eBooks-IT.org

THE EXPERT'S VOICE® IN ASP.NET

FIFTH E

Pro ASP.NET MVC 5

BUILD THE MOST MAINTAINA
COMPLIANT, AND BEST PERI
APPLICATIONS ON THE MICR

Adam Freeman

Ap

eBooks-IT.org

Adam Freeman

Ap

eBooks-IT.org

Copyright © 2013 by Adam Freeman

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specific translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and trans information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or herea Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or pa permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4302-6529-0

ISBN-13 (electronic): 978-1-4302-6530-6

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a tradema or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or respect to the material contained herein.

President and Publisher: Paul Manning
Lead Editor: Ewan Buckingham
Technical Reviewer: Fabio Claudio Ferracchiati
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Jonatha Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dor Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Kevin Shea
Copy Editor: Larissa Shmailo
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 100 SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are al most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781430257882. For detailed information about how to locate your book's source code, go to www.apress. code/.

About the Author

About the Technical Reviewer

Chapter 1: Putting ASP.NET MVC in Context
Chapter 2: Your First MVC Application
Chapter 3: The MVC Pattern
Chapter 4: Essential Language Features
Chapter 5: Working with Razor
Chapter 6: Essential Tools for MVC
Chapter 7: SportsStore: A Real Application
Chapter 8: SportsStore: Navigation
Chapter 9: SportsStore: Completing the Cart
Chapter 10: SportsStore: Mobile
Chapter 11: SportsStore: Administration
Chapter 12: SportsStore: Security & Finishing Touches
Chapter 13: Deployment
Chapter 14: Overview of MVC Projects
Chapter 15: URL Routing
Chapter 16: Advanced Routing Features
Chapter 17: Controllers and Actions
Chapter 18: Filters
Chapter 19: Controller Extensibility
Chapter 20: Views
Chapter 21: Helper Methods
Chapter 22: Templated Helper Methods
Chapter 23: URL and Ajax Helper Methods
Chapter 24: Model Binding
Chapter 25: Model Validation
Chapter 26: Bundles
Chapter 27: Web API and Single-page Applications

Index

About the Author

About the Technical Reviewer

Chapter 1: Putting ASP.NET MVC in Context

Understanding the History of ASP.NET

What Is Wrong with ASP.NET Web Forms?

Web Development Today

Web Standards and REST

Agile and Test-Driven Development

Ruby on Rails

Node.js

Key Benefits of ASP.NET MVC

MVC Architecture

Extensibility

Tight Control over HTML and HTTP

Testability

Powerful Routing System

Built on the Best Parts of the ASP.NET Platform

Modern API

ASP.NET MVC Is Open Source

What Do I Need to Know?

What Is the Structure of This Book?

Part 1: Introducing ASP.NET MVC 5

Part 2: ASP.NET MVC in Detail

What's New in this Edition?

Where Can I Get the Example Code?

What Software Do I Need for This Book?

Credits

Summary

Chapter 2: Your First MVC Application

Preparing Visual Studio

Creating a New ASP.NET MVC Project

Adding the First Controller

Understanding Routes

Rendering Web Pages

Creating and Rendering a View

Adding Dynamic Output

Creating a Simple Data-Entry Application

eBooks-IT.org

Designing a Data Model
Linking Action Methods
Building the Form
Setting the Start URL
Handling Forms
Adding Validation
Styling the Content
Completing the Example
Summary
Chapter 3: The MVC Pattern
The History of MVC
Understanding the MVC Pattern
Understanding the Domain Model
The ASP.NET Implementation of MVC
Comparing MVC to Other Patterns
Building Loosely Coupled Components
Using Dependency Injection
Using a Dependency Injection Container
Getting Started with Automated Testing
Understanding Unit Testing
Understanding Integration Testing
Summary
Chapter 4: Essential Language Features
Preparing the Example Project
Adding the System.Net.Http Assembly
Using Automatically Implemented Properties
Using Object and Collection Initializers
Using Extension Methods
Applying Extension Methods to an Interface
Creating Filtering Extension Methods
Using Lambda Expressions
Using Automatic Type Inference
Using Anonymous Types
Performing Language Integrated Queries
Understanding Deferred LINQ Queries
Using Async Methods
Applying the async and await Keywords
Summary
Chapter 5: Working with Razor

Defining the Model
Defining the Controller
Creating the View
Working with the Model Object
Working with Layouts
Creating the Layout
Applying a Layout
Using a View Start File
Demonstrating Shared Layouts
Using Razor Expressions
Inserting Data Values
Setting Attribute Values
Using Conditional Statements
Enumerating Arrays and Collections
Dealing with Namespaces
Summary
Chapter 6: Essential Tools for MVC
Preparing the Example Project
Creating the Model Classes
Adding the Controller
Adding the View
Using Ninject
Understanding the Problem
Adding Ninject to the Visual Studio Project
Getting Started with Ninject
Setting up MVC Dependency Injection
Creating Chains of Dependency
Specifying Property and Constructor Parameter Values
Using Conditional Binding
Setting the Object Scope
Unit Testing with Visual Studio
Creating the Unit Test Project
Creating the Unit Tests
Running the Unit Tests (and Failing)
Implementing the Feature
Testing and Fixing the Code
Using Moq
Understanding the Problem
Adding Moq to the Visual Studio Project
Adding a Mock Object to a Unit Test
Creating a More Complex Mock Object

eBooks-IT.org

Chapter 7: SportsStore: A Real Application

Getting Started

- Creating the Visual Studio Solution and Projects
- Installing the Tool Packages
- Adding References Between Projects
- Setting Up the DI Container
- Running the Application

Starting the Domain Model

- Creating an Abstract Repository
- Making a Mock Repository

Displaying a List of Products

- Adding a Controller
- Adding the Layout, View Start File and View
- Setting the Default Route
- Running the Application

Preparing a Database

- Creating the Database
- Defining the Database Schema
- Adding Data to the Database
- Creating the Entity Framework Context
- Creating the Product Repository

Adding Pagination

- Displaying Page Links
- Improving the URLs

Styling the Content

- Installing the Bootstrap Package
- Applying Bootstrap Styles to the Layout
- Creating a Partial View

Summary

Chapter 8: SportsStore: Navigation

Adding Navigation Controls

- Filtering the Product List
- Refining the URL Scheme
- Building a Category Navigation Menu
- Correcting the Page Count

Building the Shopping Cart

- Defining the Cart Entity
- Adding the Add to Cart Buttons
- Implementing the Cart Controller
- Displaying the Contents of the Cart

Summary

Using Model Binding
Creating a Custom Model Binder
Completing the Cart
Removing Items from the Cart
Adding the Cart Summary
Submitting Orders
Extending the Domain Model
Adding the Checkout Process
Implementing the Order Processor
Registering the Implementation
Completing the Cart Controller
Displaying Validation Errors
Displaying a Summary Page
Summary
Chapter 10: SportsStore: Mobile
Putting Mobile Web Development in Context
Doing Nothing (Or As Little As Possible)
Using Responsive Design
Creating a Responsive Header
Creating a Responsive Product List
Creating Mobile Specific Content
Creating a Mobile Layout
Creating the Mobile Views
Summary
Chapter 11: SportsStore: Administration
Adding Catalog Management
Creating a CRUD Controller
Creating a New Layout
Implementing the List View
Editing Products
Creating New Products
Deleting Products
Summary
Chapter 12: SportsStore: Security & Finishing Touches
Securing the Administration Controller
Creating a Basic Security Policy
Applying Authorization with Filters
Creating the Authentication Provider
Creating the Account Controller
Creating the View

Extending the Database
Enhancing the Domain Model
Creating the Upload User Interface Elements
Saving Images to the Database
Implementing the GetImage Action Method
Displaying Product Images
Summary
Chapter 13: Deployment
Preparing Windows Azure
Creating the Web Site and Database
Preparing the Database for Remote Administration
Creating the Schema
Deploying the Application
Summary
Chapter 14: Overview of MVC Projects
Working with Visual Studio MVC Projects
Creating the Project
Understanding MVC Conventions
Debugging MVC Applications
Preparing the Example Project
Launching the Visual Studio Debugger
Causing the Visual Studio Debugger to Break
Using Edit and Continue
Using Browser Link
Summary
Chapter 15: URL Routing
Preparing the Example Project
Creating the Example Controllers
Creating the View
Setting the Start URL and Testing the Application
Introducing URL Patterns
Creating and Registering a Simple Route
Using the Simple Route
Defining Default Values
Using Static URL Segments
Defining Custom Segment Variables
Using Custom Variables as Action Method Parameters
Defining Optional URL Segments
Defining Variable-Length Routes
Prioritizing Controllers by Namespaces

Constraining a Route Using a Regular Expression
Constraining a Route to a Set of Specific Values
Constraining a Route Using HTTP Methods
Using Type and Value Constraints
Defining a Custom Constraint
Using Attribute Routing
Enabling and Applying Attribute Routing
Creating Routes with Segment Variables
Applying Route Constraints
Using a Route Prefix
Summary
Chapter 16: Advanced Routing Features
Preparing the Example Project
Simplifying the Routes
Adding the Optimization Package
Updating the Unit Test Project
Generating Outgoing URLs in Views
Using the Routing System to Generate an Outgoing URL
Targeting Other Controllers
Passing Extra Values
Specifying HTML Attributes
Generating Fully Qualified URLs in Links
Generating URLs (and Not Links)
Generating Outgoing URLs in Action Methods
Generating a URL from a Specific Route
Customizing the Routing System
Creating a Custom RouteBase Implementation
Creating a Custom Route Handler
Working with Areas
Creating an Area
Populating an Area
Resolving the Ambiguous Controller Issue
Creating Areas with Attributes
Generating Links to Actions in Areas
Routing Requests for Disk Files
Configuring the Application Server
Defining Routes for Disk Files
Bypassing the Routing System
URL Schema Best Practices
Make Your URLs Clean and Human-Friendly
GET and POST: Pick the Right One

Chapter 17: Controllers and Actions

Preparing the Example Project

Setting the Start URL

Introducing the Controller

Creating a Controller with IController

Creating a Controller by Deriving from the Controller Class

Receiving Request Data

Getting Data from Context Objects

Using Action Method Parameters

Producing Output

Understanding Action Results

Returning HTML by Rendering a View

Passing Data from an Action Method to a View

Performing Redirections

Returning Errors and HTTP Codes

Summary

Chapter 18: Filters

Preparing the Example Project

Setting the Start URL and Testing the Application

Using Filters

Introducing the Filter Types

Applying Filters to Controllers and Action Methods

Using Authorization Filters

Applying the Custom Authorization Filter

Using the Built-in Authorization Filter

Using Authentication Filters

Understanding the IAuthenticationFilter Interface

Implementing the Authentication Check

Combining Authentication and Authorization Filters

Handling the Final Challenge Request

Using Exception Filters

Creating an Exception Filter

Applying the Exception Filter

Using a View to Respond to an Exception

Using the Built-in Exception Filter

Using Action Filters

Implementing the OnActionExecuting Method

Implementing the OnActionExecuted Method

Using Result Filters

Using the Built-in Action and Result Filter Class

Filtering Without Attributes
Using Global Filters
Ordering Filter Execution
Overriding Filters
Summary
Chapter 19: Controller Extensibility
Preparing the Example Project
Setting the Start URL
Creating a Custom Controller Factory
Dealing with the Fallback Controller
Instantiating Controller Classes
Implementing the Other Interface Methods
Registering a Custom Controller Factory
Working with the Built-in Controller Factory
Prioritizing Namespaces
Customizing DefaultControllerFactory Controller Instantiation
Creating a Custom Action Invoker
Using the Built-in Action Invoker
Using a Custom Action Name
Using Action Method Selection
Improving Performance with Specialized Controllers
Using Sessionless Controllers
Using Asynchronous Controllers
Summary
Chapter 20: Views
Creating a Custom View Engine
Preparing the Example Project
Creating a Custom IView
Creating an IViewEngine Implementation
Registering a Custom View Engine
Testing the View Engine
Working with the Razor Engine
Preparing the Example Project
Understanding Razor View Rendering
Configuring the View Search Locations
Adding Dynamic Content to a Razor View
Using Layout Sections
Using Partial Views
Using Child Actions
Summary

Preparing the Example Project
Setting the Start URL
Testing the Example Application
Creating Custom Helper Methods
Creating an Inline Helper Method
Creating an External Helper Method
Managing String Encoding in a Helper Method
Using the Built-In Form Helper Methods
Creating Form Elements
Specifying the Route Used by a Form
Using Input Helpers
Creating Select Elements
Summary
Chapter 22: Templated Helper Methods
Preparing the Example Project
Using Templated Helper Methods
Generating Label and Display Elements
Using Whole-Model Templated Helpers
Using Model Metadata
Using Metadata to Control Editing and Visibility
Using Metadata for Labels
Using Metadata for Data Values
Using Metadata to Select a Display Template
Applying Metadata to a Buddy Class
Working with Complex Type Properties
Customizing the Templated View Helper System
Creating a Custom Editor Template
Creating a Generic Template
Replacing the Built-in Templates
Summary
Chapter 23: URL and Ajax Helper Methods
Preparing the Example Project
Defining Additional CSS Styles
Installing the NuGet Packages
Creating Basic Links and URLs
Using MVC Unobtrusive Ajax
Creating the Synchronous Form View
Preparing the Project for Unobtrusive Ajax
Creating an Unobtrusive Ajax Form
Preparing the Controller

Understanding How Unobtrusive Ajax Works

Setting Ajax Options

Ensuring Graceful Degradation

Providing the User with Feedback While Making an Ajax Request

Prompting the User Before Making a Request

Creating Ajax Links

Ensuring Graceful Degradation for Links

Working with Ajax Callbacks

Working with JSON

Adding JSON Support to the Controller

Processing JSON in the Browser

Preparing Data for Encoding

Detecting Ajax Requests in the Action Method

Summary

Chapter 24: Model Binding

Preparing the Example Project

Understanding Model Binding

Using the Default Model Binder

Binding to Simple Types

Binding to Complex Types

Binding to Arrays and Collections

Manually Invoking Model Binding

Dealing with Binding Errors

Customizing the Model Binding System

Creating a Custom Value Provider

Creating a Custom Model Binder

Registering the Custom Model Binder

Summary

Chapter 25: Model Validation

Preparing the Example Project

Creating the Layout

Creating the Views

Explicitly Validating a Model

Displaying Validation Errors to the User

Displaying Validation Messages

Displaying Property-Level Validation Messages

Using Alternative Validation Techniques

Performing Validation in the Model Binder

Specifying Validation Rules Using Metadata

Defining Self-Validating Models

Enabling Client-Side Validation
Using Client-Side Validation
Understanding How Client-Side Validation Works
Performing Remote Validation
Summary
Chapter 26: Bundles
Preparing the Example Application
Adding the NuGet Packages
Creating the Model and Controller
Creating the Layout and View
Profiling Script and Style Sheet Loading
Using Script and Style Bundles
Adding the NuGet Package
Defining the Bundles
Applying Bundles
Optimizing the JavaScript and CSS Files
Summary
Chapter 27: Web API and Single-page Applications
Understanding Single-page Applications
Preparing the Example Application
Creating the Model
Adding the NuGet Packages
Adding the Controller
Adding the Layout and Views
Setting the Start Location and Testing the Example Application
Using Web API
Creating the Web API Controller
Testing the API Controller
Understanding How the API Controller Works
Understanding API Controller Action Selection
Mapping HTTP Methods to Action Methods
Using Knockout for Single-page Applications
Adding the JavaScript Libraries to the Layout
Implementing the Summary
Implementing the Create Feature
Completing the Application
Simplify the Home Controller
Manage Content Visibility
Summary

Adam Freeman is an experienced IT professional who has held senior positions in a range of companies, most rece
as chief technology officer and chief operating officer of a global bank. Now retired, he spends his time writing and

eBooks-IT.org

Fabio Claudio Ferracchiati is a senior consultant and a senior analyst/developer using Microsoft technologies. He w
Brain Force (http://www.brainforce.com) in its Italian branch (http://www.brainforce
a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microso
Professional, and a prolific author and technical reviewer. Over the past 10 years, he's written articles for Italian and
magazines and coauthored more than 10 books on a variety of computer topics.

eBooks-IT.org

Putting ASP.NET MVC in Context

ASP.NET MVC is a Web development framework from Microsoft that combines the effectiveness and tidiness of m controller (MVC) architecture, the most up-to-date ideas and techniques from agile development, and the best parts c ASP.NET platform. It is a complete alternative to traditional ASP.NET Web Forms, delivering advantages for all but trivial of Web development projects. In this chapter, you'll learn why Microsoft created ASP.NET MVC, how it com predecessors and alternatives, and, finally, what's new in ASP.NET MVC 5 and what's covered in this book.

Understanding the History of ASP.NET

ASP.NET was a huge shift when it first arrived in 2002. Figure 1-1 illustrates Microsoft's technology stack as it app

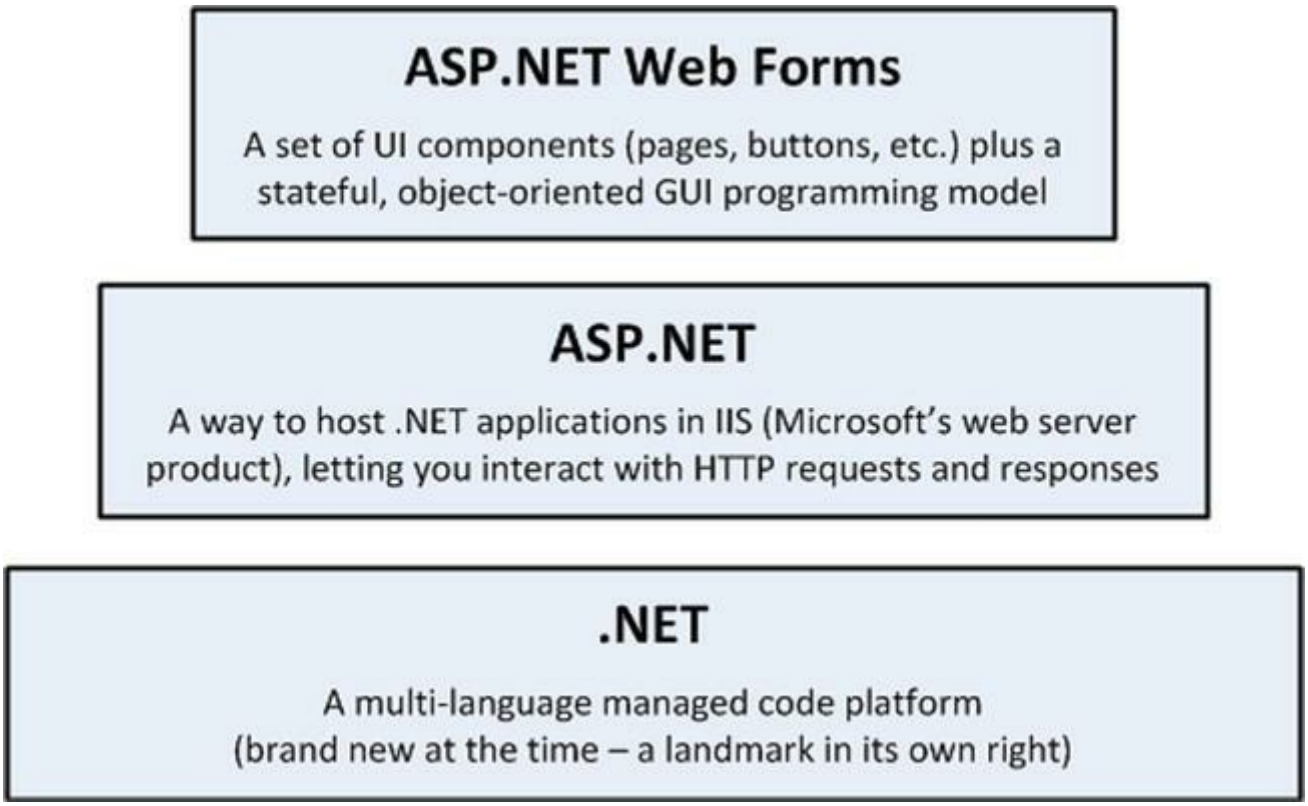


Figure 1-1. The ASP.NET Web Forms technology stack

With Web Forms, Microsoft attempted to hide both HTTP (with its intrinsic statelessness) and HTML (which at th unfamiliar to many developers) by modeling the user interface (UI) as a hierarchy of server-side control objects. Eac track of its own state across requests (using the *View State* facility), rendering itself as HTML when needed and auto connecting client-side events (for example, a button click) with the corresponding server-side event handler code. In Forms is a giant abstraction layer designed to deliver a classic event-driven graphical user interface (GUI) over the W

The idea was to make Web development feel just the same as Windows Forms development. Developers didn't ne with a series of independent HTTP requests and responses. They could think in terms of a stateful UI, and Microsoft seamlessly transition the army of Windows desktop developers into the new world of web applications.

Traditional ASP.NET Web Forms development was great in principle, but reality proved more complicated:

- *View State weight*: The actual mechanism for maintaining state across requests (known as View State) r blocks of data being transferred between the client and server. This data can reach hundreds of kilobytes ir Web applications, and it goes back and forth with *every* request, leading to slower response times and bandwidth demands of the server.
- *Page life cycle*: The mechanism for connecting client-side events with server-side event handler code, part c cycle, can be extraordinarily complicated and delicate. Few developers have success manipulating the contr runtime without getting View State errors or finding that some event handlers mysteriously fail to execute.
- *False sense of separation of concerns*: ASP.NET Web Forms' *code-behind* model provides a means to take ap out of its HTML markup and into a separate code-behind class. This has been widely applauded for separa presentation, but, in reality, developers are encouraged to mix presentation code (for example, manipulating t control tree) with their application logic (for example, manipulating database data) in these same monstrou classes. The end result can be fragile and unintelligible.
- *Limited control over HTML*: Server controls render themselves as HTML, but not necessarily the HTML you versions of ASP.NET, the HTML output failed to meet with Web standards or make good use of Cascading (CSS), and server controls generated unpredictable and complex ID attribute values that are hard to JavaScript. These problems are much improved in recent Web Forms releases, but it can still be tricky to g you expect.
- *Leaky abstraction*: Web Forms tries to hide HTML and HTTP wherever possible. As you try to imple behaviors, you frequently fall out of the abstraction, which forces you to reverse-engineer the postback eve or perform obtuse acts to make it generate the desired HTML. Plus, all this abstraction can act as a frustrat competent Web developers.
- *Low testability*: The designers of Web Forms could not have anticipated that automated testing would becon component of software development. Not surprisingly, the tightly coupled architecture they designed is unsu testing. Integration testing can be a challenge, too.

Web Forms isn't all bad and Microsoft has put a lot of effort into improving standards compliance, simplifying the process, and even taking some features from ASP.NET MVC. Web Forms excels when you need quick results, and a reasonably complex web app up and running within a day. But unless you are careful during development, you will application you create is hard to test and hard to maintain.

Note For complete details of ASP.NET Web Forms, see my *Pro ASP.NET 4.5 in C#* book, also published by Ap the complete framework and provide best-practice guidance for avoiding the most serious pitfalls.

Web Development Today

Outside Microsoft, Web development technology has been progressing rapidly and in several different directions sinc was first released.

Web Standards and REST

The drive for Web standards compliance has increased in recent years. Web sites are consumed on a greater variety and browsers than ever before, and Web standards (HTML, CSS, JavaScript, and so forth) remain the great hope fo consistent browsing experience. Modern web platforms can't afford to ignore the business case and the weight of de enthusiasm for Web standards compliance.

HTML5 has begun to enter mainstream use and provides the Web developer with rich capabilities that allow the cli perform work that was previously the exclusive responsibility of the server. These new capabilities and the increasing JavaScript libraries such as AngularJS, jQuery, jQuery UI, and jQuery Mobile means that standards have become eve important and form the critical foundation for ever richer Web apps.

eBooks-IT.org

topics
right. If you want more complete coverage, then Apress publishes my books on these subjects: *Pro AngularJS*, *Pro*
Pro JavaScript for Web Apps, and *The Definitive Guide to HTML5*.

At the same time, Representational State Transfer (REST has become the dominant architecture for application int
over HTTP, completely overshadowing SOAP (the technology behind ASP.NET's original approach to Web services
describes an application in terms of resources (URIs) representing real-world entities and standard operations (HTTP
representing available operations on those resources. For example, you might `PUT` a new
`http://www.example.com/Products/Lawnmower` or `DELETE`
`http://www.example.com/Customers/Arnold-Smith`.

Today's Web applications don't serve just HTML. Often, they must also serve JSON or XML data to client techn
as AJAX and native smartphone applications. This happens naturally with REST, which eliminates the distinction bet
services and Web applications, but requires an approach to HTTP and URL handling that has not easily been support
ASP.NET Web Forms.

Agile and Test-Driven Development

It is not just Web development that has matured. Software development as a whole has shifted toward *agile* methodo
can mean a lot of different things, but it is largely about running software projects as adaptable processes of discove
resisting excessive forward planning. Enthusiasm for agile methodologies tends to go hand-in-hand with a set of deve
practices and tools (usually open source) that promote and assist these practices.

Test-driven development (TDD), and its close relative, *behavior-driven development (BDD)*, are two examples. T
design your software by first describing examples of desired behaviors (known as *tests* or *specifications*), so at any t
verify the stability and correctness of your application by executing your suite of tests against the implementation. T
shortage of .NET tools to support TDD/BDD, but these tend to not work well with Web Forms:

- *Unit testing tools* let you specify the behavior of individual classes or other small code units in isolation.
effectively applied only to software that has been designed as a set of independent modules, so that each test
isolation. Unfortunately, few Web Forms applications can be tested this way.
- *UI automation tools* let you simulate a series of user interactions against a complete running instance of yo
These can be used with Web Forms, but they can break down whenever you make a slight change to you
Without special attention, Web Forms change the HTML structures and element IDs, breaking test suites.

The .NET open source and independent software vendor (ISV community has produced no end of top quality unit
frameworks (NUnit and xUnit), mocking frameworks (Moq and Rhino Mocks), inversion-of-control containers (Nin
AutoFac), continuous integration servers (Cruise Control and TeamCity), object-relational mappers (NHibernate and
and the like. Traditional ASP.NET Web Forms is not amenable to these tools and techniques because of its monolithi
so Web Forms gets little respect from these projects.

Ruby on Rails

In 2004, Ruby on Rails was a quiet, open source contribution from an unknown player. Suddenly fame hit, transform
of Web development. It's not that Ruby on Rails contained revolutionary technology but that the concept took existin
and blended them in such a compelling and appealing way as to put existing platforms to shame.

Ruby on Rails (or just Rails, as it is commonly called) embraced an MVC architecture, which I describe in Chapte
applying MVC and working in tune with the HTTP protocol, by promoting conventions instead of the need for confi
by integrating an object-relational mapping (ORM tool into its core, Rails applications more or less fell into place with
effort. It was as if this was how Web development should have been all along. Rails showed that Web standards con
RESTfulness don't need to be hard. It also showed that agile development and TDD work best when the framework
support them. The rest of the Web development world has been catching up ever since.

Node.js

Another significant trend is the movement toward using JavaScript as a primary programming language. AJAX first
JavaScript is important; jQuery showed us that it could be powerful and elegant; and Google's open source V8 JavaS
showed us that it could be fast. Today, JavaScript is becoming a serious server-side programming language. It serve

eBooks-IT.org

sed as
purpose language in server-side platforms such as Node.js. Node.js has been around since 2009 and gained acceptan
key innovations are as follows:

- *Using JavaScript*: Developers need to work only in a single language, from client-side code, through server-s even into data-querying logic via CouchDB or the like.
- *Being completely asynchronous*: Node.js's core API doesn't expose any way of blocking a thread whil input/output (I/O) or any other operation. All I/O is implemented by beginning the operation and then lat callback when the I/O is completed. This means that Node.js makes extremely efficient use of system resou handle tens of thousands of concurrent requests per CPU. (Alternative platforms tend to be limited to abou concurrent requests per CPU.)

Node.js remains a niche technology. Its biggest contribution to web app development has, rather oddly, been to pr consistent JavaScript engine on which development tools can be written. Many emerging client-side JavaScript fram as AngularJS, have good tooling support based on the use of Node.js.

Node.js adoption for deploying web apps has been slower. Most businesses building real applications in limited tim typically need the infrastructure in full-stack frameworks such as Ruby on Rails and ASP.NET MVC. Node.js is mer only to put some of ASP.NET MVC's design into context against industry trends. For example, ASP.NET MVC incl *asynchronous controllers* (which I describe in Chapter 19). This is a way to handle HTTP requests with non-blocking scale up to handle more requests per CPU.

Key Benefits of ASP.NET MVC

In October 2007, Microsoft announced a new MVC Web development platform, built on the core ASP.NET platform designed as a direct response to the evolution of technologies such as Rails and as a reaction to the criticisms of Web following sections describe how this new platform overcame the Web Forms limitations and brought ASP.NET back edge.

MVC Architecture

It is important to distinguish between the MVC architectural pattern and the ASP.NET MVC Framework. The MVC new—it dates back to 1978 and the Smalltalk project at Xerox PARC—but it has gained enormous popularity today a for Web applications, for the following reasons:

- User interaction with an MVC application follows a natural cycle: the user takes an action, and in response changes its data model and delivers an updated view to the user. And then the cycle repeats. This is a con Web applications delivered as a series of HTTP requests and responses.
- Web applications necessitate combining several technologies (databases, HTML, and executable code, for exa split into a set of tiers or layers. The patterns that arise from these combinations map naturally onto the conce

The ASP.NET MVC Framework implements the MVC pattern and, in doing so, provides greatly improved separat concerns. In fact, ASP.NET MVC implements a modern variant of the MVC pattern that is especially suitable for We applications. You will learn more about the theory and practice of this architecture in Chapter 3.

By embracing and adapting the MVC pattern, the ASP.NET MVC Framework provides strong competition to Rub similar platforms, and brings the MVC pattern into the mainstream of the .NET world. By capitalizing on the experien practices discovered by developers using other platforms, ASP.NET MVC has, in many ways, pushed forward beyo Rails can offer.

Extensibility

The MVC Framework is built as a series of independent components that satisfy a .NET interface or that are built on base class. You can easily replace components, such as the routing system, the view engine, and the controller facto different one of your own implementation. In general, the MVC Framework gives you three options for each compon

eBooks-IT.org

- Derive a *subclass* of the default implementation to tweak its behavior.
- *Replace* the component entirely with a new implementation of the interface or abstract base class.

You'll learn all about the various components, and how and why you might want to tweak or replace each of them Chapter 14.

Tight Control over HTML and HTTP

ASP.NET MVC produces clean, standards-compliant markup. Its built-in HTML helper methods produce standards- output, but there is a more significant philosophical change compared with Web Forms. Instead of generating out sw HTML over which you have little control, the MVC Framework encourages you to craft simple, elegant markup style

Of course, if you do want to throw in some ready-made widgets for complex UI elements such as date pickers or menus, ASP.NET MVC's "no special requirements" approach to markup makes it easy to use best-of-breed UI librar jQuery UI or the Bootstrap CSS library. ASP.NET MVC meshes so well with jQuery, for example, that Microsoft sh a built-in part of the default Visual Studio ASP.NET MVC project template, along with other popular libraries, such a Knockout and Modernizr.

Tip I don't get into the detail of these "blessed" JavaScript libraries in this book because they are not part of the Framework and do their work within the browser. Client-side development for MVC Framework applications is an in topic, however, and you can learn more in my book *Pro ASP.NET MVC 5 Client*, which will be published by Apress There are some libraries, however, that provide support for core features such as validation and Ajax requests and I in Part 2 of this book. I describe Knockout in Chapter 27 and I use Bootstrap (albeit without a detailed introduction) the book.

ASP.NET MVC–generated pages don't contain any View State data, so they are smaller than typical pages from A Forms. Despite today's fast connections, this economy of bandwidth still gives an enormously improved end-user ex helps reduce the cost of running a popular web application.

ASP.NET MVC works in tune with HTTP. You have control over the requests passing between the browser and s can fine-tune your user experience as much as you like. AJAX is made easy, and there aren't any automatic postbac with client-side state.

Testability

The MVC architecture gives you a great start in making your application maintainable and testable because you natura different application concerns into independent pieces. Yet the ASP.NET MVC designers didn't stop there. To suppo they took the framework's component-oriented design and made sure that each separate piece is structured to meet requirements of unit testing and mocking tools.

They added Visual Studio wizards to create unit test projects on your behalf, which can be integrated with open so tools such as NUnit and xUnit as well as the test tools that are included in Visual Studio, which I introduce in Chapte you have never written a unit test before, you will be off to a great start.

In this book, you will see examples of how to write clean, simple unit tests for ASP.NET MVC controllers and act supply fake or mock implementations of framework components to simulate any scenario, using a variety of testing strategies.

Testability is not only a matter of unit testing. ASP.NET MVC applications work well with UI automation testing t can write test scripts that simulate user interactions without needing to guess which HTML element structures, CSS IDs the framework will generate, and you do not have to worry about the structure changing unexpectedly.

Powerful Routing System

The style of URLs has evolved as Web application technology has improved. URLs like this one:

```
/App_v2/User/Page.aspx?action=show%20prop&prop_id=82742
```

are increasingly rare, replaced with a simpler, cleaner format like this:

```
/to-rent/chicago/2303-silver-street
```

eBooks-IT.org

keywor

URL. A search for "rent in Chicago" is much more likely to turn up the simpler URL. Second, many Web users are n enough to understand a URL, and appreciate the option of navigating by typing it into their browser's address bar. Th someone understands the structure of a URL, they are more likely to link to it, share it with a friend, or even read it a phone. Fourth, it doesn't expose the technical details, folder, and file name structure of your application to the public you are free to change the underlying implementation without breaking all your incoming links.

Clean URLs were hard to implement in earlier frameworks, but ASP.NET MVC uses a feature known as *URL rou* provide clean URLs by default. This gives you control over your URL schema and its relationship to your application the freedom to create a pattern of URLs that is meaningful and useful to your users, without the need to conform to pattern. And, of course, this means you can easily define a modern REST-style URL schema if you wish. You'll find description of URL routing in Chapters 15 and 16.

Built on the Best Parts of the ASP.NET Platform

Microsoft's existing ASP.NET platform provides a mature, well-proven set of components and facilities for developi and efficient Web applications. First and most obviously, as ASP.NET MVC is based on the .NET platform, you hav flexibility to write code in any .NET language and access the same API features—not just in MVC itself but in the ex class library and the vast ecosystem of third-party .NET libraries.

Second, ready-made ASP.NET platform features—such as authentication, membership, roles, profiles, and interna —can reduce the amount of code you need to develop and maintain any Web application, and these features are just when used in the MVC Framework as they are in a classic Web Forms project. The underlying ASP.NET platform p set of tools on which to build web applications with the MVC Framework.

Note I describe the most commonly used ASP.NET Platform features as they relate to MVC development in this platform is a topic in its own right. For complete details of the rich features that the ASP.NET platform provides, see forthcoming *Pro ASP.NET MVC 5 Platform*, which will be published by Apress in 2014.

Modern API

Microsoft's .NET platform has evolved with each major release, supporting – and even defining – the state-of-the-ar modern programming. ASP.NET MVC 5 is built for .NET 4.5.1, so its API can take full advantage of recent languag innovations, including the `await` keyword, extension methods, lambda expressions, anonymous and dynamic type Language Integrated Query (LINQ). Many of the MVC Framework's API methods and coding patterns follow a clea expressive composition than was possible with earlier platforms. Don't worry if you are not up to speed on the latest features: I provide a summary of the most important C# features for MVC development in Chapter 4.

ASP.NET MVC Is Open Source

Unlike previous Microsoft Web development platforms, you are free to download the original source code for ASP.N and even modify and compile your own version of it. This is invaluable when your debugging trail leads into a systen and you want to step into its code (and even read the original programmers' comments). It is also useful if you are b advanced component and want to see what development possibilities exist, or how the built-in components actually w

Additionally, this ability is great if you do not like the way something works, if you find a bug, or if you just want something that's otherwise inaccessible, because you can simply change it yourself. However, you'll need to keep tra changes and reapply them if you upgrade to a newer version of the framework. ASP.NET MVC is licensed under the Public License (Ms-PL, http://www.opensource.org/licenses/ms-pl.html), an Open S Initiative (OSI)–approved open source license. This means that you can change the source code, deploy it, and even your changes publicly as a derivative project. You can download the MVC source code from http://aspnetwebstack.codeplex.com.

What Do I Need to Know?

To get the most from this book, you should be familiar with the basics of web development, have an understanding o and CSS work and a working knowledge of C#. Don't worry if you are a little hazy on the client-side details. My em server-side development in this book and you can pick up what you need through the examples. In Chapter 4, I prov summary of the most useful C# language features for MVC development, which you'll find useful if you are moving

eBooks-IT.org

What Is the Structure of This Book?

This book is split into 2 parts, each of which covers a set of related topics.

Part 1: Introducing ASP.NET MVC 5

I start this book by putting the ASP.NET MVC Framework in context. I explain the benefits and practical impact of t pattern, the way in which the MVC Framework fits into modern web development and describe the tools and C# lang that every MVC Framework programmer needs.

In the next chapter you will dive right in and create a simple web application and get an idea of what the major cor building blocks are and how they fit together. Most of this part of the book, however, is given over to the developme project called SportsStore, through which I show you a realistic development process from inception to deployment, the major features of the ASP.NET MVC Framework.

Part 2: ASP.NET MVC in Detail

In Part 2, I explain the inner workings of the MVC Framework features that I used to build the SportsStore applicati you how each feature works, explain the role it plays in the MVC Framework and show you the configuration and cu options that are available. Having set the broad context in Part 1, I dig right into the details in Part 2.

What's New in this Edition?

Version 5 of the MVC Framework is a relatively minor upgrade and a lot of the changes are really to do with the way ASP.NET projects are created and managed in Visual Studio. Table 1-1 briefly describes the new MVC Framework f details where you can find more information about them in this book.

Table 1-1. *The New Features in MVC 5*

Feature	Description
Authentication Filters	A new kind of filter that can be used to include different types of authentication within the same controller
Filter Overrides	A new kind of filter that is applied to action methods to prevent filters defined globally or on the controller from taking effect
Attribute Routing	A set of attributes that allow URL routes to be defined within the controller class

ASP.NET version 4.5.1, on which the MVC Framework 5 is built, has been enhanced as well. The most importan addition of the ASP.NET Identity API, which replaces the Membership system for managing user credentials. I don' ASP.NET Identity in this book, although I do explain how authentication and authorization are applied to MVC Frame applications through the use of features like filters.

Note I will be covering ASP.NET Identity in my *Pro ASP.NET MVC 5 Platform* book, which will be published i cover all of the facilities that the ASP.NET platform provides. That said, I don't want you to have to buy a second b about something as important as user security, and so Apress has agreed to distribute the security-related chapters fr from its web site for download without charge when that book is published. Those chapters won't be available imme because I have not written the platform book yet, but it is my next major writing project after this book and my hope delay won't be too long.

A new edition is a chance to go beyond writing about new features and I have made some other changes for this b expanded the SportsStore example to show the basics of responsive and mobile web application development, I adde references to the start of all the in-depth chapters so you can find easily specific examples, and I added a chapter tha one of the open source libraries that Microsoft has embraced—Knockout—can be combined with the Web API featu Single-Page Applications (SPAs).

eBooks-IT.org

You can download all of the examples for all of the chapters in this book from Apress.com. The download is a without charge and includes all of the Visual Studio projects and their contents. You don't have to download the cod easiest way of experimenting with the examples and cutting and pasting techniques into your own projects.

What Software Do I Need for This Book?

The only software you need for MVC development is Visual Studio 2013, which contains everything you need to get including a built-in application server for running and debugging MVC applications, an administration-free edition of S for developing database-driven applications, tools for unit testing and, of course, a code editor compiler and debugge

There are several different editions of Visual Studio, but I will be using the one that Microsoft makes available free called *Visual Studio Express 2013 for Web*. Microsoft adds some nice features to the paid-for editions of Visual Stud will not need them for this book and all of the figures that you see throughout this book have been taken using the Ex which you can download from http://www.microsoft.com/visualstudio/eng/products/visual-studio-e products. There are several different versions of Visual Studio 2013 Express, each of which is used for a differ development. Make sure that you get the Web version, which supports ASP/NET applications.

