helpers action links, 46-47 forms, 44-46 HtmlHelper class, 48 input elements, 46 partial views, 47 bearer capability group, 448 BeginForm HTML helper, 43-45, 109, 116 BeginRequest event, 432 BeginRouteForm HTML helper, 43-45 <b> element, 403 <br/>
<big> element, 403 binary data, returning using action result, 295-297 Bind attribute overview, 91 using, 107 bind function, 384 binding events using jQuery, 384-385 binding layer, 78 binding, model custom type binders creating, 94-95 customizing default binder, 93-94 registering, 96 DateTime model binder code for, 99-101 controller method, 98-99 displayed data, 97-98 overview, 96

#### black-box testing

default model binder aliasing parameters, 92–93 Bind attribute, 91 binding collections, 84-89 binding complex types, 83-84 binding content from uploaded files, 89-91 binding primitive types, 80-81 blacklist of properties, 92 optional values, 81-82 value providers, 82 whitelist of properties, 92 method signature, 79 model binders, 79 black-box testing, 321 blacklist of properties, 92 block elements, 401 blur event, 385 <body> tag. 391 Bootstrap button groups, 429-430 drop-down menus, 427-429 feature detection, 407 glyph icons, 427 grid system, 425-426 images, 427 mobile-friendly websites, 412, 423 navigation bars, 426-427 overview, 423-424 setting up, 424-425 brand\_name capability, 451-452 .browser files, 444 browsers, specific content for, 452-453 btn-group style, 429 built-in action filters, 266 built-in action results returning custom status code, 285-287 returning JavaScript, 287-288 returning JSON, 288-290 returning primitive types, 290 BundleCollection class, 395 BundleConfig class, 395 BundleFileSetOrdering class, 396 bundling JavaScript overview, 394-395 resources, 395 script files, 395-397 button groups, 429-430 ByteArrayModelBinder class, 95

# С

cache capability group, 448 Cache object caching method response, 161–162 distributed caching, 161-162 HttpContext and, 152 injecting caching service, 158-160 mocking, 333-335 OutputCache attribute, 161–162 overview, 157 partial caching, 162 pros and cons, 157-158 CacheProfile property, 162 caching layer, 107 camelCasing, 361 can\_assign\_phone\_number capability, 451–452 CanReadType method, 361 CanWriteType method, 361 capabilities, device accuracy vs. performance, 458-459 processing HTTP request, 456 virtual capabilities, 457-458 WURFL capability groups, 448-449 Capabilities dictionary, 465 caret segment, 430 Cascading Style Sheets. See CSS case for URLs, and SEO, 154 Castle DynamicProxy, 328 Castle Windsor, 249 catch-all route, 172-174 CDN (content delivery network), 392, 424 cell phones, 436 <center> element, 403 centralized validation, 143 chaining operations queries, 378 wrapped sets, 384 Chakra JavaScript engine, 368 change event, 385 ChangePassword method, 198 CheckBoxFor HTML helper, 43 CheckBox HTML helper, 43 ChildActionMvcHandler class, 73 ChildActionOnly attribute, 73, 162, 266 child actions, 73-74 chips capability group, 448 chtml\_ui capability group, 448 ClassCleanup attribute, 311 ClassInitialize attribute, 311

class-level validation, 143 click event, 385 client-side validation, 130, 139-140, 403 closures (JavaScript), 374-376 Cloud API, WURFL vs. on-premise API, 465 overview, 462-463 setting up, 464-465 code coverage, 318 decoupling from serialization, 340 layers of, 3 nuggets, Razor view engine conditional nuggets, 58 overview, 55-57 special expressions of, 57 Code Contracts, 146, 236 collapsible elements HTML5, 402-403 jQuery Mobile, 422-423 collections, naming of, 87 comma (,), 381 comments in Razor code, 57 Compare annotation, 131, 137 complex types, binding, 83-84 Compress attribute, 270, 280 consumer kev/secret, 216 container style, 425 :contains filter, 382 content delivery network (CDN), 392, 424 Content-Disposition header, 296 Content-Encoding header, 269, 270 Content folder, 177 content negotiation default formatters, 361-362 defined, 360 defining formatters for types, 362-363 HTTP headers, 360-361 ContentResult class, 26, 29, 290 Content-Type header, 360 ContentType property, 296 ContextCondition property, 442, 443 Context property, 58 ControllerBase class, 49 Controller class, 110, 263, 340, 342 ControllerContext class, 273 controller factory creating, 251-252 multiple instances and, 261 registering custom, 251

replaceable components, 256, 258 Unity-based, 252-253 controllers action filters built-in, 266 custom, 267-274 embedded, 263-264 external, 263-264 global, 266-267 types of, 265-266 action results ActionResult class, 26 mechanics of, 27-28 returning binary data, 295-297 returning custom status code, 285-287 returning HTML, 28-29 returning JavaScript, 287–288 returning JSON, 29-31, 288-290 returning JSONP, 290-292 returning PDF files, 297-299 returning primitive types, 290 returning syndication feed, 293-295 types of, 26-28 action selectors action method selectors. 276-277 action name selectors, 274-275 restricting method to Ajax calls only, 277-278 restricting method to button, 277-279 classes for action methods, 20-22 Actions and HTTP verbs, 19-20 from routing to actions, 19 from routing to controllers, 17-19 dynamic loader filter adding action filter using fluent code, 280 customizing action invoker, 280-282 enabling via filter provider, 282-285 interception points, 279-280 registering custom invoker, 282 extensibility models, 255 folder for, 226 granularity of, 16 injecting data and services "Dependency Injection" pattern, 247-248 Dependency Inversion Principle, 244-245 IoC tools, 248-249 poor man's dependency injection, 249-250 "Service Locator" pattern, 245-247 input model and, 76

#### **Controllers namespace**

keeping lean action method sample, 236 action methods coded as view model builders. 231-232 short is always better, 230-231 "Lavered Architecture" pattern application layer, 240-241 domain layer, 241-242 exposing entities of domain, 242-243 infrastructure layer, 243-244 overview, 237-238 presentation laver, 239 lavering, 16 processing input data from multiple sources, 23-24 from Request object, 22-25 from route, 23 ValueProvider dictionary, 24-25 provider-based extension model alternate TempData storage example, 257 extensibility points, 256-257 using custom components, 257-259 routing requests attribute routing, 15 defining application routes, 11-12 for physical files, 14 preventing for defined URLs, 14-15 processing routes, 12-13 route handler, 13-14 simulating ASP.NET MVC runtime, 4-7 URL patterns and routes, 10 URL Routing HTTP module, 7-9 Service Locator extension model in ASP.NET MVC, 260-261 dependency resolvers, 261–262 vs. Dependency Injection, 259-260 Session object and, 156 stateless components, 16 stereotypes for Controller stereotype, 228-229 Coordinator stereotype, 229-230 request execution and, 227-228 Responsibility-Driven Design, 226-227 testability, 17 vs. Web API advantages of Web API, 340-341 Controller class, 340 overview, 339 RESTful applications, 341 Controllers namespace, 18