Once you have installed Visual Studio, you are ready to go. Microsoft has improved the scope of the features in th Studio Express in recent years and there is nothing else you need to follow along with this book. I do rely on addition packages, but these are installed through Visual Studio and don't require separate downloads and installations (and ar without cost).

Tip I have used Windows 8.1 throughout this book, but you can use Visual Studio 2013 and develop MVC applic happily on earlier versions of Windows. See the system requirements for Visual Studio 2013 for details of which ver patch levels are supported).

Credits

In Chapter 10, I use a feature of the Bootstrap CSS library called *Glyphicons Halflings*, which are a set of icons that usually available for free, but for which the creator has given an open license for their inclusion in Bootstrap. The on that the creator's URL be quoted when it is possible to do so, which seems like a fair and reasonable thing to do. Her http://glyphicons.com.

Summary

In this chapter, I explained the context in which the MVC Framework exists and how it compares to Web Forms. I benefits of using the MVC framework, the structure of this book and the software that you will require to follow the

You saw how the ASP.NET MVC platform addresses the weaknesses of ASP.NET Web Forms, and how its mod delivers advantages to developers who want to write high-quality, maintainable code. In the next chapter, you'll see t Framework in action in a simple demonstration of the features that deliver these benefits.

eBooks-IT.org

Your First MVC Application

The best way to appreciate a software development framework is to jump right in and use it. In this chapter, you'll c
data-entry application using the ASP.NET MVC Framework. I take things a step at a time so you can see how an AS
application is constructed. To keep things simple, I will skip over some of the technical details for the moment. But d
If you are new to MVC, you will find plenty to keep you interested. Where I use something without explaining it, I p
reference to the chapter in which you can find all the details.

Preparing Visual Studio

Visual Studio Express contains all of the features you need to create, test and deploy an MVC Framework application
those features are hidden away until you ask for them. To enable all of the features, select `Expert Settings`
Visual Studio `Tools` ➤ `Settings` menu.

Tip For some reason, Microsoft has decided that the top-level menus in Visual Studio should be all in uppercase,
that the menu I referred to is really `TOOLS`. I think this is rather like shouting and I will capitalize menu names as `T`
throughout this book.

Creating a New ASP.NET MVC Project

I am going to start by creating a new MVC Framework project in Visual Studio. Select `New Project` from the
to open the `New Project` dialog. If you select the Web templates in the Visual C# section, you will see the `ASP`
`Web Application` project template. Select this project type, as shown in Figure 2-1.

eBooks-IT.org

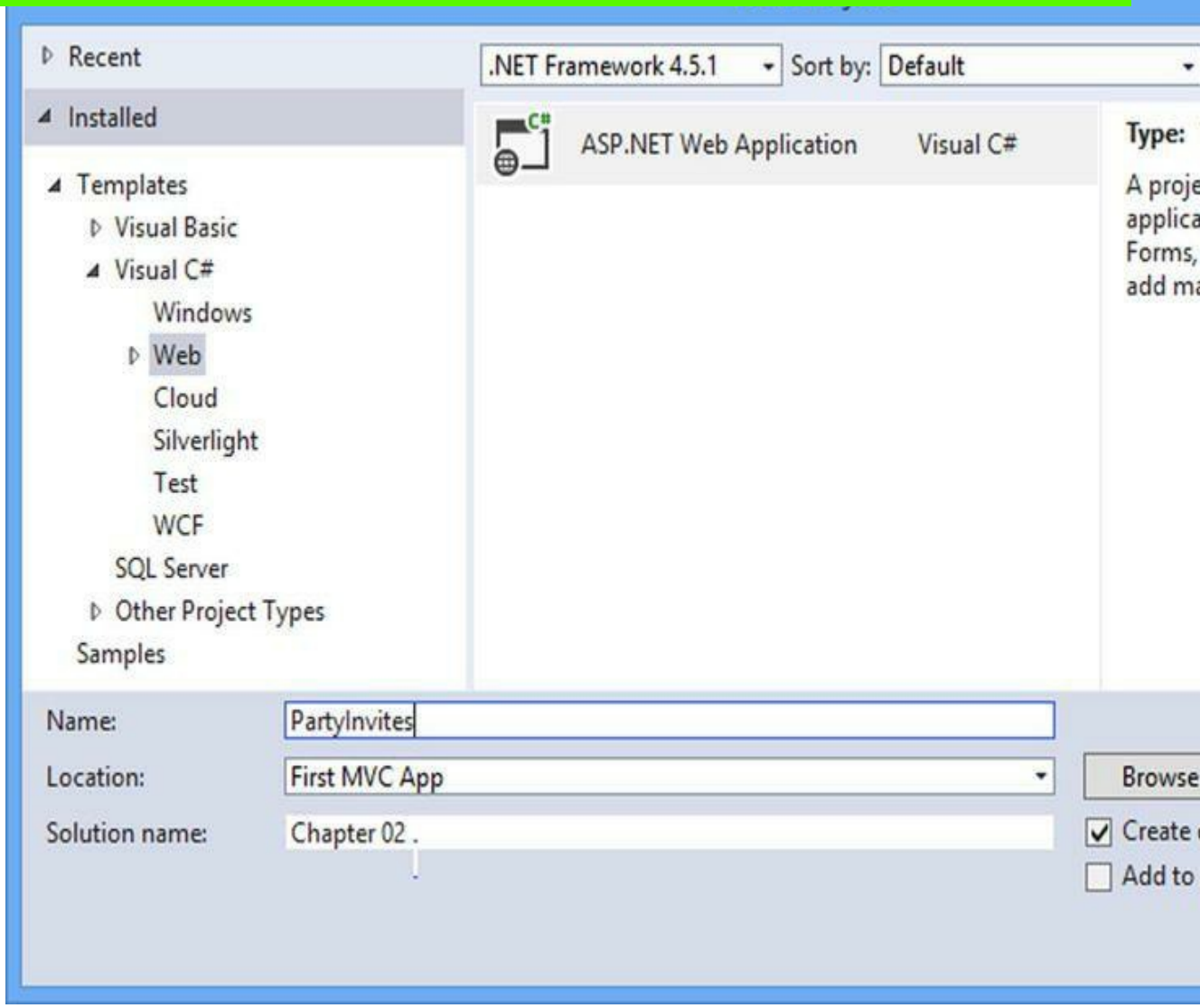


Figure 2-1. The Visual Studio ASP.NET Web Application project template

Set the name of the new project to **PartyInvites** and click the OK button to continue. You will see another dialog in Figure 2-2, which asks you to set the initial content for the ASP.NET project. This is part of the Microsoft initiativ integrate the different parts of ASP.NET into a set of consistent tools and templates.

eBooks-IT.org

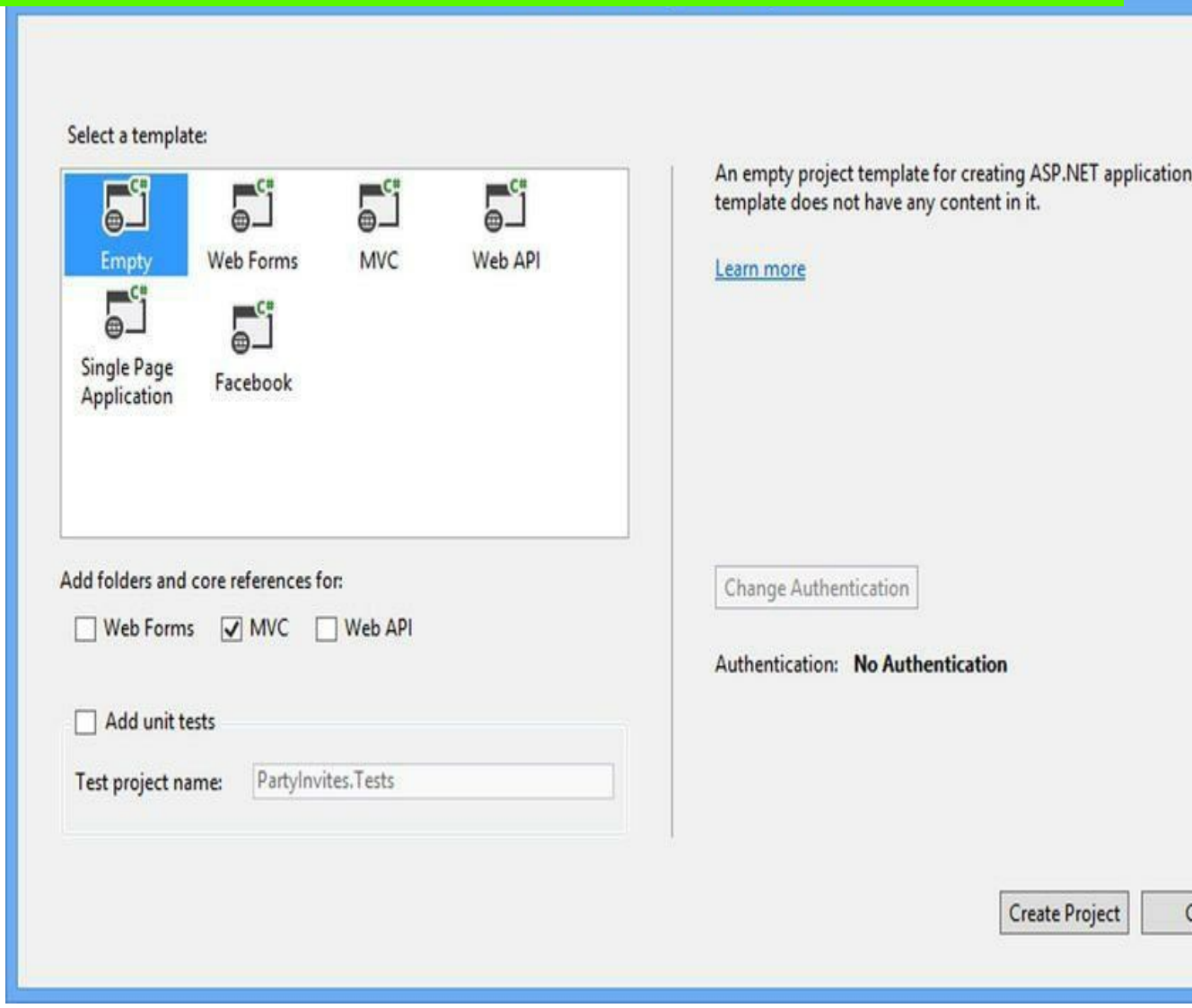


Figure 2-2. Selecting the initial project configuration

Tip Make sure you select version 4.5.1 of the .NET Framework at the top of the window. This is the latest versi and is required for some of the advanced features that I describe in this book.

The templates create projects with different starting points and configurations for features such as authentication, and visual themes. I am going to keep things simple: select the `Empty` option and check the `MVC` box in the *Add f core references* section, as shown in the figure. This will create a basic MVC project with minimal predefined conten the starting point that I use for all of the examples in this book. Click the `OK` button to create the new project.

Note The other project template options are intended to give you a more complete starting point for your ASP.N don't like these templates because they encourage developers to treat some important features, such as authenticatior boxes. My goal in this book is to give you the knowledge to understand and manage every aspect of your MVC appli a consequence, I use the `Empty` template for most of the examples in the book – the exception is in Chapter 14, w you the content that the `MVC` template adds to new projects.

Once Visual Studio creates the project, you will see a number of files and folders displayed in the **Solution Explor** as shown in Figure 2-3. This is the default project structure for a new MVC project and you will soon understand th each of the files and folders that Visual Studio creates.

eBooks-IT.org

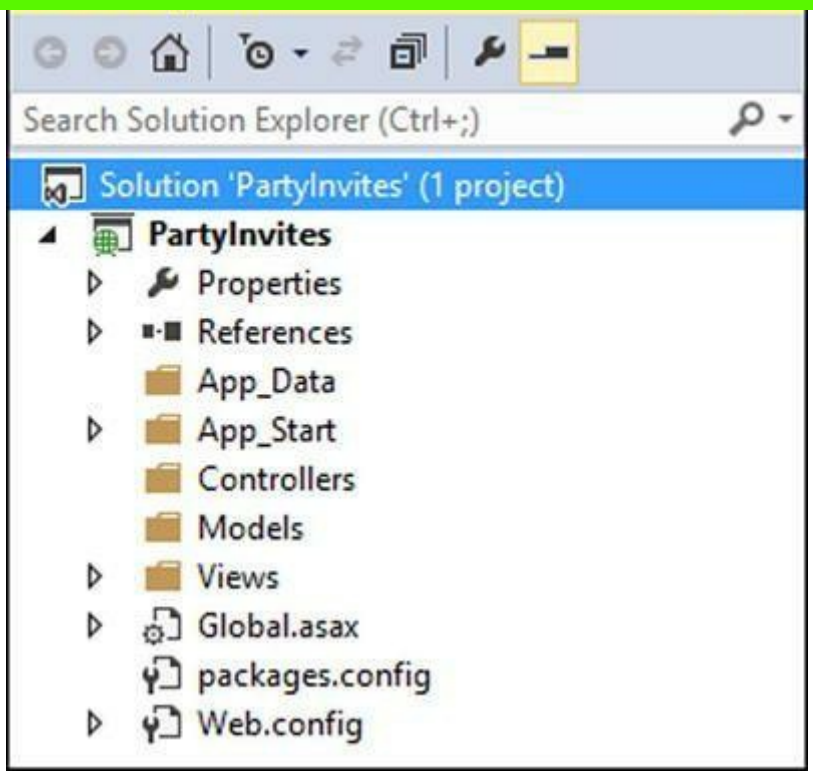


Figure 2-3. The initial file and folder structure of an MVC project

You can try to run the application now by selecting **Start Debugging** from the `Debug` menu (if it prompts you t debugging, just click the `OK` button). You can see the result in Figure 2-4. Because I started with the empty project t application does not contain anything to run, so the server generates a 404 Not Found Error.

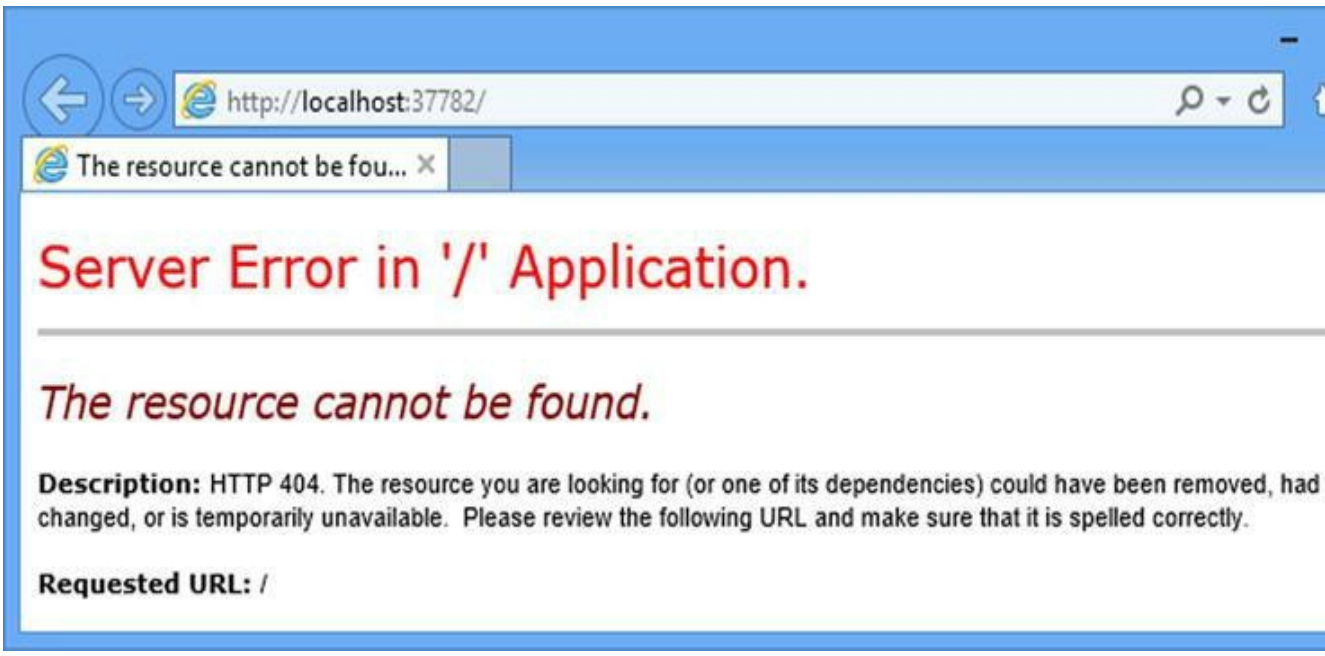


Figure 2-4. Trying to run an empty project

When you are finished, be sure to stop debugging by closing the browser window that shows the error, or by goir Visual Studio and selecting `Stop Debugging` from the `Debug` menu.

As you have just seen, Visual Studio opens the browser to display the project. The default browser is, of course, I Explorer, but you can select any browser that you have installed by using the toolbar shown in Figure 2-5. As the fig have a range of browsers installed, which I find useful for testing web apps during development.

eBooks-IT.org

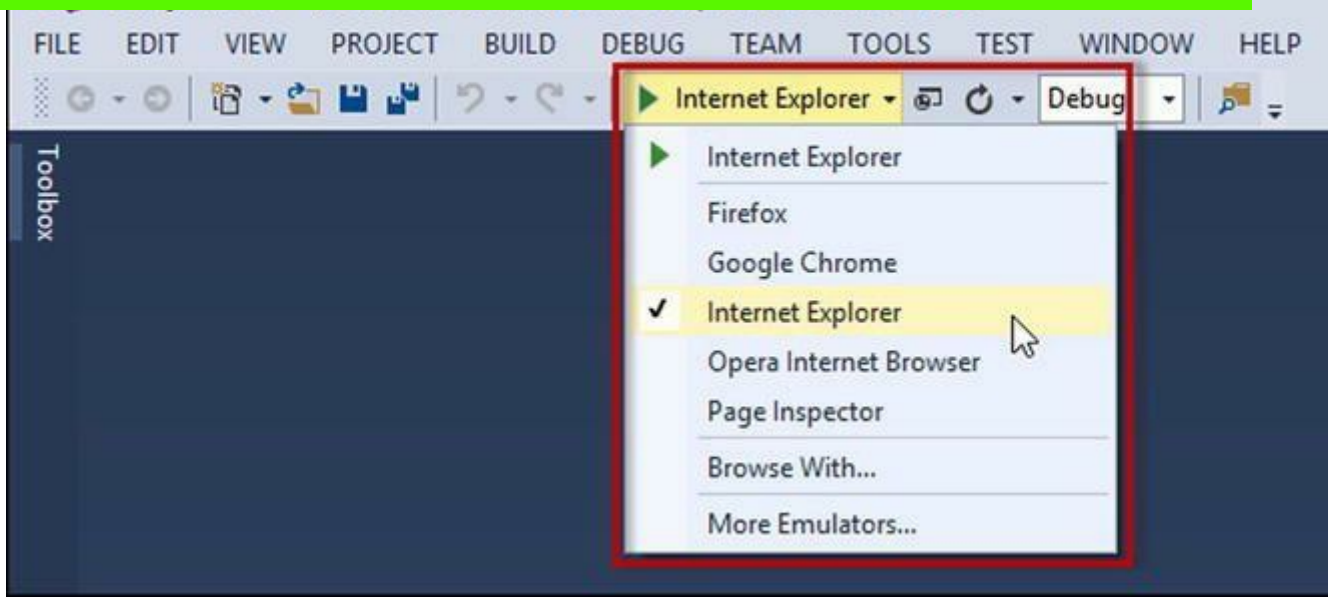


Figure 2-5. Changing the browser that Visual Studio uses to run the project

I will be using Internet Explorer 11 throughout this book, but that's just because I know that IE is so widely instal Explorer used to play fast and loose with web standards, but recent versions have been good at implementing the HT standard. Google Chrome is also a good choice for development and I tend to use it for my own projects.

Adding the First Controller

In MVC architecture, incoming requests are handled by *controllers*. In ASP.NET MVC, controllers are just C# classe inheriting from `System.Web.Mvc.Controller`, the framework's built-in controller base class).

Each public method in a controller is known as an *action method*, meaning you can invoke it from the Web via sor perform an action. The MVC convention is to put controllers in the `Controllers` folder, which Visual Studio c it set up the project.

Tip You do not need to follow this or most other MVC conventions, but I recommend that you do—not least bec help you make sense of the examples in this book.

To add a controller to the project, right-click the `Controllers` folder in the Visual Studio `Solution Ex` window and choose `Add` and then `Controller` from the pop-up menus, as shown in Figure 2-6.

eBooks-IT.org

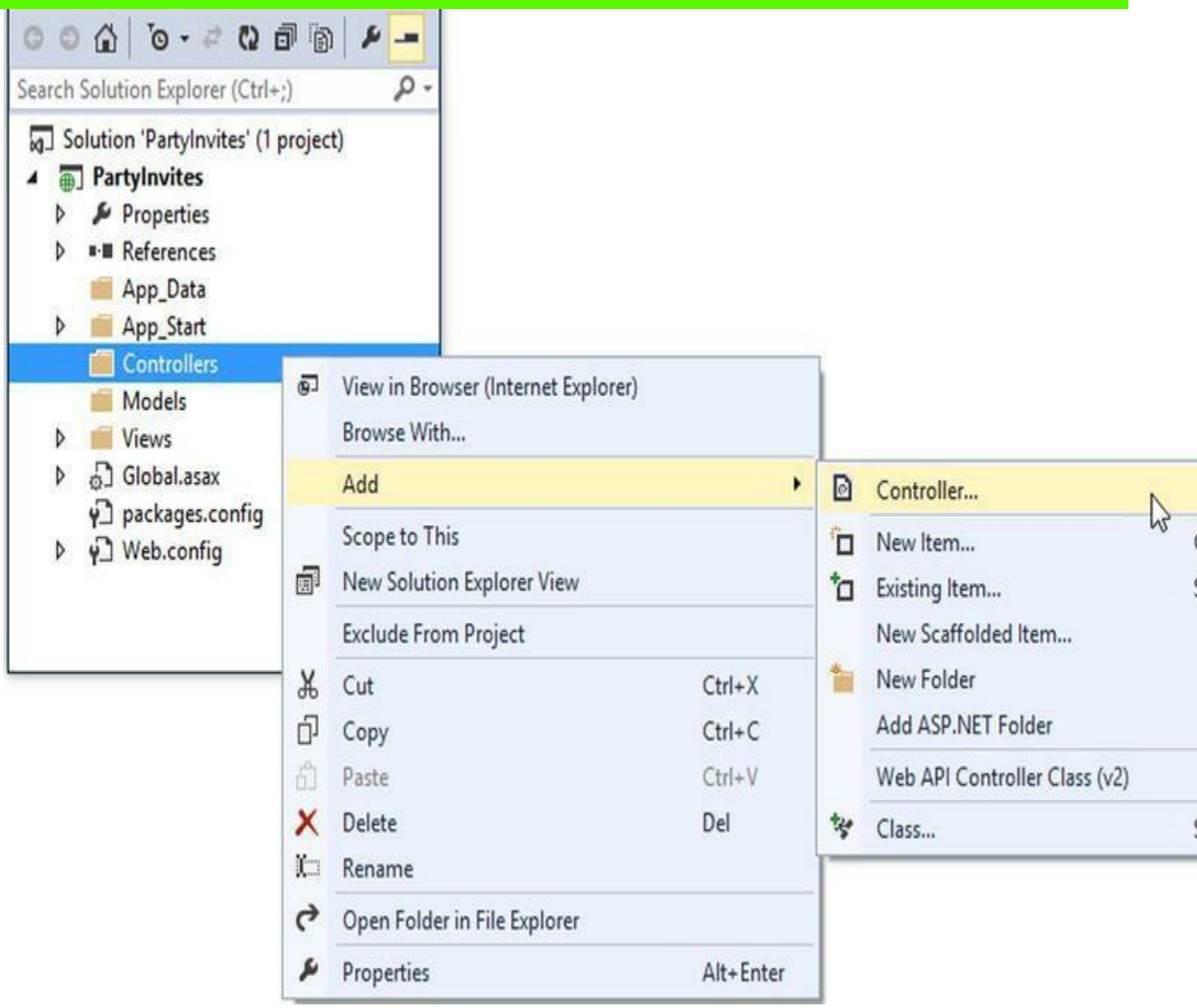


Figure 2-6. Adding a controller to the MVC project

When the `Add Scaffold` dialog appears, select the `MVC 5 Controller - Empty` option, as sho
2-7, and click the `Add` button.

eBooks-IT.org

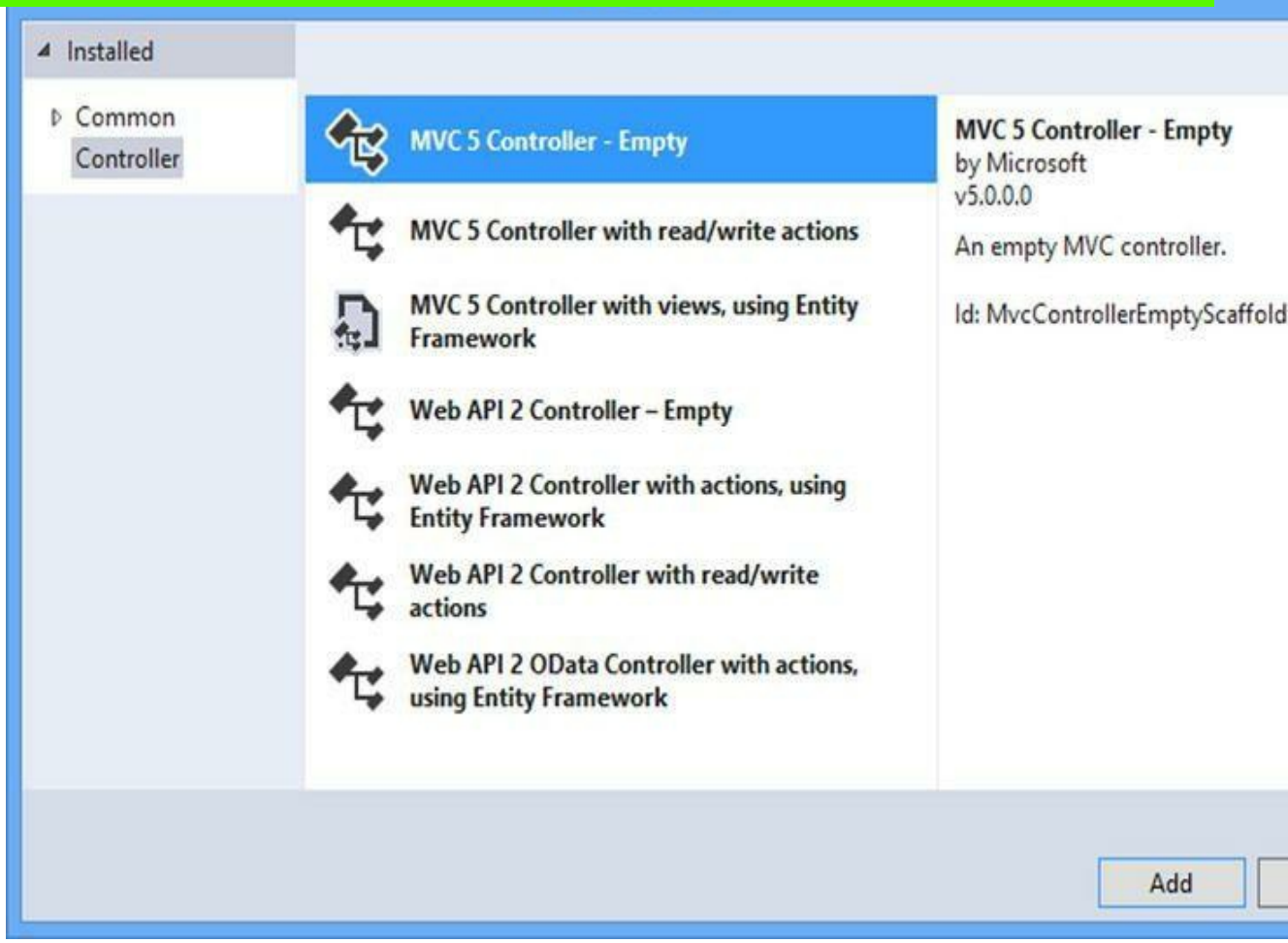


Figure 2-7. Selecting an empty controller from the Add Scaffold dialog

The `Add Controller` dialog will appear. Set the name to `HomeController` and click the `Add` butt several conventions represented in this name: names given to controllers should indicate their purpose; the default co called `Home` and controller names have the suffix `Controller`.

Tip If you have used earlier versions of Visual Studio to create MVC applications, then you will notice that the pr slightly different. Microsoft has changed the way that Visual Studio can populate a project with preconfigured classe items.

Visual Studio will create a new C# file in the `Controllers` folder called `HomeController.cs` and o editing. I have listed the default contents that Visual Studio puts into the class file in Listing 2-1. You can see that the `HomeController` and it is derived from the `Controller` class, which is found in the `System.Web.` namespace.

Listing 2-1. The Default Contents of the HomeController.cs File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
```

eBooks-IT.org

```
        }
    }
}
```

A good way of getting started with MVC is to make a couple of simple changes to the controller class. Edit the co `HomeController.cs` file so that it matches Listing 2-2. I have highlighted the statements that have changed easier to see.

Listing 2-2. Modifying the HomeController.cs File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Hello World";
        }
    }
}
```

These changes don't have a dramatic effect, but they make for a nice demonstration. I have changed the action m `Index` so that it returns the string "`Hello World`". Run the project again by selecting `Start Debuggi` Visual Studio `Debug` menu. The browser will display the result of the `Index` action method, as shown in Figure

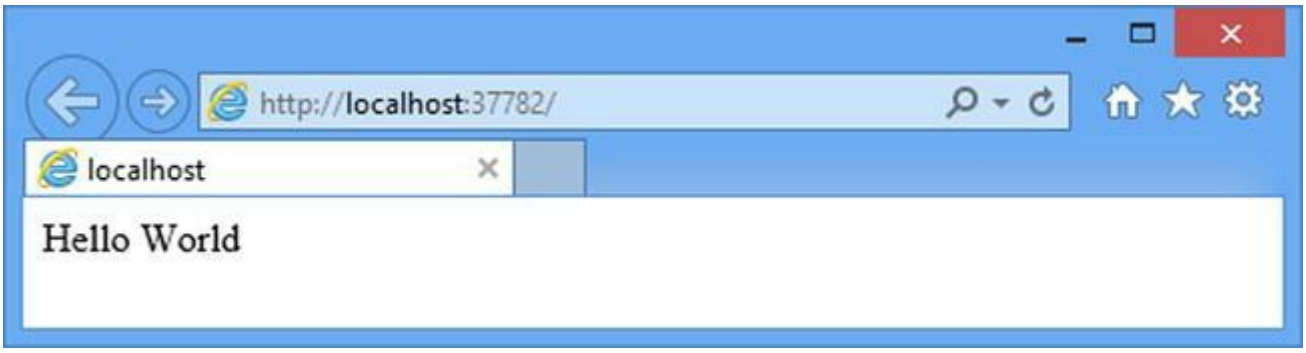


Figure 2-8. The output from the controller action method

Tip Notice that Visual Studio has directed the browser to port 37782. You will almost certainly see a different po the URL that your browser requests because Visual Studio allocates a random port when the project is created. If yo Windows task bar notification area, you will find an icon for IIS Express. This is a cut-down version of the full IIS a server which is included with Visual Studio and is used to deliver ASP.NET content and services during developmen you how to deploy an MVC project into a production environment in Chapter 13.

Understanding Routes

As well as models, views, and controllers, MVC applications use the ASP.NET *routing system*, which decides how U controllers and actions. When Visual Studio creates the MVC project, it adds some default routes to get us started. Y request any of the following URLs, and they will be directed to the `Index` action on the `HomeController`:

- /

- /Home/Index

So, when a browser requests http://yoursite/ or http://yoursite/Home, it gets back the HomeController's Index method. You can try this yourself by changing the URL in the browser. At the r be http://localhost:37782/, except that the port part may be different. If you append /Home or /Home/Index to the URL and hit return, you will see the same Hello World result from the MVC applica

This is a good example of benefiting from following MVC conventions. In this case, the convention is that I will h controller called HomeController and that it will be the starting point for my MVC application. The default ro Visual Studio creates for a new project assume that I will follow this convention. And since I *did* follow the convent automatically got support for the URLs in the preceding list.

If I had *not* followed the convention, I would need to modify the routes to point to whatever controller I had creat For this simple example, the default configuration is all I need.

Tip You can see and edit your routing configuration by opening the RouteConfig.cs file in the App_St I explain what the entries in this file do in Chapters 16 and 17.

Rendering Web Pages

The output from the previous example wasn't HTML—it was just the string "Hello World". To produce an H response to a browser request, I need a *view*.

Creating and Rendering a View

The first thing I need to do is modify my Index action method, as shown in Listing 2-3.

Listing 2-3. Modifying the Controller to Render a View in the HomeController.cs File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View();
        }
    }
}
```

The changes in Listing 2-3 are shown in bold. When I return a ViewResult object from an action method, I MVC to *render* a view. I create the ViewResult by calling the View method with no parameters. This tells M the *default* view for the action.

If you run the application at this point, you can see the MVC Framework trying to find a default view to use, as sh error message displayed in Figure 2-9.

eBooks-IT.org

Figure 2-9. The MVC Framework trying to find a default view

This error message is quite helpful. It explains not only that MVC could not find a view for the action method, but where it looked. This is another nice illustration of an MVC convention: views are associated with action methods by convention. The action method is called `Index` and the controller is called `Home` and you can see from Figure 2-9 trying to find different files in the `Views` folder that have that name.

The simplest way to create a view is to ask Visual Studio to do it for you. Right-click anywhere in the definition of action method in code editor window for the `HomeController.cs` file and select `Add View` from the po as shown in Figure 2-10.

eBooks-IT.org

```
using System.Web.Mvc;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View();
        }
    }
}
```

Go To View Ctrl+M, Ctrl+G
Add View...
Go To Definition F12
Find All References Ctrl+K, R
Breakpoint
Add Watch
Add Parallel Watch

Figure 2-10. Asking Visual Studio to create a view for an action method

Visual Studio displays the `Add View` dialog, which allows you to configure the initial contents of the view file created. Set `View Name` to `Index` (the name of the action method that the view will be associated with—anotl convention), set `Template` to `Empty (without model)`, and leave the `Create as a parti` and `Use a layout page` boxes unchecked, as shown in Figure 2-11. Don't worry about what all of these at the moment—I'll explain all of the details in later chapters. Click the `Add` button to create the new view file.

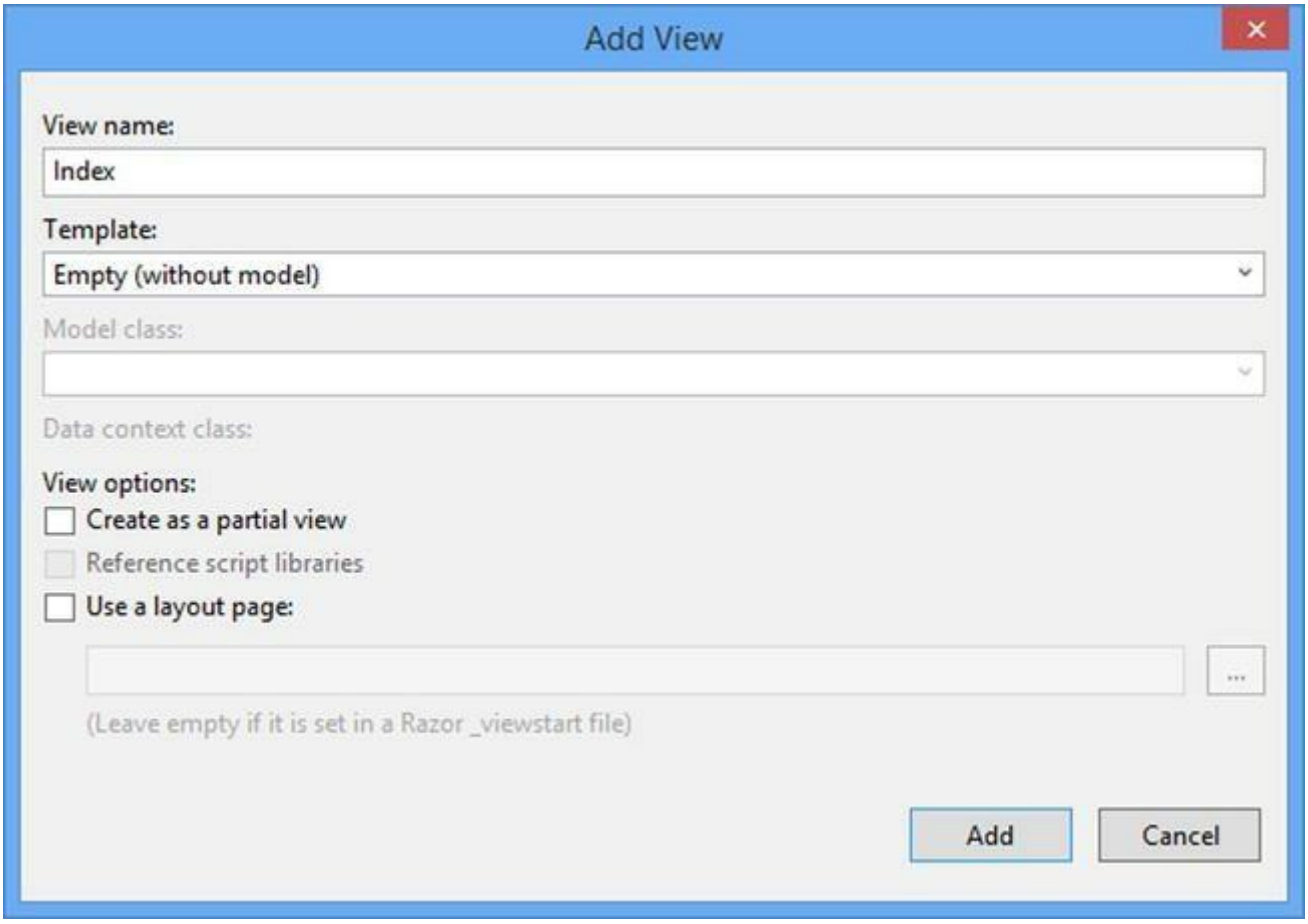


Figure 2-11. Configuring the initial contents of the view file

eBooks-IT.org

then delete the file you created and try again. This is another MVC Framework convention. Views are placed in the V folder, organized in folders that correspond to the name of the controller they are associated with.

Tip The `.cshtml` file extension denotes a C# view that will be processed by Razor. Early versions of MVC re ASPX view engine, for which view files have the `.aspx` extension.

The effect of the values I told you to enter into the `Add View` dialog tell Visual Studio to create the most basic contents of which are shown in Listing 2-4.

Listing 2-4. The Initial Contents of the Index.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
    </div>
</body>
</html>
```

Visual Studio opens the `Index.cshtml` file for editing. You'll see that this file contains mostly HTML. The e the part that looks like this:

```
...
@{
    Layout = null;
}
...
```

This is an expression that will be interpreted by the Razor view engine, which processes the contents of views and HTML that is sent to the browser. This is a simple Razor expression and it tells Razor that I chose not to use a layou like a template for the HTML that will be sent to the browser (and which I describe in Chapter 5). I am going to igno the moment and come back to it later. Make the addition to the `Index.cshtml` file that is shown in bold in List

Listing 2-5. Adding to the View HTML in the Index.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
```

eBooks-IT.org

```
    <div>
        Hello World (from the view)
    </div>
</body>
</html>
```

The addition displays another simple message. Select `Start Debugging` from the `Debug` menu to run th and test the view. You should see something similar to Figure 2-12.

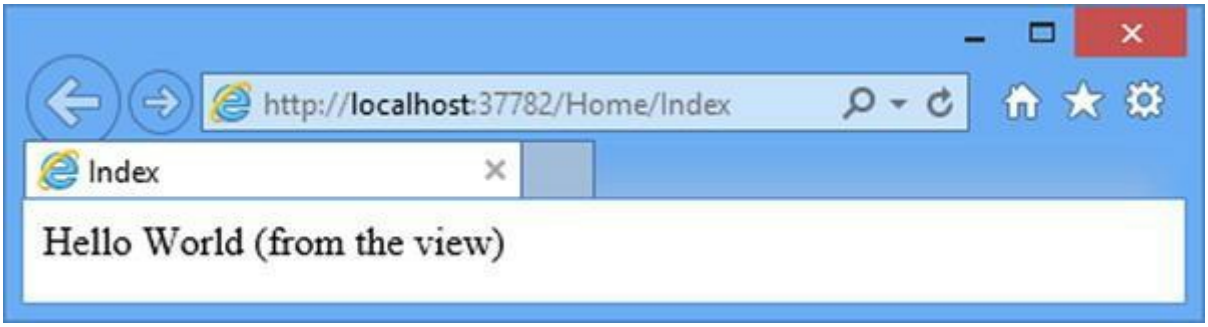


Figure 2-12. Testing the view

When I first edited the `Index` action method, it returned a string value. This meant that MVC did nothing except string value as is to the browser. Now that the `Index` method returns a `ViewResult`, the MVC Framework r and returns the HTML it produces. I didn't tell MVC which view should be used, so it used the naming convention t automatically. The convention is that the view has the name of the action method and is contained in a folder named controller: `/Views/Home/Index.cshtml`.

I can return other results from action methods besides strings and `ViewResult` objects. For example, if I retu `RedirectResult`, the browser will be redirected to another URL. If I return an `HttpUnauthorizedR` force the user to log in. These objects are collectively known as *action results*, and they are all derived from the `ActionResult` class. The action result system lets us encapsulate and reuse common responses in actions. I'll about them and show more complex uses in Chapter 17.

Adding Dynamic Output

The whole point of a web application platform is to construct and display *dynamic* output. In MVC, it is the controlle construct some data and pass it to the view, which is responsible for rendering it to HTML.

One way to pass data from the controller to the view is by using the `ViewBag` object, which is a member of the `Controller` base class. `ViewBag` is a dynamic object to which you can assign arbitrary properties, making available in whatever view is subsequently rendered. Listing 2-6 demonstrates passing some simple dynamic data in t `HomeController.cs` file.

Listing 2-6. Setting Some View Data in the HomeController.cs File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
                ViewBag.Greeting = hour < 12 ? "Good Morning"
```

eBooks-IT.org

```
            return View();
        }
    }
}
```

I provide data for the view when I assign a value to the `ViewBag.Greeting` property. The `Greeting` didn't exist until the moment I assigned the value—this allows me to pass data from the controller to the view in a fr manner, without having to define classes ahead of time. I refer to the `ViewBag.Greeting` property again in t get the data value, as illustrated in Listing 2-7, which shows the corresponding change to the Index.cshtml file.

Listing 2-7. Retrieving a ViewBag Data Value in the Index.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
    </div>
</body>
</html>
```

The addition to Listing 2-7 is a Razor expression. When I call the `View` method in the controller's `Index` meth framework locates the `Index.cshtml` view file and asks the Razor view engine to parse the file's content. Raz expressions like the one I added in the listing and processes them. In this example, processing the expression means value assigned to the `ViewBag.Greeting` property in the action method into the view.

There's nothing special about the property name `Greeting`; you could replace this with any property name an work the same, just as long as the name you use in the controller matches the name you use in the view. You can pa data values from your controller to the view by assigning values to more than one property. You can see the effect o changes by starting the project, as shown in Figure 2-13.

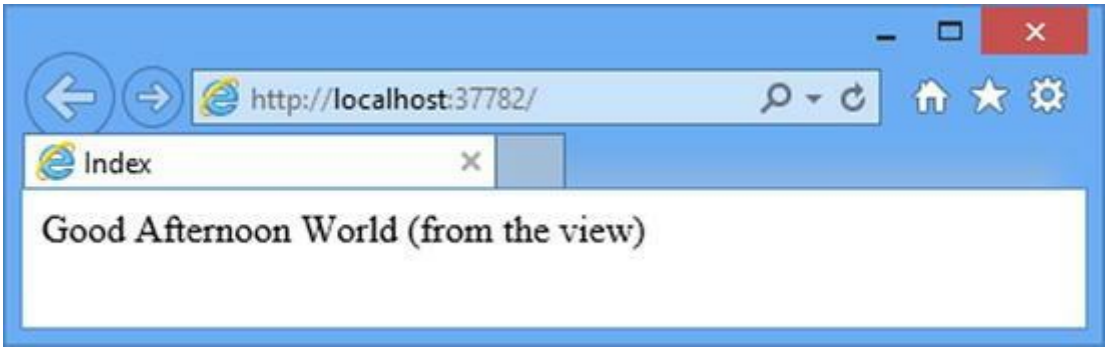


Figure 2-13. A dynamic response from MVC

Creating a Simple Data-Entry Application

In the rest of this chapter, I will explore more of the basic MVC features by building a simple data-entry application.

eBooks-IT.org

lanatior

things work behind the scenes. But don't worry, I'll revisit these topics in depth in later chapters.

Setting the Scene

Imagine that a friend has decided to host a New Year's Eve party and that she has asked me to create a web app that invitees to electronically RSVP. She has asked for four key features:

- A home page that shows information about the party
- A form that can be used to RSVP
- Validation for the RSVP form, which will display a thank-you page
- RSVPs e-mailed to the party host when complete

In the following sections, I will build up the MVC project I created at the start of the chapter and add these feature check the first item off the list by applying what I covered earlier and add some HTML to my existing view to give d party. Listing 2-8 shows the additions I made to the `Views/Home/Index.cshtml` file.

Listing 2-8. Displaying Details of the Party in the Index.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
       <p>We're going to have an exciting party.<br />
       (To do: sell it better. Add pictures or something.)
       </p>
    </div>
</body>
</html>
```

I am on my way. If you run the application, you'll see the details of the party—well, the placeholder for the details the idea—as shown in Figure 2-14.

Designing a Data Model

In MVC, the *M* stands for *model*, and it is the most important part of the application. The model is the representation world objects, processes, and rules that define the subject, known as the *domain*, of the application. The model, ofte as a *domain model*, contains the C# objects (known as *domain objects*) that make up the universe of the application methods that manipulate them. The views and controllers expose the domain to the clients in a consistent manner and designed MVC application starts with a well-designed model, which is then the focal point as controllers and views a

I don't need a complex model for the `PartyInvites` application because it is such a simple application and create just one domain class which I will call `GuestResponse`. This object will be responsible for storing, vali confirming an RSVP.

Adding a Model Class

The MVC convention is that the classes that make up a model are placed inside the `Models` folder, which Visual S as part of the initial project setup. Right-click `Models` in the `Solution Explorer` window and select `Ad` by `Class` from the pop-up menus. Set the file name to `GuestResponse.cs` and click the `Add` button to c class. Edit the contents of the class to match Listing 2-9.

Tip If you don't have a `Class` menu item, then you probably left the Visual Studio debugger running. Visual S the changes you can make to a project while it is running the application.

Listing 2-9. The GuestResponse Domain Class Defined in the GuestResponse.cs File

```
namespace PartyInvites.Models {
    public class GuestResponse {
        public string Name { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
        public bool? WillAttend { get; set; }
    }
}
```

Tip You may have noticed that the `WillAttend` property is a *nullable* `bool`, which means that it can be t `false`, or `null`. I explain the rationale for this in the Adding Validation section later in the chapter.

Linking Action Methods

One of my application goals is to include an RSVP form, so I need to add a link to it from my `Index.cshtml` shown in Listing 2-10.

Listing 2-10. Adding a Link to the RSVP Form in the Index.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
```

```
    <div>
        @ViewBag.Greeting World (from the view)
       <p>We're going to have an exciting party.<br />
       (To do: sell it better. Add pictures or something.)
       </p>
       @Html.ActionLink("RSVP Now", "RsvpForm")
    </div>
</body>
</html>
```

`Html.ActionLink` is an HTML *helper method*. The MVC Framework comes with a collection of built-in h that are convenient for rendering HTML links, text inputs, checkboxes, selections, and other kinds of content. The `ActionLink` method takes two parameters: the first is the text to display in the link, and the second is the action when the user clicks the link. I explain the complete set of HTML helper methods in Chapters 21-23. You can see th helper creates by starting the project, as shown in Figure 2-15.

Figure 2-15. Adding a link to the view

If you roll your mouse over the link in the browser, you will see that the link points to `http://yourserver/Home/RsvpForm`. The `Html.ActionLink` method has inspected the app URL routing configuration and determined that `/Home/RsvpForm` is the URL for an action called `RsvpFor` controller called `HomeController`.

Tip Notice that, unlike traditional ASP.NET applications, MVC URLs do not correspond to physical files. Each a has its own URL, and MVC uses the ASP.NET routing system to translate these URLs into actions.

Creating the Action Method

You will see a `404 Not Found` error if you click the link. That's because I have not yet created the action me corresponds to the `/Home/RsvpForm` URL. I do this by adding a method called `RsvpForm` to the `HomeController` class, as shown in Listing 2-11.

Listing 2-11. Adding a New Action Method in the HomeController.cs File

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

namespace PartyInvites.Controllers {
```

```
        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
                ViewBag.Greeting = hour < 12 ? "Good Morning"
Afternoon";
            return View();
        }

        public ViewResult RsvpForm() {
            return View();
        }
    }
}
```

Adding a Strongly Typed View

I am going to add a view for the `RsvpForm` action method, but in a slightly different way—I am going to create a *typed* view. A strongly typed view is intended to render a specific domain type, and if I specify the type I want to wo (`GuestResponse` in this case), MVC can create some helpful shortcuts to make it easier.

Caution Make sure your MVC project is compiled before proceeding. If you have created the `GuestRespo` but not compiled it, MVC won't be able to create a strongly typed view for this type. To compile your application, se `Solution` from the Visual Studio `Build` menu.

Right-click the `RsvpForm` method in the code editor and select `Add View` from the pop-up menu to open t `View` dialog window. Ensure that the `View Name` is set as `RsvpForm`, set `Template` to `Empty` and se `GuestResponse` from the drop-down list for the `Model Class` field. Leave the `View Options` box unchecked, as shown in Figure 2-16.

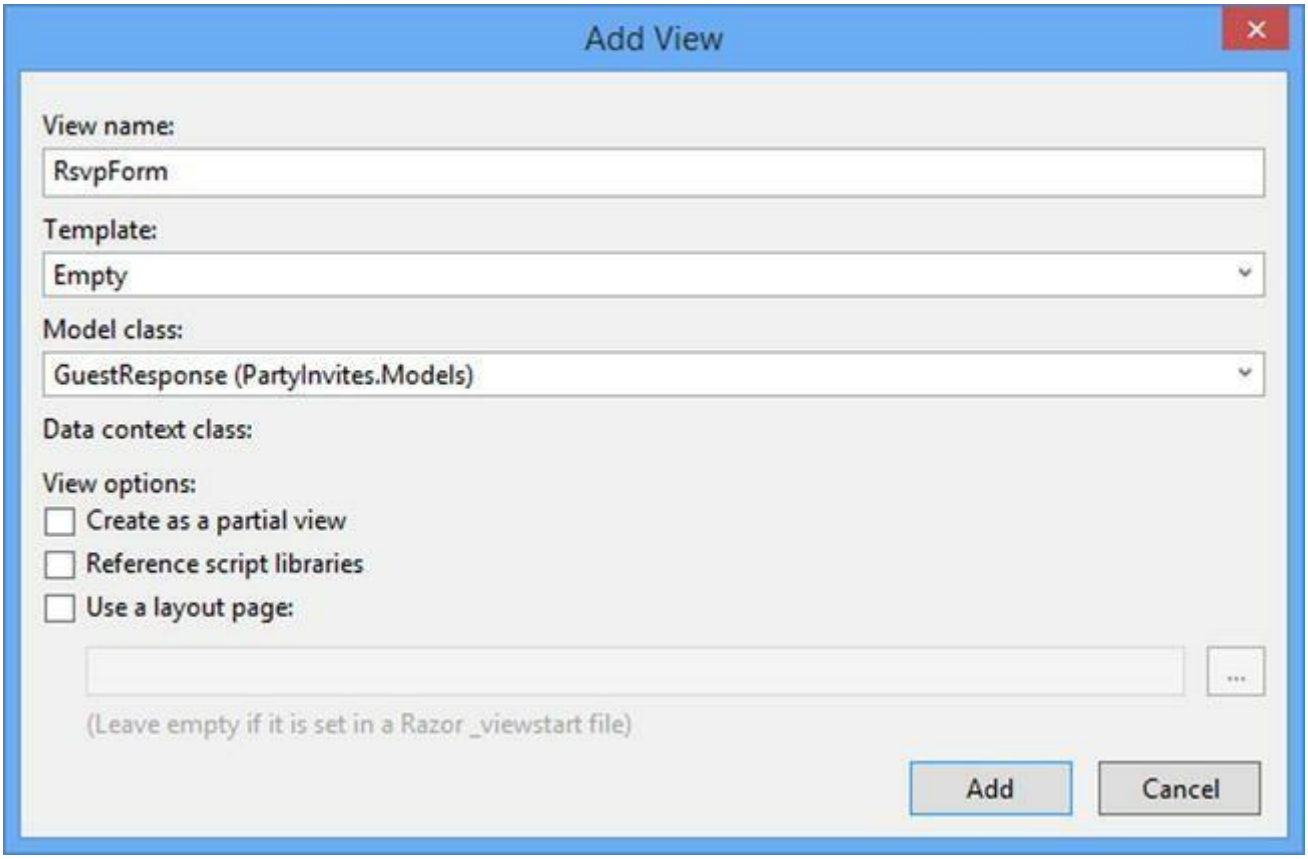


Figure 2-16. Adding a new view to the project

eBooks-IT.org

and open it for editing. You can see the initial contents in Listing 2-12. This is another skeletal HTML file, but it conta
@model Razor expression. As you will see in a moment, this is the key to a strongly typed view and the convenien

Listing 2-12. The Initial Contents of the RsvpForm.cshtml File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <div>
    </div>
</body>
</html>
```

Tip The options that you select and check when you create a view determine the initial content of a view file, bu
You can change from regular to strongly typed views, for example, just by adding or removing the @model direct
editor.

Building the Form

Now that I have created the strongly typed view, I can build out the contents of RsvpForm.cshtml to make
HTML form for editing GuestResponse objects, as shown in Listing 2-13.

Listing 2-13. Creating a Form View in the RsvpForm.cshtml File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm()) {
        <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
        <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
        <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
        <p>
```

eBooks-IT.org

```
                @Html.DropDownListFor(x => x.WillAttend, new[] {
                    new SelectListItem() {Text = "Yes, I'll be there",
                        Value = bool.TrueString},
                    new SelectListItem() {Text = "No, I can't come",
                        Value = bool.FalseString}
                }, "Choose an option")
            </p>
            <input type="submit" value="Submit RSVP" />
        }
</body>
</html>
```

For each property of the `GuestResponse` model class, I use an HTML helper method to render a suitable H `input` control. These methods let you select the property that the `input` element relates to using a lambda expre this:

```
...
@Html.TextBoxFor(x => x.Phone)
...
```

The HTML `TextBoxFor` helper method generates the HTML for an `input` element, sets the `type` param `text`, and sets the `id` and `name` attributes to `Phone` (the name of the selected domain class property) like this

```
<input id="Phone" name="Phone" type="text" value="" />
```

This handy feature works because the `RsvpForm` view is strongly typed, and I have told MVC that `GuestR` the type that I want to render with this view. This provides the HTML helper methods with the information they nee understand which data type I want to read properties from via the `@model` expression.

Don't worry if you aren't familiar with C# lambda expressions. I provide an overview in Chapter 4, but an alternat lambda expressions is to refer to the name of the model type property as a string, like this:

```
...
@Html.TextBox("Email")
...
```

I find that the lambda expression technique prevents me from mistyping the name of the model type property, bec Studio IntelliSense pops up and lets me pick the property automatically, as shown in Figure 2-17.

eBooks-IT.org

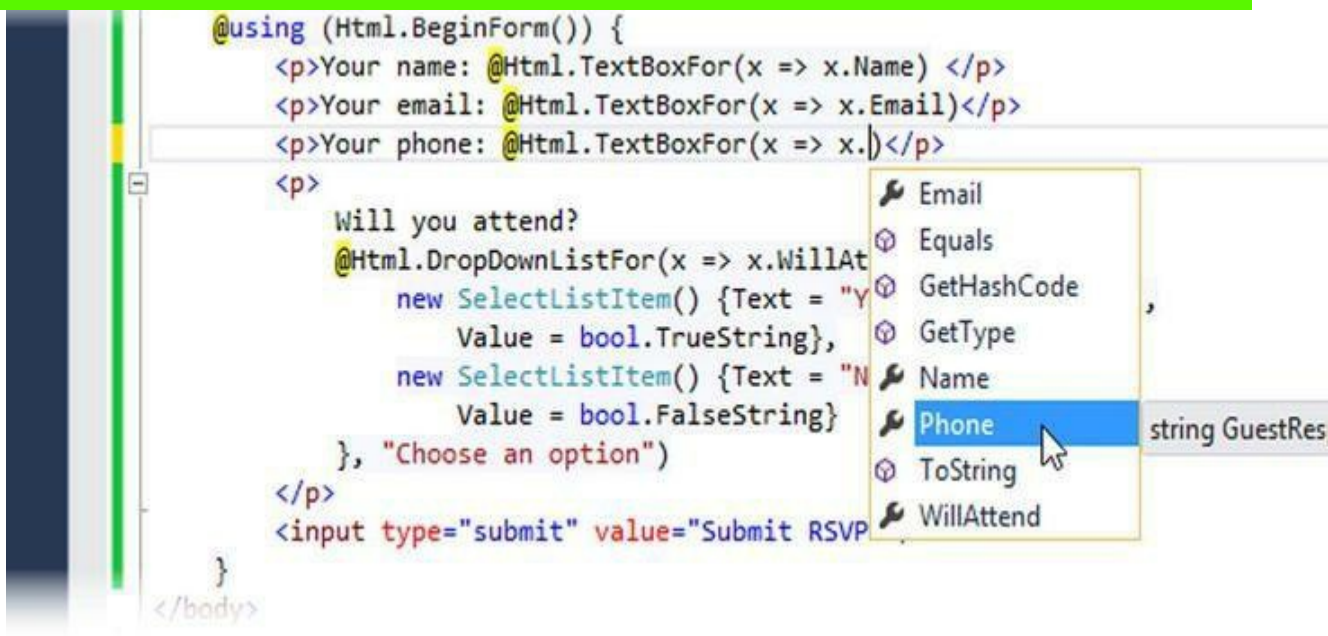


Figure 2-17. Visual Studio IntelliSense for lambda expressions in HTML helper methods

Another convenient helper method is `Html.BeginForm`, which generates an HTML `form` element configu back to the action method. Because I have not passed any arguments to the helper method, it assumes I want to post same URL that the HTML document was requested from. A neat trick is to wrap this in a C# `using` statement, lik

```
...
@using (Html.BeginForm()) {
    ... form contents go here ...
}
...
```

Normally, when applied like this, the `using` statement ensures that an object is disposed of when it goes out of commonly used for database connections, for example, to make sure that they are closed as soon as a query has con application of the `using` keyword is different from the kind that brings classes in a namespace into scope in a clas

Instead of disposing of an object, the `HtmlBeginForm` helper closes the HTML `form` element when it goe scope. This means that the `Html.BeginForm` helper method creates both parts of a form element, like this:

```
<form action="/Home/RsvpForm" method="post">
    ... form contents go here ...
</form>
```

Don't worry if you are not familiar with disposing of C# objects. The point here is to demonstrate how to create a the HTML helper method.

Setting the Start URL

Visual Studio will, in an effort to be helpful, make the browser request a URL based on the view that is currently bei is a hit-and-miss feature because it doesn't work when you are editing other kinds of file and because you can't just point in most complex web apps.

To set a fixed URL for the browser to request, select `PartyInvites Properties` from the Visual Stu `Project` menu, select the `Web` section and check the `Specific Page` option in the `Start Action` shown in Figure 2-18. You don't have to enter a value into the field–Visual Studio will request the default URL for th which will be directive to the `Index` action method on the `Home` controller. (I show you how to use the URL rou to change the default mapping in Chapters 15 and 16).

eBooks-IT.org

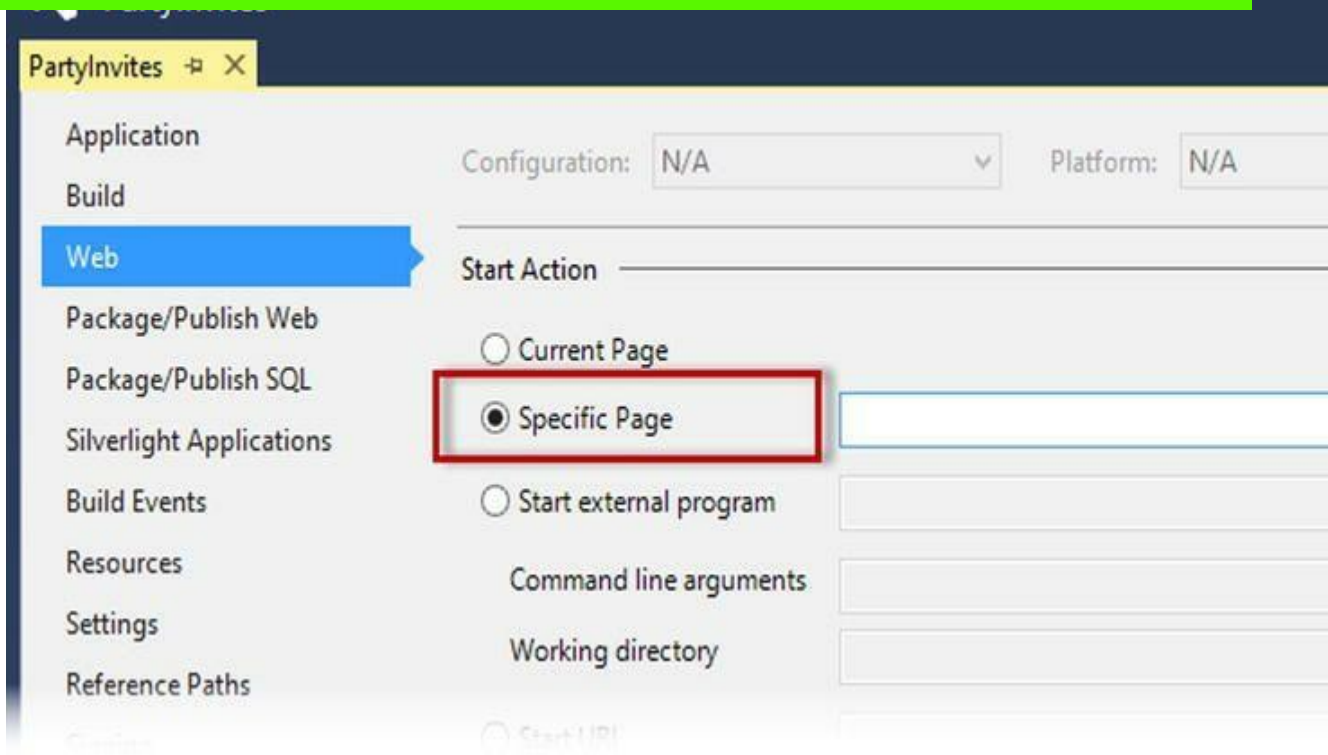


Figure 2-18. Setting the default start URL for the project

You can see the form in the `RsvpForm` view when you run the application and click the `RSVP Now` link. Fig shows the result.

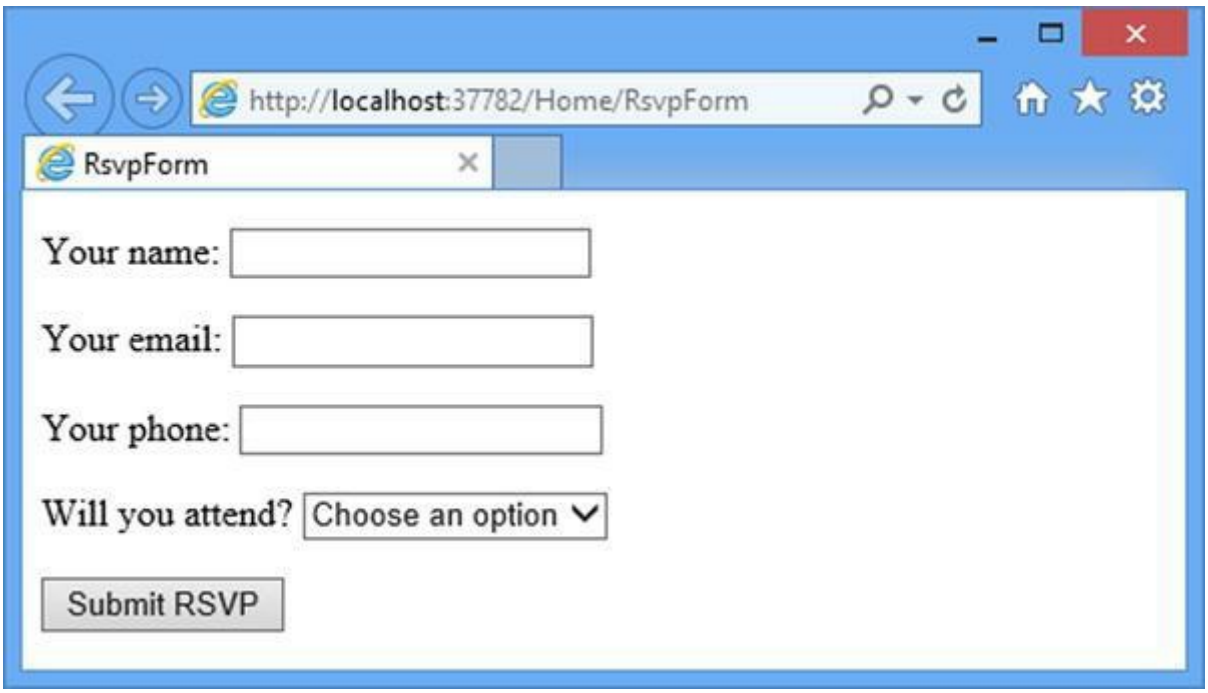


Figure 2-19. The RspvForm view

Handling Forms

I have not yet told MVC what I want to do when the form is posted to the server. As things stand, clicking the `Sub` `RSVP` button just clears any values you have entered into the form. That is because the form posts back to the `Rsv` action method in the `Home` controller, which just tells MVC to render the view again.

eBooks-IT.org

robably developing applications with ASP.NET Web Forms, which automatically preserves data in this situation. I will show achieve the same effect with MVC shortly.

To receive and process submitted form data, I am going to use a clever feature. I will add a second `RsvpForm` method in order to create the following:

- *A method that responds to HTTP* `GET` *requests*: A `GET` request is what a browser issues normally each clicks a link. This version of the action will be responsible for displaying the initial blank form when some `/Home/RsvpForm`.
- *A method that responds to HTTP* `POST` *requests*: By default, forms rendered using `Html.BeginF` submitted by the browser as a `POST` request. This version of the action will be responsible for receiving s and deciding what to do with it.

Handing `GET` and `POST` requests in separate C# methods helps to keep my controller code tidy, since the two m different responsibilities. Both action methods are invoked by the same URL, but MVC makes sure that the appropria called, based on whether I am dealing with a `GET` or `POST` request. Listing 2-14 shows the changes I applied to th `HomeController` class.

Listing 2-14. Adding an Action Method to Support POST Requests in the HomeController.cs File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            int hour = DateTime.Now.Hour;
                 ViewBag.Greeting = hour < 12 ? "Good Morning"
Afternoon";
            return View();
        }

        [HttpGet]
        public ViewResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestResponse) {
            // TODO: Email response to the party organizer
            return View("Thanks", guestResponse);
        }
    }
}
```

I have added the `HttpGet` attribute to my existing `RsvpForm` action method. This tells MVC that this metho used only for `GET` requests. I then added an overloaded version of `RsvpForm`, which takes a `GuestRespon` parameter and applies the `HttpPost` attribute. The attribute tells MVC that the new method will deal with `POST` also imported the `PartyInvites.Models` namespace—this is just so I can refer to the `GuestRespon`

Using Model Binding

The first overload of the RsvpForm action method renders the same view as before–the RsvpForm.cshtm generate the form shown in Figure 2-18.

The second overload is more interesting because of the parameter, but given that the action method will be invoke to an HTTP POST request, and that the GuestResponse type is a C# class, how are the two connected?

The answer is *model binding*, an extremely useful MVC feature whereby incoming data is parsed and the key/valu HTTP request are used to populate properties of domain model types. This process is the opposite of using the HTM methods; that is, when creating the form data to send to the client, I generated HTML input elements where the id and name attributes were derived from the model class property names.

In contrast, with model binding, the names of the input elements are used to set the values of the properties in the model class, which is then passed to the POST-enabled action method.

Model binding is a powerful and customizable feature that eliminates the grind and toil of dealing with HTTP reque and lets us work with C# objects rather than dealing with Request.Form[] and Request.QueryStri values. The GuestResponse object that is passed as the parameter to the action method is automatically popul data from the form fields. I dive into the detail of model binding, including how it can be customized, in Chapter 24.

Rendering Other Views

The second overload of the RsvpForm action method also demonstrates how to tell MVC to render a specific vie to a request, rather than the default view. Here is the relevant statement:

```
...
return View("Thanks", guestResponse);
...
```

This call to the View method tells MVC to find and render a view called Thanks and to pass the GuestRes object to the view. To create the view I specified, right-click on any of the HomeController methods and se View from the pop-up menu and use the Add View dialog to create a strongly typed view called Thanks tha GuestResponse model class and that is based on the Empty template. (See the Adding a Strongly Typed Vi step-by-step details if needed). Visual Studio will create the view as Views/Home/Thanks.cshtml. Edit so that it matches Listing 2-15—I have highlighted the markup you need to add.

Listing 2-15. The Contents of the Thanks.cshtml File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
</head>
<body>
    <div>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
```

eBooks-IT.org

```
fridge!
        } else {
            @:Sorry to hear that you can't make it, but thanks for
us know.
        }
    </div>
</body>
</html>
```

The `Thanks` view uses Razor to display content based on the value of the `GuestResponse` properties tha the `View` method in the `RsvpForm` action method. The Razor `@model` expression specifies the domain model view is strongly typed with. To access the value of a property in the domain object, I use `Model.PropertyN` example, to get the value of the `Name` property, I call `Model.Name`. Don't worry if the Razor syntax doesn't n explain it in more detail in Chapter 5.

Now that I have created the `Thanks` view, I have a basic working example of handling a form with MVC. Start application in Visual Studio, click the `RSVP Now` link, add some data to the form, and click the `Submit RSV` will see the result shown in Figure 2-20 (although it will differ if your name is not Joe or you said you could not atte

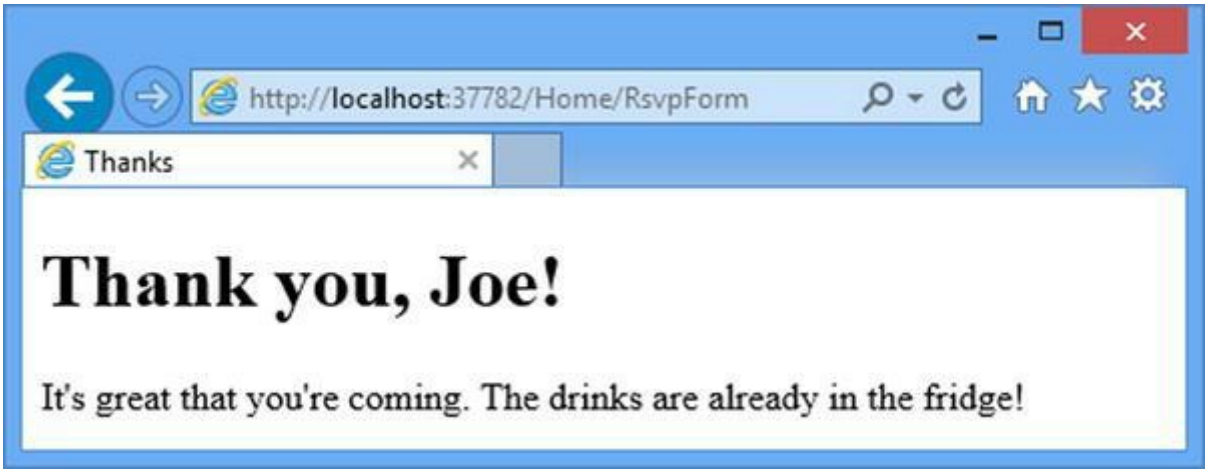


Figure 2-20. The Thanks view

Adding Validation

I am now in a position to add validation to my application. Without validation, users could enter nonsense data or eve empty form. In an MVC application, validation is typically applied in the domain model, rather than in the user interfa means that I am able to define validation criteria in one place and have it take effect anywhere in the application that t class is used. ASP.NET MVC supports *declarative validation rules* defined with attributes from the `System.ComponentModel.DataAnnotations` namespace, meaning that validation constraints are using the standard C# attribute features. Listing 2-16 shows how I applied these attributes to the `GuestRespon` class.

Listing 2-16. Applying Validation in the GuestResponse.cs File

```
using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {

    public class GuestResponse {

        [Required(ErrorMessage = "Please enter your name")]
        public string Name { get; set; }
```

eBooks-IT.org

```
        [RegularExpression(".+\\@.+\\..+",
            ErrorMessage = "Please enter a valid email address")]
        public string Email { get; set; }

        [Required(ErrorMessage = "Please enter your phone number")
        public string Phone { get; set; }

        [Required(ErrorMessage = "Please specify whether you'll at
        public bool? WillAttend { get; set; }
    }
}
```

The validations rules are shown in bold. MVC automatically detects the attributes and uses them to validate data du model-binding process. Notice that I have imported the namespace that contains the validations, so I can refer to the needing to qualify their names.

Tip As noted earlier, I used a nullable `bool` for the `WillAttend` property. I did this so that I could apply t `Required` validation attribute. If I had used a regular `bool`, the value I received through model binding could b or `false`, and I wouldn't be able to tell if the user had selected a value. A nullable `bool` has three possible values `false`, and `null`. The `null` value will be used if the user hasn't selected a value, and this causes the `Requi` to report a validation error. This is a nice example of how the MVC Framework elegantly blends C# features with HT HTTP.

I check to see if there has been a validation problem using the `ModelState.IsValid` property in the con Listing 2-17 shows how I have done this in the `POST`-enabled `RsvpForm` action method in the `Home` controller

Listing 2-17. Checking for Form Validation Errors in the HomeController.cs File

```
...
[HttpPost]
public ViewResult RsvpForm(GuestResponse guestResponse) {
    if (ModelState.IsValid) {
        // TODO: Email response to the party organizer
        return View("Thanks", guestResponse);
    } else {
        // there is a validation error
        return View();
    }
}
...
```

If there are no validation errors, I tell MVC to render the `Thanks` view, just as I did previously. If there *are* vali I re-render the `RsvpForm` view by calling the `View` method without any parameters.

Just displaying the form when there is an error is not helpful—I also need to provide the user with some indication problem is and why I could not accept their form submission. I do this by using the `Html.ValidationSum` method in the `RsvpForm` view, as shown in Listing 2-18.

Listing 2-18. Using the Html.ValidationSummary Helper Method in the RsvpForm.cshtml File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}
```

eBooks-IT.org

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm()) {
        @Html.ValidationSummary()
        <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
        <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
        <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
        <p>
            Will you attend?
            @Html.DropDownListFor(x => x.WillAttend, new[] {
                new SelectListItem() {Text = "Yes, I'll be there",
                    Value = bool.TrueString},
                new SelectListItem() {Text = "No, I can't come",
                    Value = bool.FalseString}
            }, "Choose an option")
        </p>
        <input type="submit" value="Submit RSVP" />
    }
</body>
</html>
```

If there are no errors, the `Html.ValidationSummary` method creates a hidden list item as a placeholde MVC makes the placeholder visible and adds the error messages defined by the validation attributes. You can see hov in Figure 2-21.

eBooks-IT.org

Figure 2-21. The validation summary

The user won't be shown the `Thanks` view until all of the validation constraints applied to the `GuestRespo` have been satisfied. Notice that the data entered into the form was preserved and displayed again when the view was with the validation summary. This is another benefit of the model binding feature and it simplifies working with form

Note If you have worked with ASP.NET Web Forms, you will know that Web Forms has a concept of *server c* retain state by serializing values into a hidden form field called `__VIEWSTATE`. ASP.NET MVC model binding is the Web Forms concepts of server controls, postbacks, or View State. ASP.NET MVC does not inject a hidden `__VIEWSTATE` field into your rendered HTML pages.

Highlighting Invalid Fields

The HTML helper methods that create text boxes, drop-downs, and other elements have a handy feature that can be conjunction with model binding. The same mechanism that preserves the data that a user entered in a form can also highlight individual fields that failed the validation checks.

When a model class property has failed validation, the HTML helper methods will generate slightly different HTML example, here is the HTML that a call to `Html.TextBoxFor(x => x.Name)` generates when there is n error:

```
<input data-val="true" data-val-required="Please enter your
id="Name" name="Name"
    type="text" value="" />
```

And here is the HTML the same call generates when the user doesn't provide a value (which is a validation error b applied the `Required` attribute to the `Name` property in the `GuestResponse` model class):

```
<input class="input-validation-error" data-val="true" data-val-required="Plea
your name" id="Name" name="Name" type="text" value="" />
```

I have highlighted the difference in bold: the helper method added a class called `input-validation-er`

eBooks-IT.org

ass and

that different HTML helper methods apply.

The convention in MVC projects is that static content, such as CSS style sheets, is placed into a folder called Cor
Create this folder by right-clicking on the `PartyInvites` item in the Solution Explorer, selecting `Add` ➤ `New`
from the menu and setting the name to `Content`.

To create the CSS file, right click on the newly created `Content` folder, select `Add` ➤ `New Item` from th
choose `Style Sheet` from the set of item templates. Set the name of the new file to `Styles.css`, as show
2-22.

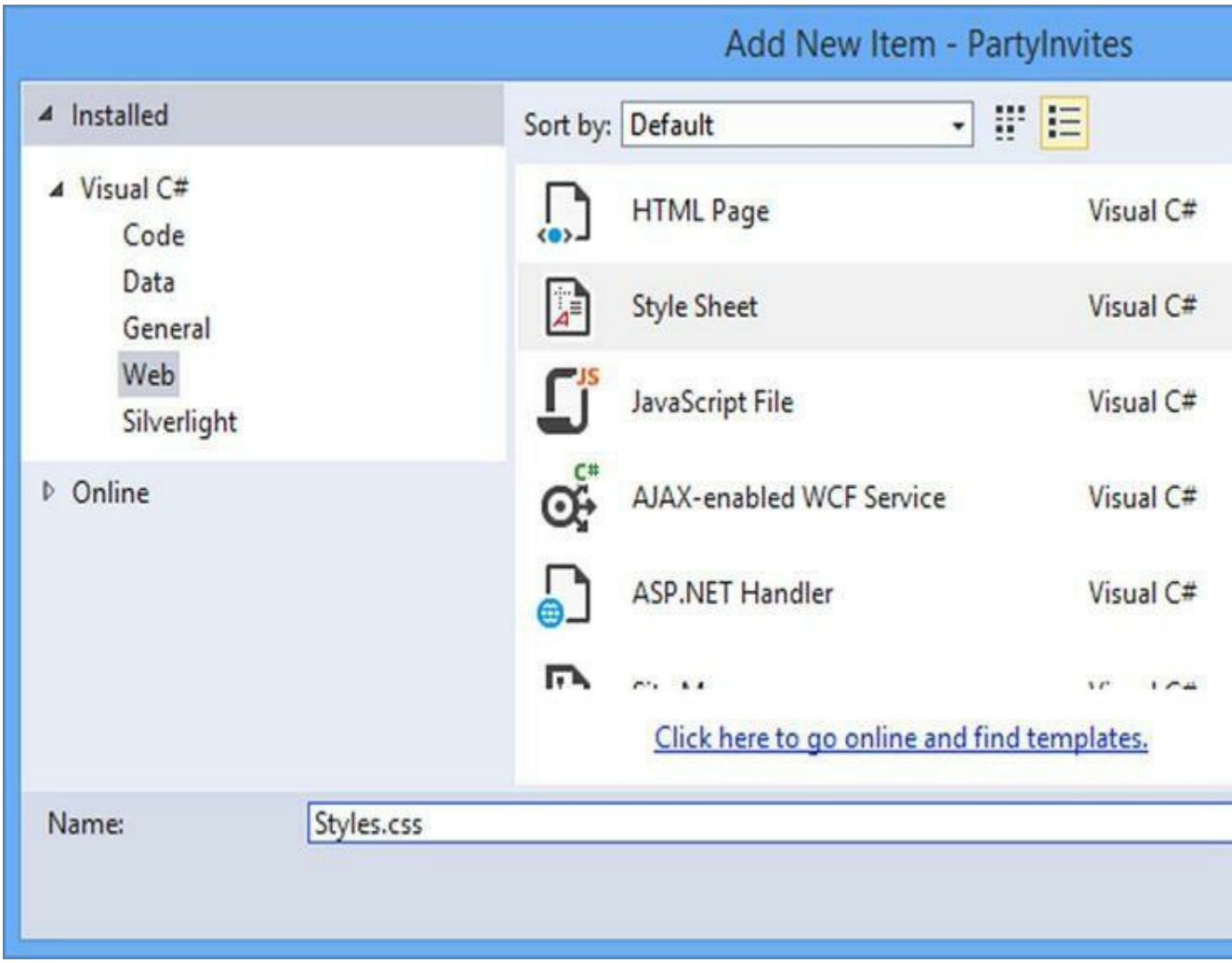


Figure 2-22. Creating a new style sheet

Click the `Add` button and Visual Studio will create the `Content/Styles.css` file. Set the content of the
match Listing 2-19.