Controller stereotype, 228-229 controls assigning same ID to, 85 properties for, 77 controls attribute, 407 Convention-over-Configuration pattern, 70 Cookies collection, 22, 25 cookie support capability, 452 Coordinator stereotype, 227, 229-230 CORS (cross-origin resource sharing), 358-359 CountrylsValid method, 111 coupling and testability, 307 CreateController method, 251 CreateMetadata method, 124 Create/Read/Update/Delete. See CRUD CreateRequest method, 211 CreateUser method, 198 Credentials property, 170 cross-origin resource sharing (CORS), 358 cross-property validation, 135-137 CRUD (Create/Read/Update/Delete) REST and, 341, 342 validation and, 130 .cshtml files, 29, 39 css capability group, 448 CSS (Cascading Style Sheets) Editor/Display helpers and, 120 jQuery, 378, 379 media queries, 408-410 Media Oueries Level 4, 410 RWD and, 439 view templates, 39 CssMinify transformer, 397 culture changing programmatically, 183-186 validation based on. 140-141 CultureAttribute class, 185, 271 Culture property, 58, 140, 182 curly brackets { }, 10 CurrentCulture property, 183 Current property, 124 CurrentUICulture property, 183 custom action filters adding response header, 267-268 compressing response, 268-271 view selector, 271-275 custom action results returning binary data, 295-297 returning JSONP, 290-292 returning PDF files, 297-299 returning syndication feed, 293-295

custom binders, 80 custom helpers Ajax helper example, 53 HTML helper example, 52 MvcHtmlString wrapper object, 51–53 structure of, 51 CustomValidation annotation, 131, 136–138, 143 custom view engines, 71–72

# D

data access layer, 321 DataAnnotationsModelMetadataProvider class. 124 DataAnnotationsModelValidatorProvider class, 131 data-\* attributes. 414-415 databases, localized resources, 186 data-collapsed attribute, 422 data-content-theme attribute, 422 data-driven tests, 313-314, 327-328 data entry patterns Post-Redirect-Get pattern overview, 112-113 saving data across redirects, 114-117 splitting POST and GET actions, 113–114 syncing content and URL, 112-113 updating via POST, 113-114 Select-Edit-Post pattern editing data, 106-108 overview, 104-105 presenting data, 105-106 saving data, 108-111 data-filter attribute, 419 data-icon attribute, 416 data-id attribute. 417 data-inset attribute, 419 datalist element, 405 data model, 76 data-position attribute, 417 data-role attribute, 415-416, 418 DataSource attribute, 313 data-theme attribute, 414-415 data-title attribute, 416 data-toggle attribute, 429 data-transfer objects (DTOs), 242, 342 DataType annotation, 118, 123 Date object, 368 DateTime model binder code for, 99-101 controller method, 98-99

displayed data, 97-98 overview, 96 date type, 404-405 DbContext class, 203 dblclick event, 385 DDD (Domain-Driven Design), 241 DDR (Device Description Repository), 446 debugging vs. testing, 302 decision coverage, 318 declarative helpers, 63-65 DefaultActionInvoker class, 256 DefaultBundleOrderer class, 395 DefaultControllerFactory class, 251, 256 DefaultDisplayMode class, 442, 443 default model binder aliasing parameters, 92-93 Bind attribute, 91 binding collections, 84-89 binding complex types, 83-84 binding content from uploaded files, 89-91 binding primitive types, 80-81 blacklist of properties, 92 class for, 80, 84, 93 optional values, 81-82 value providers, 82 whitelist of properties, 92 defaultRedirect attribute, 172 default route, 173 defer attribute. 391 DELETE method (HTTP) expected behavior for Web API, 347-348 REST and, 341 dependencies, testing data access code, 327-328 fake objects, 326-327 mock objects, 326-327 Dependency Injection. See DI DependencyResolver class, 262 dependency resolvers, 125, 261-262 design and testability coupling and, 307 Design for Testability control, 303 simplicity, 303-304 visibility, 303 interface-based programming, 304-306 object-oriented design and, 308-309 relativity of testability, 306-307 <details> element, 401, 402

#### **Developer Tools (IE)**

Developer Tools (IE), 441 device-aspect-ratio property, 409 Device Description Repository (DDR), 446 <device> elements, 447 device-height property, 409, 410 device identification accuracy vs. performance, 458-459 overview, 451-452 processing HTTP request, 456 virtual capabilities, 457-458 device\_os capability, 451, 452 device os version capability, 451, 452 device-width property, 409, 410 DfT (Design for Testability) control, 303 defined, 302 simplicity, 303-304 visibility, 303 dictionaries, in JavaScript, 371 Dictionary values, 256, 258 DI (Dependency Injection) overview, 247-248 poor man's dependency injection, 249-250 vs. Service Locator extension model, 259-260 die method, 386 **DIP** (Dependency Inversion Principle) overview, 244-245 SOLID principles, 306 DisableCors attribute, 359 display capability group, 448 DisplayColumn attribute, 130 DisplayConfig class, 460 DisplayFor HTML helper, 49 DisplayFormat annotation, 118 DisplayForModel HTML helper, 49, 51, 119, 125 Display HTML helper, 49 DisplayModeProvider class, 441 display modes built-in support for mobile views, 440-441 custom, 444-446 default configuration for mobile views, 441-442 defined, 271 example using, 461-462 listing available, 443-444 matching rules, 461-462 naming, 442 overview, 440 selecting, 459-461 DisplayName annotation, 118

display templates folder for, 122 for forms, 117 distributed caching, 161-162 <div> elements, 401 DLR (Dynamic Language Runtime), 66 DNOA (DotNetOpenAuth) library, 209 Document Object Model. See DOM document root object, 386 Domain-Driven Design (DDD), 241 domain laver exposing entities, 242-243 overview, 241-242 testina, 321 domain model, 75, 242 DOM (Document Object Model) control IDs. 85 querying with jQuery, 377-379 readiness of, 386-387 selectors, 379 SPAs, 367 dot(.), 380 DotNetOpenAuth (DNOA) library, 209 DOTX files, 298 drm capability group, 448 dropdown class, 428 dropdown-header class, 429 DropDownListFor HTML helper, 43 DropDownList HTML helper, 43 drop-down menus, 427-429 DTOs (data-transfer objects), 242, 342 Duration property, 162 Dynamic Language Runtime (DLR), 66 dynamic loader filter adding action filter using fluent code, 280 customizing action invoker, 280-282 enabling via filter provider, 282-285 interception points, 279-280 registering custom invoker, 282

# Ε

Echo method, 23 ECMA-262 standard, 368 EcmaScript, 368 Edit-and-Post pattern, 104 Edit method, 20 EditorForModel HTML helper, 51, 119, 125, 132 Editor helpers, 49–51 editor templates folder for, 122 for forms, 117 ELMAH (Error Logging Modules And Handlers), 170 email type, 403, 405 embedded action filters, 263-264 Embedded Resource build action, 178 :empty filter, 382 EmptyResult class, 26, 29, 290 em unit. 412 EnableClientValidation method, 48 EnableCorsAttribute class, 359 EnableOptimization property, 395 EnableUnobtrusiveJavaScript method, 48 Encode method, 48 endpoints, HTTP, 339 Engines collection, 35, 40 engines, JavaScript, 368 Enterprise Library Validation Application Block, 142 Entity Framework, 203, 242, 243 EnumDataType annotation, 131, 133 :eg filter, 381 Equals method, 137 ErrorController class, 174 Error event, 169 error handling exceptions HandleError attribute, 167–168 handling directly, 163-165 intercepting model-binding exceptions, 171-172 overriding OnException method, 165-168 route exceptions, 171-172 global error handling from global.asax, 169-170 using HTTP module, 170 input validation messages, 134–135 missing content catch-all route, 172-174 overriding IIS policies, 174 overview, 163-164 Error Logging Modules And Handlers (ELMAH), 170 ErrorMessage property, 134 ErrorMessageResourceName property, 135 ErrorMessageResourceType property, 135 Error object, 368 error pages, built-in still showing, 168 ErrorViewModel class, 173 :even filter, 381 EvenNumber attribute, 140, 144