Listing 2-19. The Contents of the Styles.css File

```
.field-validation-error     {color: #f00;}
.field-validation-valid     { display: none;}
.input-validation-error       { border: 1px solid #f00; backgroun
#fee; }
.validation-summary-errors { font-weight: bold; color: #f00;}
.validation-summary-valid  { display: none;}
```

To use this style sheet, I add a new reference to the `head` section of `RsvpForm` view, as shown in Listing 2-
link elements to views just as you would to a regular static HTML file, although in Chapter 27, I show you the *bundl*
allows JavaScript and CSS style sheets to be consolidated and delivered to the browsers over a single HTTP request.

eBooks-IT.org

Studio will create `script` and `link` elements for the files you have selected.

Listing 2-20. Adding the Link Element in the RsvpForm.cshtml File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" type="text/css" href="~/Content/Styles.
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm()) {
        @Html.ValidationSummary()
        <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
        <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
        <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
        <p>
            Will you attend?
            @Html.DropDownListFor(x => x.WillAttend, new[] {
                new SelectListItem() {Text = "Yes, I'll be there",
                    Value = bool.TrueString},
                new SelectListItem() {Text = "No, I can't come",
                    Value = bool.FalseString}
            }, "Choose an option")
        </p>
        <input type="submit" value="Submit RSVP" />
    }
</body>
</html>
```

Tip If you have moved to MVC 5 directly from MVC 3, you might have been expecting us to have added the CS view by specifying the `href` attribute as `@Href("~/Content/Site.css")` or `@Url.Content("~/Content/Site.css")`. As of MVC 4, Razor detects attributes that begin with ~ automatically inserts the `@Href` or `@Url` call for you.

With the application of the style sheet, a more visually obvious validation error will be displayed when data is subm causes a validation error, as shown in Figure 2-23.

eBooks-IT.org

Figure 2-23. Automatically highlighted validation errors

Styling the Content

The basic functionality of the application is in place – except for sending emails, which I'll get to shortly–but the ove appearance is pretty poor. Although this is a book focused on server-side development, Microsoft has adopted a num source libraries and included them in some of the Visual Studio project templates.

I am not a fan of these templates, but I do like some of the libraries they use and one of the new adoptees in MVC Bootstrap, which is a nice CSS library originally developed by Twitter that has become widely used.

You don't have to use the Visual Studio project templates to use libraries like Bootstrap, of course. You can downl directly from project web sites or use NuGet, which is integrated into Visual Studio and provides access to a catalog packaged software that can be downloaded and installed automatically. One of the best NuGet features is that it mana dependencies between packages such that if you install Bootstrap, for example, NuGet will also download and install some Bootstrap features depend on.

Using NuGet to Install Bootstrap

To install the Bootstrap package, select `Library Package Manager` ➤ `Package Manager Co` from the Visual Studio `Tools` menu. Visual Studio will open the NuGet command line. Enter the following comma return:

```
Install-Package -version 3.0.0 bootstrap
```

The `Install-Package` command tells NuGet to download a package and its dependencies and add them t The name of the package I want is called `bootstrap`, and you can search for package names either from the Nu (http://www.nuget.org) or using the Visual Studio NuGet user interface (select `Tools` ➤ `Librar` `Package Manager` ➤ `Manage NuGet Packages for Solution`).

I have used `-version` to specify that I want Bootstrap version 3, which is the latest stable version available as Without `-version`, NuGet would have downloaded the latest version of the package, but I want to make sure th able to recreate the examples exactly as I have shown them and so installing a specific version helps me to ensure co

NuGet will download all of the files required for Bootstrap and for jQuery, which Bootstrap relies on. CSS files are `Content` folder and a `Scripts` folder is created (which is the standard MVC location for JavaScript files) and with Bootstrap and jQuery files. (A `fonts` folder is also created–this is a quirk of the Bootstrap typography feature

Note The reason that I am showing you Bootstrap in this chapter is to illustrate how readily the HTML generated Framework can be used with popular CSS and JavaScript libraries. I don't want to lose my focus on server-side dev however, and so if you want complete details of the *client*-side aspects of working with the MVC Framework, then *Pro ASP.NET MVC 5 Client*, which will be published by Apress in 2014.

Styling the Index View

The basic Bootstrap features work by applying classes to elements that correspond to CSS selectors defined in the fi the `Content` folder. You can get full details of the classes that Bootstrap defines from `http://getbootstrap.com`, but you can see how I have applied some basic styling to the `Index.cs` file in Listing 2-21.

Listing 2-21. Adding Bootstrap to the Index.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <title>Index</title>
    <style>
        .btn a { color: white; text-decoration: none}
        body { background-color: #F1F1F1; }
    </style>
</head>
<body>
    <div class="text-center">
        <h2>We're going to have an exciting party!</h2>
        <h3>And you are invited</h3>
        <div class="btn btn-success">
            @Html.ActionLink("RSVP Now", "RsvpForm")
        </div>
    </div>
</body>
</html>
```

I have added `link` elements for the `bootstrap.css` and `bootstrap-theme.css` files in the `Co` folder. These are the Bootstrap files required for the basic CSS styling that the library provides and there is a corresp JavaScript file in the `Scripts` folder, but I won't need it in this chapter. I have also defined a `style` element th background color for the `body` element and styles the text for `a` elements.

Tip You will notice that each of the Bootstrap files in the `Content` folder has a twin with the prefix `min`–e.g `bootstrap.css` and `bootstrap.min.cs`. It is common practice to *minify* JavaScript and CSS files w deploying an application into production, which is a process of removing all of the whitespace and, in the case of Jav replacing the function and variable names with shorter labels. The goal of minification is to reduce the amount of ban required to deliver your content to the browser and in Chapter 27, I describe the ASP.NET features for managing thi automatically. For this chapter–and most of the other chapters in this book–I will use the regular files, which is norm

Having imported the Bootstrap styles and defined a couple of my own, I need to style my elements. This is a simpl and so I only need to use three Bootstrap CSS classes: `text-center`, `btn` and `btn-success`.

The `text-center` class centers the content of an element and its children. The `btn` class styles a `butto` or `a` element as a pretty button and the `btn-success` specifies which of a range of colors I want the button to of the button depends on the theme that is being used–I have the default theme (as defined by the `bootstrap-theme.css` file), but there are endless replacements available with a search online. You can see the effect that I in Figure 2-24.

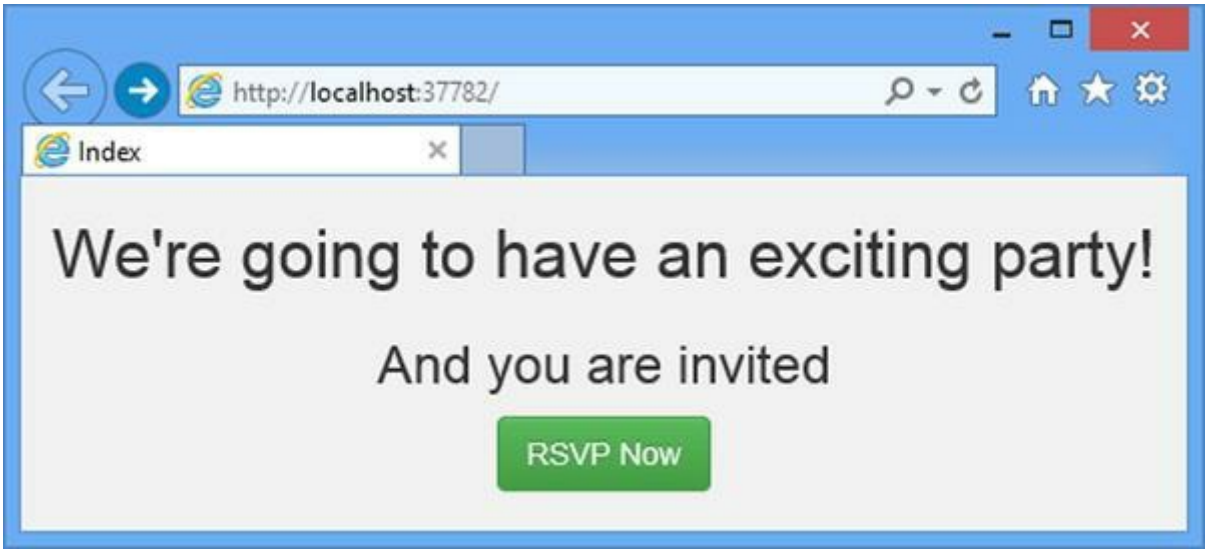


Figure 2-24. Styling the Index view

It will be obvious to you that I am not a web designer. In fact, as a child, I was excused from art lessons on the b absolutely no talent whatsoever. This had the happy result of making more time for math lessons but meant that my have not developed beyond those of the average 10 year old. For a real project, I would seek a professional to help d style the content, but for this example I am going it alone and that means applying Bootstrap with as much restraint a consistency as I can muster.

Styling the RsvpForm View

Bootstrap defines classes that can be used to style forms. I am not going to go into detail, but you can see how I hav these classes in Listing 2-22.

Listing 2-22. Adding Bootstrap to the RsvpForm.cshtml File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Styles.css" rel="stylesheet" />
    <title>RsvpForm</title>
```

```
<body>
    <div class="panel panel-success">
        <div class="panel-heading text-center"><h4>RSVP</h4></div>
        <div class="panel-body">
            @using (Html.BeginForm()) {
                @Html.ValidationSummary()
                <div class="form-group">
                    <label>Your name:</label>
                      @Html.TextBoxFor(x => x.Name, new { @class =
control"})
                </div>
                <div class="form-group">
                    <label>Your email:</label>
                     @Html.TextBoxFor(x => x.Email, new { @class
control"})
                </div>
                <div class="form-group">
                    <label>Your phone:</label>
                     @Html.TextBoxFor(x => x.Phone, new { @class
control"})
                </div>
                <div class="form-group">
                    <label>Will you attend?</label>
                    @Html.DropDownListFor(x => x.WillAttend, new[]
                                new SelectListItem() {Text = "Yes,
there",
                                 Value = bool.TrueString},
                                 new SelectListItem() {Text = "No,
come",
                                 Value = bool.FalseString}
                            }, "Choose an option", new { @class =
control" })
                </div>
                <div class="text-center">
                            <input class="btn btn-success" type=
value="Submit RSVP" />
                </div>
            }
        </div>
    </div>
</body>
</html>
```

The Bootstrap classes in this example create a panel with a header, just to give structure to the layout. To style the used the `form-group` class, which is used to style the element that contains the `label` and the associated in `select` element.

These elements are created using HTML helper methods, which means that there are not statically defined element which I can apply the required `form-control` class. Fortunately, the helper methods take an optional object ar lets me specify attributes on the elements that they create, as follows:

```
...
@Html.TextBoxFor(x => x.Name, new { @class = "form-control"})
...
```

eBooks-IT.org

he `cl`
should be set to `form-control` on the element that the `TextBoxFor` helper generates. The properties defi
object are used for the name of the attribute added to the HTML element and `class` is a reserved word in the C# l
have to prefix it with `@`. This is a standard C# feature that allows keywords to be used in expressions. You can see t
my styles in Figure 2-25.

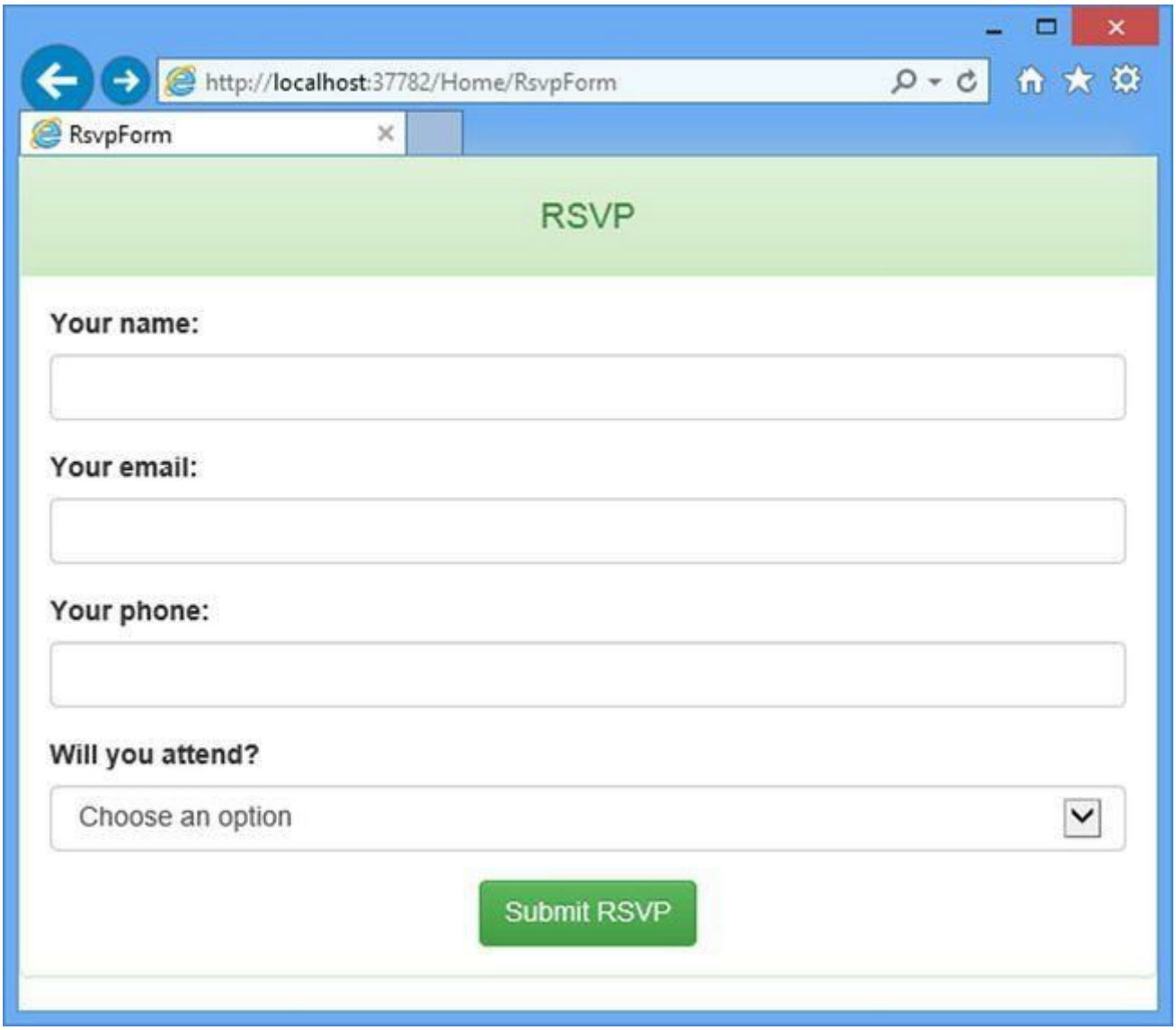


Figure 2-25. Styling the RsvpForm view

Styling the Thanks View

The last view file to style is `Thanks.cshtml` and you can see how I have done this in Listing 2-23. You will n
markup I have added is similar to that in the `Index.cshtml` view. To make an application easier to manage, it
principal to avoid duplicating code and markup wherever possible and in Chapter 5 I will introduce you to Razor *layo*
Chapter 20, I describe *partial views*, both of which can be used to reduce duplication of markup.

Listing 2-23. Applying Bootstrap to the Thanks.cshtml File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}
```

```
<html>
<head>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
    <style>
        body { background-color: #F1F1F1; }
    </style>
</head>
<body>
    <div class="text-center">
        <h1>Thank you, @Model.Name!</h1>
        <div class="lead">
            @if (Model.WillAttend == true) {
                @:It's great that you're coming. The drinks are al
the fridge!
            } else {
                @:Sorry to hear that you can't make it, but th
letting us know.
            }
        </div>
    </div>
</body>
</html>
```

The `lead` class applies one of the Bootstrap typographic styles and you can see the effect in Figure 2-26.

Figure 2-26. Styling the Thanks View

Completing the Example

The last requirement for my example application is to e-mail completed RSVPs to the party organizer. I could do this action method to create and send an e-mail message using the e-mail classes in the .NET Framework–and that would technique which is most consistent with the MVC pattern. Instead, I am going to use the `WebMail` helper method part of the MVC framework, but it does let me complete this example without getting mired in the details of setting u of sending e-mail. I want the e-mail message to be sent as I render the `Thanks` view. Listing 2-24 show the chang to apply.

Listing 2-24. Using the WebMail Helper in the Thanks.cshtml File

eBooks-IT.org

```
<body>
    @{
        try {
            WebMail.SmtpServer = "smtp.example.com";
            WebMail.SmtpPort = 587;
            WebMail.EnableSsl = true;
            WebMail.UserName = "mySmtpUsername";
            WebMail.Password = "mySmtpPassword";
            WebMail.From = "rsvps@example.com";

            WebMail.Send("party-host@example.com", "RSVP Notificat
                Model.Name + " is " + ((Model.WillAttend ?? false
"not")
                    + "attending");

        } catch (Exception) {
             @:<b>Sorry - we couldn't send the email to confirm yo
</b>
        }
    }
    <div class="text-center">
        <h1>Thank you, @Model.Name!</h1>
        <div class="lead">
            @if (Model.WillAttend == true) {
                @:It's great that you're coming. The drinks are al
the fridge!
            } else {
                  @:Sorry to hear that you can't make it, but th
letting us know.
            }
        </div>
    </div>
</body>
...
```

Note I used the `WebMail` helper because it lets us demonstrate sending an e-mail message with a minimum of Typically, however, I would prefer to put this functionality in an action method. I will explain why when I describe t architecture pattern in Chapter 3.

I have added a Razor expression that uses the `WebMail` helper to configure the details of my e-mail server, incl server name, whether the server requires SSL connections, and account details. Once I have configured all of the de `WebMail.Send` method to send the e-mail.

I enclosed all of the e-mail code in a `try...catch` block so that I can alert the user if the e-mail is not sent. adding a block of text to the output of the `Thanks` view. A better approach would be to display a separate error vi e-mail message cannot be sent, but I wanted to keep things simple for this first MVC application.

Summary

In this chapter, I created a new MVC project and used it to construct a simple MVC data-entry application, giving yo glimpse of the MVC Framework architecture and approach. I skipped over some key features (including Razor synta and automated testing), but I come back to these topics in depth in later chapters. In the next chapter, I describe the architecture, design patterns, and techniques that I use throughout the rest of this book and which form the foundati effective development with the MVC Framework.

eBooks-IT.org

The MVC Pattern

Before I start digging into the details of the ASP.NET MVC Framework, I want to make sure you are familiar with th design pattern and the thinking behind it. In this chapter, I describe the following:

- The MVC architecture pattern
- Domain models and repositories
- Creating loosely coupled systems using dependency injection (DI)
- The basics of automated testing

You might be familiar with some of the ideas and conventions I discuss in this chapter, especially if you have done ASP.NET or C# development. If not, I encourage you to read carefully—a good understanding of what lies behind M put the features of the framework into context as you continue through the book.

The History of MVC

The term *model-view-controller* has been in use since the late 1970s and arose from the Smalltalk project at Xerox P was conceived as a way to organize some early GUI applications. Some of the fine detail of the original MVC pattern Smalltalk-specific concepts, such as *screens* and *tools*, but the broader concepts are still applicable to applications—a especially well suited to Web applications.

Interactions with an MVC application follow a natural cycle of user actions and view updates, where the view is a stateless. This fits nicely with the HTTP requests and responses that underpin a Web application.

Further, MVC forces a *separation of concerns*—the domain model and controller logic are decoupled from the use a Web application, this means that the HTML is kept apart from the rest of the application, which makes maintenanc simpler and easier. It was Ruby on Rails that led to renewed mainstream interest in MVC and it remains the implemen template for the MVC pattern. Many other MVC frameworks have since emerged and demonstrated the benefits of M including, of course, ASP.NET MVC.

Understanding the MVC Pattern

In high-level terms, the MVC pattern means that an MVC application will be split into at least three pieces:

- *Models*, which contain or represent the data that users work with. These can be simple *view models*, which data being transferred between views and controllers; or they can be *domain models*, which contain the data domain as well as the operations, transformations, and rules for manipulating that data.
- *Views,* which are used to render some part of the model as a user interface.
- *Controllers*, which process incoming requests, perform operations on the model, and select views to render t

Models are the definition of the universe your application works in. In a banking application, for example, the mod everything in the bank that the application supports, such as accounts, the general ledger, and credit limits for custom as the operations that can be used to manipulate the data in the model, such as depositing funds and making withdraw accounts. The model is also responsible for preserving the overall state and consistency of the data—for example, m

eBooks-IT.org

to or n

Models are also defined by what they are *not* responsible for: models don't deal with rendering UIs or processing r those are the responsibilities of *views* and *controllers. Views* contain the logic required to display elements of the mod —and nothing more. They have no direct awareness of the model and do not directly communicate with the model i *Controllers* are the bridge between views and the model—requests come in from the client and are serviced by the co which selects an appropriate view to show the user and, if required, an appropriate operation to perform on the mode

Each piece of the MVC architecture is well-defined and self-contained—this is referred to as the *separation of con* logic that manipulates the data in the model is contained *only* in the model; the logic that displays data is *only* in the v code that handles user requests and input is contained *only* in the controller. With a clear division between each of th application will be easier to maintain and extend over its lifetime, no matter how large it becomes.

Understanding the Domain Model

The most important part of an MVC application is the domain model. We create the model by identifying the real-wo operations, and rules that exist in the industry or activity that the application must support, known as the *domain*.

We then create a software representation of the domain: the *domain model*. For the purposes of the ASP.NET MV Framework, the domain model is a set of C# types (classes, structs, etc.), collectively known as the *domain types*. T from the domain are represented by the methods defined in the domain types, and the domain rules are expressed in t inside of these methods—or, as you saw in the previous chapter, by applying C# attributes. When an instance of a do created to represent a specific piece of data, it is called a *domain object*. Domain models are usually persistent and lo there are lots of different ways of achieving this, but relational databases remain the most common choice.

In short, a domain model is the single, authoritative definition of the business data and processes within your applic *persistent* domain model is also the authoritative definition of the state of your domain representation.

The domain model approach solves many of the problems that arise when maintaining an application. If you need t the data in your model or add a new process or rule, the domain model is the only part of your application that has to

Tip A common way of enforcing the separation of the domain model from the rest of an ASP.NET MVC applica place the model in a separate C# assembly. In this way, you can create references *to* the domain model from other pa application but ensure that there are no references in the other direction. This is particularly useful in large-scale proje demonstrate this approach in the example I start building in Chapter 7.

The ASP.NET Implementation of MVC

In MVC, controllers are C# classes, usually derived from the `System.Web.Mvc.Controller` class. Eac method in a class derived from `Controller` is an *action method*, which is associated with a configurable URL t ASP.NET routing system. When a request is sent to the URL associated with an action method, the statements in the class are executed in order to perform some operation on the domain model and then select a view to display to the c 3-1 shows the interactions between the controller, model, and view.

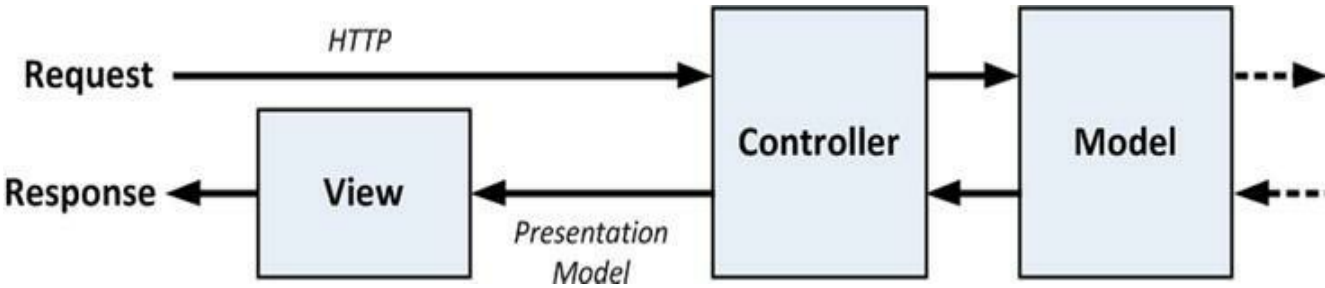


Figure 3-1. The interactions in an MVC application

The ASP.NET MVC Framework uses a *view engine*, which is the component responsible for processing a view in generate a response for the browser. Earlier versions of MVC used the standard ASP.NET view engine, which proce pages using a streamlined version of the Web Forms markup syntax. MVC 3 introduced the Razor view engine, whic in MVC 4 (and unchanged in MVC5) and that uses a different syntax entirely, which I describe in Chapter 5).

Tip Visual Studio provides IntelliSense support for Razor, making it a simple matter to inject and respond to view by the controller.

eBooks-IT.org

e a mod
regular C# objects and implement persistence using any of the databases, object relational mapping frameworks, or c
tools supported by .NET.

Comparing MVC to Other Patterns

MVC is not the only software architecture pattern, of course. There are many others and some of them are, or at lea extremely popular. We can learn a lot about MVC by looking at the alternatives. In the following sections, I briefly de different approaches to structuring an application and contrast them with MVC. Some of the patterns are close variat MVC theme, whereas others are entirely different.

I am not suggesting that MVC is the perfect pattern for all situations. I am a proponent of picking the best approac problem at hand. As you will see, there are situations where some competing patterns are as useful as or better than encourage you to make an *informed* and *deliberate* choice when selecting a pattern. The fact that you are reading thi suggests that you already have a certain commitment to the MVC pattern, but it is always helpful to maintain the wid perspective.

Understanding the Smart UI Pattern

One of the most common design patterns is known as the *smart user interface* (smart UI). Most programmers have smart UI application at some point in their careers—I certainly have. If you have used Windows Forms or ASP.NET you have too.

To build a smart UI application, developers construct a user interface, often by dragging a set of *components* or *co* design surface or canvas. The controls report interactions with the user by emitting events for button presses, keystr movements, and so on. The developer adds code to respond to these events in a series of *event handlers*: small block are called when a specific event on a specific component is emitted. This creates a monolithic application, as shown The code that handles the user interface and the business is all mixed together with no separation of concerns at all. defines the acceptable values for a data input, that queries for data or modifies a user account, ends up in little pieces together by the order in which events are expected.

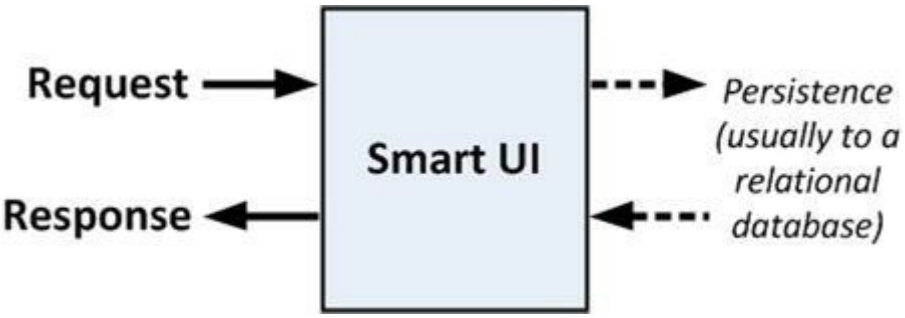


Figure 3-2. The Smart UI pattern

Smart UIs are ideal for simple projects because you can get some good results fast (by comparison to MVC devel which, as you'll see in Chapter 7, requires some careful preparation and initial investment before getting results). Sm also suited to user interface prototyping. These design surface tools can be *really* good, although I always find the W design surface in Visual Studio to be awkward and unpredictable. If you are sitting with a customer and want to cap requirements for the look and flow of the interface, a Smart UI tool can be a quick and responsive way to generate a different ideas.

The biggest drawback is that Smart UIs are difficult to maintain and extend. Mixing the domain model and busines in with the user interface code leads to duplication, where the same fragment of business logic is copied and pasted t newly added component. Finding all of the duplicate parts and applying a fix can be difficult. It can be almost impos new feature without breaking an existing one. Testing a Smart UI application can also be difficult. The only way is to interactions, which is far from ideal and a difficult basis from which to provide full test coverage.

In the world of MVC, the Smart UI is often referred to as an *anti-pattern*: something that should be avoided at all antipathy arises, at least in part, because people come to MVC looking for an alternative after spending part of their c to develop and maintain Smart UI applications.

Although it is a common point of view, it is overly simplistic and it is a mistake to reject the Smart UI pattern out everything is rotten in the Smart UI pattern and there are positive aspects to this approach. Smart UI applications are

eBooks-IT.org

nt expe
pleasant one, and even the most inexperienced programmer can produce something professional-looking and reasona
in just a few hours.

The biggest weakness of Smart UI applications—maintainability—doesn't arise in small development efforts. If yo
producing a simple tool for a small audience, a Smart UI application can be a perfect solution. The additional complex
MVC application simply isn't warranted.

Understanding the Model-View Architecture

The area in which maintenance problems tend to arise in a Smart UI application is in the business logic, which ends u
across the application that making changes or adding features becomes a fraught process. An improvement in this a
by the *model-view* architecture, which pulls out the business logic into a separate domain model. In doing this, the da
and rules are all concentrated in one part of the application, as shown in Figure 3-3.

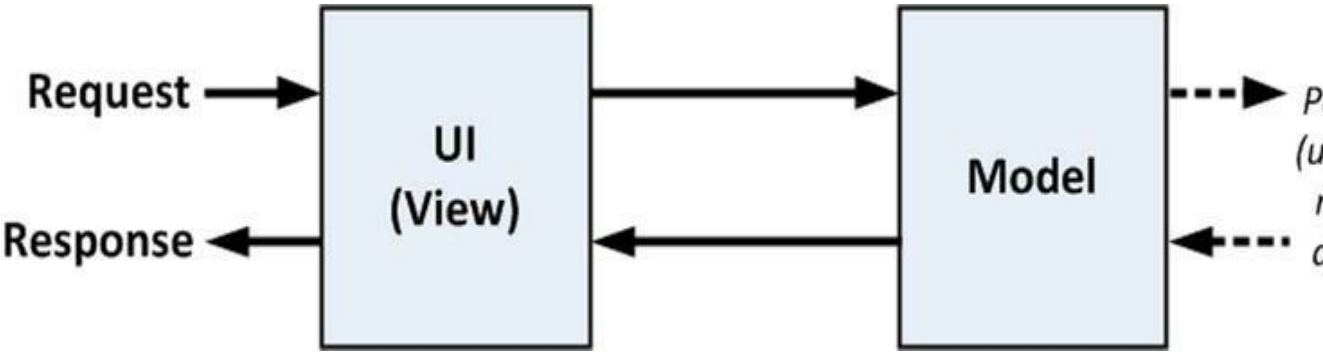


Figure 3-3. The model-view pattern

The model-view architecture can be an improvement over the monolithic Smart UI pattern—it is much easier to m
example–but two problems arise. The first is that since the UI and the domain model are closely integrated, it can be
perform unit testing on either. The second problem arises from practice, rather than the definition of the pattern. Th
typically contains a mass of data access code—this need not be the case, but it usually is—and this means that the da
does not contain just the business data, operations, and rules.

Understanding Classic Three-Tier Architectures

To address the problems of the model-view architecture, the *three-tier* or *three-layer* pattern separates the persistenc
the domain model and places it in a new component called the *ddata access layer* (DAL). This is shown in Figure 3-4

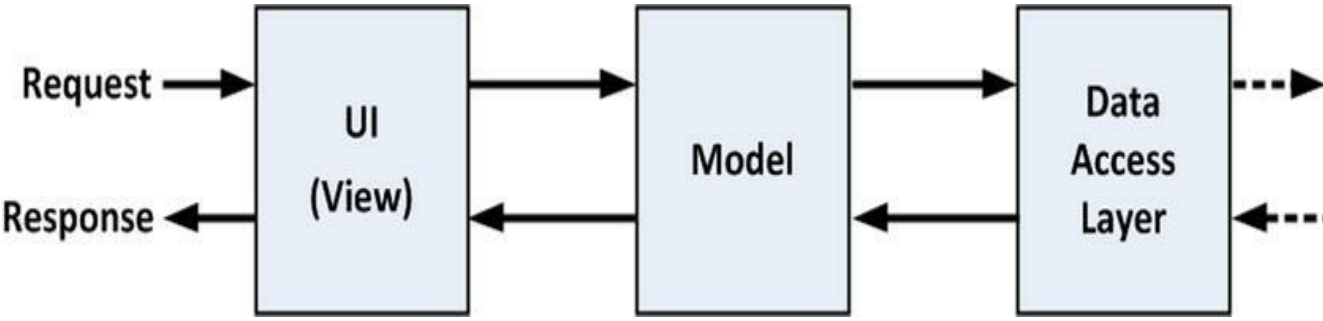


Figure 3-4. The three-tier pattern

The three-tier architecture is the most widely used pattern for business applications. It has no constraints on how
implemented and provides good separation of concerns without being too complicated. And, with some care, the DA
created so that unit testing is relatively easy. You can see the obvious similarities between a classic three-tier applicat
MVC pattern. The difference is that when the UI layer is directly coupled to a click-and-event GUI framework (such
Forms or ASP.NET Web Forms), it becomes almost impossible to perform automated unit tests. And because the UI
three-tier application can be complex, there's a lot of code that can't be rigorously tested.

In the worst scenario, the three-tier pattern's lack of enforced discipline in the UI tier means that many such applic

eBooks-IT.org

utcome

untestable, unmaintainable application that is excessively complex.

Understanding Variations on MVC

I have already described the core design principles of MVC applications, especially as they apply to the ASP.NET MV implementation. Others interpret aspects of the pattern differently and have added to, adjusted, or otherwise adapted the scope and subject of their projects. In the following sections, I provide a brief overview of the two most prevalen on the MVC theme. Understanding these variations is not essential to working with ASP.NET MVC and I have includ information just for completeness because you will hear the terms used in most discussions of software patterns.

Understanding the Model-View-Presenter Pattern

Model-view-presenter (MVP) is a variation on MVC that is designed to fit more easily with stateful GUI platforms su Windows Forms or ASP.NET Web Forms. This is a worthwhile attempt to get the best aspects of the Smart UI patt the problems it usually brings.

In this pattern, the presenter has the same responsibilities as an MVC controller, but it also takes a more direct rela stateful view, directly managing the values displayed in the UI components according to the user's inputs and actions two implementations of this pattern:

- The *passive view* implementation, in which the view contains no logic—it is a container for UI controls th manipulated by the presenter.
- The *supervising controller* implementation, in which the view may be responsible for some elements of pres such as data binding, and has been given a reference to a data source from the domain models.

The difference between these two approaches relates to how intelligent the view is. Either way, the presenter is de the GUI framework, which makes the presenter logic simpler and suitable for unit testing.

Understanding the Model-View-View Model Pattern

The *model-view-view model* (MVVM) pattern is the most recent variation on MVC. It originated from Microsoft and Windows Presentation Foundation (WPF). In the MVVM pattern, models and views have the same roles as they do i difference is the MVVM concept of a *view model*, which is an abstract representation of a user interface—typically a exposes both properties for the data to be displayed in the UI and operations on the data that can be invoked from the MVC controller, an MVVM view model has no notion that a view (or any specific UI technology) exists. An MVVM WPF *binding* feature to bi-directionally associate properties exposed by controls in the view (items in a drop-down n effect of pressing a button) with the properties exposed by the view model.

Tip MVC also uses the term *view model* but refers to a simple model class that is used only to pass data from a c view, as opposed to *domain models*, which are sophisticated representations of data, operations, and rules.

Building Loosely Coupled Components

One of most important features of the MVC pattern is that it enables separation of concerns. I want the components applications to be as independent as possible and to have as few interdependencies as I can arrange.

In an ideal situation, each component knows nothing about any other component and only deals with other areas o application through abstract interfaces. This is known as *loose coupling*, and it makes testing and modifying applicat

A simple example will help put things in context. If I am writing a component called `MyEmailSender` that w mails, I would implement an interface that defines all of the public functions required to send an e-mail, which I wou `IEmailSender`.

Any other component of my application that needs to send an e-mail—let's say a password reset helper called `PasswordResetHelper`—can then send an e-mail by referring only to the methods in the interface. There i dependency between `PasswordResetHelper` and `MyEmailSender`, as shown by Figure 3-5.

eBooks-IT.org

Figure 3-5. Using interfaces to decouple components

By introducing `IEmailSender`, I ensure that there is no direct dependency between `PasswordReset` `MyEmailSender`. I could replace `MyEmailSender` with another e-mail provider or even use a mock imp for testing purposes without needing to make any changes to `PasswordResetHelper`. (I introduce mock implementations later in this chapter and return to them again in Chapter 6).

Using Dependency Injection

Interfaces help decouple components, but I still face a problem: C# doesn't provide a built-in way to easily create ob implement interfaces, except to create an instance of the concrete component with the `new` keyword. I end up with this:

```
public class PasswordResetHelper {

    public void ResetPassword() {

        IEmailSender mySender = new MyEmailSender() ;

        //...call interface methods to configure e-mail details...

        mySender.SendEmail();
    }
}
```

This undermines my goal of being able to replace `MyEmailSender` without having to change `Password` helper and means that I am only part of the way to loosely coupled components. The `PasswordResetHelpe` configuring and sending e-mails through the `IEmailSender` interface, but to create an object that implements t it had to create an instance of `MyEmailSender`. In fact, I have made things worse for myself because `PasswordResetHelper` now depends on the `MyEmailSender` class *and* the `IEmailSender` int shown in Figure 3-6.

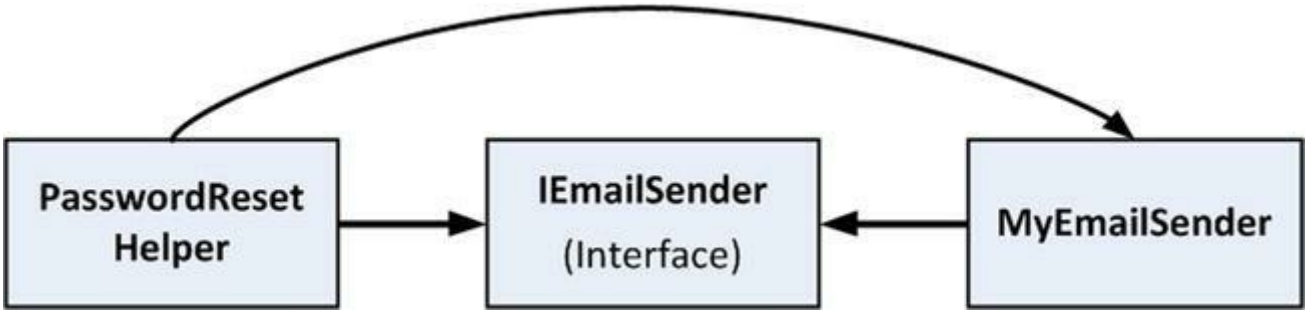


Figure 3-6. Components which are tightly coupled after all

What I need is a way to get objects that implement an interface *without* having to create the object directly. The so problem is called *dependency injection* (DI), also known as *Inversion of Control* (IoC).

DI is a design pattern that completes the loose coupling process. As I describe DI, you might wonder what the fus but bear with me—this is an important concept that is central to effective MVC development and it can cause a lot of

Breaking and Declaring Dependencies

There are two parts to the DI pattern. The first is that I remove any dependencies on concrete classes from my com

eBooks-IT.org

I need as arguments, like this:

```
public class PasswordResetHelper {
    private IEmailSender emailSender;

    public PasswordResetHelper(IEmailSender emailSenderParam) {
        emailSender = emailSenderParam;
    }

    public void ResetPassword() {
        // ...call interface methods to configure e-mail details..
        emailSender.SendEmail();
    }
}
```

The constructor for the `PasswordResetHelper` class is now said to *declare a dependency* on the `IEmailSender` interface, meaning that it can't be created and used unless it receives an object that implements `IEmailSender` interface. In declaring its dependency, the `PasswordResetHelper` class no longer ha knowledge of `MyEmailSender`, it only depends on the `IEmailSender` interface. In short, the `PassworsResetHelper` no longer knows or cares how the `IEmailSender` interface is implemented.

Injecting Dependencies

The second part of the DI pattern is to *inject* the dependencies declared by the `PasswordResetHelper` clas create instances of it, hence the term *dependency injection*.

All this really means is that I need to decide which class that implements the `IEmailSender` interface I am g create an object from that class and then pass the object as an argument to the `PasswordResetHelper` con

Note The `PasswordResetHelper` class declares its dependencies through its constructor. This is know *constructor injection*. I could also declare dependencies to be injected through a public property, known as *setter inje*

The dependencies are injected into the `PasswordResetHelper` at runtime; that is to say, an instance of s that implements the `IEmailSender` interface will be created and passed to the `PasswordResetHelpe` during instantiation. There is no compile-time dependency between `PasswordResetHelper` and any class t implements the interfaces it depends on.

Because the dependencies are dealt with at runtime, I can decide which interface implementations are going to be u run the application. I can choose between different e-mail providers or inject a special mocked implementation for tes Dependency injection lets me achieve the relationships I was aiming for in Figure 3-5.

Using a Dependency Injection Container

I have resolved my dependency issue, but how do I instantiate the concrete implementation of interfaces without cre dependencies somewhere else in the application? As it stands, I still have to have statements somewhere in the applic these:

```
...
IEmailSender sender = new MyEmailSender();
helper = new PasswordResetHelper(sender);
...
```

The answer is to use a *dependency injection container*, also known as an IoC container. This is a component that broker between the dependencies that a class like `PasswordResetHelper` declares and the classes that can resolve those dependencies, such as `MyEmailSender`.

I register the set of interfaces or abstract types that my application uses with the DI container, and specify which implementation classes should be instantiated to satisfy dependencies. So, I would register the `IEmailSender`

eBooks-IT.org

IEmailSender is required.

When I want a PasswordResetHelper object in my application, I ask the DI container to create one for that the PasswordResetHelper has declared a dependency on the IEmailSender interface and it kn I have specified that I want to use the MyEmailSender class as the implementation of that interface. The DI c these two pieces of information together, creates the MyEmailSender object and then uses it as an argument t PasswordResetHelper object, which I am then able to use in the application.

Note It is important to note that I no longer create the objects in my application myself using the new keyword. to the DI container and request the objects I need. It can take a while to get used to this when you are new to DI, bu see, the MVC Framework provides some features to make the process simpler.

I do not need to write my own DI container—there are some great open source and freely licensed implementation The one I like and use in my own projects is called Ninject and you can get details at www.ninject.org. I'll to using Ninject in Chapter 6 and show you how to install the package using NuGet.

Tip Microsoft has created its own DI container, called Unity. I are going to use Ninject, however, because I like demonstrate the ability to mix and match tools when using MVC. If you want more information about Unity, see unity.codeplex.com.

The role of a DI container may seem simple and trivial, but that is not the case. A good DI container, such as Ninj clever features:

- *Dependency chain resolution*: If you request a component that has its own dependencies (e.g., constructor pa container will satisfy those dependencies, too. So, if the constructor for the MyEmailSender clas implementation of the INetworkTransport interface, the DI container will instantiate the default imp that interface, pass it to the constructor of MyEmailSender and return the result as the default imp IEmailSender.
- *Object lifecycle management*: If you request a component more than once, should you get the same instance fresh new instance? A good DI container will let you configure the lifecycle of a component, allowing you t predefined options including *singleton* (the same instance each time), *transient* (a new instance each time), *thread*, *instance-per-HTTP-request*, *instance-from-a-pool*, and many others.
- *Configuration of constructor parameter values*: If the constructor for my implementat INetworkTransport interface requires a string called serverName, for example, you should b value for it in your DI container configuration. It is a crude but simple configuration system that removes your code to pass around connection strings, server addresses, and so forth.

Writing your own DI container is an excellent way to understand how C# and .NET handle types and reflection an recommend it as a good project for a rainy weekend. But don't be tempted to deploy your code in a real project. Wri robust and high-performance DI container is difficult and you should find a proven and tested package to use. I like there are plenty of others available and you are sure to find something that suits your development style.

Getting Started with Automated Testing

The ASP.NET MVC Framework is designed to make it as easy as possible to set up automated tests and use develop methodologies such as test-driven development (TDD), which I explain later in this chapter. ASP.NET MVC provide platform for automated testing and Visual Studio has some solid testing features. Between them they make designing tests simple and easy.

In broad terms, Web application developers today focus on two kinds of automated testing. The first is *unit testing* way to specify and verify the behavior of individual classes (or other small units of code) in isolation from the rest of application. The second type is *integration testing*, which is a way to specify and verify the behavior of multiple com working together, up to and including the entire Web application.

Both kinds of testing can be valuable in Web applications. Unit tests, which are simple to create and run, are brillia when you are working on algorithms, business logic, or other back-end infrastructure. The value of integration testin can model how a user will interact with the UI, and can cover the entire technology stack that your application uses,

eBooks-IT.org

atures;

as *regression testing*.

Understanding Unit Testing

In the .NET world, you create a separate test project in your Visual Studio solution to hold *test fixtures* . This project created when you first add a unit test, or can be set up automatically when you use an MVC project template. A *test* class that defines a set of test methods: one method for each behavior you want to verify. A test project can contain fixture classes.

GETTING THE UNIT TEST FEVER

Being able to perform unit testing is one of the benefits of working with the MVC Framework, but it isn't for ev have no intention of pretending otherwise. If you have not encountered unit testing before, then I encourage you try and see how it works out.

I like unit testing and I use it in my own projects, but not all of them and not as consistently as you might expec focus on writing unit tests for features and functions that I know will be hard to write and that are likely to be t bugs in deployment. In these situations, unit testing helps me structure my thoughts about how to best impleme need. I find that just thinking about what I test helps throw up ideas about potential problems–and that's before with actual bugs and defects.

That said, unit testing is a tool and not a religion and only you know how much testing–and what kind of testing If you don't find unit testing useful or if you have a different methodology that suits you better, then don't feel unit test just because it is fashionable. (Although if you *don't* have a better methodology and you are not testing you are probably letting users find your bugs and you are officially a *bad person.* You don't have to unit test, bu should do *some* testing of *some* kind).

Note I show you how to create a test project and populate it with unit tests in Chapter 6. The goal for this chapt introduce the concept of unit testing and give you an idea of what a test fixture looks like and how it is used.

To get started, I have created a class from an imaginary application, as shown in Listing 3-1. The class is called `AdminController` and it defines the `ChangeLoginName` method, which allows my imaginary users to passwords.

Listing 3-1. The Definition of the AdminController Class

```
using System.Web.Mvc;

namespace TestingDemo {

    public class AdminController : Controller {
        private IUserRepository repository;

        public AdminController(IUserRepository repo) {
            repository = repo;
        }

             public ActionResult ChangeLoginName(string oldName,
newName) {
            User user = repository.FetchByLoginName(oldName);
            user.LoginName = newName;
            repository.SubmitChanges();
            // render some view to show the result
            return View();
        }
    }
```

Tip I created the classes for this demonstration in a new Visual Studio project called `TestingDemo`. You do recreate the examples in this section to follow along but I have included the project in the source code download ava Apress.com.

The controller relies on some model classes and an interface, which you can see in Listing 3-2. Once again, these real project and I have simplified these classes to make demonstrating the test easier. I am not suggesting that you cr classes that have just a single string property, for example.

Listing 3-2. The Model Classes and Interface that the AdminController Relies On

```
namespace TestingDemo {

    public class User {
        public string LoginName { get; set; }
    }

    public interface IUserRepository {
        void Add(User newUser);
        User FetchByLoginName(string loginName);
        void SubmitChanges();
    }

    public class DefaultUserRepository : IUserRepository {

        public void Add(User newUser) {
            // implement me
        }

        public User FetchByLoginName(string loginName) {
            // implement me
            return new User() { LoginName = loginName };
        }

        public void SubmitChanges() {
            // implement me
        }
    }
}
```

The `User` class represents a user in my application. Users are created, managed and stored through a repository functionality is defined by the `IUserRepository` interface and there is a partially complete implementation of in the `DefaultUserRepository` class.

My goal in this section is to write a unit test for the functionality provided by the `ChangeLoginName` metho the `AdminController`, as shown in Listing 3-3.

Listing 3-3. A Test Fixture for the AdminController.ChangeLoginName Method

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestingDemo.Tests {

    [TestClass]
```

```
        [TestMethod]
        public void CanChangeLoginName() {

            // Arrange (set up a scenario)
            User user = new User() { LoginName = "Bob" };
            FakeRepository repositoryParam = new FakeRepository();
            repositoryParam.Add(user);
            AdminController target = new AdminController(repositor
            string oldLoginParam = user.LoginName;
            string newLoginParam = "Joe";

            // Act (attempt the operation)
            target.ChangeLoginName(oldLoginParam, newLoginParam);

            // Assert (verify the result)
            Assert.AreEqual(newLoginParam, user.LoginName);
            Assert.IsTrue(repositoryParam.DidSubmitChanges);
        }
    }

    class FakeRepository : IUserRepository {
        public List<User> Users = new List<User>();
        public bool DidSubmitChanges = false;

        public void Add(User user) {
            Users.Add(user);
        }

        public User FetchByLoginName(string loginName) {
            return Users.First(m => m.LoginName == loginName);
        }

        public void SubmitChanges() {
            DidSubmitChanges = true;
        }
    }
}
```

The test fixture is the `CanChangeLoginName` method. Notice that the method is decorated with the `Tes` attribute and that the class it belongs to—called `AdminControllerTests`—is decorated with the `TestCl` attribute. This is how Visual Studio finds the test fixture.

The `CanChangeLoginName` method follows a pattern known as *arrange/act/assert* (A/A/A). *Arrange* refer up the conditions for the test, *act* refers to performing the test, and *assert* refers to verifying that the result was the c required. Being consistent about the structure of your unit test methods makes them easier to read, something you'll when your project contains hundreds of unit tests.

The test fixture uses a test-specific fake implementation of the `IUserRepository` interface to simulate a sp condition—in this case, when there is a single member, `Bob`, in the repository. Creating the fake repository and the done in the *arrange* section of the test.

Next, the method being tested—`AdminController.ChangeLoginName`—is called. This is the *act* s test. Finally, I check the results using a pair of `Assert` calls (this is the *assert* part of the test). The `Assert` me provided by the Visual Studio test suite and lets me check for specific outcomes. I run the test from the Visual Studi menu and receive visual feedback about the tests as they are performed, as shown in Figure 3-7.

eBooks-IT.org

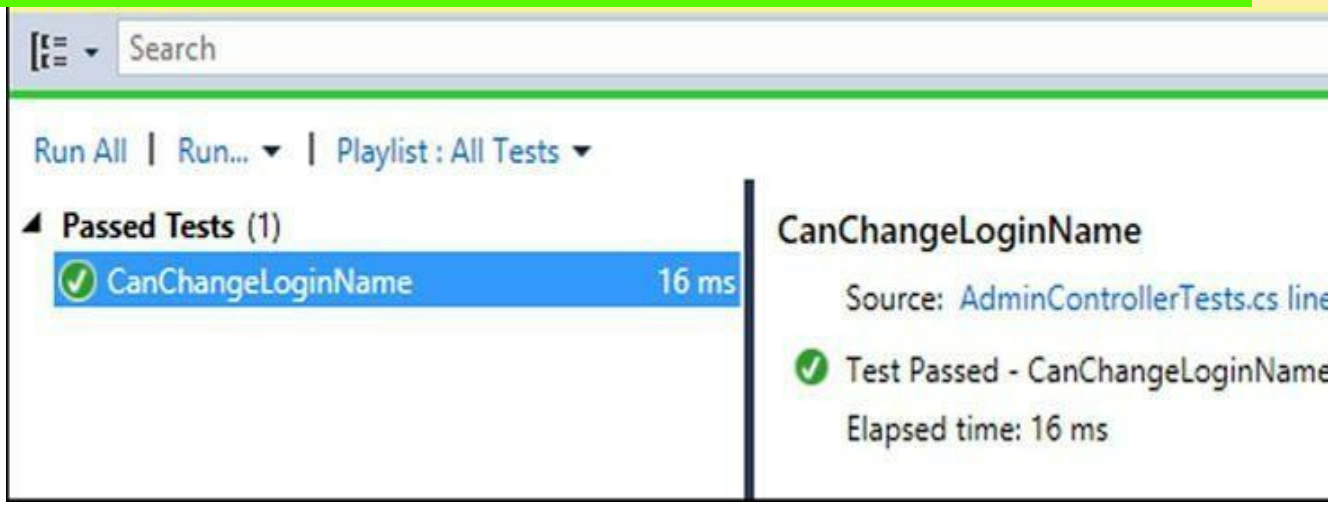


Figure 3-7. Visual feedback on the progress of unit tests

If the test runs without throwing any unhandled exceptions and all of the `Assert` statements pass without prob `Test Explorer` window shows a green light. If not, you get a red light and details of what went wrong.

Note You can see how my use of DI has helped with unit testing. I was able to create a fake implementation of t and inject it into the controller to create a specific scenario. I am a big fan of DI and this is one of the reasons.

It might seem like I have gone to a lot of effort to test a simple method, but it wouldn't require much more code t something far more complex. If you find yourself considering skipping small tests like this one, remember that test f uncover bugs that can sometimes be hidden in more complex tests. One improvement I could have made to my test test-specific fake classes like `FakeMembersRepository` by using a *mocking tool*—I show you how to do Chapter 6.

Using TDD and the Red-Green-Refactor Workflow

With test-driven development (TDD), you use unit tests to help design your code. This can be an odd concept if you testing after you have finished coding, but there is a lot of sense in this approach. The key concept is a development called red-green-refactor. It works like this:

- Determine that you need to add a new feature or method to your application.
- Write the test that will validate the behavior of the new feature when it is written.
- Run the test and get a red light
- Write the code that implements the new feature.
- Run the test again and correct the code until you get a green light.
- Refactor the code if required. For example, reorganize the statements, rename the variables, and so on.
- Run the test to confirm that your changes have not changed the behavior of your additions.

This workflow is repeated for every feature you add. TDD inverts the traditional development process: you start b tests for the perfect implementation of a feature, knowing that the tests will fail. You then implement the feature, cre aspect of its behavior to pass one or more tests.

This cycle is the essence of TDD. There is a lot to recommend it as a development style, not least because it make programmer think about how a change or enhancement should behave *before* the coding starts. You always have a c in view and a way to check that you are there. And if you have unit tests that cover the rest of your application, you that your additions have not changed the behavior elsewhere.

TDD seems a little odd when you first try it, but it is strangely empowering, and writing the tests first make you c a perfect implementation should do before you become biased by the techniques that you use to write the code.

eBooks-IT.org

start writing code or, as I have witnessed several times on projects, sneakily discard problematic tests to make code ard TD better shape than it really is. For these reasons, TDD should be used in established and mature development teams w generally a high level of skill and discipline or in teams where there the team leads can enforce good practice, even in time constraints.

Tip You can see a simple example of TDD in Chapter 6 when I demonstrate the testing tools built into Visual Stu

Understanding Integration Testing

For Web applications, the most common approach to integration testing is *UI automation*, which means simulating o a Web browser to exercise the application's entire technology stack by reproducing the actions that a user would per pressing buttons, following links, and submitting forms. The two best-known open source browser automation optio developers are

- *Selenium RC* (`http://seleniumhq.org/`), which consists of a Java "server" application t automation commands to Internet Explorer, Firefox, Safari, or Opera, plus clients for .NET, Python, Ruby others so that you can write test scripts in the language of your choice. Selenium is powerful and ma drawback is that you have to run its Java server.
- *WatiN* (`http://watin.org`), a .NET library that can send automation commands to Internet Explor Its API isn't as powerful as Selenium, but it comfortably handles most common scenarios and is easy to set only reference a single DLL.

Integration testing is an ideal complement to unit testing. Although unit testing is well suited to validating the behav individual components at the server, integration testing lets you create tests that are client-focused, recreating the act As a result, it can highlight problems that come from the interaction between components, hence the term *integratio* because integration testing for a Web application is done through the browser, you can test that JavaScript behaviors way they are supposed to, something that is difficult with unit testing.

There are some drawbacks. Integration testing takes more time. It takes longer to create the tests and longer to pe And integration tests can be brittle. If you change the `id` attribute of an element that is checked in a test, for exampl (and usually will) fail.

As a consequence of the additional time and effort required, integration testing is often done at key project milesto after a weekly source code check-in, or when major functional blocks are completed. Integration testing is every bit unit testing and it can highlight problems that unit testing cannot. The time required to set up and run integration testi worthwhile, and I encourage you to add it to your development process.

I am not going to get into integration testing in this book. It is outside of my focus on the MVC Framework. Any benefit from integration testing and there are no special features in the MVC Framework to support this activity. Inte is a separate art and what is true when performing integration testing on any Web application is also true for MVC.

Summary

In this chapter, I introduced you to the MVC architectural pattern and compared it to some other patterns you may h heard of before. I discussed the significance of the domain model and introduced dependency injection, which allows decouple components to enforce a strict separation between the parts of an application. I demonstrated a simple unit saw how decoupled components and dependency injection make unit testing simple and easy. In the next chapter, I c essential C# language features that are used in MVC Framework applications.

eBooks-IT.org

Essential Language Features

C# is a feature-rich language and not all programmers are familiar with all of the features I rely on in this book. In th
describe the C# language features that a good MVC programmer needs to know and that I use in examples throughou

I provide only a short summary of each feature. If you want more in-depth coverage of C# or LINQ, three of my
of interest. For a complete guide to C#, try *Introducing Visual C#;* for in-depth coverage of LINQ, check out *Pro L*
and for a detailed examination of the .NET support for asynchronous programming see *Pro .Net Parallel Programm*
of these books are published by Apress. Table 4-1 provides the summary for this chapter.

Table 4-1. *Chapter Summary*

Problem	Solution	Listing
Simplify C# properties	Use automatically implemented properties	1–7
Create an object and sets its properties in a single step	Use an object or collection initializer	8–10
Add functionality to a class which cannot be modified	Use an extension method	11–18
Simplify the use of delegates	Use a lambda expression	19–23
Use implicit typing	Use the `var` keyword	24
Create objects without defining a type	Use an anonymous type	25–26
Query collections of objects as though there were a database	Use LINQ	27–31
Simplify the use of asynchronous methods	Use the `async` and `await` keywords	32–33

Preparing the Example Project

To demonstrate the language features in this part of the book, I have created a new Visual Studio project called `LanguageFeatures` using the `ASP.NET MVC Web Application` template. I selected the `Empt` the initial content and checked the option for `MVC` folders and references, just as I did in Chapter 2. The language fe describe in this chapter are not specific to MVC, but Visual Studio Express 2013 for Web doesn't support creating p can write to the console, so you will have to create an MVC app if you want to follow along with the examples. I nee controller to demonstrate these language features, so I created the `HomeController.cs` file in the `Contro` folder–I did this by right-clicking on the `Controllers` folder in the Solution Explorer, selecting `Add` ➤ `Con` from the pop-up menu, selecting `MVC 5 Controller-Empty` from the `Add Scaffold` menu, and c `Add` button. I set the name to `HomeController` in the `Add Controller` dialog and clicked the `Add` create the controller class file, the edited contents of which you can see in Listing 4-1.

Listing 4-1. The Initial Content of the HomeController.cs File

```
using System;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
```

eBooks-IT.org

```
        }
    }
}
```

I will create action methods for each example, so the result from the `Index` action method is a basic message to project simple.

Caution The `HomeController` class won't compile at the moment because it imports the `LanguageFeatures.Models` namespace. This namespace won't create until I add a class to the `Model` which I do as part of the first example in the next section.

To display the results from my action methods, I right-clicked the `Index` action method, selected `Add View` a new view called `Result`. You can see the contents of the view file in Listing 4-2. (It doesn't matter which optio in the `Add View` dialog because you will replace the initial content of the file with the markup shown in the listing

Listing 4-2. The Contents of the Result.cshtml File

```
@model String

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Result</title>
</head>
<body>
    <div>
        @Model
    </div>
</body>
</html>
```

You can see that this is a strongly typed view, where the model type is `String`–for the most part, the examples are not complex examples and I can represent the results as a simple string.

Adding the System.Net.Http Assembly

Later in the chapter, I'll be using an example that relies on the `System.Net.Http` assembly, which isn't adde projects by default. Select `Add Reference` from the Visual Studio `Project` menu to open the `Referen Manager` window. Ensure that the `Assemblies` section is selected on the left-hand side and locate and chec `System.Net.Http` item, as shown in Figure 4-1.

eBooks-IT.org

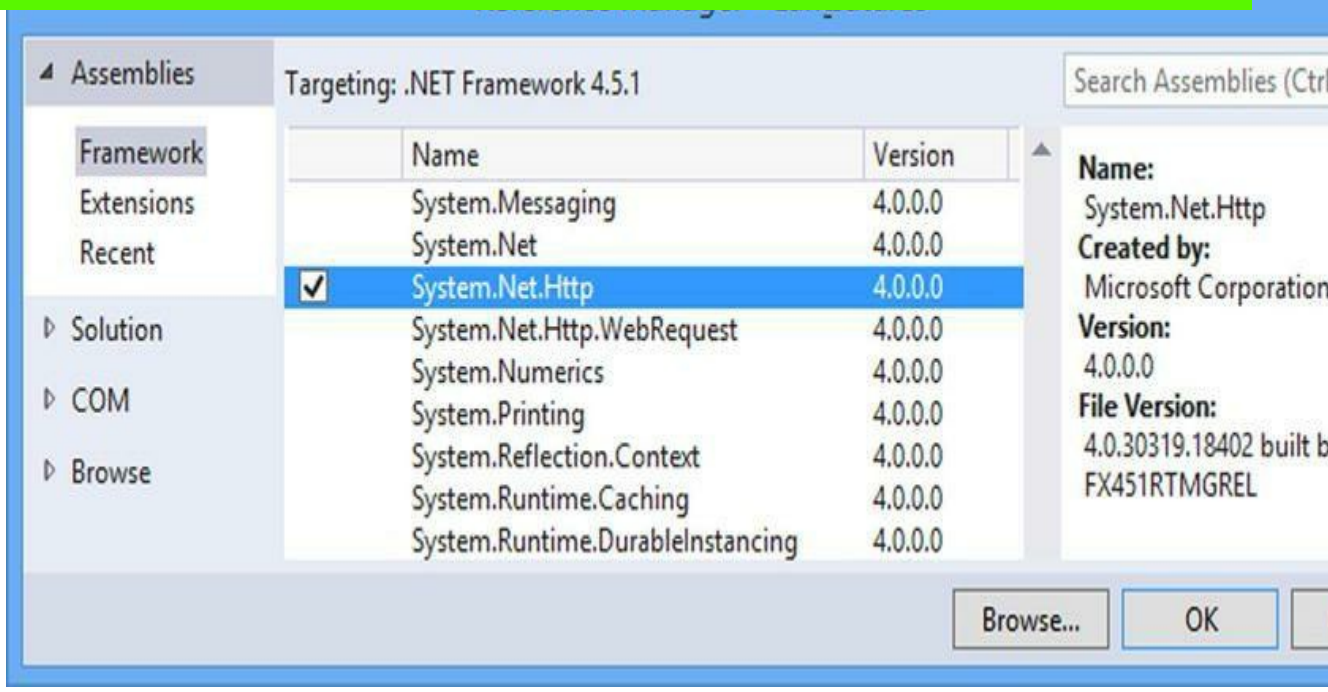


Figure 4-1. Adding an assembly to the project

Using Automatically Implemented Properties

The regular C# property feature lets you expose a piece of data from a class in a way that decouples the data from h
and retrieved. Listing 4-3 contains a simple example in a class called `Product`, which I added to the `Models` fo
`LanguageFeatures` project in a class file called `Product.cs`

Listing 4-3. Defining a Property in the Product.cs File

```
namespace LanguageFeatures.Models {
    public class Product {
        private string name;

        public string Name {
            get { return name; }
            set { name = value; }
        }
    }
}
```

The property, called `Name`, is shown in bold. The statements in the `get` code block (known as the *getter*) are p
when the value of the property is read, and the statements in the `set` code block (known as the *setter*) are performe
value is assigned to the property (the special variable `value` represents the assigned value). A property is consume
classes as though it were a field, as shown in Listing 4-4, which shows an `AutoProperty` action method I add
`Home` controller.

Listing 4-4. Consuming a Property in the HomeController.cs File

```
using System;
using System.Web.Mvc;
using LanguageFeatures.Models;
```

eBooks-IT.org

```
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        public ViewResult AutoProperty() {
            // create a new Product object
            Product myProduct = new Product();

            // set the property value
            myProduct.Name = "Kayak";

            // get the property
            string productName = myProduct.Name;

            // generate the view
            return View("Result",
                (object)String.Format("Product name: {0}", product
        }
    }
}
```

You can see that the property value is read and set just like a regular field. Using properties is preferable to using fi
you can change the statements in the `get` and `set` blocks without needing to change the classes that depend on th

Tip You may notice that I cast the second argument to the `View` method to an `object` in Listing 4-4. This i
`View` method has an overload that accepts two `String` arguments and which has a different meaning to the ove
accepts a `String` and an `object`. To avoid calling the wrong one, I explicitly cast the second argument. I retu
`View` method and its overloads in Chapter 20.

You can see the effect of this example by starting the project and navigating to `/Home/AutoProperty` (w
the `AutoProperty` action method and will be the pattern for testing each example in this chapter). Because I pa
from the action method to the view, I am going to show you the results as text, rather than a screen shot. Here is the
targeting the action method in Listing 4-4:

```
Product name: Kayak
```

Properties are all well and good, but they become tedious when you have a class that has a lot of properties, all of
mediate access to a field, producing a class file that is needlessly verbose, as shown in Listing 4-5, which shows son
properties I added to the `Product` class in the `Product.cs` file.

Listing 4-5. Verbose Property Definitions in the Product.cs File

```
namespace LanguageFeatures.Models {

    public class Product {
        private int productID;
        private string name;
        private string description;
        private decimal price;
        private string category;

        public int ProductID {
            get { return productID; }
```

```
        }

        public string Name {
            get { return name; }
            set { name = value; }
        }

        public string Description {
            get { return description; }
            set { description = value; }
        }

        //...and so on...
    }
}
```

What I want is the flexibility of properties without having to duplicate the getters and setters. The solution is an *au implemented property*, also known as an *automatic property*. With an automatic property, you can create the pattern backed property, without defining the field or specifying the code in the getter and setter, as Listing 4-6 shows.

Listing 4-6. Using Automatically Implemented Properties in the Product.cs File

```
namespace LanguageFeatures.Models {

    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

Notice that I do not define the bodies of the getter and setter or the field that the property is backed by. Both of th for me by the C# compiler when the class is compiled. Using an automatic property is no different from using a regu the code in the action method in Listing 4-4 will work without any modification.

By using automatic properties, I save myself some typing, create code that is easier to read, but still preserve the f property provides. If the day comes when I need to change the way a property is implemented, I can return to the re property format. As a demonstration, Listing 4-7 shows what I would have to do if I needed to change the way the N property is composed.

Listing 4-7. Reverting from an Automatic to a Regular Property in the Product.cs File

```
namespace LanguageFeatures.Models {

    public class Product {
        private string name;

        public int ProductID { get; set; }

        public string Name {
            get {
                return ProductID + name;
            }
```

eBooks-IT.org

```
                name = value;
            }
        }

        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

Note Notice that I must implement both the getter and setter to return to a regular property. C# does not support automatic- and regular-style getters and setters in a single property.

Using Object and Collection Initializers

Another tiresome programming task is constructing a new object and then assigning values to the properties, as illust Listing 4-8, which shows the addition of a `CreateProduct` action method to the `Home` controller.

Listing 4-8. Constructing and Initializing an Object with Properties in the HomeController.cs File

```
using System;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        public ViewResult AutoProperty() {
            // ...statements omitted for brevity...
        }

        public ViewResult CreateProduct() {

            // create a new Product object
            Product myProduct = new Product();

            // set the property values
            myProduct.ProductID = 100;
            myProduct.Name = "Kayak";
            myProduct.Description = "A boat for one person";
            myProduct.Price = 275M;
            myProduct.Category = "Watersports";

            return View("Result",
                                    (object)String.Format("Category
myProduct.Category));
        }
    }
}
```

eBooks-IT.org

meter

then call the View method so I can display the result through the view. Fortunately, I can use the *object initializer* which allows me to create and populate the Product instance in a single step, as shown in Listing 4-9.

Listing 4-9. Using the Object Initializer Feature in the HomeController.cs File

```
...
public ViewResult CreateProduct() {

    // create and populate a new Product object
    Product myProduct = new Product {
        ProductID = 100, Name = "Kayak",
        Description = "A boat for one person",
        Price = 275M, Category = "Watersports"
    };

    return View("Result",
        (object)String.Format("Category: {0}", myProduct.Category)
}
...
```

The braces ({ }) after the call to the Product name form the *initializer*, which I use to supply values to the pa part of the construction process. The same feature let me initialize the contents of collections and arrays as part of th construction process, as demonstrated by Listing 4-10.

Listing 4-10. Initializing Collections and Arrays in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult CreateCollection() {
            string[] stringArray = { "apple", "orange", "plum" };

            List<int> intList = new List<int> { 10, 20, 30, 40 };

            Dictionary<string, int> myDict = new Dictionary<string
                { "apple", 10 }, { "orange", 20 }, { "plum", 30 }
            };

            return View("Result", (object)stringArray[1]);
        }
    }
}
```

eBooks-IT.org

feature is a syntax convenience—it just makes C# more pleasant to use but does not have any other impact or benefi

Using Extension Methods

Extension methods are a convenient way of adding methods to classes that you do not own and cannot modify direct 11 shows a `ShoppingCart` class, which I added to the `Models` folder in a file called `ShoppingCart.` which represents a collection of `Product` objects.

Listing 4-11. The ShoppingCart Class in the ShoppingCart.cs File

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart {
        public List<Product> Products { get; set; }
    }
}
```

This is a simple class that acts as a wrapper around a `List` of `Product` objects (I only need a basic class for example). Suppose I need to be able to determine the total value of the `Product` objects in the `ShoppingCar` cannot modify the class itself, perhaps because it comes from a third party and I do not have the source code. I can extension method to add the functionality I need. Listing 4-12 shows the `MyExtensionMethods` class that I `Models` folder in the `MyExtensionMethods.cs` file.

Listing 4-12. Defining an Extension Method in the MyExtensionMethods.cs File

```
namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this ShoppingCart cartPa
            decimal total = 0;
            foreach (Product prod in cartParam.Products) {
                total += prod.Price;
            }
            return total;
        }
    }
}
```

The `this` keyword in front of the first parameter marks `TotalPrices` as an extension method. The first p .NET which class the extension method can be applied to—`ShoppingCart` in this case. I can refer to the insta `ShoppingCart` that the extension method has been applied to by using the `cartParam` parameter. My met enumerates the `Products` in the `ShoppingCart` and returns the sum of the `Product.Price` property shows how I apply an extension method in a new action method called `UseExtension` I added to the `Home` c

Listing 4-13. Applying an Extension Method in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using LanguageFeatures.Models;
```

eBooks-IT.org

```
namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult UseExtension() {
            // create and populate ShoppingCart
            ShoppingCart cart = new ShoppingCart {
                Products = new List<Product> {
                    new Product {Name = "Kayak", Price = 275M},
                    new Product {Name = "Lifejacket", Price = 48.9
                    new Product {Name = "Soccer ball", Price = 19.
                    new Product {Name = "Corner flag", Price = 34.
                }
            };

            // get the total value of the products in the cart
            decimal cartTotal = cart.TotalPrices();

            return View("Result",
                (object)String.Format("Total: {0:c}", cartTotal));
        }
    }
}
```

Note Extension methods do not let you break through the access rules that classes define for their methods, field properties. You can extend the functionality of a class by using an extension method, but only using the class membe had access to anyway.

The key statement is this one:

```
...
decimal cartTotal = cart.TotalPrices();
...
```

I call the `TotalPrices` method on a `ShoppingCart` object as though it were part of the `Shopping` even though it is an extension method defined by a different class altogether. .NET will find extension classes if they scope of the current class, meaning that they are part of the same namespace or in a namespace that is the subject o statement. Here is the result from the `UseExtension` action method, which you can see by starting the applica navigating to the `/Home/UseExtension` URL:

```
Total: $378.40
```

Applying Extension Methods to an Interface

I can also create extension methods that apply to an interface, which allows me to call the extension method on all of that implement the interface. Listing 4-14 shows the `ShoppingCart` class updated to implement the `IEnumerable<Product>` interface.

Listing 4-14. Implementing an Interface in the ShoppingCart.cs File

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart : IEnumerable<Product> {

        public List<Product> Products { get; set; }

        public IEnumerator<Product> GetEnumerator() {
            return Products.GetEnumerator();
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }
    }
}
```

I can now update my extension method so that it deals with `IEnumerable<Product>`, as shown in Listi

Listing 4-15. An Extension Method That Works on an Interface in the MyExtensionMethods.cs File

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

            public static decimal TotalPrices(this IEnumerable<
productEnum) {
            decimal total = 0;
            foreach (Product prod in productEnum) {
                total += prod.Price;
            }
            return total;
        }
    }
}
```

The first parameter type has changed to `IEnumerable<Product>`, which means that the `foreach` lo method body works directly on `Product` objects. The switch to the interface means that I can calculate the total `Product` objects enumerated by any `IEnumerable<Product>`, which includes instances of `Shoppi` also arrays of `Products`, as shown in Listing 4-16.

Listing 4-16. Extension Methods Applies to Implementations of an Interface in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
```

eBooks-IT.org

```
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult UseExtensionEnumerable() {

            IEnumerable<Product> products = new ShoppingCart {
                Products = new List<Product> {
                    new Product {Name = "Kayak", Price = 275M},
                    new Product {Name = "Lifejacket", Price = 48.9
                    new Product {Name = "Soccer ball", Price = 19.
                    new Product {Name = "Corner flag", Price = 34.
                }
            };

            // create and populate an array of Product objects
            Product[] productArray = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M}
                new Product {Name = "Corner flag", Price = 34.95M}
            };

            // get the total value of the products in the cart
            decimal cartTotal = products.TotalPrices();
            decimal arrayTotal = products.TotalPrices();

            return View("Result",
                (object)String.Format("Cart Total: {0}, Array Tota
                    cartTotal, arrayTotal));
        }
    }
}
```

Note The way that C# arrays implement the `IEnumerable<T>` interface is a little unusual. You will not fin in the list of implemented interfaces in the MSDN documentation. The support is handled by the compiler so that coc versions C# will still compile. Odd, but true. I could have used another generic collection class in this example, but I show off my knowledge of the dark corners of the C# specification. Also odd, but true.

If you start the project and target the action method, you will see the following results, which demonstrate that I g result from the extension method, irrespective of how the `Product` objects are collected:

```
Cart Total: 378.40, Array Total: 378.40
```

Creating Filtering Extension Methods

The last thing I want to show you about extension methods is that they can be used to filter collections of objects. A method that operates on an `IEnumerable<T>` and that also returns an `IEnumerable<T>` can use the `y` keyword to apply selection criteria to items in the source data to produce a reduced set of results. Listing 4-17 demo a method, which I have added to the `MyExtensionMethods` class.

Listing 4-17. A Filtering Extension Method in the MyExtensionMethods.cs File

```
namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

            public static decimal TotalPrices(this IEnumerable<
productEnum) {
            decimal total = 0;
            foreach (Product prod in productEnum) {
                total += prod.Price;
            }
            return total;
        }

        public static IEnumerable<Product> FilterByCategory(
                                this IEnumerable<Product> productEnum,
categoryParam) {

            foreach (Product prod in productEnum) {
                if (prod.Category == categoryParam) {
                    yield return prod;
                }
            }
        }
    }
}
```

This extension method, called `FilterByCategory`, takes an additional parameter that allows me to inject a condition when I call the method. Those `Product` objects whose `Category` property matches the parameter in the result `IEnumerable<Product>` and those that do not match are discarded. Listing 4-18 shows this r used.

Listing 4-18. Using the Filtering Extension Method in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {

    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ... other action methods omitted for brevity...

        public ViewResult UseFilterExtensionMethod() {

            IEnumerable<Product> products = new ShoppingCart {
                Products = new List<Product> {
                    new Product {Name = "Kayak", Category = "Water
```

eBooks-IT.org

```
                                new Product {Name = "Lifejacket", Ca
"Watersports",
                        Price = 48.95M},
                            new Product {Name = "Soccer ball", Ca
"Soccer",
                        Price = 19.50M},
                            new Product {Name = "Corner flag", Ca
"Soccer",
                        Price = 34.95M}
                }
            };

            decimal total = 0;
             foreach (Product prod in products.FilterByCategory("S
{
                total += prod.Price;
            }

                return View("Result", (object)String.Format("Total
total));
        }
    }
}
```

When I call the `FilterByCategory` method on the `ShoppingCart`, only those `Products` in the
category are returned. If you start the project and target the `UseFilterExtensionMethod` action metho
the following result, which is the sum of the `Soccer` product prices:

```
Total: 54.45
```

Using Lambda Expressions

I can use a delegate to make my `FilterByCategory` method more general. That way, the delegate that will
against each `Product` can filter the objects in any way I choose, as illustrated by Listing 4-19, which shows the
extension method I added to the `MyExtensionMethods` class.

Listing 4-19. Using a Delegate in an Extension Method in the MyExtensionMethods.cs File

```
using System;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

             public static decimal TotalPrices(this IEnumerable<
productEnum) {
            decimal total = 0;
            foreach (Product prod in productEnum) {
                total += prod.Price;
            }
            return total;
        }
```

```
        public static IEnumerable<Product> FilterByCategory(
                                this IEnumerable<Product> productEnum,
categoryParam) {

            foreach (Product prod in productEnum) {
                if (prod.Category == categoryParam) {
                    yield return prod;
                }
            }
        }

        public static IEnumerable<Product> Filter(
                this IEnumerable<Product> productEnum, Func<Produc
selectorParam) {

            foreach (Product prod in productEnum) {
                if (selectorParam(prod)) {
                    yield return prod;
                }
            }
        }
    }
}
```

I used a `Func` as the filtering parameter, which means that I do not need to define the delegate as a type. The del
`Product` parameter and returns a `bool`, which will be `true` if that `Product` should be included in the resu
end of this arrangement is a little verbose, as illustrated by Listing 4-20, which shows the changes I made to the
`UseFilterExtensionMethod` action method in the `Home` controller.

Listing 4-20. Using the Filtering Extension Method with a Func in the HomeController.cs File

```
...
public ViewResult UseFilterExtensionMethod() {

    // create and populate ShoppingCart
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
              new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
             new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
             new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
        }
    };

    Func<Product, bool> categoryFilter = delegate(Product prod) {
        return prod.Category == "Soccer";
    };

    decimal total = 0;
```

eBooks-IT.org

```
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", tota
}
...
```

I have taken a step forward, in the sense that I can now filter the `Product` objects using any criteria specified delegate, but I must define a `Func` for each kind of filtering that I want, which is not ideal. The less verbose alterna a *lambda expression*, which is a concise format for expressing a method body in a delegate. I can use it to replace m definition in the action method, as shown in Listing 4-21.