Event object, 385 events, iOuerv binding and unbinding, 384-385 DOM readiness and, 386-387 live event binding, 385-386 Exception class, 164 ExceptionContext class, 166 exception filters, 265 exception handling HandleError attribute, 167-168 handling directly, 163-165 intercepting model-binding exceptions, 171-172 invalid values passed, 81 overriding OnException method, 165-168 testing for, 312 Exclude attribute, 91-92 ExecuteAsync method, 344 ExecuteResult method, 285, 292, 296 ExpectedException attribute, 313 ExtendedMembershipProvider class, 200 extensibility models for controllers, 255 extensibility point, 255 Extensible Markup Language. See XML external action filters, 263-264 ExternalLoginCallback method, 219, 222 ExternalLogin method, 218 ExternalLoginResult class, 218 ExtraData property, 222

#### F

Facebook authentication using, 194 Client SDK for C#, 222 social authentication importance, 215 SSO, 214 fake objects testing dependencies, 326-327 unit testing, 315-316 Fakes, 308 fall\_back attribute, 447 feature detection defined, 439 overview, 407-408 <figure> element, 401 FileContentResult class, 26-27, 296 FileDownloadName property, 296 FileExtensions property, 54–55 FilePathResult class, 26-27, 296-297

#### **FileResult class**

FileResult class. 296 Files collection, 82 FileSetOrderList property, 396 FileStream property, 297 FileStreamResult class, 26-27, 296 file uploads, 90 FilterAttribute class, 266-267 Filter class, 284 FilterInfo class. 279 filters, authentication, 194-195 FilterScope enumeration, 284 filters, jQuery overview, 381-383 vs. find, 383-384 find method, jQuery, 383-384 FindPartialView method, 36-37 FindView method, 36-37 Firefox, 367-368 :first-child filter, 382 :first filter, 381 Flags attribute, 20 flash lite capability group, 448 Flickr, 208 fluid lavout jQuery Mobile, 420-421 overview, 411-412 focus event, 385 footer HTML element, 401 iOuerv Mobile, 416-418 FormattedModelValue property, 120 formatters default, 361-362 defined, 340 defining for types, 362-363 Formatters collection, 361 Form collection, 22, 25, 82 <form> element, 46 forms data entry patterns Post-Redirect-Get pattern, 112-117 Select-Edit-Post pattern, 104–111 HTML helpers, 44-46 templates annotating data members, 117-120 custom, 122-124, 126-129 default templates, 120-121 display and editor templates, 117 nested models, 128-129

read-only members, 123-125 tabular templates, 126-128 validating input centralized validation advantages, 143 client-side validation, 139-140 cross-property validation, 135–137 culture-based validation. 140-141 custom validation attributes, 137-139 decorating model class, 132-133 enumerated types, 133-134 error messages, 134-135 IClientValidatable interface, 143-145 IValidatableObject interface, 142–143 overview, 130-131 server-side validation, 145-148 validating properties remotely, 141-142 validation provider, 130-131 FormsAuthentication class, 198 FormsCookieName property, 214 forward slash (/), 10, 13 Foundation framework, 424 <frame> element, 403 FriendlyIdentifierForDisplay property, 211 full\_flash\_support capability, 452 functional programming, 372 function coverage, 318 function filter, 383 functions (JavaScript), 372-373 Function type, 368

# G

Gecko JavaScript engine, 368 geolocation, 183, 186 GetAllCapabilities method, 457 GetAll method, 244, 345 GetCapability method, 457 GetControllerInstance method, 251 GetControllerSessionBehavior method, 251 GetControllerType method, 251 GetDeviceForRequest method, 456, 459 GetDeviceInfo method, 465 GetFilters method, 279, 283 get function, 379 GetGlobalResourceObject method, 178 GetHttpHandler method, 14 GetHttpMethodOverride method, 276 getJSON function, 293 GetLocalizedUrl method, 324

GET method (HTTP) defined, 20 example using, 85 Post-Redirect-Get pattern, 113-114 simulating in test, 329 Web API, 341, 345 GetOrderByld method, 15 GetPreferredEncoding method, 270 GetSingle function, 244 GetUser method, 198 GetValue method, 24-25 GetWebResourceUrl method, 179 gif capability, 452 global action filters, 266-267 global.asax file Cache object, 157 global error handling from, 169–170 model binders, 96 registering custom filters, 283 routes, 12 global error handling from global.asax, 169-170 using HTTP module, 170 GlobalFilters collection, 267 glyph icons, 427 Google case for URLs, 154 OpenID protocol, 208 Google Chrome, 367-368, 402, 404 granularity of controllers, 16 greyscale capability, 452 grid system, 425-426 :gt filter, 381 Gumby framework, 424 gzip compression, 269, 392

# Η

HandleError attribute, 167–168, 266 has\_cellular\_radio capability, 452 :has filter, 382 header HTML5 element, 401 jQuery Mobile, 416–418 :header filter, 381 Headers collection, 25 headers, HTTP, 360–361 HEAD requests, 348 height property, 409–410

@helper keyword, 64 HiddenFor HTML helper, 43 Hidden HTML helper, 43 HiddenInput annotation, 118, 130 hoisting in JavaScript, 370-371 HomeController class, 341 Href property, 58 HTML5 (Hypertext Markup Language 5) audio and video, 406-407 <datalist> element, 405 input types, 403-405 local storage, 406 native collapsible element, 402-403 semantic markup, 400-401 HtmlHelper class, 43, 48, 51 HTML helpers basic helpers action links, 46-47 forms. 44-46 HtmlHelper class, 48 input elements, 46 partial views, 47 custom helpers Ajax helper example, 53 HTML helper example, 52 MvcHtmlString wrapper object, 51-53 structure of, 51 overview, 42-43 templated helpers Display helpers, 49 Editor helpers, 49-51 types of, 48-49 HTML (Hypertext Markup Language) returning from action result, 28-29 viewport meta attribute, 453 Html property, 58 html ui capability group, 448 HttpApplication class, 169 HttpClient class, 351 HttpConfiguration class, 344, 359 HTTP context, mocking Cache object, 333-335 HttpContext object, 329 overview, 328-329 Request object, 329-330 Response object, 330-331 Session object, 331-333 HttpCookieCollection class, 331 HttpDelete attribute, 353, 355

#### **HttpGet** attribute

HttpGet attribute, 19-20, 85, 353, 355 HTTP handler behavior of, 5-6 invoking, 6-7 HTTP (Hypertext Transfer Protocol) 200 response code, 347 202 response code, 348 204 response code, 347 301 response code, 152-153, 155 302 response code, 8, 113, 152, 172 400 response code, 163 403 response code, 172, 286 404 response code, 8, 172-173 500 response code, 163, 345 compression, 268 CRUD operations and, 341 endpoints, 339 headers and content negotiation, 360-361 HEAD requests, 348 online resources, 152 returning custom status code, 285-287 stateless protocol, 107 verbs for, and Actions, 19-20 HttpMethodOverride method, 46, 48 HTTP module, 170 HttpNotFoundResult class, 26-27 HttpPost attribute, 19–20, 85, 353, 355 HttpPostedFileBase class, 89 HttpPostedFileBaseModelBinder class, 95 HttpPut attribute, 19, 353, 355 HttpRequestBase class, 276 HttpRequest class, 152, 276 HttpResponse class devising URLs, 153-154 HttpContext and, 152 mocking, 330 permanent redirection, 152-153 trailing slash, 154-155 HttpResponseMessage class, 346–347 httpRuntime section, 90 HttpSessionState class HttpContext and, 152 overview, 155 HTTPS (Hypertext Transfer Protocol Secure), 357 HttpStatusCodeResult class, 285 HttpUnauthorizedResult class, 26-27, 286 HttpVerbs enum, 20 Hypertext Markup Language. See HTML Hypertext Markup Language 5. See HTML5