Listing 4-21. Using a Lambda Expression to Replace a Delegate Definition in the HomeController.cs File

```
...
public ViewResult UseFilterExtensionMethod() {

    // create and populate ShoppingCart
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
             new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
        }
    };

       Func<Product, bool> categoryFilter = prod => prod.Cate
"Soccer";

    decimal total = 0;

    foreach (Product prod in products.Filter(categoryFilter)) {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", tota
}
...
```

The lambda expression is shown in bold. The parameter is expressed without specifying a type, which will be infe automatically. The => characters are read aloud as "goes to" and links the parameter to the result of the lambda expr example, a `Product` parameter called `prod` goes to a `bool` result, which will be `true` if the `Category` p `prod` is equal to `Soccer`. I can make my syntax even tighter by doing away with the `Func` entirely, as shown 22.

Listing 4-22. A Lambda Expression Without a Func in the HomeController.cs File

```
...
public ViewResult UseFilterExtensionMethod() {

    IEnumerable<Product> products = new ShoppingCart {
```

eBooks-IT.org

```
                new Product {Name = "Kayak", Category = "Watersports",
275M},
                 new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
                new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
                new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
            }
        };

        decimal total = 0;

         foreach (Product prod in products.Filter (prod => prod.Cate
"Soccer")) {
            total += prod.Price;
        }
        return View("Result", (object)String.Format("Total: {0}", tota
    }
...
```

In this example, I supplied the lambda expression as the parameter to the `Filter` method. This is a nice and nat
expressing the filter I want to apply. I can combine multiple filters by extending the result part of the lambda express
in Listing 4-23.

Listing 4-23. Extending the Filtering Expressed by the Lambda Expression in the HomeController.cs File

```
...
public ViewResult UseFilterExtensionMethod() {

    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
                 new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
                new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
                new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
            }
        };

        decimal total = 0;

        foreach (Product prod in products
                .Filter(prod => prod.Category == "Soccer" || prod.Pric
{
            total += prod.Price;
        }
        return View("Result", (object)String.Format("Total: {0}", tota
    }
...
```

This revised lambda expression will match `Product` objects that are in the `Soccer` category or whose `Pri`

eBooks-IT.org

I don't need to express the logic of my delegate in the lambda expression. I can as easily call a method, like this

```
prod => EvaluateProduct(prod)
```

If I need a lambda expression for a delegate that has multiple parameters, I must wrap the parameters in parenth

```
(prod, count) => prod.Price > 20 && count > 0
```

And, finally, if I need logic in the lambda expression that requires more than one statement, I can do so by using and finishing with a `return` statement, like this:

```
(prod, count) => {
  //...multiple code statements
  return result;
}
```

You do not need to use lambda expressions in your code, but they are a neat way of expressing complex functio and in a manner that is readable and clear. I like them a lot, and you will see them used liberally throughout this

Using Automatic Type Inference

The C# `var` keyword allows you to define a local variable without explicitly specifying the variable type, as demons Listing 4-24. This is called *type inference*, or *implicit typing*.

Listing 4-24. Using Type Inference

```
..
var myVariable = new Product { Name = "Kayak", Category = "Water
Price = 275M };

string name = myVariable.Name;   // legal
int count = myVariable.Count;    // compiler error
...
```

It is not that `myVariable` does not have a type. It is just that I am asking the compiler to infer it from the coo see from the statements that follow that the compiler will allow only members of the inferred class—`Product` in be called.

Using Anonymous Types

By combining object initializers and type inference, I can create simple data-storage objects without needing to define corresponding class or struct. Listing 4-25 shows an example.

Listing 4-25. Creating an Anonymous Type

```
...
var myAnonType = new {
    Name = "MVC",
    Category = "Pattern"
};
```

In this example, myAnonType is an anonymously typed object. This does not mean that it is dynamic in the se JavaScript variables are dynamic. It just means that the type definition will be created automatically by the compiler. is still enforced. You can get and set only the properties that have been defined in the initializer, for example.

The C# compiler generates the class based on the name and type of the parameters in the initializer. Two anonymo objects that have the same property names and types will be assigned to the same automatically generated class. This create arrays of anonymously typed objects, as demonstrated by Listing 4-26, which shows the CreateAnonAr method I added to the Home controller.

Listing 4-26. Creating an Array of Anonymously Typed Objects in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult CreateAnonArray() {

            var oddsAndEnds = new[] {
                new { Name = "MVC", Category = "Pattern"},
                new { Name = "Hat", Category = "Clothing"},
                new { Name = "Apple", Category = "Fruit"}
            };

            StringBuilder result = new StringBuilder();
            foreach (var item in oddsAndEnds) {
                result.Append(item.Name).Append("");
            }

            return View("Result", (object)result.ToString());
        }
    }
}
```

Notice that I use var to declare the variable array. I must do this because I do not have a type to specify, as I w regularly typed array. Even though I have not defined a class for any of these objects, I can still enumerate the conte array and read the value of the Name property from each of them. This is important, because without this feature, I able to create arrays of anonymously typed objects at all. Or, rather, I *could* create the arrays, but I would not be abl anything useful with them. You will see the following results if you run the example and target the action method:

```
MVC Hat Apple
```

Performing Language Integrated Queries

eBooks-IT.org

.NET. If you have never used LINQ, you have been missing out. LINQ is a SQL-like syntax for querying data in clas mpellin that I have a collection of `Product` objects, and I want to find the three highest prices and pass them to the `Vie` Without LINQ, I would end up with something similar to Listing 4-27, which shows the `FindProducts` action added to the `Home` controller.

Listing 4-27. Querying Without LINQ in the HomeController.cs File

```
...
public ViewResult FindProducts() {

    Product[] products = {
          new Product {Name = "Kayak", Category = "Watersports",
275M},
         new Product {Name = "Lifejacket", Category = "Watersports
= 48.95M},
          new Product {Name = "Soccer ball", Category = "Soccer",
19.50M},
          new Product {Name = "Corner flag", Category = "Soccer",
34.95M}
    };

    // define the array to hold the results
    Product[] foundProducts = new Product[3];
    // sort the contents of the array
    Array.Sort(products, (item1, item2) => {
                    return  Comparer<decimal>.Default.Compare(item
item2.Price);
    });
    // get the first three items in the array as the results
    Array.Copy(products, foundProducts, 3);

    // create the result
    StringBuilder result = new StringBuilder();
    foreach (Product p in foundProducts) {
        result.AppendFormat("Price: {0} ", p.Price);
    }

    return View("Result", (object)result.ToString());
}
...
```

With LINQ, I can significantly simplify the querying process, as demonstrated in Listing 4-28.

Listing 4-28. Using LINQ to Query Data in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
```

eBooks-IT.org

```
        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult FindProducts() {

            Product[] products = {
              new Product {Name = "Kayak", Category = "Watersports
= 275M},
               new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
             new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
             new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
            };

            var foundProducts = from match in products
                                orderby match.Price descending
                                select new { match.Name, match.Pri

            // create the result
            int count = 0;
            StringBuilder result = new StringBuilder();
            foreach (var p in foundProducts) {
                result.AppendFormat("Price: {0} ", p.Price);
                if (++count == 3) {
                    break;
                }
            }

            return View("Result", (object)result.ToString());
        }
    }
}
```

This is a lot neater. You can see the SQL-like query shown in bold. I order the `Product` objects in descending the `select` keyword to return an anonymous type that contains just the `Name` and `Price` properties. This styl known as *query syntax*, and it is the kind that developers find most comfortable when they start using LINQ. The wr query is that it returns one anonymously typed object for every `Product` in the array that I used in the source que to play around with the results to get the first three and print out the details.

However, if you are willing to forgo the simplicity of the query syntax, you can get a lot more power from LINQ. alternative is the *dot-notation syntax*, or *dot notation*, which is based on extension methods. Listing 4-29 shows this syntax used to process the `Product` objects.

Listing 4-29. Using LINQ Dot Notation in the HomeController.cs File

```
...
public ViewResult FindProducts() {

    Product[] products = {
          new Product {Name = "Kayak", Category = "Watersports",
```

eBooks-IT.org

```
            new Product {Name = "Lifejacket", Category = "Watersports
= 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer",
19.50M},
            new Product {Name = "Corner flag", Category = "Soccer",
34.95M}
        };

        var foundProducts = products.OrderByDescending(e => e.Price)
                                    .Take(3)
                                    .Select(e => new { e.Name, e.Price });

        StringBuilder result = new StringBuilder();
        foreach (var p in foundProducts) {
            result.AppendFormat("Price: {0} ", p.Price);
        }

        return View("Result", (object)result.ToString());
    }
...
```

This LINQ query, shown in bold, is not as nice to look at as the one expressed in query syntax, but not all LINQ f corresponding C# keywords. For serious LINQ queries, I need to switch to using extension methods. Each of the LI methods in the listing is applied to an `IEnumerable<T>` and returns an `IEnumerable<T>` too, which al chain the methods together to form complex queries.

Note All of the LINQ extension methods are in the `System.Linq` namespace, which you must bring into sc `using` statement before you can make queries. Visual Studio adds the `System.Linq` namespace to controller automatically, but you may need to add it manually elsewhere in an MVC project.

The `OrderByDescending` method rearranges the items in the data source. In this case, the lambda expres the value I want used for comparisons. The `Take` method returns a specified number of items from the front of the is what I couldn't do using query syntax). The `Select` method allows me to project my results, specifying the str want. In this case, I am projecting an anonymous object that contains the `Name` and `Price` properties.

Tip Notice that I have not needed to specify the names of the properties in the anonymous type. C# has inferred properties I picked in the `Select` method.

Table 4-2 describes the most useful LINQ extension methods. I use LINQ liberally throughout the rest of this book may find it useful to return to this table when you see an extension method that you have not encountered before. All methods shown in the table operate on `IEnumerable<T>`.

***Table 4-2**. Some Useful LINQ Extension Methods*

Extension Method	Description
`All`	Returns `true` if all the items in the source data match the predicate
`Any`	Returns `true` if at least one of the items in the source data matches the predicate
`Contains`	Returns `true` if the data source contains a specific item or value
`Count`	Returns the number of items in the data source
`First`	Returns the first item from the data source
`FirstOrDefault`	Returns the first item from the data source or the default value if there are no items
`Last`	Returns the last item in the data source
`LastOrDefault`	Returns the last item in the data source or the default value if there are no items

OrderBy OrderByDescending	Sorts the source data based on the value returned by the lambda expression
Reverse	Reverses the order of the items in the data source
Select	Projects a result from a query
SelectMany	Projects each data item into a sequence of items and then concatenates all of those resulting sequences into a single sequence
Single	Returns the first item from the data source or throws an exception if there are multiple matches
SingleOrDefault	Returns the first item from the data source or the default value if there are no items, or throws an exception there are multiple matches
Skip SkipWhile	Skips over a specified number of elements, or skips while the predicate matches
Sum	Totals the values selected by the predicate
Take TakeWhile	Selects a specified number of elements from the start of the data source or selects items while the predicate matches
ToArray ToDictionary ToList	Converts the data source to an array or other collection type
Where	Filters items from the data source that do not match the predicate

Understanding Deferred LINQ Queries

You will notice that Table 2 includes a Deferred column. There is an interesting variation in the way that the ext methods are executed in a LINQ query. A query that contains only deferred methods is not executed until the items i are enumerated, as demonstrated by Listing 4-30, which shows a simple change to the FindProducts action r

Listing 4-30. Using Deferred LINQ Extension Methods in a Query in the HomeController.cs File

```
...
public ViewResult FindProducts() {

    Product[] products = {
          new Product {Name = "Kayak", Category = "Watersports",
275M},
         new Product {Name = "Lifejacket", Category = "Watersports
= 48.95M},
          new Product {Name = "Soccer ball", Category = "Soccer",
19.50M},
          new Product {Name = "Corner flag", Category = "Soccer",
34.95M}
    };

    var foundProducts = products.OrderByDescending(e => e.Price)
                            .Take(3)
                            .Select(e => new {
                                e.Name,
                                e.Price
                            });

    products[2] = new Product { Name = "Stadium", Price = 79600M }

    StringBuilder result = new StringBuilder();
    foreach (var p in foundProducts) {
        result.AppendFormat("Price: {0} ", p.Price);
    }
```

eBooks-IT.org

```
    return View("Result", (object)result.ToString());
}
...
```

Between defining the LINQ query and enumerating the results, I changed one of the items in the `products` arr
output from this example is as follows:

```
Price: 79600 Price: 275 Price: 48.95
```

You can see that the query is not evaluated until the results are enumerated, and so the change I made—introducin
into the `Product` array—is reflected in the output.

Tip One interesting feature that arises from deferred LINQ extension methods is that queries are evaluated from
time the results are enumerated, meaning that you can perform the query repeatedly as the source data for the chang
results that reflect the current state of the source data.

By contrast, using any of the non-deferred extension methods causes a LINQ query to be performed immediately.
shows the `SumProducts` action method I added to the `Home` controller.

Listing 4-31. An Immediately Executed LINQ Query in the HomeController.cs File

```
...
public ViewResult SumProducts() {

    Product[] products = {
          new Product {Name = "Kayak", Category = "Watersports",
275M},
         new Product {Name = "Lifejacket", Category = "Watersports
= 48.95M},
          new Product {Name = "Soccer ball", Category = "Soccer",
19.50M},
          new Product {Name = "Corner flag", Category = "Soccer",
34.95M}
    };

    var results = products.Sum(e => e.Price);

    products[2] = new Product { Name = "Stadium", Price = 79500M }

    return View("Result",
        (object)String.Format("Sum: {0:c}", results));
}
...
```

This example uses the `Sum` method, which is not deferred, and produces the following result:

```
Sum: $378.40
```

You can see that the `Stadium` item, with its much higher price, has not been included in the results—this is bec
results from the `Sum` method are evaluated as soon as the method is called, rather than being deferred until the resul

Using Async Methods

One of the big recent additions to C# is improvements in the way that *asynchronous methods* are dealt with. Asynchr
methods go off and do work in the background and notify you when they are complete, allowing your code to take c

eBooks-IT.org

bottlen

code and allow applications to take advantage of multiple processors and processor cores to perform work in parallel

C# and .NET have excellent support for asynchronous methods, but the code tends to be verbose and developers used to parallel programming often get bogged down by the unusual syntax. As an example, Listing 4-32 shows an a method called `GetPageLength`, which I defined in a class called `MyAsyncMethods` and added to the Mo folder in a class file called `MyAsyncMethods.cs`.

Listing 4-32. A Simple Asynchronous Method in the MyAsyncMethods.cs File

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public static Task<long?> GetPageLength() {

            HttpClient client = new HttpClient();

            var httpTask = client.GetAsync("http://apress.com");

            // we could do other things here while we are waiting
            // for the HTTP request to complete

                  return  httpTask.ContinueWith((Task<HttpResponse
antecedent) => {
                 return antecedent.Result.Content.Headers.ContentLe
            });
        }
    }
}
```

Caution This example requires the `System.Net.Http` assembly, which I added to the project at the start chapter.

This is a simple method that uses a `System.Net.Http.HttpClient` object to request the contents of home page and returns its length. I have highlighted the part of the method that tends to cause confusion, which is a *task continuation*.

.NET represents work that will be done asynchronously as a `Task`. `Task` objects are strongly typed based on that the background work produces. So, when I call the `HttpClient.GetAsync` method, what I get back `Task<HttpResponseMessage>`. This tells me that the request will be performed in the background and of the request will be an `HttpResponseMessage` object.

Tip When I use words like *background*, I am skipping over a lot of detail in order to make the key points that are the world of MVC. The .NET support for asynchronous methods and parallel programming in general is excellent an you to learn more about it if you want to create truly high-performing applications that can take advantage of multico multiprocessor hardware. I come back to asynchronous methods for MVC in Chapter 19.

The part that most programmers get bogged down with is the *continuation*, which is the mechanism by which yo what you want to happen when the background task is complete. In the example, I have used the `ContinueWit` process the `HttpResponseMessage` object I get from the `HttpClient.GetAsync` method, which lambda expression that returns the value of a property that returns the length of the content I get from the Apress W Notice that I use the `return` keyword twice:

```
...
```

eBooks-IT.org

```
    return antecedent.Result.Content.Headers.ContentLength;
});
...
```

This is the part that makes heads hurt. The first use of the `return` keyword specifies that I am returning a `Task<HttpResponseMessage>` object, which, when the task is complete, will `return` the length of t `ContentLength` header. The `ContentLength` header returns a `long?` result (a nullable long value) ar that the result of my `GetPageLength` method is `Task<long?>`, like this:

```
...
public static Task<long?> GetPageLength() {
...
```

Do not worry if this does not make sense—you are not alone in your confusion. And this is a simple example—co asynchronous operations can chain large numbers of tasks together using the `ContinueWith` method, which c that can be hard to read and harder to maintain.

Applying the async and await Keywords

Microsoft introduced two keywords to C# that are specifically intended to simplify using asynchronous methods like `HttpClient.getAsync`. The keywords are `async` and `await` and you can see how I have used them my example method in Listing 4-33.

Listing 4-33. Using the Async and Await Keywords in the MyAsyncMethods.cs File

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public async static Task<long?> GetPageLength() {

            HttpClient client = new HttpClient();

            var httpMessage = await client.GetAsync("http://apress

            // we could do other things here while we are waiting
            // for the HTTP request to complete
            return httpMessage.Content.Headers.ContentLength;
        }
    }
}
```

I used the `await` keyword when calling the asynchronous method. This tells the C# compiler that I want to wai result of the `Task` that the `GetAsync` method returns and then carry on executing other statements in the same

Applying the `await` keyword means I can treat the result from the `GetAsync` method as though it were a re and just assign the `HttpResponseMessage` object that it returns to a variable. And, even better, I can then u `return` keyword in the normal way to produce a result from other method—in this case, the value of the `ContentLength` property. This is a much more natural technique and it means I do not have to worry about th `ContinueWith` method and multiple uses of the `return` keyword.

When you use the `await` keyword, you must also add the `async` keyword to the method signature, as I have example. The method result type does not change—my example `GetPageLength` method still returns a `Task` This is because the `await` and `async` are implemented using some clever compiler tricks, meaning that they allo

eBooks-IT.org

who is
GetPageLength method still has to deal with a Task<long?> result because there is still a background o
produces a nullable long—although, of course, that programmer can also choose to use the await and asy
keywords as well.

Note You will have noticed that I did not provide an MVC example for you to test out the async and await
This is because using asynchronous methods in MVC controllers requires a special technique, and I have a lot of info
present to you before I introduce it in Chapter 19.

Summary

In this chapter, I gave you an overview of the key C# language features that an effective MVC programmer needs to
features are combined in LINQ, which I use to query data throughout this book. As I said, I am a big fan of LINQ, a
important role in MVC applications. I also showed you the async and await keywords, which make it easier to
asynchronous methods—this is a topic that I return to in Chapter 19 when I show you an advanced technique for int
asynchronous programming into your MVC controllers.

In the next chapter, I turn my attention to the Razor View Engine, which is the mechanism by which dynamic data
into views.

eBooks-IT.org

Working with Razor

A *view engine* procezses ASP.NET content and looks for instructions, typically to insert dynamic content into the ou browser and *Razor* is the name of the MVC Framework view engine. There are no changes to Razor in MVC 5 and i already familiar with the syntax from earlier versions you can skip ahead.

In this chapter, I give you a quick tour of the Razor syntax so you can recognize Razor expressions when you see not going to supply an exhaustive Razor reference in this chapter; think of this more as a crash course in the syntax. Razor in depth as I continue through the book, within the context of other MVC Framework features. Table 5-1 prov summary for this chapter.

***Table 5-1**. Chapter Summary*

Problem	Solution	Listing
Define and access the model type	Use the `@model` and `@Model` expressions	1–4, 15
Reduce duplication in views	Use a layout	5–7, 10–12
Specify a default layout	Use the view start view	8, 9
Pass data values to the view from the controller	Pass a view model object or the view bag	13, 14
Generate different content based on data values	Use Razor conditional statements	16, 17
Enumerate an array or a collection	Use a `@foreach` expression	18, 19
Add a namespace to a view	Use a `@using` expression	20

Preparing the Example Project

To demonstrate Razor, I created a new Visual Studio project called Razor using the `ASP.NET MVC Web Application` template. I selected the `Empty` option and checked the box to get the initial configuration for a project.

Defining the Model

I am going to start with a simple domain model called `Product`, defined in a class file called `Product.cs`, w to the `Models` folder. You can see the contents of the new file in Listing 5-1.

***Listing 5-1**.* The Contents of the Product.cs File

```
namespace Razor.Models {

    public class Product {

        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
```

Defining the Controller

I am going to follow the MVC Framework convention and define a controller called Home as the initial starting poin project. Right-click the Controllers folder in the Solution Explorer, select Add ➤ Controller, select Controller-Empty, click Add and set the name to HomeController. When you click the second A Visual Studio will create the Controllers/HomeController.cs file. Edit the contents to match those Listing 5-2.

Listing 5-2. The Content of the HomeController.cs File

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers {

    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

        public ActionResult Index() {
            return View(myProduct);
        }
    }
}
```

The controller defines an action method called Index, in which I create and populate the properties of a Prod pass the Product to the View method so that it is used as the model when the view is rendered. I do not specif a view file when I call the View method, so the default view for the action method will be used.

Creating the View

Right-click on the Index method in the HomeController class and select Add View from the pop-up m that the name of the view is Index, change the Template to Empty and select Product for the model cl don't see Product as an option for the model, compile the project and start over). Uncheck the View Optio click Add to create the Index.cshtml file in the Views/Home folder. The initial contents of the new view in Listing 5-3.

Listing 5-3. The Contents of the Index.cshtml File

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
```

eBooks-IT.org

```
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>

    </div>
</body>
</html>
```

In the sections that follow, I go through the different parts of a Razor view and demonstrate some of the different can do with one. When learning about Razor, it is helpful to bear in mind that views exist to express one or more par model to the user—and that means generating HTML that displays data that is retrieved from one or more objects. If remember that I am always trying to build an HTML page that can be sent to the client, then everything that Razor do make sense.

Note I repeat some information in the following sections that I already touched on in Chapter 2. I want to provid single place in the reference that you can turn to when you need to look up a Razor feature and I thought that a smal duplication made this worthwhile.

Working with the Model Object

Let us start with the first line in the view:

```
...
@model Razor.Models.Product
...
```

Razor statements start with the `@` character. In this case, the `@model` statement declares the type of the model will pass to the view from the action method. This allows me to refer to the methods, fields, and properties of the vi object through `@Model`, as shown in Listing 5-4, which shows a simple addition to the `Index` view.

Listing 5-4. Referring to a View Model Object Property in the Index.cshtml File

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @Model.Name
    </div>
</body>
</html>
```

eBooks-IT.org

me pr

`@Model` (upper case `M`). This is slightly confusing when you start working with Razor, but it becomes second nat quickly.

If you start the project, you'll see the output shown in Figure 5-1.

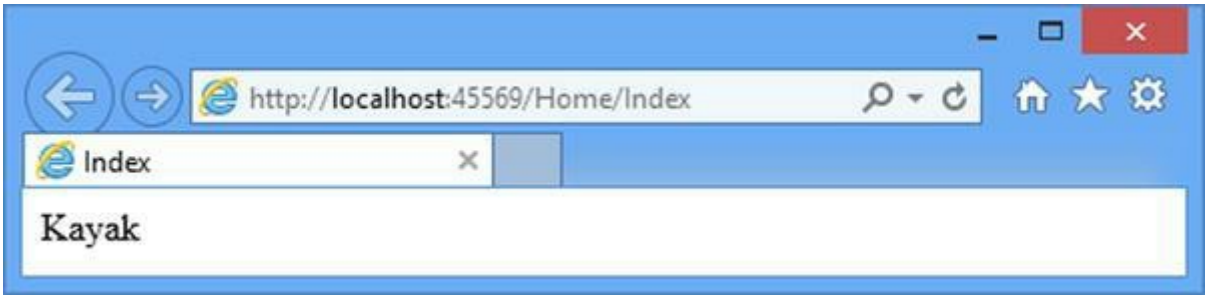


Figure 5-1. The Effect of Reading a Property Value in the View

By using the `@model` expression, I tell MVC what kind of object I will be working with and Visual Studio takes this in a couple of ways. First, as you are writing your view, Visual Studio will pop up suggestions of member name type `@Model` followed by a period, as shown in Figure 5-2. This is similar to the way that autocomplete for lambd passed to HTML helper methods works, which I described in Chapter 4.

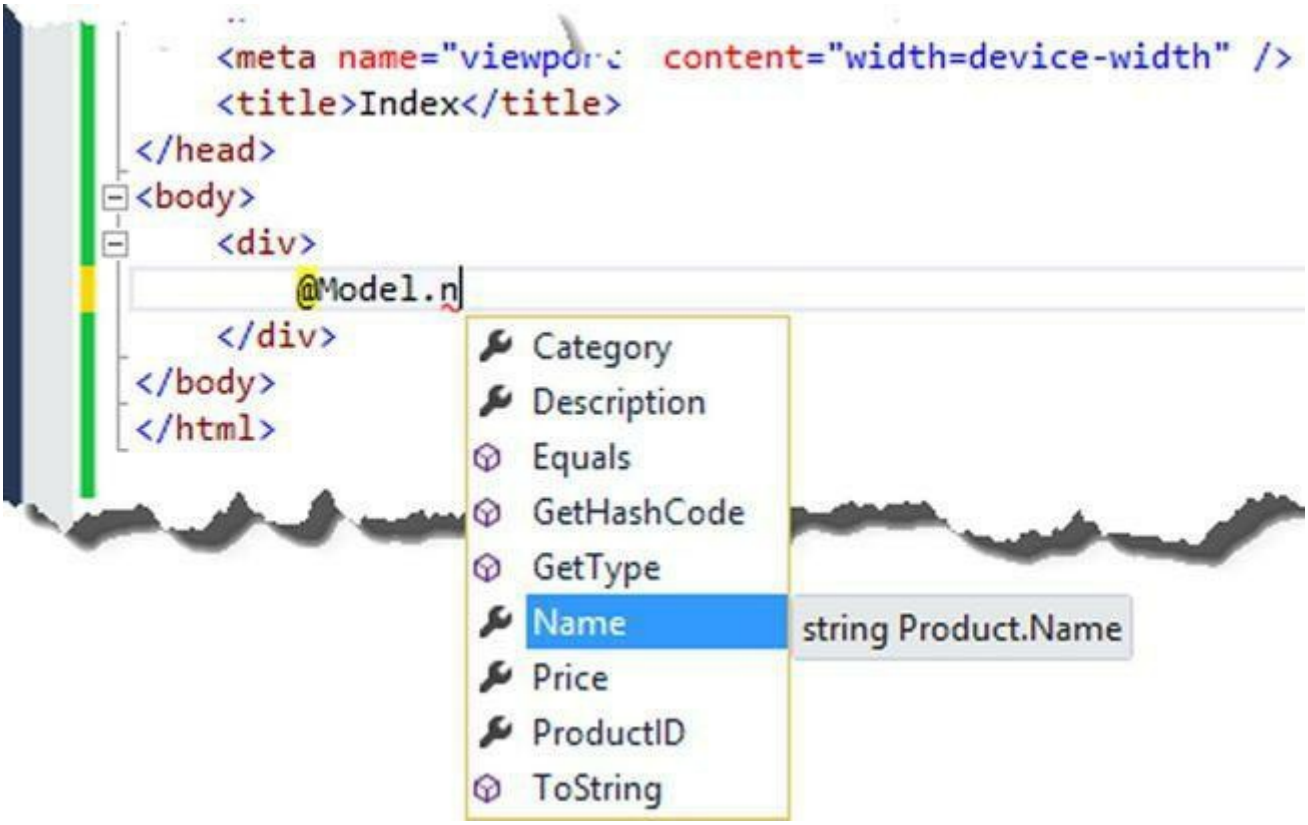


Figure 5-2. Visual Studio offering suggestions for member names based on the @Model expression

Equally useful is that Visual Studio will flag errors when there are problems with the view model object members y referring to. You can see an example of this in Figure 5-3, where I have tried to reference `@Model.NotARealProperty`. Visual Studio has realized that the `Product` class I specified at the mod not have such a property and has highlighted an error in the editor.

eBooks-IT.org

```
    <title>Index</title>
</head>
<body>
    <div>
        @Model.NotARealProperty
    </div>
</body>
</html>
```

Figure 5-3. Visual Studio reporting a problem with an @Model expression

Working with Layouts

The other Razor expression in the `Index.cshtml` view file is this one:

```
...
@{
    Layout = null;
}
...
```

This is an example of a Razor *code block*, which allows me to include C# statements in a view. The code block is `@{` and closed with `}` and the statements it contains are evaluated when the view is rendered.

This code block sets the value of the `Layout` property to `null`. As I explain in detail in Chapter 20, Razor vie compiled into C# classes in an MVC application and the base class that is used defines the `Layout` property. I'll sl this all works in Chapter 20, but the effect of setting the `Layout` property to `null` is to tell the MVC framework is self-contained and will render all of the content required for the client.

Self-contained views are fine for simple example apps, but a real project can have dozens of views. Layouts are ef templates that contain markup that you use to create consistency across your app—this could be to ensure that the r JavaScript libraries are included in the result or that a common look and feel is used throughout.

Creating the Layout

To create a layout, right-click on the `Views` folder in the Solution Explorer, click `Add` ➤ `New Item` from the and select the `MVC 5 Layout Page (Razor)` template, as shown in Figure 5-4.

eBooks-IT.org

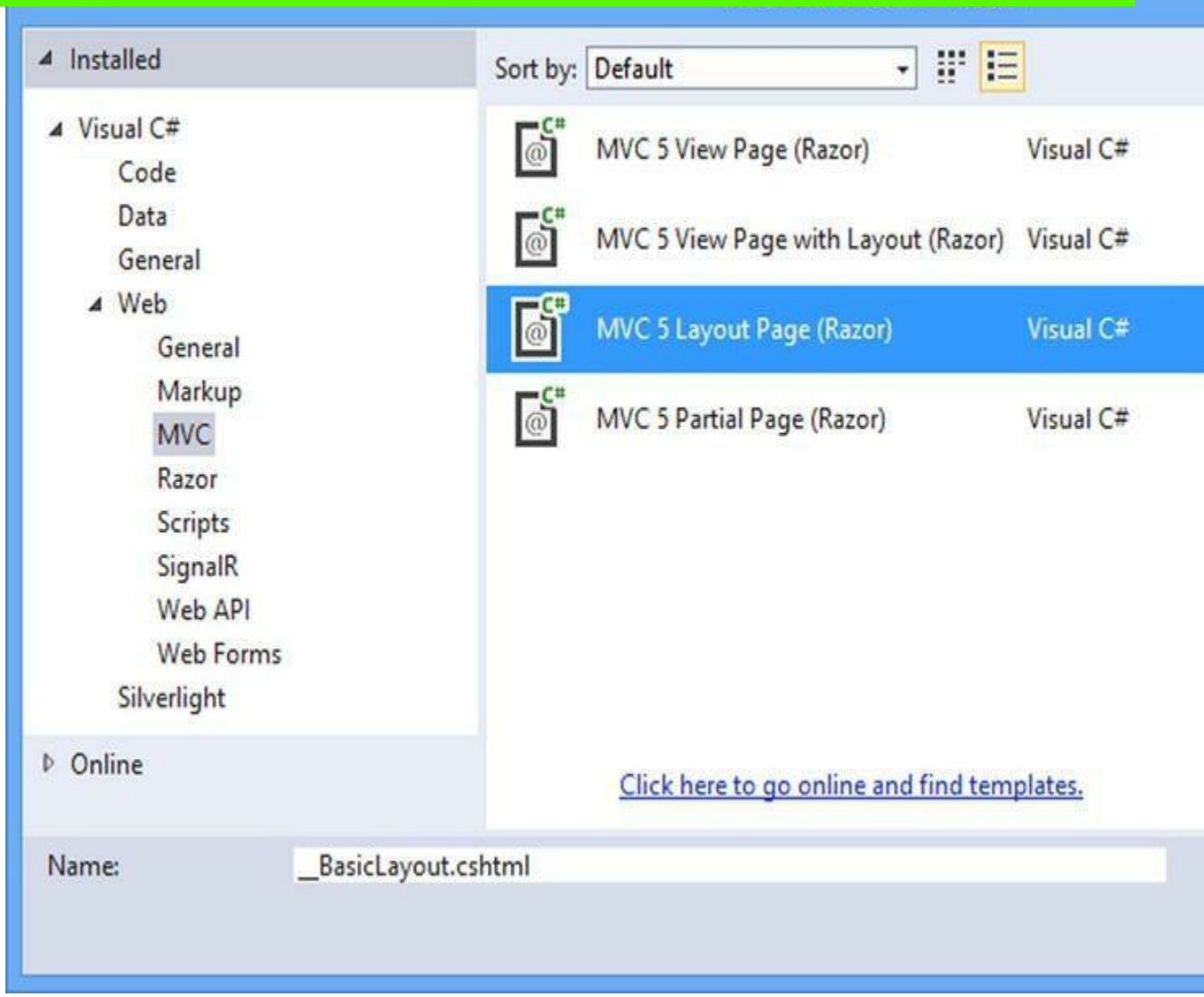


Figure 5-4. Creating a new layout

Set the name of the file to `_BasicLayout.cshtml` (notice the first character is an underscore) and click button to create the file. Listing 5-5 shows the contents of the file as it is created by Visual Studio.

Note Files in the `Views` folder whose names begin with an underscore (_) are not returned to the user, which name to differentiate between views that you want to render and the files that support them. Layouts, which are supp prefixed with an underscore.

Listing 5-5. The Initial Contents of the _BasicLayout.cshtml File

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title> @ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
```

Layouts are a specialized form of view and I have highlighted the @ expressions in the listing. The call to the @Re method inserts the contents of the view specified by the action method into the layout markup. The other Razor expr layout looks for a property called Title in the ViewBag in order to set the contents of the title element.

The elements in the layout will be applied to any view that uses the layout and this is why layouts are essentially te Listing 5-6, I have added some simple markup to the layout to demonstrate how this works.

Listing 5-6. Adding Elements to the _BasicLayout.cshtml File

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <h1>Product Information</h1>
      <div style="padding: 20px; border: solid medium black; fo
20pt">
        @RenderBody()
    </div>
    <h2>Visit <a href="http://apress.com">Apress</a></h2>
</body>
</html>
```

I have added a couple of header elements and applied some CSS styles to the div element which contains the @RenderBody expression, just to make it clear what content comes from the layout and what comes from the v

Applying a Layout

To apply the layout to the view, I just need to set the value of the Layout property. The layout contains the HTM that define the structure of the response to the browser, so I can also remove those elements from the view. You ca have applied the layout in Listing 5-7, which shows a drastically simplified Index.cshtml file.

Tip I also set a value for the ViewBag.Title property, which will be used as the contents of the title elem HTML document sent back to the user—this is optional, but good practice. If there is no value for the property, the framework will return an empty title element.

Listing 5-7. Using the Layout Property in the Index.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "Product Name";
    Layout = "~/Views/_BasicLayout.cshtml";
}

Product Name: @Model.Name
```

The transformation is dramatic, even for such a simple view. What I am left with is focused on presenting data fr model object to the user, which is ideal–not only does it let me work with simpler markup, but it means that I don't h duplicate common elements in every view that I create. To see the effect of the layout, run the example app. The res shown in Figure 5-5.

eBooks-IT.org

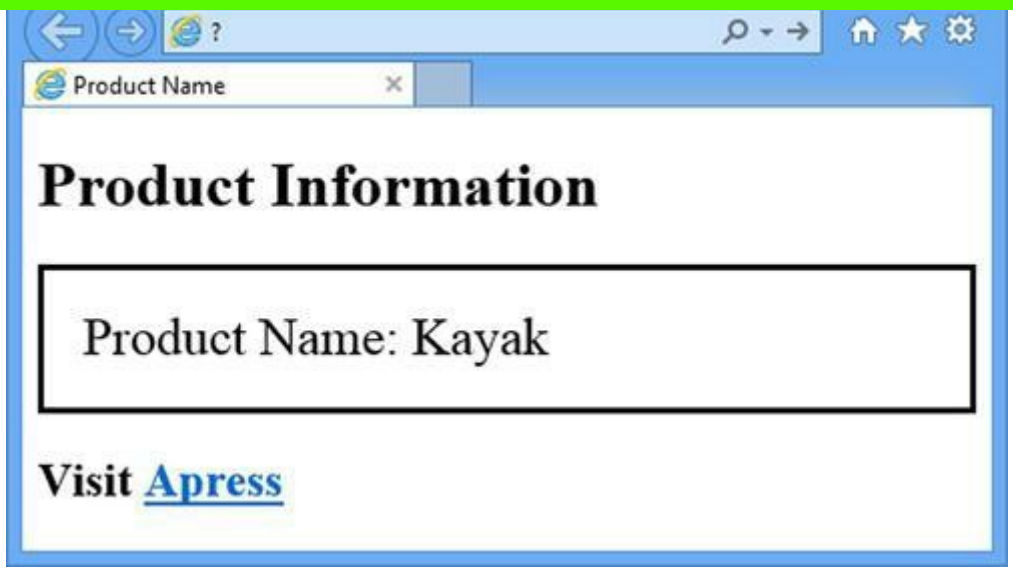


Figure 5-5. The Effect of Applying a Simple Layout to a View

Using a View Start File

I still have a tiny wrinkle to sort out, which is that I have to specify the layout file I want in every view. That means to rename the layout file, I am going to have to find every view that refers to it and make a change, which will be an process and counter to the general theme of easy maintenance that runs through the MVC framework.

I can resolve this by using a *view start file*. When it renders a view, the MVC framework will look for a file called `_ViewStart.cshtml`. The contents of this file will be treated as though they were contained in the view file can use this feature to automatically set a value for the `Layout` property.

To create a view start file, add a new layout file to the `Views` folder and set the name of the file to `_ViewStart.cshtml` (once again, notice the leading underscore). Edit the contents of the new file so that th those shown in Listing 5-8.

Listing 5-8. The contents of the _ViewStart.cshtml File

```
@{
    Layout = "~/Views/_BasicLayout.cshtml";
}
```

My view start file contains a value for the `Layout` property, which means that I can remove the corresponding from the `Index.cshtml` file, as shown in Listing 5-9.