Hypertext Transfer Protocol. See HTTP Hypertext Transfer Protocol Secure (HTTPS), 357

# 

IActionFilter interface, 263, 265 IAuthenticationFilter interface, 194, 265 IAuthenticationManager interface, 203 IAuthorizationFilter interface, 265 IBundleOrderer interface, 395 IBundleTransform interface, 397 ICacheService interface, 158, 333 IClientValidatable interface, 140, 143–145 IControllerFactory interface, 251 id attribute, 447 IdentityDbContext class, 203 identity system, 201-204 IdentityUser class, 202 IDevice interface, 461 IE. See Internet Explorer <i> element, 403 IExceptionFilter interface, 167, 265 <iframe> element, 403 if statements, 52 Ignore attribute, 312 IgnoreList property, 397 IHtmlString interface, 52 IHttpHandler interface, 5 IHttpRouteConstraint interface, 355 IIS (Internet Information Services) HTTP compression, 268 requests and, 8 rewrite module, 155 security mechanisms and, 189 version 7.5, 91 Virtual Accounts feature, 91 Web API and, 341 ILogger interface, 305, 315 image\_format capability group, 448 image\_inlining capability, 452–453 images, in Bootstrap, 427 img-responsive class, 427 <img> tag, 393 IModelBinder interface, 93–94, 256 IModelBinderProvider interface, 256 @Import directive, 68 Include method, 395 Include property, 91–92 index parameter, 345-346

Information holder stereotype, 227 infrastructure laver, 243-244 Inherits attribute, 68 injecting data and services "Dependency Injection" pattern, 247-248 Dependency Inversion Principle, 244-245 loC tools, 248-249 poor man's dependency injection, 249-250 "Service Locator" pattern, 245-247 InMemoryConfigurer class, 455 <input> element, 403 input elements, 46 input model defined, 75 model binders, 78 server controls role, 76-77 view state, 77-78 input types in HTML5, 403-405 input validation. See validation, input Instance property, 456 IntelliSense, 376 interface-based programming, 304-306 Interfacer stereotype, 227 Interface Segregation Principle, 199, 306 InternalsVisibleTo attribute, 317 Internet Explorer Developer Tools, 441 HTML5 and, 402, 404 JavaScript engine, 368 Internet Information Services. See IIS intrinsic objects Cache object caching method response, 161-162 distributed caching, 161-162 injecting caching service, 158-160 OutputCache attribute, 161–162 overview, 157 partial caching, 162 pros and cons, 157-158 HttpResponse class devising URLs, 153-154 permanent redirection, 152-153 trailing slash, 154-155 overview, 151-152 Session object controller and, 156 overview, 155-156 InvalidOperationException, 165

InvokeActionMethodWithFilters method, 279-280 IoC (Inversion of Control) extensibility model and, 255 injecting caching service, 158 Select-Edit-Post pattern, 105 tools for, 248-249 iOS. 399 IP addresses, 186 IResourceProvider interface, 187 IResultFilter interface, 265 IRouteHandler interface, 13 IsAjax property, 58 is android capability, 458 is\_app capability, 458 is full desktop capability, 458 is html preferred capability, 458 is\_ios capability, 458 is largescreen capability, 458 is\_mobile capability, 458 IsMobileDevice method, 444 ISO/IEC 16262:201 standard, 368 isolation, testing in, 314 IsPost property, 59 is\_robot capability, 458 IsSectionDefined method, 62 is\_smartphone capability, 458 is smartty capability, 451 is tablet capability, 450, 451, 452 is\_touchscreen capability, 458 IsValidForRequest method, 277 IsValid method, 138 is\_windows\_phone capability, 458 is\_wireless\_device capability, 451, 452, 459 is wml preferred capability, 458 is\_xhtmlmp\_preferred capability, 458 ITempDataProvider interface, 256 iTextSharp, 298 IUseFixture interface, 311 IUser interface, 201 IUserStore interface, 202 IValidatableObject interface, 142-143 IValueProvider interface, 25, 256 IViewEngine interface, 36, 256 IViewSelector interface, 273 IWURFLConfigurer interface, 455

#### j2me capability group

#### J

j2me capability group, 448 JavaScript bundlina overview, 394-395 resources, 395 script files, 395-397 invoking Web API from, 349-350 iQuery DOM queries, 377-379 events, 384-386 overview, 377 root object, 377-378 selectors, 379-384 wrapped sets, 378-379 language functions, 372-373 hoisting, 370-371 local and global variables, 369-370 null vs. undefined values, 369 objects, 371-372 type system, 368 loading scripts and resources download is synchronous, 391 scripts at bottom, 391 sprites, 393-394 static files, 392 minification, 397 object-orientation in closures, 374-376 making objects look like classes, 374 prototypes, 375-376 overview, 367 packaging for reuse Module pattern, 389-391 Namespace pattern, 388-389 returning using action result, 287-288 unobtrusive code, 387-388 JavaScript Object Notation. See JSON JavaScript Object Notation with Padding (JSONP), 290-292 JavaScriptResult class, 26-27, 287 JavaScriptSerializer class, 29, 288, 350 jpg capability, 452 jQM. See jQuery Mobile jQuery. See also jQuery Mobile Ajax calls, 293 Bootstrap requirements, 425 chained queries, 378

client-side validation, 403 date picker, 96 documentation, 377 DOM queries, 377-379 events binding and unbinding, 384-385 DOM readiness and, 386-387 live event binding, 385-386 globalization plugin, 140 in Ajax helper, 53 Mobile framework, 399 modal dialogs, 104 older browsers and, 407 overview, 377 prototypes and, 376 root object, 377-378 selectors basic selectors, 379-380 chaining operations on wrapped set, 384 compound selectors, 380-381 filters, 381-383 filter vs. find, 383-384 Validation plugin, 116, 140, 144 wrapped sets, 378-379 jQuery Mobile collapsible panels, 422-423 data-\* attributes, 414-415 fluid layout, 420-421 header and footer, 416-418 lists, 418-421 overview, 413-414 pages in, 415-416 themes, 414 JScript, 368 JScript.NET, 368 JSLint, 370, 373 JsMinify transformer, 397 JsonCamelCaseFormatter class, 361 JSON (JavaScript Object Notation) formatters, 340 returned by method, 21 returning from action result, 29-31, 288-290 return payload, 141 WCF and, 338 Web storage, 406 JsonMediaTypeFormatter class, 361 Json method, 289, 340 JSONP (JavaScript Object Notation with Padding), 290-292 JsonpResult class, 291, 293 JsonResult class, 27, 288, 340

#### Microsoft.AspNet.Identity.EntityFramework namespace

# K

Kendo UI, 414, 423 keyup event, 385 KnockoutJS, 349, 367

# L

LabelFor HTML helper, 43 Label HTML helper, 43 lambda expressions, 117 LanguageController class, 185 :last-child filter, 382 :last filter, 381 latency, 186 layer, defined, 237 "Layered Architecture" pattern application layer, 240-241 domain layer, 241-242 exposing entities of domain, 242–243 infrastructure layer, 243-244 overview, 237-238 presentation layer, 239 layering controllers, 16 layout breakpoint, 408 Layout property, 59, 60 Leaner CSS (LESS), 410, 424 Least-Recently-Used (LRU) algorithm, 457 length property, 379 LESS (Leaner CSS), 410, 424 LinkedIn authentication using, 194 SSO, 214 k> element, 154, 410 Liskov's Substitution Principle, 306 ListBoxFor HTML helper, 43 ListBox HTML helper, 43 lists, using jQuery Mobile, 418-421 listview role, 418 live event binding using jQuery, 385-386 live method, 386 localization auto-adapting applications, 182-183 changing culture programmatically, 183-186 files for, 177-178 getting localized data from service, 187 multilingual applications, 183 referencing embedded files, 178-180 storing resources in database, 186 text, 175-177

unit testing, 323–325 views, 180–181 local storage, HTML5, 406 localStorage property, 406 local variables in JavaScript, 369–370 Location property, 162 Logoff method, 196 Logon Method, 196 LogonViewModel class, 198 LRU (Least-Recently-Used) algorithm, 457 :lt filter, 381

# Μ

maintainability, 304 MapHttpAttributeRoutes method, 354 MapRoute method, 11 marketing\_name capability, 451 markup capability group, 448 MasterLocationFormats property, 54, 55 Master property, 168, 193 master view overview, 42 Razor view engine, 60-61 matching rules for display modes, 443, 461-462 Math object, 368 max image width capability, 451 maxRequestLength attribute, 90 media attribute, 409 Media Queries Level 4, CSS, 410 MediaTypeFormatter class, 361 Membership class, 198 MembershipProvider class, 198 membership system (authentication) identity system, 201-204 integrating with roles, 200-201 Membership API, 198–199 overview, 195 SimpleMembership API, 200–201 social authentication and, 220-221 validating user credentials, 196-198 Memcached, 157 MemoryCache class, 161 message handlers, 357 Message Queuing (MSMQ), 338 metadata provider, 117 Microsoft.AspNet.Identity.Core namespace, 202 Microsoft.AspNet.Identity.EntityFramework namespace, 202

#### **Microsoft IntelliSense**

Microsoft IntelliSense, 376 Microsoft Internet Explorer. See Internet Explorer Microsoft Internet Information Services. See IIS Microsoft.Owin.Security namespace, 203 Microsoft Passport, 208 MIME (Multipurpose Internet Mail Extensions), 453 minifying JavaScript, 397 missing content catch-all route, 172-174 overriding IIS policies, 174 mms capability group, 448 mobile.browser file, 444 mobile-first approach, 413 mobile-friendly websites Bootstrap button groups, 429-430 drop-down menus, 427-429 alvph icons, 427 grid system, 425-426 images, 427 navigation bars, 426-427 overview, 423-424 setting up, 424-425 HTML5 audio and video, 406-407 <datalist> element, 405 input types, 403–405 local storage, 406 native collapsible element, 402-403 semantic markup, 400-401 jQuery Mobile collapsible panels, 422-423 data-\* attributes, 414-415 fluid layout, 420-421 header and footer, 416-418 lists, 418-421 overview, 413-414 pages in, 415-416 themes. 414 routing users from existing site configuration files, 435-436 implementing routing, 432-434 overview, 430-431 routing algorithm, 432 tracking chosen route, 434-435 RWD CSS media gueries, 408-410 feature detection, 407-408 fluid layout, 411-412 overview, 412-413

mobile views built-in support for, 440-441 default configuration for, 441-442 mock objects mocking Cache object, 333-335 mocking HttpContext object, 329 mocking Request object, 329-330 mocking Response object, 330-331 mocking Session object, 331-333 testing dependencies, 326-327 unit testing, 315-316 modal dialogs, 104 mode attribute, 458 model bindina custom type binders creating, 94-95 customizing default binder, 93-94 reaisterina, 96 DateTime model binder code for. 99-101 controller method, 98-99 displayed data, 97-98 overview, 96 default model binder aliasing parameters, 92-93 Bind attribute, 91 binding collections, 84-89 binding complex types, 83-84 binding content from uploaded files, 89-91 binding primitive types, 80-81 blacklist of properties, 92 optional values, 81-82 value providers, 82 whitelist of properties, 92 exception handling for, 171–172 method signature, 79 model binders, 79 replaceable components, 256, 258 Model metadata, 256, 258 ModelMetadataProvider class, 125, 256 model name capability, 451, 452 Model property, 59 models. See also model binding data model, 76 domain model, 75 input model defined, 75 model binders, 78 server controls role, 76–77 view state, 77-78

#### object\_download capability group

modeling view packaging view-model classes, 70 strongly typed view models, 67-70 ViewBag dictionary, 66-67 ViewData dictionary, 65–66 types of, 78 view model. 75 ModelState dictionary, 110, 115, 132 @ModelType keyword, 60 Model validator, 256, 258 ModelValidatorProvider class, 256 Model-View-Controller pattern. See MVC pattern Modernizr, 367 Module pattern, 389-391 Moles, 308, 333 MooTools, 376 Mog, 316, 326 MOV codec. 407 Mozilla Firefox, 367-368 MP4 codec, 407 MSMQ (Message Queuing), 338 MSTest, 309, 313, 318 multi-device sites display modes built-in support for mobile views, 440-441 custom, 444-446 default configuration for mobile views, 441-442 example using, 461-462 listing available, 443-444 matching rules, 443, 461-462 naming, 442 overview, 440 selecting, 459-461 server-side solution advantages, 466-467 WURFL database capability groups, 448-449 Cloud API, 462-465 detecting device capabilities, 456-459 identifying current device, 451-452 initializing runtime, 456 installing NuGet package, 454-455 overview, 446-447 patch files, 450-451 referencing device database, 455-456 serving browser-specific content, 452-453 XML schema, 447 multilingual applications, 183. See also localization Multipurpose Internet Mail Extensions (MIME), 453

multitransport services, 338 MvcHtmlString wrapper object, 51–53 MVC (Model-View-Controller) pattern ASP.NET runtime and, 3 history, 75 myOpenID, 209

## Ν

naked domains, 155 Namespace pattern, 388-389 navbar style, 426 nav class, 426 navigation bars, 426-427 NCache, 157 nested layouts, 63-64 nested models, 128-129 NETWORKSERVICE account, 91 new operator, 373-374 next operator, 381 Ninject, 249 NMock2, 316 Node.is, 367 NonAction attribute, 19, 276, 346 NoSQL, 239 NoStore property, 162 :not filter, 381 NotSupported exception, 14 :nth-child filter, 382 NuGet, 139 NullReferenceException, 165 null type JavaScript primitive types, 368 vs. undefined type, 369 number type, 368, 405 NUnit, 309

# 0

OAuth ASP.NET identity, 204 authentication filters, 194 vs. OpenID protocol, 214–216 Web API, 358–359 OAuthWebSecurity class, 217 ObjectCache class, 161 Object.cshtml file, 121 object\_download capability group, 448