Listing 5-9. Updating the Index.cshtml File to Reflect the Use of a View Start File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "Product Name";
}

Product Name: @Model.Name
```

I do not have to specify that I want to use the view start file. The MVC framework will locate the file and use its c automatically. The values defined in the view file take precedence, which makes it easy to override the view start file

Caution It is important to understand the difference between omitting the `Layout` property from the view file to `null`. If your view is self-contained and you do not want to use a layout, then set the `Layout` property to `nu`

eBooks-IT.org

ld use

finds in the view start file.

Demonstrating Shared Layouts

As a quick and simple demonstration of how layouts are shared, I have added a new action method to the `Home` co `NameAndPrice`. You can see the definition of this method in Listing 5-10, which shows the changes I made to `HomeController.cs` file.

Listing 5-10. Adding a New Action Method to the HomeController.cs File

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers {

    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

        public ActionResult Index() {
            return View(myProduct);
        }

        public ActionResult NameAndPrice() {
            return View(myProduct);
        }
    }
}
```

The action method passes the `myProduct` object to the view method, just like the `Index` action method doe something that you would do in a real project, but I am demonstrating Razor functionality and so a simple example su Right-click on the `NameAndPrice` action method and select `Add View` from the pop-up menu. Fill in the A dialog to match Figure 5-6: set `View Name` to `NameAndPrice`; set `Template` to `Empty` and set the M `Class` to `Product`.

eBooks-IT.org

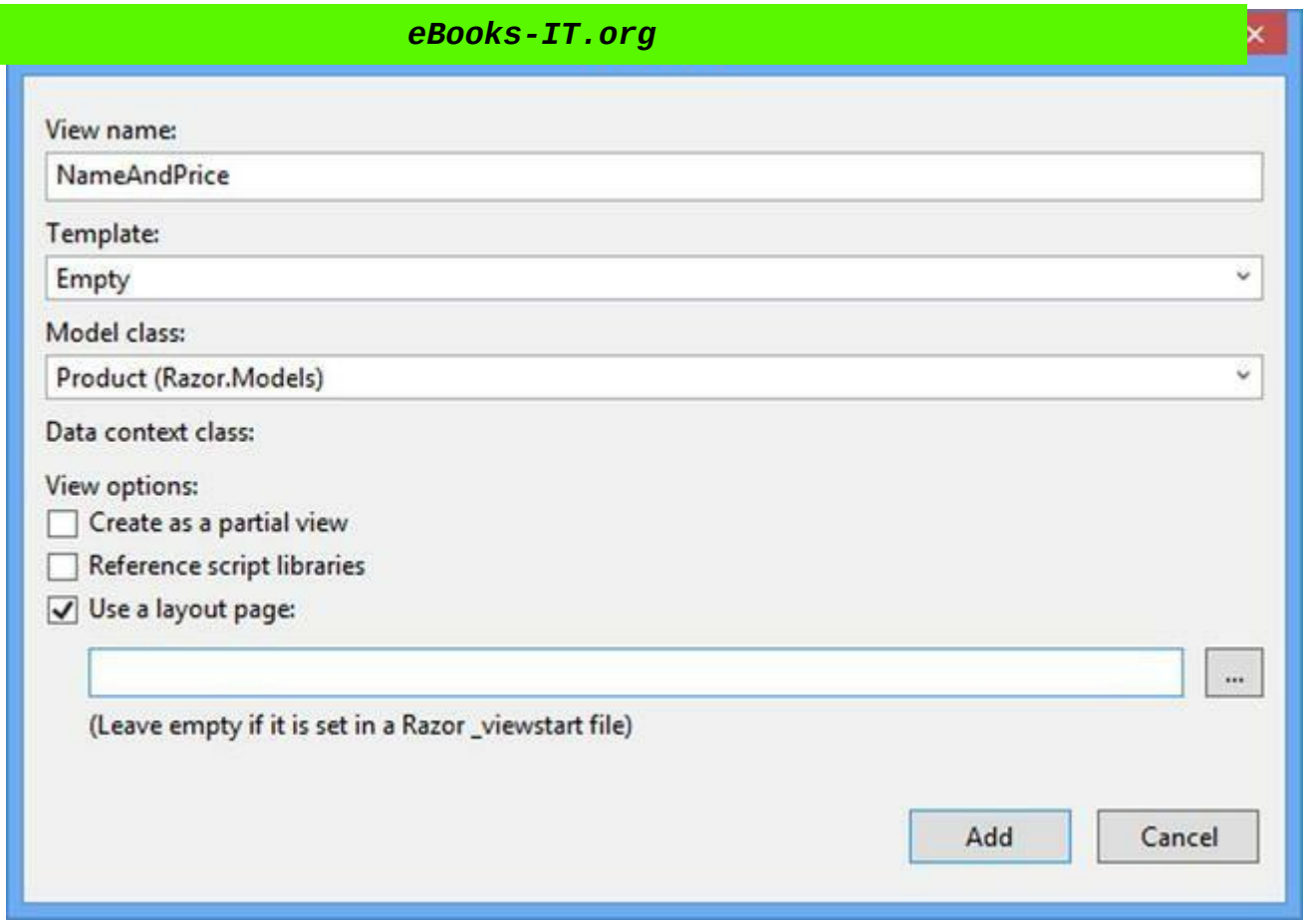


Figure 5-6. Adding a view that relies on a layout

Ensure that the `Use a Layout Page` box is checked–and notice the text beneath the text field. It says tha leave the textbox empty if you have specified the view you want to use in a view start file. If you were to click the A this point, the view would be created without a value for the `Layout` property.

I want to explicitly specify the view, so click on the button with an ellipsis label (`...`) that is to the right of the te Visual Studio will present you with a dialog that allows you to select a layout file, as shown in Figure 5-7.

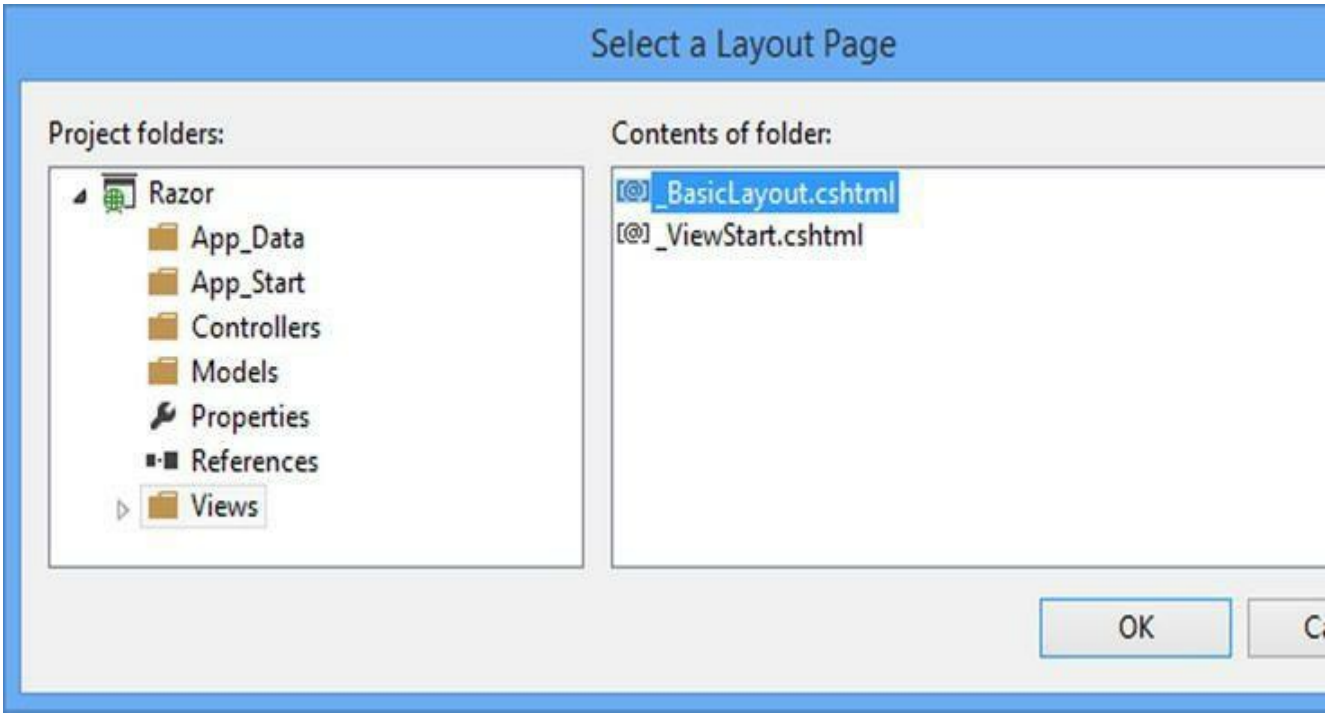


Figure 5-7. Selecting the layout file

eBooks-IT.org

which

left-hand panel of the dialog lets you navigate around the project, just in case you have decided to put them somewhe

I have only defined one layout file, so select `_BasicLayout.cshtml` and click the `OK` button to return to `View` dialog. You will see that the name of the layout file has been placed in the textbox, as shown in Figure 5-8.

Figure 5-8. Specifying a layout file when creating a view

Click the `Add` button to create the `/Views/Home/NameAndPrice.cshtml` file. You can see the co file in Listing 5-11.

Listing 5-11. The Contents of the NameAndPrice.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "NameAndPrice";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<h2>NameAndPrice</h2>
```

Visual Studio uses slightly different default content for view files when you specify a layout, but you can see that t contains the same Razor expressions I used when I applied the layout to a view directly. To complete this example, I shows a simple addition to the `NameAndPrice.cshtml` file that displays data values from the view model ob

Listing 5-12. Adding to the NameAndPrice.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "NameAndPrice";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<h2>NameAndPrice</h2>
The product name is @Model.Name and it costs $@Model.Price
```

If you start the app and navigate to `/Home/NameAndPrice`, you will see the results illustrated by Figure 5 might have expected, the common elements and styles defined in the layout have been applied to the view, demonstra layout can be used as a template to create a common look and feel (albeit a simple and unattractive one in this examp

eBooks-IT.org

Figure 5-9. The content in the layout file applied to the NameAndPrice view

Note I would have gotten the same result if I had left the text field in the `Add View` dialog empty and relied o start file. I specified the file explicitly only because I wanted to show you the Visual Studio feature which helps you r selection.

Using Razor Expressions

Now that I have shown you the basics of views and layouts, I am going to turn to the different kinds of expressions supports and how you can use them to create view content. In a good MVC Framework application, there is a clear between the roles that the action method and view perform. The rules are simple and I have summarized them in Tab

Table 5-2. The Roles Played by the Action Method and the View

Component	Does Do	Doesn't Do
Action Method	Passa view model object to the view	Pass formatted data to the view
View	Use the view model object to present content to the user	Change any aspect of the view model object

I am going to come back to this theme throughout this book. To get the best from the MVC Framework, you need and enforce the separation between the different parts of the app. As you will see, you can do quite a lot with Razor, using C# statements—but you should not use Razor to perform business logic or manipulate your domain model obje way.

Equally, you should not format the data that your action method passed to the view. Instead, let the view figure ou to display. You can see a simple example of this in the previous section of this chapter. I defined an action method ca `NameAndPrice`, which displays the value of the `Name` and `Price` properties of a `Product` object. Even knew which properties I needed to display, I passed the complete `Product` object to the view model, like this:

```
...
public ActionResult NameAndPrice() {
```

eBooks-IT.org

```
}
...
```

I then used the Razor `@Model` expression in the view to get the value of the properties I was interested in, like t

```
...
The product name is @Model.Name and it costs $ @Model.Price
...
```

I could have created the string I wanted to display in the action method and passed it as the view model object to t would have worked, but taking this approach undermines the benefit of the MVC pattern and reduces my ability to re changes in the future. As I said, I will return to this theme again, but you should remember that the MVC Framework enforce proper use of the MVC pattern and that you must remain aware of the effect of the design and coding decisi make.

PROCESSING VERSUS FORMATTING DATA

It is important to differentiate between *processing* data and *formatting* it. Views *format* data, which is why I pas `Product` object in the previous section to the view, rather than formatting the object's properties into a displ Processing data–including selecting the data objects to display–is the responsibility of the controller, which will model to get and modify the data it requires. It can sometimes be hard to figure out where the line between proc formatting is, but as a rule of thumb, I recommend erring on the side of caution and pushing anything but the m expressions out of the view and into the controller.

Inserting Data Values

The simplest thing you can do with a Razor expression is to insert a data value into the markup. You can do this usin `@Model` expression to refer to properties and methods defined by the view model object or use the `@ViewBag` refer to properties you have defined dynamically using the view bag feature (which I introduced in Chapter 2).

You have already seen examples of both these expressions, but for completeness, I have added a new action meth `Home` controller called `DemoExpressions` that passes data to the view using a model object and the view ba see the definition of the new action method in Listing 5-13.

Listing 5-13. The DemoExpression Action Method in the HomeController.cs File

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers {
    public class HomeController : Controller {
        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

        public ActionResult Index() {
            return View(myProduct);
        }

        public ActionResult NameAndPrice() {
            return View(myProduct);
        }
```

```
            ViewBag.ProductCount = 1;
            ViewBag.ExpressShip = true;
            ViewBag.ApplyDiscount = false;
            ViewBag.Supplier = null;

            return View(myProduct);
        }
    }
}
```

I have created a strongly typed view called `DemoExpression.cshtml` in the `Views/Home` folder to basic expressions and you can see the contents of the view file in Listing 5-14.

Listing 5-14. The Contents of the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr><td>Stock Level</td><td>@ViewBag.ProductCount</td></tr
    </tbody>
</table>
```

I created a simple HTML table and used the properties from the model object and the view bag to populate some o values. You can see the result of starting the app and navigating to the `/Home/DemoExpression` URL, as s Figure 5-10. This is just a reconfirmation of the basic Razor expressions that I have been using in the examples so fa

eBooks-IT.org

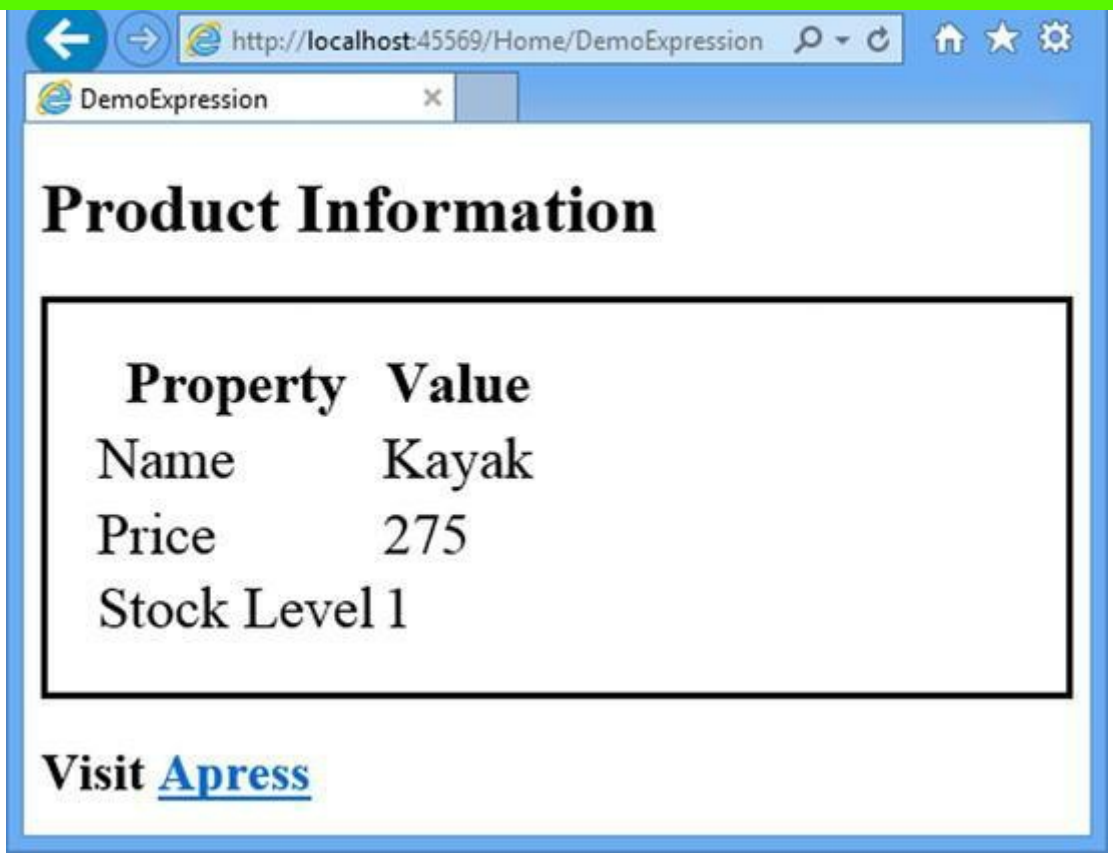


Figure 5-10. Using basic Razor expressions to insert data values into the HTML markup

The result is not pretty because I have not applied any CSS styles to the HTML elements that the view and the layo but the example serves to reinforce the way in which the basic Razor expressions can be used to display the data pas action method to the view.

Setting Attribute Values

All of my examples so far have set the content of elements, but you can also use Razor expressions to set the value o *attributes*. Listing 5-15 shows how I have changed the `DemoExpression.cshtml` view to use view bag p set attribute values.

Listing 5-15. Using a Razor Expression to Set an Attribute Value in the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr><td>Stock Level</td><td>@ViewBag.ProductCount</td></tr
    </tbody>
```

```
<div                data-discount="@ViewBag.ApplyDiscount"
express="@ViewBag.ExpressShip"
    data-supplier="@ViewBag.Supplier">
    The containing element has data attributes
</div>

Discount:<input type="checkbox" checked="@ViewBag.ApplyDiscount" /
Express:<input type="checkbox" checked="@ViewBag.ExpressShip" />
Supplier:<input type="checkbox" checked="@ViewBag.Supplier" />
```

I used basic Razor expressions to set the value for some `data` attributes on a `div` element.

Tip Data attributes, which are attributes whose names are prefixed by `data-`, have been an informal way of c
custom attributes for many years and have been made part of the formal standard as part of HTML5. I have used the
`ApplyDiscount, ExpressShip` and `Supplier` view bag properties to set the value of these attribu

Start the example app, target the action method, and look at the HTML source that has been used to render the pag
see that Razor has set the value of the attributes like this:

```
...
<div data-discount="False" data-express="True" data-supplier="" >
    The containing element has data attributes
</div>
...
```

The `False` and `True` values correspond to the Boolean view bag values, but Razor has done something sensib
property whose value is `null`, which is to render an empty string.

But things get interesting in the second set of additions to the view, which are a series of checkboxes whose `che`
attribute is set to the same view bag properties that I used for the data attributes. The HTML that Razor has rendered

```
...
Discount: <input type="checkbox" />
Express:  <input type="checkbox" checked="checked" />
Supplier: <input type="checkbox" />
...
```

Razor is aware of the way that attributes such as `checked` are used, where the presence of the attribute rather
changes the configuration of the element (known as a *Boolean attribute* in the HTML specification). If Razor had ins
`False` or `null` or the empty string as the value of the `checked` attribute, then the checkbox that the browser
would be checked. Instead, Razor has deleted the attribute from the element entirely when the value is `false` or `n`
creating an effect that is consistent with the view data, as shown in Figure 5-11.

eBooks-IT.org

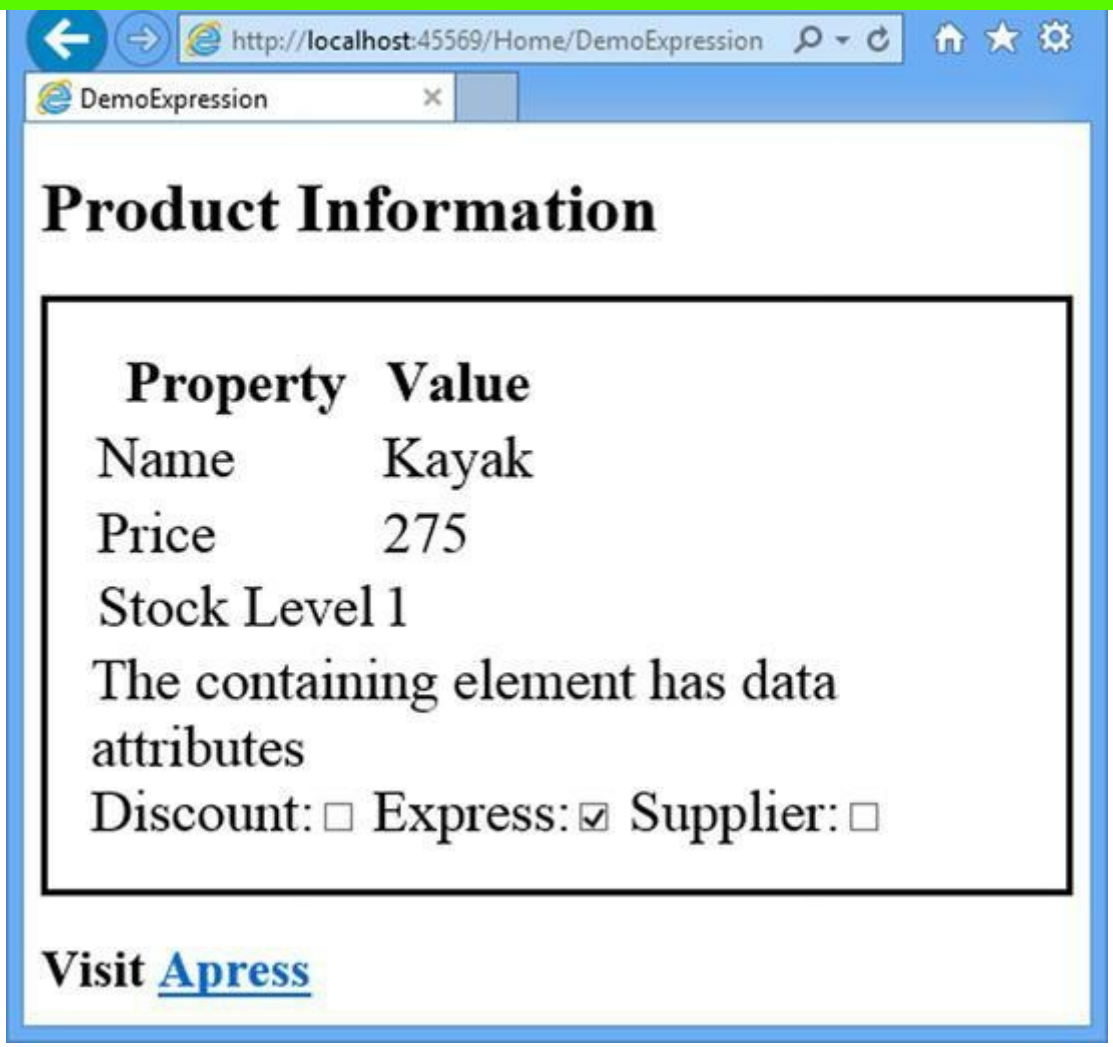


Figure 5-11. The effect of deleting attributes whose presence configures their element

Using Conditional Statements

Razor is able to process conditional statements, which means that I can tailor the output from a view based on values data. This kind of technique is at the heart of Razor, and allows you to create complex and fluid layouts that are still simple to read and maintain. In Listing 5-16, I have updated the `DemoExpression.cshtml` view file to inc conditional statement.

Listing 5-16. Using a Conditional Razor Statement in the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
```

```
            <tr>
                <td>Stock Level</td>
                <td>
                @switch ((int)ViewBag.ProductCount) {
                    case 0:
                        @: Out of Stock
                        break;
                    case 1:
                        <b>Low Stock (@ViewBag.ProductCount)</b>
                        break;
                    default:
                        @ViewBag.ProductCount
                        break;
                }
                </td>
            </tr>
        </tbody>
</table>
```

To start a conditional statement, you place an `@` character in front of the C# conditional keyword, which is `swit` example. You terminate the code block with a close brace character (`}`) just as you would with a regular C# code bl

Tip Notice that I had to cast the value of the `ViewBag.ProductCount` property to an `int` in order to `switch` statement. This is required because the Razor `switch` expression cannot evaluate a dynamic property– to a specific type so that it knows how to perform comparisons.

Inside of the Razor code block, you can include HTML elements and data values into the view output just by defin and Razor expressions, like this:

```
...
<b>Low Stock (@ViewBag.ProductCount)</b>
...
```

Or like this:

```
...
@ViewBag.ProductCount
...
```

I do not have to put the elements or expressions in quotes or denote them in any special way—the Razor engine w these as output to be processed. However, if you want to insert literal text into the view when it is not contained in a element, then you need to give Razor a helping hand and prefix the line like this:

```
...
@: Out of Stock
...
```

The `@:` characters prevent Razor from interpreting this as a C# statement, which is the default behavior when it e text. You can see the result of the conditional statement in Figure 5-12.

eBooks-IT.org

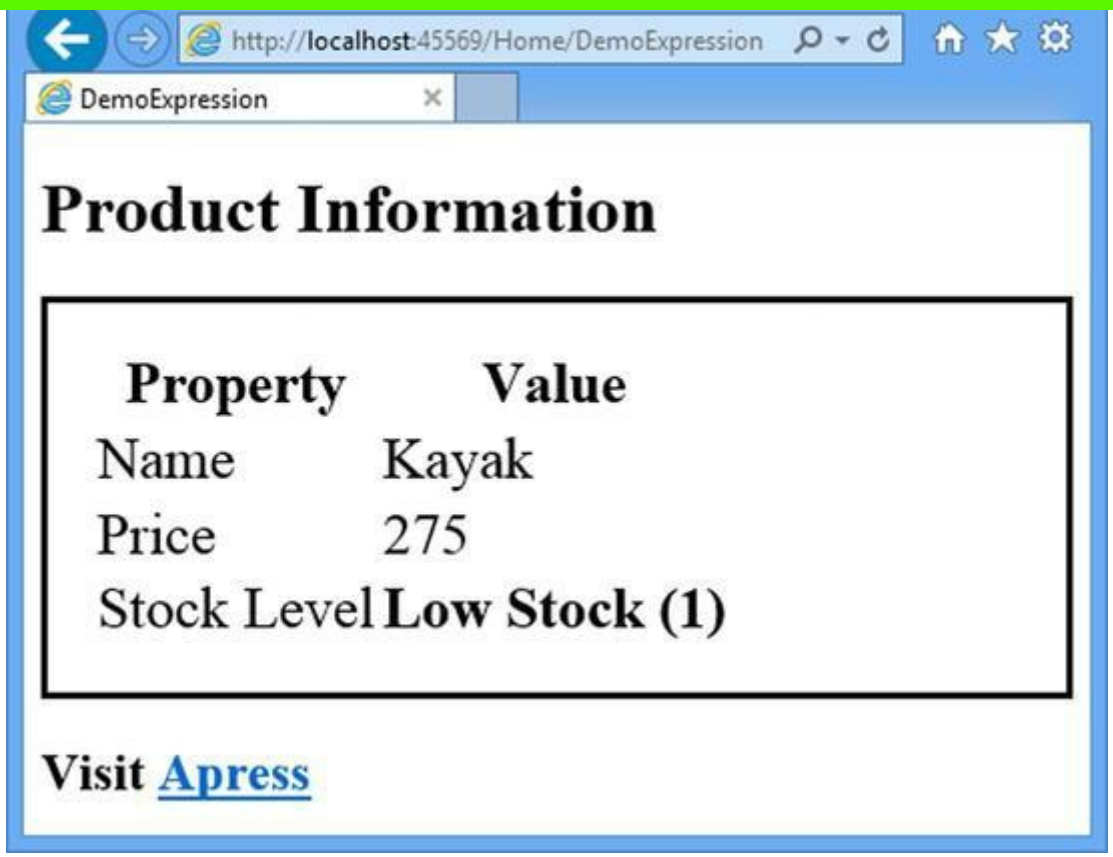


Figure 5-12. Using a switch statement in a Razor view

Conditional statements are important in Razor views because they allow content to be varied based on the data valu view receives from the action method. As an additional demonstration, Listing 5-17 shows the addition of an `if` stat `DemoExpression.cshtml` view. As you might imagine, this is a commonly used conditional statement.

Listing 5-17. Using an if Statement in a Razor View in the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr><td>Name</td><td>@Model.Name</td></tr>
        <tr><td>Price</td><td>@Model.Price</td></tr>
        <tr>
            <td>Stock Level</td>
            <td>
                @if (ViewBag.ProductCount == 0) {
                    @:Out of Stock
                } else if (ViewBag.ProductCount == 1) {
                    <b>Low Stock (@ViewBag.ProductCount)</b>
```

eBooks-IT.org

```
                    @ViewBag.ProductCount
                }
            </td>
        </tr>
    </tbody>
</table>
```

This conditional statement produces the same results as the `switch` statement, but I wanted to demonstrate how mesh C# conditional statements with Razor views. I explain how this all works in Chapter 20, when I explore views

Enumerating Arrays and Collections

When writing an MVC application, you will often want to enumerate the contents of an array or some other kind of c objects and generate content that details each one. To demonstrate how this is done, I have defined a new action met `Home` controller called `DemoArray` which you can see in Listing 5-18.

Listing 5-18. The DemoArray Action Method in the HomeController.cs File

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers {

    public class HomeController : Controller {

        Product myProduct = new Product {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            Category = "Watersports",
            Price = 275M
        };

        // ...other action methods omitted for brevity...

        public ActionResult DemoArray() {

            Product[] array = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M}
                new Product {Name = "Corner flag", Price = 34.95M}
            };
            return View(array);
        }
    }
}
```

This action method creates a `Product[]` object that contains simple data values and passes them to the `View` that the data is rendered using the default view. The Visual Studio scaffold feature won't let you specify an array as a (I don't know why, since Razor itself happily supports array.) To create a view for an action method that passes an best approach is to create a view without a model and then manually add the `@model` expression after the file has b In Listing 5-19, you can see the contents of the `DemoArray.cshtml` file that I created this way in the `View` folder and then edited.

```
@model Razor.Models.Product[]

@{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}

@if (Model.Length > 0) {
    <table>
        <thead><tr><th>Product</th><th>Price</th></tr></thead>
        <tbody>
            @foreach (Razor.Models.Product p in Model) {
                <tr>
                    <td>@p.Name</td>
                    <td>$@p.Price</td>
                </tr>
            }
        </tbody>
    </table>
} else {
    <h2>No product data</h2>
}
```

I used an `@if` statement to vary the content based on the length of the array that I receive as the view model and `@foreach` expression to enumerate the contents of the array and generate a row in an HTML table for each of th see how these expressions match their C# counterpart. You can also see how I created a local variable called `p` in th loop and then referred to its properties using the Razor expressions `@p.Name` and `@p.Price`.

The result is that I generate an `h2` element if the array is empty and produce one row per array item in an HTML otherwise. Because my data is static in this example, you will always see the same result, which I have shown in Fig

eBooks-IT.org

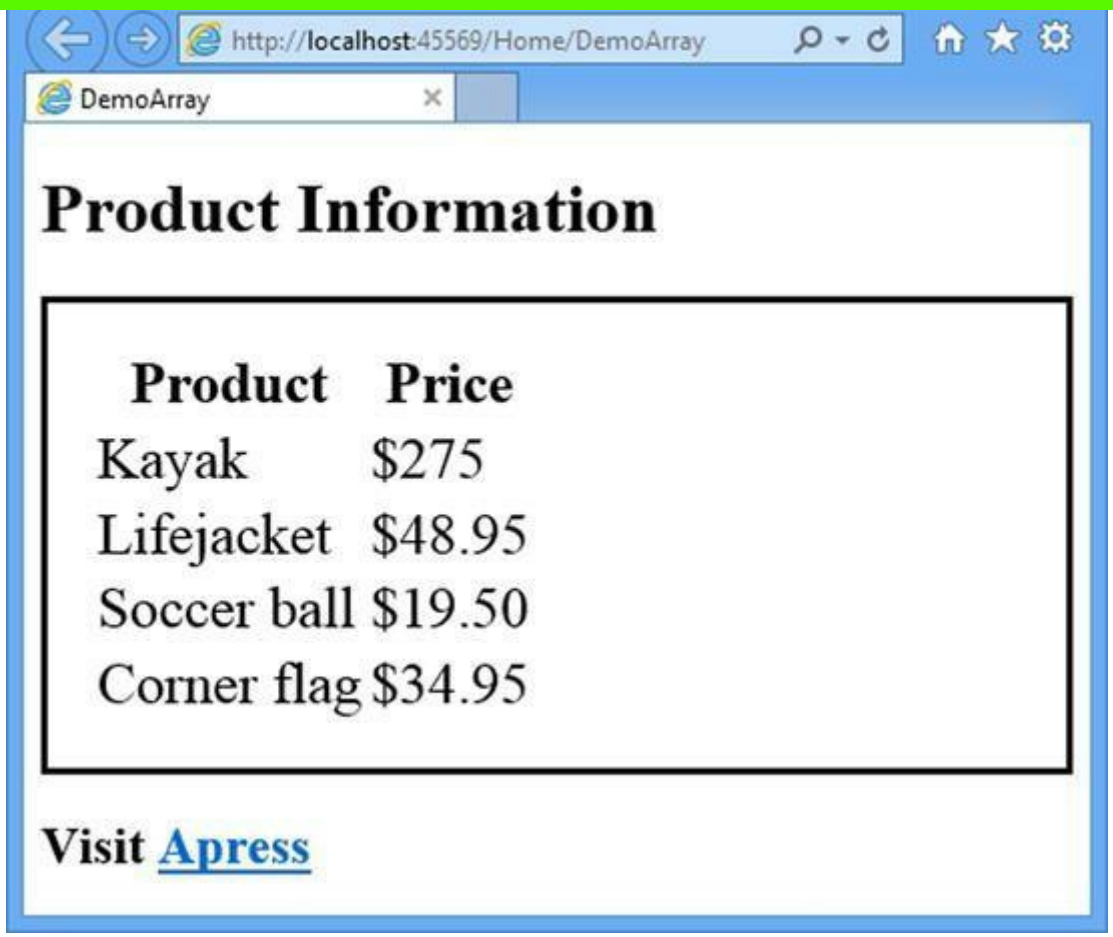


Figure 5-13. Generating elements using a foreach statement

Dealing with Namespaces

You will notice that I had to refer to the `Product` class by its fully qualified name in the `foreach` loop in the l
like this:

```
...
@foreach (Razor.Models.Product p in Model) {
...
```

This can become annoying in a complex view, where you will have many references to view model and other clas
up my views by applying the `@using` expression to bring a namespace into context for a view, just like I would in
class. Listing 5-20 shows how I have applied the `@using` expression to the `DemoArray.cshtml` view.

Listing 5-20. Applying the @using Expression in the DemoArray.cshtml File

```
@using Razor.Models
@model Product[]

@{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}

@if (Model.Length > 0) {
    <table>
        <thead><tr><th>Product</th><th>Price</th></tr></thead>
```

eBooks-IT.org

```
                @foreach (Product p in Model) {
                    <tr>
                        <td>@p.Name</td>
                        <td>$@p.Price</td>
                    </tr>
                }
            </tbody>
        </table>
    } else {
        <h2>No product data</h2>
    }
```

A view can contain multiple `@using` expressions. I have used the `@using` expression to import the `Razor` namespace, which means that I can remove the namespace from the `@model` expression and from within the `fo`

Summary

In this chapter, I have given you an overview of the Razor view engine and how it can be used to generate HTML. I how to refer to data passed from the controller via the view model object and the view bag, and how Razor expressi used to tailor responses to the user based on data values. You will see many different examples of how Razor can be rest of the book and I describe how the MVC view mechanism works in detail in Chapter 20. In the next chapter, I c essential development and testing tools that underpin the MVC Framework and that help you to get the best from you

eBooks-IT.org

Essential Tools for MVC

In this chapter, I am going to look at three tools that should be part of every MVC programmer's arsenal: a depender (DI) container, a unit test framework, and a mocking tool.

I have picked three specific implementations of these tools for this book, but there are many alternatives for each t you cannot get along with the ones I use, do not worry. There are so many out there, that you are certain to find sor suits the way your mind and workflow operate.

As I noted in Chapter 3, Ninject is my preferred DI container. It is simple, elegant, and easy to use. There are mor sophisticated alternatives, but I like the way that Ninject works with a minimum of configuration. If you do not like N recommend trying Unity, which is an alternative from Microsoft.

For unit testing, I am going to be using the built-in Visual Studio testing tools. I used to use NUnit, which is a popu unit-testing framework, but Microsoft has made a big push to improve the unit-testing support in Visual Studio and n in the free Visual Studio editions. The result is a unit test framework that is tightly integrated into the rest of the IDE has actually become pretty good.

The third tool I selected is Moq, which is a mocking tool kit. I use Moq to create implementations of interfaces to tests. Programmers either love or hate Moq; there is nothing in the middle. Either you will find the syntax elegant and or you will be cursing every time you try to use it. If you just cannot get along with it, I suggest looking at Rhino Mo a nice alternative.

I introduce each of these tools and demonstrate their core features, but I do not provide exhaustive coverage of th Each could easily fill a book in its own right. I have given you enough to get started and, critically, to follow the exar rest of the book. Table 6-1 provides the summary for this chapter.

Table 6-1. *Chapter Summary*

Problem	Solution
Decouple classes	Introduce interfaces and declare dependencies on them in the class constructor
Automatically resolve dependencies expressed using interfaces	Use Ninject or another dependency injection container
Integrate Ninject into an MVC application	Create an implementation of the `IDependencyResolver` interface that calls the Ninject kernel and register it as a resolver by calling the `System.Web.Mvc.DependencyResolver.SetResolver` method
Inject property and constructor values into newly created objects	Use the `WithPropertyValue` and `WithConstructorArgument` methods
Dynamically select an implementation class for an interface	Use an Ninject conditional binding
Control the lifecycle of the objects that Ninject creates	Set an object scope
Create a unit test	Add a unit test project to the solution and annotate a class file with `TestClass` and `TestMethod` attributes
Check for expected outcomes in a unit test	Use the `Assert` class
Focus a unit test on a single feature of component	Isolate the test target using mock objects

eBooks-IT.org

ing an a
that supports lots of testing and an emphasis on creating applications that are easily modified and maintained. I love t
know that some readers will just want to understand the features that the MVC Framework offers and not get into th
development philosophy and methodology. I am not going to try to convert you. It is a personal decision and you kn
need to do to deliver your projects. I suggest that you at least have a quick skim through this chapter to see what is a
if you are not the unit-testing type, then you can skip to the next chapter and see how to build a more realistic examp

Preparing the Example Project

I am going to start by creating a simple example project, called `EssentialTools`, which I will use throughou
I used the `ASP.NET MVC Web Application` template, selected the `Empty` option and checked the bo
basic MVC project content.

Creating the Model Classes

I added a class file to the `Models` project folder called `Product.cs` and set the content as shown in Listing 6
model class from in previous chapters and the only change is that the namespace matches that of the `Essential`
project.

Listing 6-1. The Contents of the Product.cs File

```
namespace EssentialTools.Models {

    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

I also want to add a class that will calculate the total price of a collection of `Product` objects. I added a new cl
`Models` folder called `LinqValueCalculator.cs` and set the contents to match Listing 6-2.

Listing 6-2. The Contents of the LinqValueCalculator.cs File

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

namespace EssentialTools.Models {

    public class LinqValueCalculator {

        public decimal ValueProducts(IEnumerable<Product> products
            return products.Sum(p => p.Price);
        }
    }
}
```

The `LinqValueCalculator` class defines a single method called `ValueProducts`, which uses the
method to add together the value of the `Price` property of each `Product` object in an enumerable passed to the
nice LINQ feature that I use often).

My final model class is `ShoppingCart` and it represents a collection of `Product` objects and uses a

eBooks-IT.org

added the statements shown in Listing 6-3.

Listing 6-3. The contents of the ShoppingCart.cs File

```
using System.Collections.Generic;

namespace EssentialTools.Models {
    public class ShoppingCart {
        private LinqValueCalculator calc;

        public ShoppingCart(LinqValueCalculator calcParam) {
            calc = calcParam;
        }

        public IEnumerable<Product> Products { get; set; }

        public decimal CalculateProductTotal() {
            return calc.ValueProducts(Products);
        }
    }
}
```

Adding the Controller

I added a new controller to the `Controllers` folder called `HomeController` and set the content to matc
The `Index` action method creates an array of `Product` objects and uses a `LinqValueCalculator` ob
produce the total value, which I then pass to the `View` method. I do not specify a view when I call the `View` meth
MVC Framework will select the default view associated with the action method (the `Views/Home/Index.c`

Listing 6-4. The Contents of the HomeController.cs File

```
using System.Web.Mvc;
using System.Linq;
using EssentialTools.Models;

namespace EssentialTools.Controllers {
    public class HomeController : Controller {
        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
              new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
             new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
             new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
        };

        public ActionResult Index() {

            LinqValueCalculator calc = new LinqValueCalculator();

                ShoppingCart cart = new ShoppingCart(calc) { Pro
products };
```

eBooks-IT.org

```
            decimal totalValue = cart.CalculateProductTotal();

            return View(totalValue);
        }
    }
}
```

Adding the View

The last addition to the project is the view, called `Index`. It does not matter which options you check as you creat long as you set the contents to match those shown in Listing 6-5.

Listing 6-5. The Contents of the Index.cshtml File

```
@model decimal

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Value</title>
</head>
<body>
    <div>
        Total value is $@Model
    </div>
</body>
</html>
```

This view uses the `@Model` expression to display the value of the `decimal` passed from the action method. the project, you will see the total value, as calculated by the `LinqValueCalculator` class, illustrated by Fig is a simple project, but it sets the scene for the different tools and techniques that I describe in this chapter.