#### object model

object model, 242 object-oriented programming. See OOP Object/Relational Mapper (O/RM), 241, 244 objects in JavaScript, 371-372 Object type, 368 :odd filter, 381 off function, 386 Office automation, 298 OnActionExecuted method, 263, 265 OnActionExecuting method, 185, 263, 265, 282 OnAuthenticationChallenge method, 194 OnAuthentication method, 194, 265 OnAuthorization method, 193, 265 onclick attribute, 429 OnException method, 165-168, 265 on function, 386 onload event, 386 :only-child filter, 382 OnModelUpdated method, 94 OnPropertyValidating method, 94 OnResultExecuted method, 265 OnResultExecuting method, 265, 272 OOP (object-oriented programming) in JavaScript closures, 374-376 making objects look like classes, 374 prototypes, 375-376 testability and, 308-309 open attribute, 402 Open/Closed Principle, 306 OpenID protocol vs. OAuth, 214-216 overview, 208 using, 209-214 OpenIdRelyingParty class, 211 Open Web Interface for .NET (OWIN), 203 Opera, 368, 404 orchestration, defined, 240 orchestration layer, 321 Orderer property, 395 Order property, 266 orientation property, 409 O/RM (Object/Relational Mapper), 241, 244 OutputCache attribute, 73, 161–162, 192, 266 output caching, 192 OWIN (Open Web Interface for .NET), 203

#### Ρ

packaging JavaScript for reuse Module pattern, 389-391 Namespace pattern, 388–389 Page class, 39 pageinit event, 415, 416 parameters, aliasing, 92-93 :parent filter, 382 partial caching, 162 Partial HTML helper, 43, 47, 108, 180 PartialViewLocationFormats property, 54-55 PartialViewResult action result type, 27 partial views folder for, 71-72 HTML helpers, 47 PascalCasing, 361 PasswordFor HTML helper, 43 Password HTML helper, 43 patch files for WURFL database, 450-451 path coverage, 318 pathInfo parameter, 15 pdf capability group, 449 PDF files, returning using action result, 297-299 Performance mode, 459 Pex add-in, 319 PhoneGap, 367 pipe symbol (|), 370 placeholder attribute, 405 playback capability group, 449 png capability, 452 pointing\_method capability, 451-452 poster attribute, 407 POST method (HTTP) defined, 20 example using, 85 expected behavior for Web API, 346-347 Post-Redirect-Get pattern, 113-114 REST and, 341 simulating in test, 329 X-HTTP-Method-Override, 46 Post-Redirect-Get pattern overview, 112-113 saving data across redirects, 114-117 splitting POST and GET actions, 113–114 syncing content and URL, 112-113 updating via POST, 113-114 PostResolveRequestCache event, 9 preferred\_markup capability, 452-453 Prefix attribute, 91, 92

presentation layer, 238, 239 PRG pattern. See Post-Redirect-Get pattern primitive types binding with default model binder, 80-81 returning using action result, 290 PrivateObject class, 318 product info capability group, 449 properties, whitelist/blacklist of, 92 proportional layout, 412 prototypes in JavaScript, 375-376 performance, 376 provider-based extension model alternate TempData storage example, 257 extensibility points, 256-257 using custom components, 257-259 PUT method (HTTP) expected behavior for Web API, 347 REST and, 341

# Q

QueryString collection, 22, 25, 82

# R

RadioButtonFor HTML helper, 43 RadioButton HTML helper, 43 RAD (Rapid Application Development), 301 Range annotation, 131 Raw HTML helper, 57 Raw method, 48 RawValue property, 25 Razor view engine code nuggets conditional nuggets, 58 overview, 55-57 special expressions of, 57 comments, 57 declarative helpers, 63-65 default for ASP.NET MVC 5, 35 master view, 60-61 model for view, 59 nested layouts, 63-64 Razor view object, 58-59 search locations, 54-55 sections default content for, 62 overview, 61-62

RazorViewEngine class, 441 RDD (Responsibility-Driven Design) defined, 105 overview. 226-227 readability, 304 ReadFromStreamAsync method, 361 ReadOnly attribute, 118, 123 read-only members, 123-125 ready event, 386, 415 readyState property, 386 Really Simple Syndication. See RSS Redirect method, 152 RedirectPermanent method, 153 RedirectResult class, 27, 153, 325 redirects, HTTP permanent, 152-153 Post-Redirect-Get pattern, 114–117 unit testina, 325 RedirectToRouteResult class, 27, 325 refreshing page, avoiding with Ajax, 116 RegExp object, 368 RegisterCacheService method, 160 RegisterDisplayModes method, 460–461 RegisterRoutes method, 11, 18, 325 RegularExpression annotation, 131 ReleaseController method, 251 ReleaseView method, 36-37 RelyingParty property, 211 Remember-Me feature and Ajax reproducing problem, 204-205 solving, 205-207 Remote annotation, 131, 141 remote procedure calls. See RPC remote property validation, 141-142 RenderAction HTML helper, 72, 162 render actions, 72-73, 156 RenderBody method, 60-61 RenderPage method, 62 RenderPartial HTML helper, 43, 47, 62, 108, 180 RenderSection method, 61-62 RepeatWithPrecedence method, 82 replaceable components listing of, 256 registering, 258 Reporting Services, 298 Representational State Transfer. See REST RequestAuthentication method, 218 Request class mocking, 329-330 processing input from, 22-25

#### **RequestContext class**

RequestContext class, 13, 324 Required annotation, 127, 131 RequireHttps filter, 266 ResetAll, 396 resolution height capability, 451-452 resolution width capability, 451-452 resources folder for, 176 separate assembly for, 177 storing in database, 186 Response class, 330-331 response format ASP.NET MVC approach, 359-360 content negotiation default formatters, 361-362 defining formatters for types, 362-363 HTTP headers, 360-361 Responsibility-Driven Design, See RDD Responsive Web Design. See RWD **REST (Representational State Transfer)** application routes and, 9 vs. RPC, 352 Web API ApiController class, 344 vs. MVC controllers, 341 naming conventions, 346 resource type, 342-343 routing to action methods, 344-346 result filters. 265 .resx files, 175 rewrite module, 155 RFC 2616, 270 Rhino Mocks, 316 Rich Site Summary (RSS), 293 RoleProvider class, 200 roles authorization of, 191 integrating membership system, 200-201 RouteData collection, 13, 23, 82 route exceptions, 171-172 RouteExistingFiles property, 14 route handler overview, 13-14 processing input from, 23 RouteLink HTML helper, 18, 43, 46 Routes collection, 11 routing to action methods REST, 344-346 RPC, 353

from existing website to mobile-friendly configuration files, 435-436 implementing routing, 432-434 overview, 430-431 routing algorithm, 432 tracking chosen route, 434-435 unit testing routes, 325-326 routing requests attribute routing, 15 defining application routes, 11-12 for physical files, 14 preventing for defined URLs, 14-15 processing routes, 12-13 route handler, 13-14 simulating ASP.NET MVC runtime behavior of HTTP handler, 5-6 invoking HTTP handler, 6-7 syntax of recognized URLs, 4-5 subdomains and, 47 URL patterns and routes, 10 **URL Routing HTTP module** internal structure of, 9 routing requests, 8-9 superseding URL rewriting, 7-8 row style, 425 RPC (remote procedure calls) action attributes, 353 attribute routing enabling, 354-355 overview, 353-354 routing to action methods, 353 vs. REST, 352 Web API and, 337 rss capability group, 449 RSS (Rich Site Summary), 293 rulesets, VAB, 145 runtime, simulating behavior of HTTP handler, 5-6 invoking HTTP handler, 6-7 syntax of recognized URLs, 4-5 RWD (Responsive Web Design) CSS media queries, 408-410 feature detection, 407-408 feature-detection and, 439 fluid layout, 411-412 mobile-friendly technologies, 399 overview, 412-413 pitfalls, 466

#### sms capability group

# S

same-origin policy, 358 ScaffoldColumn attribute, 130 ScaleOut, 157 ScientiaMobile, 447 ScriptBundle class, 397 <script> element, 288, 293, 391, 416 Script property, 287 scripts, loading of bottom of page, 391 download is synchronous, 391 SearchedLocations property, 37 search engine optimization. See SEO sections, Razor view engine, 61-62 security authentication authentication filters, 194-195 configuring, 190 membership system, 195-204 OpenID protocol, 208-215 Remember-Me feature and Ajax, 204-207 using social networks, 215-223 authorization action methods restrictions, 190-191 allowing anonymous callers, 191 anonymous vs. not authorized, 193 hiding user interface elements, 192 output caching and, 192 Web API access tokens, 357 basic authentication, 357-358 CORS, 358-359 host handles, 355-356 OAuth, 358-359 security capability group, 449 segmented buttons, 429 Select-Edit-Post pattern defined, 104 editing data, 106-108 overview, 104-105 presenting data, 105-106 saving data, 108-111 selectors, jQuery basic selectors, 379-380 chaining operations on wrapped set, 384 compound selectors, 380-381 filters, 381-383 filter vs. find, 383-384

self-validation centralized validation advantages, 143 IClientValidatable interface, 143-145 IValidatableObject interface, 142–143 server-side validation, 145-148 semantic markup of HTML5, 400-401 Sencha, 414, 423 SEO (search engine optimization) case for URLs, 154 HttpResponse class and devising URLs, 153-154 permanent redirection, 152-153 trailing slash, 154-155 subdomains, 155 URLs for, 7 Separation of Concerns (SoC), 16, 230 serialization, 340 server controls, 43 server-side validation, 141, 145-148 ServerVariables collection, 22 service layer, 238 Service Locator extension model in ASP.NET MVC, 260-261 dependency resolvers, 261-262 vs. Dependency Injection, 259-260 "Service Locator" pattern, 245-247 Service-Oriented Architecture (SOA), 240, 338 Service provider stereotype, 227 Session object controller and, 156 mocking, 331-333 overview, 155-156 saving temporary data, 114 sessionStorage, 406 setAction function, 278 SetResolver method, 262 Shared folder, 42 SimpleMembership API, 200-201 Simple Object Access Protocol (SOAP), 338 single-page applications (SPAs), 239 Single-Page Applications (SPAs), 367 Single Responsibility Principle, 306 single sign-on (SSO), 214 size function, 379 Skeleton framework, 424 <small> element, 427 smartphones, 399, 400, 436 smarttv capability group, 449 Smart TVs, 436 sms capability group, 449

#### SmtpClient class

SmtpClient class, 170 SOAP (Simple Object Access Protocol), 338 SOA (Service-Oriented Architecture), 240, 338 social authentication access tokens, 221-223 enabling, 217-218 importance of, 215 membership system, 220-221 popularity, 189 registering application with Twitter, 215-216 starting process, 218-219 Twitter response, 219 SoC (Separation of Concerns), 16, 230 SOLID principles, 199, 230, 259, 306 SortEncodings method, 271 sound\_format capability group, 449 <span> element, 427 span style, 425 Spark view engine, 28 SPAs (Single-Page Applications), 239, 367 split method, 368 Spring.NET, 249 sprites, 393-394 SqlDependency property, 162 SSO (single sign-on), 214 stateless components, 16 statement coverage, 318 stereotypes for controllers Controller stereotype, 228-229 Coordinator stereotype, 229–230 request execution and, 227-228 Responsibility-Driven Design, 226-227 StopRoutingHandler class, 14 storage capability group, 449 streaming capability group, 449 StringBuilder class, 43 StringLength annotation, 131 string type, 368 StructureMap, 249 Structurer stereotype, 227 StyleBundle class, 397 subdomains routing and, 47 SEO and, 155 substring method, 368 <summary> element, 402 SwitchToErrorView method, 166 syndication feed, returning using action result, 293-295 SyndicationItem class, 295

SyndicationResult class, 293, 295 System.ComponentModel.DataAnnotations namespace, 118 System.ComponentModel namespace, 118 System.Net.Http namespace, 351 System.ServiceModel assembly, 293 System.ServiceModel.Syndication namespace, 293 System.Web.Http assembly, 342, 344 System.Web.Mvc namespace, 279 System.Web.Routing namespace, 13

#### Т

tablets, 399, 400, 436 tabular templates, 126–128 TagBuilder class, 51 tel type, 403, 405 TempData controller extensibility example, 257 replaceable components, 256, 258 saving data across redirects, 114-115 TemplateDepth property, 129 templated helpers Display helpers, 49 Editor helpers, 49-51 types of, 48–49 TemplateHint property, 124 TemplateInfo property, 121 templates, for forms annotating data members, 117-120 custom, 122-124, 126-129 default templates, 120-121 display and editor templates, 117 nested models, 128-129 read-only members, 123–125 tabular templates, 126–128 testability. See testing TestClass attribute, 311 TestCleanup attribute, 311 TestContext variable, 313 test doubles, 314, 326 test fixtures, 310-311 testing controllers testability, 17 vs. debugging, 302 dependencies data access code, 327-328 fake objects, 326-327 mock objects, 326-327

design and testability coupling and, 307 Design for Testability, 302-304 interface-based programming, 304-306 object-oriented design and, 308-309 relativity of testability, 306-307 importance of, 301-302 mocking HTTP context mocking Cache object, 333-335 mocking HttpContext object, 329 mocking Request object, 329-330 mocking Response object, 330-331 mocking Session object, 331-333 overview, 328-329 unit testing arrange, act, assert, 311-312 assertions per test, 317 choosing environment, 309-310 code coverage, 318 data-driven tests, 313-314 defined, 308-309 fakes and mocks, 315-316 inner members, 317-318 limited scope, 314 localization, 323-325 overview, 321 redirections, 325 routes, 325-326 test fixtures, 310-311 testing in isolation, 314 using test harness, 309 views, 322-323 which code to test data access layer, 321 domain layer, 321 orchestration layer, 321 overview, 319-320 TestInitialize attribute, 311 TestMethod attribute, 313 TextAreaFor HTML helper, 43 TextArea HTML helper, 43 TextBoxFor HTML helper, 43 TextBox HTML helper, 43 text, localizing, 175-177 themes for jQuery Mobile, 414 this keyword, 370 ThreadAbortException, 166 tier, defined, 237 tiff capability, 452 tilde (~), 177, 381, 455

TimeSpan class, 98 <title> element, 416 ToBool method, 462 ToInt method, 462 trailing slash, 154-155 Transact-SQL (T-SQL), 16 transcoding capability group, 449 trv/catch blocks, 163 TrySkiplisCustomErrors property, 174 T-SQL (Transact-SQL), 16 TweetSharp, 222 Twitter authentication response, 219 OAuth, 194 registering application with, 215-216 social authentication importance, 215 SSO, 214 testing authentication, 221 Twitter Bootstrap. See Bootstrap Typemock, 308, 316, 333 typeof method, 369 type system in JavaScript, 368

# U

<u> element, 403 UICulture property, 59, 182 UIHint annotation, 118, 123-124 UI (user interface) hiding elements, 192 mobile-friendly websites, 399 unbind function, 384 unbinding events in jQuery, 384-385 undefined type JavaScript primitive types, 368 vs. null type, 369 Uniform Resource Identifier (URI), 4 Uniform Resource Locators. See URLs Uniform Resource Name (URN), 4 unit testing arrange, act, assert, 311-312 assertions per test, 317 choosing environment, 309-310 code coverage, 318 data-driven tests, 313-314 defined, 308-309 fakes and mocks, 315-316 inner members, 317-318 limited scope, 314 localization, 323-325

#### Unity

overview, 321 quality of numbers, 322 redirections, 325 routes. 325-326 test fixtures, 310–311 testing in isolation, 314 using test harness, 309 views, 322-323 white-box testing, 321 Unity controller factory based on, 252-253 dependency resolver, 261 IoC frameworks, 249 online resources, 249 unobtrusive code, 387-388 untestable code, 307 URI (Uniform Resource Identifier), 4 Url.Content method, 177 UrlHelper class, 47, 324 Url property, 47 **URL Routing HTTP module** internal structure of, 9 routing requests, 8-9 superseding URL rewriting, 7-8 URLs (Uniform Resource Locators) case for, 154 defined, 4 parameters, 10 patterns and routes, 10 Post-Redirect-Get pattern, 112-113 preventing routing for defined, 14-15 SEO and, 153-154 syntax of recognized, 4-5 url type, 403, 405 URN (Uniform Resource Name), 4 user agents, 447, 452 UserData property, 212, 214 User Experience First (UXF), 240 user interface. See UI UserManager class, 201–202 UserStore class, 202 "Use-That-Not-This" pattern, 373 UXF (User Experience First), 240

#### V

V8 JavaScript engine, 368 VAB (Validation Application Block), 145 ValidateAntiForgeryToken filter, 266 ValidateInput filter, 266 Validate, iOuerv, 116 Validate method, 143 ValidateUser method, 198–199 Validation Application Block (VAB), 145 ValidationAttribute class, 131 ValidationContext parameter, 136 validation, input data annotations client-side validation, 139-140 cross-property validation, 135–137 culture-based validation, 140-141 custom validation attributes, 137-139 decorating model class, 132-133 enumerated types, 133-134 error messages, 134-135 validating properties remotely, 141-142 validation provider, 130–131 overview. 130-131 self-validation centralized validation advantages, 143 IClientValidatable interface, 143-145 IValidatableObject interface, 142–143 server-side validation, 145-148 ValidationMessageFor HTML helper, 43 ValidationMessage HTML helper, 43, 46, 109-110, 132 ValidationSummary HTML helper, 43, 137 ValueProvider dictionary, 24-25 ValueProviderResult type, 24 value providers, 82 ValuesController class, 341 var keyword, 369, 371, 376 Varnish, 392 VaryByContentEncoding property, 162 VaryByCustom property, 162 VaryByHeader property, 162 VaryByParam property, 162 .vbhtml files, 29, 39 VerifyAuthentication method, 219 video in HTML5, 406-407 ViewBag dictionary, 42, 49, 59, 66-67, 156 ViewData dictionary, 42, 59, 65-66, 68, 121, 156, 173 View engine, 256, 258 ViewEngineResult class, 37 ViewEngines class, 35 ViewLocationFormats property, 54, 55 view model, 75 ViewName property, 273 viewport meta attribute, 424, 453

viewport supported capability, 452-453 View property, 168, 193 ViewResult class, 27, 274 views child actions, 73-74 custom view engines, 71-72 HTML helpers basic helpers, 43-48 custom helpers, 51-53 overview, 42-43 templated helpers, 48-51 localizing, 180-181 modeling view packaging view-model classes, 70 strongly typed view models, 67-70 ViewBag dictionary, 66-67 ViewData dictionary, 65-66 Razor view engine code nuggets, 55-57 conditional nuggets, 58 declarative helpers, 63-65 master view, 60-61 model for view, 59 nested layouts, 63-64 Razor view object, 58-59 search locations, 54-55 sections, 61-62 sections, default content for, 62 special expressions of code nuggets, 57 render actions, 72-73 unit testing, 322-323 view engine action invoker and, 37-38 anatomy of, 36-37 detecting, 34-36 view object, 38-39 view template default conventions and folders, 39-41 master view, 42 overview, 41-42 resolving, 39 view selector, 271-275 view state, 77-78 Virtual Accounts feature, 91 virtual capabilities, 457-458 VirtualPathProviderViewEngine class, 441 Visual Studio, 195 .vsdoc.js files, 397

#### W

W3C (World Wide Web Consortium), 153, 348 wap push capability group, 449 WCF (Windows Communication Foundation), 232, 338-339 Web API asynchronous calls, 351-352 client applications and, 339 expected method behavior DELETE method, 347-348 other methods, 348 POST method, 346-347 PUT method, 347 importance of, 338 invoking from JavaScript, 349-350 invoking from server-side code, 350-351 vs. MVC controllers advantages of Web API, 340-341 Controller class, 340 overview, 339 RESTful applications, 341 response format ASP.NET MVC approach, 359-360 default formatters, 361-362 defining formatters for types, 362-363 HTTP headers, 360-361 REST ApiController class, 344 naming conventions, 346 resource type, 342-343 routing to action methods, 344-346 RPC action attributes, 353 attribute routing, 353-354 routing to action methods, 353 vs. REST, 352 security access tokens, 357 basic authentication, 357-358 CORS (cross-origin resource sharing), 358-359 host handles, 355-356 OAuth, 358-359 WCF and, 338-339 Web Forms applications, 339 WebApiConfig class, 344 web.config file adding mobile router to site, 435 authentication in, 190 client-side validation, 139

#### Web Forms

custom error flag, 171 error handling, 163 globalization section, 182 httpRuntime section, 90 IoC configuration, 262 maxRequestLength attribute, 90 Unity configuration, 252 WURFL in, 450 Web Forms moving to input model server controls role, 76-77 view state, 77-78 switching views, 274 view templates and, 41 Web API and, 339 WebFormsViewEngine class, 441 webHttpBinding binding, 338 WebMatrix, 39 WebSecurity class, 200 web service, 338 Web storage, 406 WebViewPage class, 60, 62 white-box testing, 321 whitelist of properties, 92 width property, 409-410 window object, 370 Windows 7, 91 Windows 8, 91 Windows authentication defined, 190 overview, 195 Windows Communication Foundation (WCF), 232, 338-339 Windows Phone, 399 Windows Server 2008 R2, 91 Windows Server AppFabric Caching Services, 157, 161 wml\_ui capability group, 449 worker services, 231-236 World Wide Web Consortium (W3C), 153, 348 wrapped sets, jQuery chaining operations, 384 defined, 378 overview, 378-379 WriteFile method, 296 WriteToStreamAsync method, 361 WS-\* protocols, 338

WURFL database capability groups, 448-449 Cloud API vs. on-premise API, 465 overview, 462-463 setting up, 464-465 detecting device capabilities accuracy vs. performance, 458-459 processing HTTP request, 456 virtual capabilities, 457-458 display modes example using, 461-462 matching rules, 461-462 selecting, 459-461 identifying current device, 451–452 initializing runtime, 456 installing NuGet package, 454–455 overview, 446-447 patch files, 450-451 referencing device database, 455-456 return values, 457 serving browser-specific content, 452-453 XML schema, 447 WURFLManagerBuilder class, 456

# Х

XHR (XmlHttpRequest), 116, 204, 277
XHTML MP format, 453
xhtml\_ui capability group, 449
X-HTTP-Method-Override header, 46, 276
XML (Extensible Markup Language)
 formatters, 340
 requesting format, 360
 VAB rulesets, 145
 WURFL database schema, 447
XMLHttpRequest class, 116, 204, 277
xUnit.net, 309, 311, 313

# Υ

YAGNI (You Aren't Gonna Need It) principle, 158 Yahoo!, 208