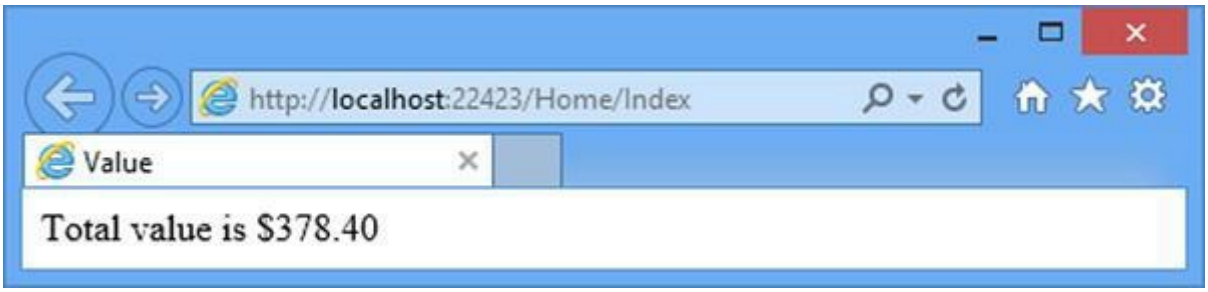


Figure 6-1. Testing the example app

Using Ninject

I introduced dependency injection (DI) in Chapter 3. To recap, the idea is to decouple the components in an MVC ap a combination of interfaces and DI container that creates instances of objects by creating implementations of the inte depend on and injecting them into the constructor.

eBooks-IT.org

Ninject my preferred DI container to solve it. Do not worry if you find that you cannot get along with Ninject–the b
principles are the same for all DI containers and there are many alternatives from which to choose.

Understanding the Problem

In the example app, I created an example of the basic problem that DI addresses: tightly coupled classes. The `ShoppingCart` class is tightly coupled to the `LinqValueCalculator` class and the `HomeContro` is tightly coupled to both `ShoppingCart` and `LinqValueCalculator`.

This means that if I want to replace the `LinqValueCalculator` class, I have to locate and the change th in the classes that are tightly coupled to it. This is not a problem with such a simple project, but it becomes a tedious prone process in a real project, especially if I want to switch between different calculator implementations (for testin example), rather than just replace one class with another.

Applying an Interface

I can solve part of the problem by using a C# interface to abstract the definition of the calculator functionality from i implementation. To demonstrate this, I have added the `IValueCalculator.cs` class file to the `Models` f created the interface shown in Listing 6-6.

Listing 6-6. The Contents of the IValueCalculator.cs File

```
using System.Collections.Generic;

namespace EssentialTools.Models {
    public interface IValueCalculator {

        decimal ValueProducts(IEnumerable<Product> products);
    }
}
```

I can then implement this interface in the `LinqValueCalculator` class, as shown in Listing 6-7.

Listing 6-7. Applying an Interface in the LinqValueCalculator.cs File

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

namespace EssentialTools.Models {

    public class LinqValueCalculator : IValueCalculator {

        public decimal ValueProducts(IEnumerable<Product> products
            return products.Sum(p => p.Price);
        }
    }
}
```

The interface allows me to break the tight coupling between the `ShoppingCart` and `LinqValueCalcu` class, as shown in Listing 6-8.

Listing 6-8. Applying the Interface in the ShoppingCart.cs File

```
using System.Collections.Generic;
```

eBooks-IT.org

```
    public class ShoppingCart {
        private IValueCalculator calc;

        public ShoppingCart(IValueCalculator calcParam) {
            calc = calcParam;
        }

        public IEnumerable<Product> Products { get; set; }

        public decimal CalculateProductTotal() {
            return calc.ValueProducts(Products);
        }
    }
}
```

I have made some progress, but C# requires me to specify the implementation class for an interface during instant is understandable because it needs to know *which* implementation class I want to use–but it means I still have a prob Home controller when I create the LinqValueCalculator object, as shown in Listing 6-9.

Listing 6-9. Applying the Interface to the HomeController.cs File

```
...
public ActionResult Index() {

    IValueCalculator calc = new LinqValueCalculator();

    ShoppingCart cart = new ShoppingCart(calc) { Products = produc

    decimal totalValue = cart.CalculateProductTotal();

    return View(totalValue);
}
...
```

My goal with Ninject is to reach the point where I specify that I want to instantiate an implementation of the IValueCalculator interface, but the details of which implementation is required are not part of the code in controller.

This will mean telling Ninject that LinqValueCalculator is the implementation of the IValueCalc interface that I want it to use and updating the HomeController class so that it obtains its objects via Ninject, by using the new keyword.

Adding Ninject to the Visual Studio Project

The simplest way to add Ninject to an MVC project is to use the integrated Visual Studio support for *NuGet*, which r to install a wide range of packages and keep them up to date. I used NuGet in Chapter 2 to install the Bootstrap librar a huge catalog of packages available, including Ninject.

Select Tools ➤ Library Package Manager ➤ Package Manager Console in Visual open the NuGet command line and enter the following commands:

```
Install-Package Ninject -version 3.0.1.10
Install-Package Ninject.Web.Common -version 3.0.0.7
Install-Package Ninject.MVC3 -Version 3.0.0.6
```

The first command installs the Ninject core package and the others install extensions to the core that makes Ninjec with ASP.NET applications (as I explain shortly). Do not be put off by the reference to MVC3 in the last package na

eBooks-IT.org

I have used the `version` argument to install specific versions of these packages. These are the latest versions a
You should use the `version` argument to ensure that you get the right results in the examples for this chapter, bu
omit the argument and get the latest (and potentially more recent) releases for real projects.

Getting Started with Ninject

There are three stages to getting the basic Ninject functionality working, and you can see all of them in Listing 6-10,
changes I have made to the `Home` controller.

Tip I am going to go slowly in this section and the sections that follow. Dependency Injection can take a while to
and I do not want to skip over anything that might help reduce confusion.

Listing 6-10. Adding the Basic Ninject Functionality to the Index Action Method in the HomeController.cs File

```
using System.Web.Mvc;
using EssentialTools.Models;
using Ninject;

namespace EssentialTools.Controllers {

    public class HomeController : Controller {

        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
              new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
             new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
             new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
        };

        public ActionResult Index() {

            IKernel ninjectKernel = new StandardKernel();
            ninjectKernel.Bind<IValueCalculator>().To<LinqValueCal
();

            IValueCalculator calc = ninjectKernel.Get<IValueCalcul

                ShoppingCart cart = new ShoppingCart(calc) { Pro
products };

            decimal totalValue = cart.CalculateProductTotal();

            return View(totalValue);
        }
    }
}
```

The first stage is to prepare Ninject for use. To do this, I create an instance of a Ninject *kernel*, which is the objec
responsible for resolving dependencies and creating new objects. When I need an object, I will use the kernel instead
keyword. Here is the statement that creates the kernel from the listing:

eBooks-IT.org

```
IKernel ninjectKernel = new StandardKernel();
...
```

I need to create an implementation of the `Ninject.IKernel` interface, which I do by creating a new instar `StandardKernel` class. Ninject can be extended and customized to use different kinds of kernel, but I only n in `StandardKernel` in this chapter. (In fact, I have been using Ninject for years and I have only ever needed `StandardKernel`).

The second stage of the process is to configure the Ninject kernel so that it understands which implementation obj use for each interface I am working with. Here is the statement from the listing that does that:

```
...
ninjectKernel.Bind< IValueCalculator >().To< LinqValueCalculator >
...
```

Ninject uses C# type parameters to create a relationship: I set the interface I want to work with as the type parame `Bind` method and call the `To` method on the result it returns. I set the implementation class I want instantiated as t parameter for the `To` method. This statement tells Ninject that dependencies on the `IValueCalculator` inte be resolved by creating an instance of the `LinqValueCalculator` class. The last step is to use Ninject to c object, which I do through the kernel `Get` method, like this:

```
...
IValueCalculator calc = ninjectKernel.Get<IValueCalculator>() ;
...
```

The type parameter used for the `Get` method tells Ninject which interface I am interested in and the result from t an instance of the implementation type I specified with the `To` method a moment ago.

Setting up MVC Dependency Injection

The result of the three steps I showed you in the previous section is that the knowledge about the implementation cla be instantiated to fulfill requests for the `IValueCalculator` interface has been set up in Ninject. Of course, improved my application because that knowledge remains defined in the `Home` controller, meaning that the `Home` c still tightly coupled to the `LinqValueCalculator` class.

In the following sections, I will show you how to embed Ninject at the heart of the MVC application, which will a simplify the controller, expand the influence Ninject has so that it works across the app, and move the configuration controller.

Creating the Dependency Resolver

The first change I am going to make is to create a *custom dependency resolver*. The MVC Framework uses a depend to create instances of the classes it needs to service requests. By creating a custom resolver, I can ensure that the M Framework uses Ninject whenever it creates an object–including instances of controllers, for example.

To set up the resolver, I created a new folder called `Infrastructure`, which is the folder that I use to put do not fit into the other folders in an MVC application. I added a new class file to the folder called `NinjectDependencyResolver.cs`, the contents of which you can see in Listing 6-11.

Listing 6-11. The Contents of the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using EssentialTools.Models;
using Ninject;
```

eBooks-IT.org

```
    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            kernel.Bind<IValueCalculator>().To<LinqValueCalculator
        }
    }
}
```

The `NinjectDependencyResolver` class implements the `IDependencyResolver` interface, of the `System.Mvc` namespace and which the MVC Framework uses to get the objects it needs. The MVC Fram call the `GetService` or `GetServices` methods when it needs an instance of a class to service an incoming job of a dependency resolver is to create that instance, a task that I perform by calling the Ninject `TryGet` and `Ge` methods. The `TryGet` method works like the `Get` method I used previously, but it returns `null` when there is binding rather than throwing an exception. The `GetAll` method supports multiple bindings for a single type, which when there are several different implementation objects available.

My dependency resolver class is also where I set up my Ninject binding. In the `AddBindings` method, I use and `To` methods to configure up the relationship between the `IValueCalculator` interface and the `LinqValueCalculator` class.

Register the Dependency Resolver

It is not enough to simply create an implementation of the `IDependencyResolver` interface–I also have to Framework that I want to use it. The Ninject packages I added with NuGet created a file called `NinjectWebCo` in the `App_Start` folder that defines methods called automatically when the application starts, in order to integra ASP.NET request lifecycle. (This is to provide the *scopes* feature that I describe later in the chapter.) In the `RegisterServices` method of the `NinjectWebCommon` class, I add a statement that creates an insta `NinjectDependencyResolver` class and uses the static `SetResolver` method defined by the `System.Web.Mvc.DependencyResolver` class to register the resolver with the MVC Framework, a Listing 6-12. Do not worry if this does not make complete sense. The effect of this statement is to create a bridge be and the MVC Framework support for DI.

Listing 6-12. Registering the Resolver in the NinjectWebCommon.cs File

```
...
private static void RegisterServices(IKernel kernel) {
    System.Web.Mvc.DependencyResolver.SetResolver(new
        EssentialTools.Infrastructure.NinjectDependencyResolver(ke
}
...
```

The final step is to refactor the Home controller so that it takes advantage of the facilities I set up in the previous se can see the changes I made in Listing 6-13.

Listing 6-13. Refactoring the Controller in the HomeController.cs File

```
using System.Web.Mvc;
using EssentialTools.Models;

namespace EssentialTools.Controllers {

    public class HomeController : Controller {
        private IValueCalculator calc;
        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
              new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
             new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
             new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
        };

        public HomeController(IValueCalculator calcParam) {
            calc = calcParam;
        }

        public ActionResult Index() {

                ShoppingCart cart = new ShoppingCart(calc) { Pr
products };

            decimal totalValue = cart.CalculateProductTotal();

            return View(totalValue);
        }
    }
}
```

The main change I have made is to add a class constructor that accepts an implementation of the IValueCal interface, changing the HomeController class so that it declares a dependency. Ninject will provide an object implements the IValueCalculator interface when it creates an instance of the controller, using the configu in the NinjectDependencyResolver class in Listing 6-10.

The other change I made is to remove any mention of Ninject or the LinqValueCalculator class from At last, I have broken the tight coupling between the HomeController and LinqValueCalculator

If you run the example, you will see the result shown in Figure 6-2. Of course, I got this same result when I was the LinqValueCalculator class directly in the controller.

eBooks-IT.org

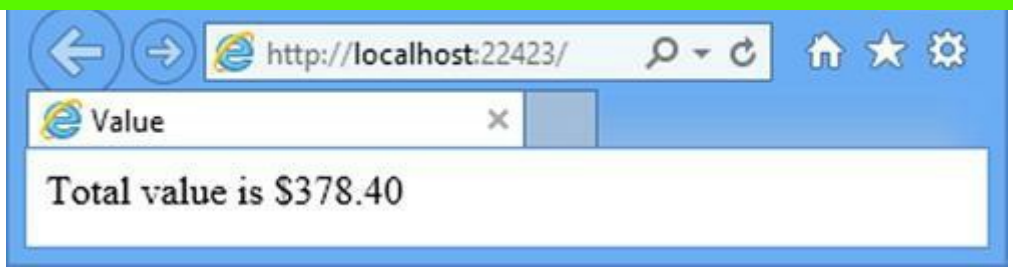


Figure 6-2. The effect of running the example app

I have created an example of *constructor injection*, which is one form of dependency injection. Here is what happe you ran the example app and Internet Explorer made the request for the root URL of the app:

1. The MVC Framework received the request and figured out that the request is intended for the `Home` controll explain how the MVC framework figures this out in Chapter 17).
2. The MVC Framework asked my custom dependency resolver class to create a new instance of the `HomeController` class, specifying the class using the `Type` parameter of the `GetService` meth
3. My dependency resolver asked Ninject to create a new `HomeController` class, passing on the `Type` `TryGet` method.
4. Ninject inspected the `HomeController` constructor and found that it has declared a dependency on the `IValueCalculator` interface, for which it has a binding.
5. Ninject creates an instance of the `LinqValueCalculator` class and uses it to create a new instance `HomeController` class.
6. Ninject passes the `HomeController` instance to the custom dependency resolver, which returns it to t Framework. The MVC Framework uses the controller instance to service the request.

I labored this slightly because DI can be a bit mind-bending when you see it used for the first time. One benefit of I have taken here is that *any* controller in the application can declare a dependency and the MVC Framework will use resolve it.

The best part is that I only have to modify my dependency resolver class when I want to replace the `LinqValueCalculator` with another implementation, because this is the only place where I have to specify implementation used to satisfy dependencies on the `IValueCalculator` interface.

Creating Chains of Dependency

When you ask Ninject to create a type, it examines the dependencies that the type has declared. It also looks at those to see if they rely on other types–or, put another way, if they declare their own dependencies. If there *are* additional Ninject automatically resolves them and creates instances of all of the classes that are required, working its way alon dependencies so that it can ultimately create an instance of the type you asked for.

To demonstrate this feature, I have added a file called `Discount.cs` to the `Models` folder and used it to d interface and a class that implements it, as shown in Listing 6-14.

Listing 6-14. The Contents of the Discount.cs File

```
namespace EssentialTools.Models {

    public interface IDiscountHelper {
        decimal ApplyDiscount(decimal totalParam);
    }

    public class DefaultDiscountHelper : IDiscountHelper {
```

eBooks-IT.org

```
        public decimal ApplyDiscount(decimal totalParam) {
            return (totalParam - (10m / 100m * totalParam));
        }
    }
}
```

The `IDiscountHelper` defines the `ApplyDiscount` method, which will apply a discount to a dec
The `DefaultDiscounterHelper` class implements the `IDiscountHelper` interface and applies a
percent discount. I have modified the `LinqValueCalculator` class so that it uses the `IDiscountHel`
interface when it performs calculations, as shown in Listing 6-15.

Listing 6-15. Adding a Dependency in the LinqValueCalculator.cs File

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

namespace EssentialTools.Models {

    public class LinqValueCalculator: IValueCalculator {
        private IDiscountHelper discounter;

        public LinqValueCalculator(IDiscountHelper discountParam)
            discounter = discountParam;
        }

        public decimal ValueProducts(IEnumerable<Product> products
            return discounter.ApplyDiscount (products.Sum(p => p.P
        }
    }
}
```

The new constructor declares a dependency on the `IDiscountHelper` interface. I assign the implementati
the constructor receives to a field and use it in the `ValueProducts` method to apply a discount to the cumulat
the `Product` objects.

I bind the `IDiscountHelper` interface to the implementation class with the Ninject kernel in the
`NinjectDependencyResolver` class, just as I did for `IValueCalculator`, as shown in Listing

Listing 6-16. Binding Another Interface to Its Implementation in the NinjectDependencyResolver.cs File

```
...
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>().To<DefaultDiscountHelper>();
}
...
```

I have created a dependency chain. My `Home` controller depends on the `IValueCalculator` interface, w
told Ninject to resolve using the `LinqValueCalculator` class. The `LinqValueCalculator` class
the `IDiscountHelper` interface, which I have told Ninject to resolve using the `DefaultDiscountHe`

Ninject resolves the chain of dependencies seamlessly, creating the objects it needs to resolve every dependency a
in this example, create an instance of the `HomeController` class to service an HTTP request.

Specifying Property and Constructor Parameter Values

eBooks-IT.org

n I bind
to its implementation. To demonstrate this feature, I have revised the DefaultDiscountHelper class so t
a DiscountSize property, which I use to calculate the discount amount, as shown in Listing 6-17.

Listing 6-17. Adding a Property in the Discount.cs File

```
namespace EssentialTools.Models {

    public interface IDiscountHelper {
        decimal ApplyDiscount(decimal totalParam);
    }

    public class DefaultDiscountHelper : IDiscountHelper {

        public decimal DiscountSize { get; set; }

        public decimal ApplyDiscount(decimal totalParam) {
            return (totalParam - (DiscountSize / 100m * totalParam
        }
    }
}
```

When I tell Ninject which class to use for an interface, I can use the WithPropertyValue method to set the DiscountSize property in the DefaultDiscountHelper class. You can see the change I made AddBindings method in the NinjectDependencyResolver class to set this up, as shown in Listi
Notice that I supply the name of the property to set as a string value.

Listing 6-18. Using the Ninject WithPropertyValue Method in the NinjectDependencyResolver.cs File

```
...
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
            .To<DefaultDiscountHelper>().WithPropertyValue("Discou
50M);
}
...
```

I do not need to change any other binding, nor change the way I use the Get method to obtain an instance of the ShoppingCart class. The property value is set following construction of the DefaultDiscountHel and has the effect of halving the total value of the items. Figure 6-3 shows the result of this change.

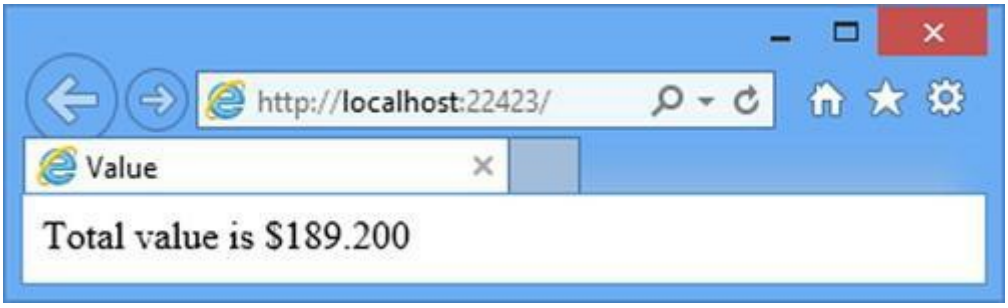


Figure 6-3. The effect of applying a discount through a property when resolving the dependency chain

If you have more than one property value you need to set, you can chain calls to the WithPropertyValue cover them all. I can do the same thing with constructor parameters. Listing 6-19 shows the DefaultDiscounterHelper class reworked so that the size of the discount is passed as a constructor p

```
namespace EssentialTools.Models {

    public interface IDiscountHelper {
        decimal ApplyDiscount(decimal totalParam);
    }

    public class DefaultDiscountHelper : IDiscountHelper {
        public decimal discountSize;

        public DefaultDiscountHelper(decimal discountParam) {
            discountSize = discountParam;
        }

        public decimal ApplyDiscount(decimal totalParam) {
            return (totalParam - (discountSize / 100m * totalParam
        }
    }
}
```

To bind this class using Ninject, I specify the value of the constructor parameter using the `WithConstructorArgument` method in the `AddBindings` method, as shown in Listing 6-20.

Listing 6-20. Specifying a Constructor Parameter in the NinjectDependencyResolver.cs File

```
...
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
                                                .To<DefaultDiscoun
().WithConstructorArgument("discountParam", 50M);
}
...
```

Once again, I can chain these method calls together to supply multiple values and mix and match with dependencie figure out what I need and create it accordingly.

Tip Notice that I did not just change the `WithPropertyValue` call to `WithConstructorArgun` changed the name of the member targeted so that it matches the C# convention for parameter names.

Using Conditional Binding

Ninject supports a number of conditional binding methods that allow me to specify which classes the kernel should u to requests for particular interfaces. To demonstrate this feature, I have added a new file to the `Models` folder of t project called `FlexibleDiscountHelper.cs`, the contents of which you can see in Listing 6-21.

Listing 6-21. The Contents of the FlexibleDiscountHelper.cs File

```
namespace EssentialTools.Models {
    public class FlexibleDiscountHelper : IDiscountHelper {

        public decimal ApplyDiscount(decimal totalParam) {
            decimal discount = totalParam > 100 ? 70 : 25;
            return (totalParam - (discount / 100m * totalParam));
        }
```

eBooks-IT.org

}

The FlexibleDiscountHelper class applies different discounts based on the magnitude of the total. N have a choice of classes that implement the IDiscountHelper interface, I can modify the AddBinding the NinjectDependencyResolver to tell Ninject when I want to use each of them, as shown in Listing

Listing 6-22. Using Conditional Binding in the NinjectDependencyResolver.cs File

```
...
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
                                                .To<DefaultDiscoun
().WithConstructorArgument("discountParam", 50M);
    kernel.Bind<IDiscountHelper>().To<FlexibleDiscountHelper>()
        .WhenInjectedInto<LinqValueCalculator>();

}
...
```

The new binding specifies that the Ninject kernel should use the FlexibleDiscountHelper class as th implementation of the IDiscountHelper interface when it is creating a LinqValueCalculator ob that I have left the original binding for IDiscountHelper in place. Ninject tries to find the best match and it l a default binding for the same class or interface, so that Ninject has a fallback if the criteria for a conditional binding satisfied. Ninject supports a number of different conditional binding methods, the most useful of which I have listed

Table 6-2. Ninject Conditional Binding Methods

Method	Effect
When(predicate)	Binding is used when the predicate—a lambda expression—evaluates to true.
WhenClassHas<T>()	Binding is used when the class being injected is annotated with the attribute whose type is specified by T
WhenInjectedInto<T>()	Binding is used when the class being injected into is of type T.

Setting the Object Scope

The last Ninject feature helps tailor the lifecycle of the objects that Ninject creates to match the needs of your applica default, Ninject will create a new instance of every object needed to resolve every dependency each time you request

To demonstrate what happens, I have modified the constructor for the LinqValueCalculator class so a message to the Visual Studio Output window when a new instance is created, as shown in Listing 6-23.

Listing 6-23. Adding a Constructor in the LinqValueCalculator.cs File

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

namespace EssentialTools.Models {

    public class LinqValueCalculator : IValueCalculator {
        private IDiscountHelper discounter;
        private static int counter = 0;

        public LinqValueCalculator(IDiscountHelper discountParam)
            discounter = discountParam;
```

eBooks-IT.org

```
                    String.Format("Instance {0} created", ++counter))
        }

        public decimal ValueProducts(IEnumerable<Product> products
            return discounter.ApplyDiscount(products.Sum(p => p.Pr
        }
    }
}
```

The System.Diagnostics.Debug class contains a number of methods that can be used to write out d messages and I find them useful when following code through to see how it works. I am, sadly, old enough that deb not sophisticated when I started writing code and I still find myself going back to basics when it comes to debugging

In Listing 6-24, I have modified the Home controller so that it demands two implementations of the IValueCalculator interface from Ninject.

Listing 6-24. Using Multiple Instances of the Calculator Class in the HomeController.cs File

```
...
public HomeController(IValueCalculator calcParam, IValueCalculato
{
    calc = calcParam;
}
...
```

I do not perform any useful task with the object that Ninject provides–what is important it that I asked for two imp of the interface. If you run the example and look at the Visual Studio Output window, you will see messages that sho created two instances of the LinqValueCalculator class:

```
Instance 1 created
Instance 2 created
```

The LinqValueCalculator can be instantiated repeatedly without any problems–but that is not true for For some classes, you will want to share a single instance throughout the entire application and for others, you will w a new instance for each HTTP request that the ASP.NET platform receives. Ninject lets you control the lifecycle of t you create using a feature called a *scope*, which is expressed using a method call when setting up the binding betwee and its implementation type. In Listing 6-25, you can see how I applied the most useful scope for MVC Framework a the *request scope* to the LinqValueCalculator class in the NinjectDependencyResolver.

Listing 6-25. Using the Request Scope in the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using EssentialTools.Models;
using Ninject;
using Ninject.Web.Common;

namespace EssentialTools.Infrastructure {
    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }
```

eBooks-IT.org

```
        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
                    kernel.Bind<IValueCalculator>().To<LinqValueCal
().InRequestScope();
            kernel.Bind<IDiscountHelper>()
                                                .To<DefaultDiscoun
().WithConstructorArgument("discountParam", 50M);
            kernel.Bind<IDiscountHelper>().To<FlexibleDiscountHelp
              .WhenInjectedInto<LinqValueCalculator>();
        }
    }
}
```

The `InRequestScope` extension method, which is in the `Ninject.Web.Common` namespace, tells N should only create one instance of the `LinqValueCalculator` class for each HTTP request that ASP.NET Each request will get its own separate object, but multiple dependencies resolved within the same request will be reso single instance of the class. You can see the effect of this change by starting the application and looking at the Visual Output window, which will show that Ninject has created only one instance of the `LinqValueCalculator` reload the browser window without restarting the application, you will see Ninject create a second object. Ninject pro of different object scopes and I have summarized the most useful in Table 6-3.

***Table 6-3**. Ninject Scope Methods*

Name	Effect
`InTransientScope()`	This is the same as not specifying a scope and creates a new object for each dependency that is resolved.
`InSingletonScope()` `ToConstant(object)`	Creates a single instance which is shared throughout the application. Ninject will create the instance if yo `InSingletonScope` or you can provide it with the `ToConstant` method.
`InThreadScope()`	Creates a single instance which is used to resolve dependencies for objects requested by a single thread.
`InRequestScope()`	Creates a single instance which is used to resolve dependencies for objects requested by a single HTTP req

Unit Testing with Visual Studio

In this book, I use the built-in unit test support that comes with Visual Studio, but there are other .NET unit-test pack available. The most popular is probably NUnit, but all of the test packages do much the same thing. The reason I hav Visual Studio test tools is that I like the integration with the rest of the IDE.

To demonstrate the Visual Studio unit-test support, I added a new implementation of the `IDiscountHelper` the example project. Create a new file in the `Models` folder called `MinimumDiscountHelper.cs` and e contents match those shown in Listing 6-26.

***Listing 6-26**.* The Contents of the MinumumDiscountHelper.cs File

```
using System;

namespace EssentialTools.Models {
    public class MinimumDiscountHelper : IDiscountHelper {
```

eBooks-IT.org

```
                throw new NotImplementedException();
            }
        }
    }
```

My objective in this example is to make the `MinimumDiscountHelper` demonstrate the following behav

- If the total is greater than $100, the discount will be 10 percent.
- If the total is between $10 and $100 inclusive, the discount will be $5.
- No discount will be applied on totals less than $10.
- An `ArgumentOutOfRangeException` will be thrown for negative totals.

The `MinimumDiscountHelper` class does not implement any of these behaviors yet. I am going to follo Driven Development (TDD) approach of writing the unit tests and only then implement the code, as described in Cha

Creating the Unit Test Project

The first step I take is to create the unit test project, which I do by right-clicking the top-level item in the Solution Ex is labeled `Solution 'EssentialTools'` for my example app) and selecting `Add` ➤ `New Projec` pop-up menu.

Tip You can choose to create a test project when you create a new MVC project: there is an `Add Unit Te` on the dialog where you choose the initial content for the project.

This will open the `Add New Project` dialog. Select `Test` from the `Visual C#` templates section in t and ensure that `Unit Test Project` is selected in the middle panel, as shown in Figure 6-4.

eBooks-IT.org

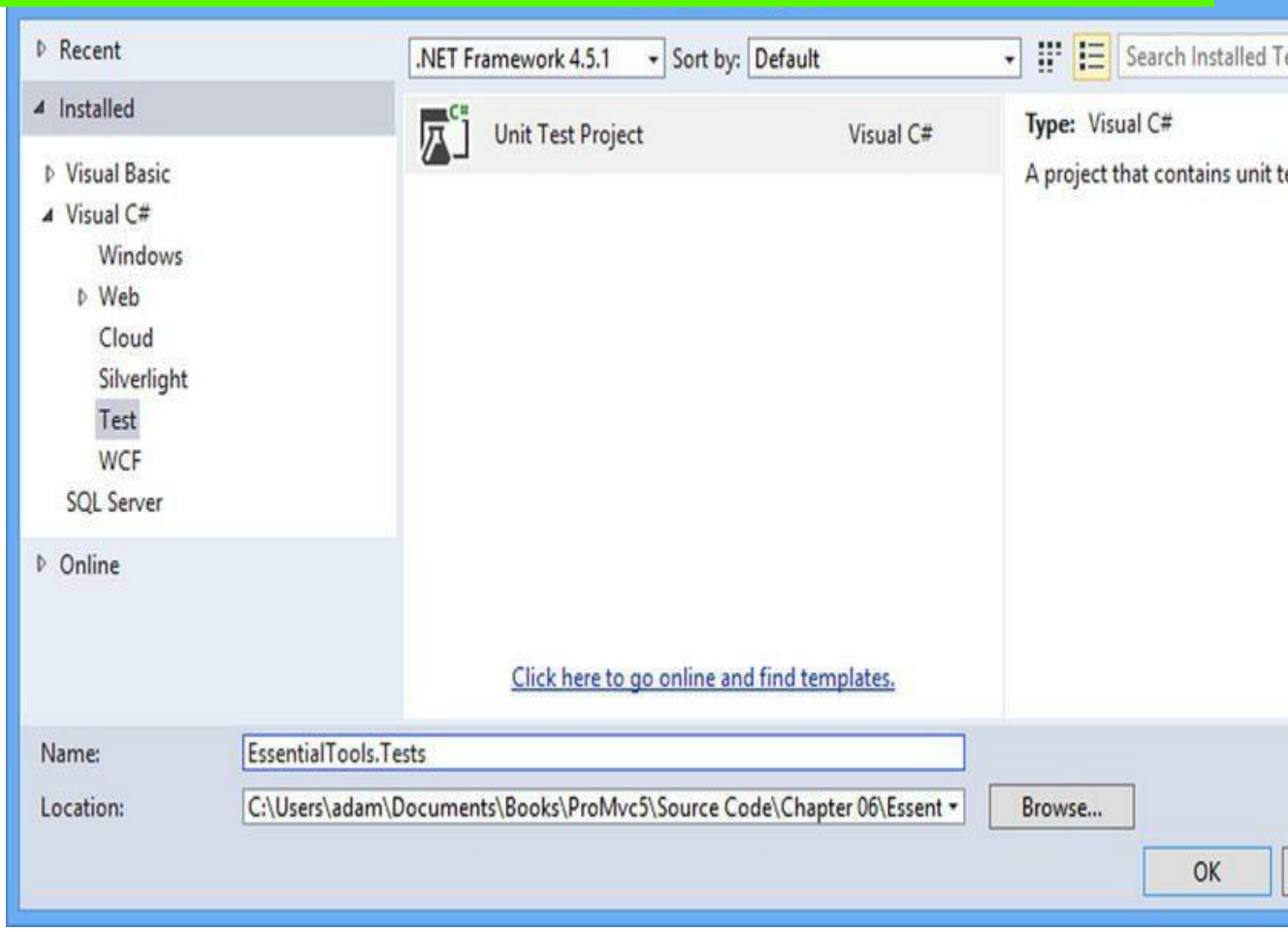


Figure 6-4. Creating the unit test project

Set the project name to `EssentialTools.Tests` and click the `OK` button to create the new project, wh
Studio will add to the current solution alongside the MVC application project.

I need to give the test project a reference to the application project so that it can access the classes and perform te
them. Right-click the `References` item for the `EssentialTools.Tests` project in the Solution Explo
select `Add Reference` from the pop-up menu. Click `Solution` in the left panel and check the box next to
`EssentialTools` item, as shown in Figure 6-5.

eBooks-IT.org

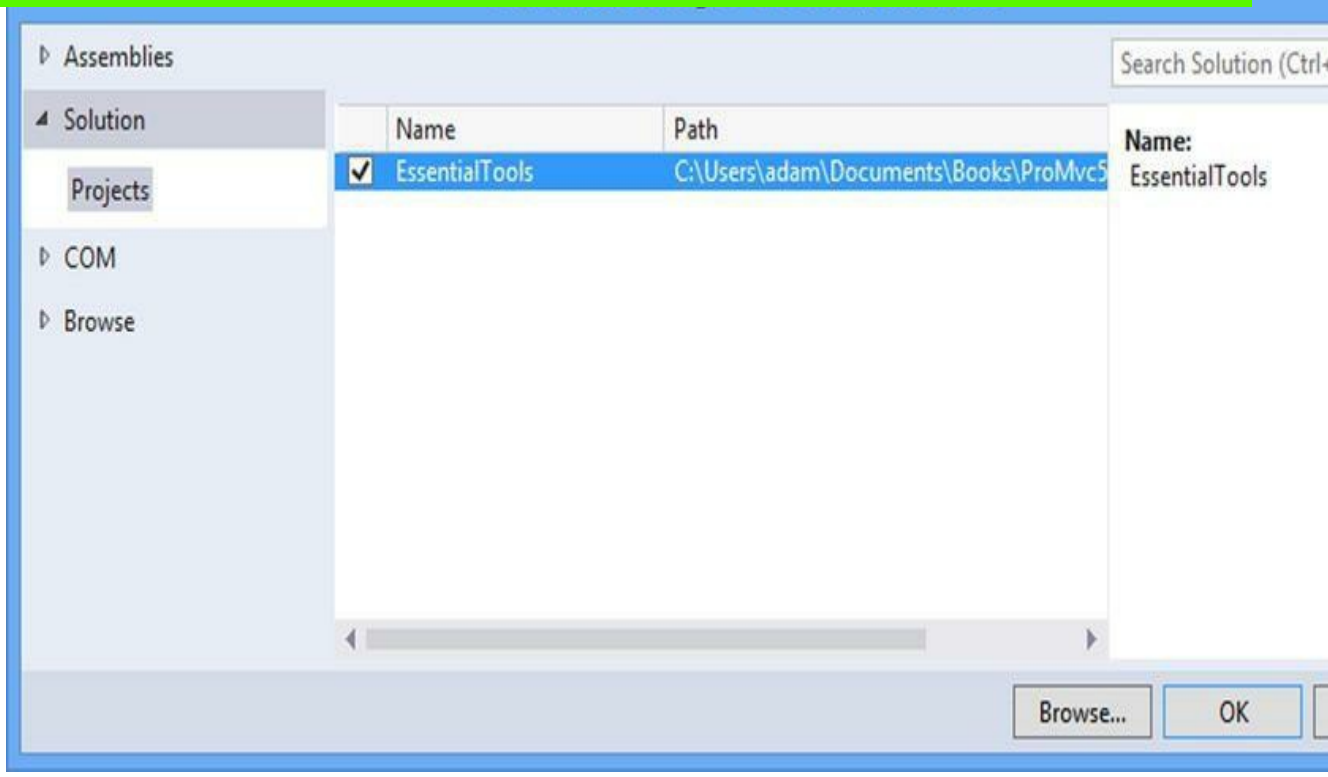


Figure 6-5. Adding a reference to the MVC project

Creating the Unit Tests

I will add my unit tests to the `UnitTest1.cs` file in the `EssentialTools.Tests` project. The paid-f Studio editions have some nice features for automatically generating test methods for a class that are not available in edition, but I can still create useful and meaningful tests. To get started, I made the changes shown in Listing 6-27.

Listing 6-27. Adding the Test Methods to the UnitTest1.cs File

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;

namespace EssentialTools.Tests {

    [TestClass]
    public class UnitTest1 {

        private IDiscountHelper getTestObject() {
            return new MinimumDiscountHelper();
        }

        [TestMethod]
        public void Discount_Above_100() {
            // arrange
            IDiscountHelper target = getTestObject();
            decimal total = 200;

            // act
            var discountedTotal = target.ApplyDiscount(total);
```

eBooks-IT.org

```
            // assert
            Assert.AreEqual(total * 0.9M, discountedTotal);
        }
    }
}
```

I have added a single unit test. A class that contains tests is annotated with the `TestClass` attribute and individu methods annotated with the `TestMethod` attribute. Not all methods in a unit test class have to be unit tests. To this, I have defined the `getTestObject` method, which I will use to arrange my tests. Because this method do `TestMethod` attribute, Visual Studio will not treat it as a unit test.

Tip Notice that I had to add a `using` statement to import the `EssentialTools.Models` namespace class. Test classes are just regular C# classes and have no special knowledge about the MVC project. It is the `Test` `TestMethod` attributes which add the testing magic to the project.

You can see that I have followed the arrange/act/assert (A/A/A) pattern in the unit test method that I described in There are countless conventions about how to name unit tests, but my advice is simply that you use names that mak the test is checking. My unit test method is called `Discount_Above_100`, which is clear and meaningful to that really matters is that you (and your team) understand whatever naming pattern you settle on, so you adopt a diff scheme if you do not like mine.

I set up the test method by calling the `getTestObject` method, which creates an instance of the object I an test: the `MinimumDiscountHelper` class in this case. I also define the `total` value with which I am goi This is the *arrange* section of the unit test.

For the *act* section of the test, I call the `MinimumDiscountHelper.ApplyDiscount` method and result to the `discountedTotal` variable. Finally, for the `assert` section of the test, I use the `Assert.` method to check that the value I got back from the `ApplyDiscount` method is 90% of the total that I started

The `Assert` class has a range of static methods that you can use in your tests. The class is in the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace along with some additional can be useful for setting up and performing tests. You can learn more about classes in the namespace at http://msdn.microsoft.com/en-us/library/ms182530.aspx.

The `Assert` class is the one that I use the most and I have summarized the most important methods in Table 6-

***Table 6-4**. Static Assert Methods*

Method	Description
`AreEqual<T>(T, T)` `AreEqual<T>(T, T, string)`	Asserts that two objects of type `T` have the same value.
`AreNotEqual<T>(T, T)` `AreNotEqual<T>(T, T, string)`	Asserts that two objects of type `T` do not have the same
`AreSame<T>(T, T)` `AreSame<T>(T, T, string)`	Asserts that two variables refer to the same object.
`AreNotSame<T>(T, T)` `AreNotSame<T>(T, T, string)`	Asserts that two variables refer to different objects.
`Fail()` `Fail(string)`	Fails an assertion: no conditions are checked.
`Inconclusive()` `Inconclusive(string)`	Indicates that the result of the unit test cannot be definitive established.
`IsTrue(bool)` `IsTrue(bool, string)`	Asserts that a `bool` value is true. Most often used to ev expression that returns a `bool` result.
`IsFalse(bool)` `IsFalse(bool, string)`	Asserts that a `bool` value is false.
`IsNull(object)` `IsNull(object, string)`	Asserts that a variable is not assigned an object reference.
`IsNotNull(object)` `IsNotNull(object, string)`	Asserts that a variable is assigned an object reference.
`IsInstanceOfType(object, Type)` `IsInstanceOfType(object, Type, string)`	Asserts that an object is of the specified type or is derived f specified type.
`IsNotInstanceOfType(object, Type)` `IsNotInstanceOfType(object, Type, string)`	Asserts that an object is not of the specified type.

eBooks-IT.org

e meth

exception if the check fails. All of the assertions have to succeed for the unit test to pass.

Each of the methods in the table has an overloaded version that takes a `string` parameter. The `string` is in message element of the exception if the assertion fails. The `AreEqual` and `AreNotEqual` methods have a n overloads that cater to comparing specific types. For example, there is a version that allows strings to be compared v case into account.

Tip One noteworthy member of the `Microsoft.VisualStudio.TestTools.UnitTestin` is the `ExpectedException` attribute. This is an assertion that succeeds only if the unit test throws an excep type specified by the `ExceptionType` parameter. This is a neat way of ensuring that exceptions are thrown w to mess around with `try...catch` blocks in your unit test.

Now that I have shown you how to put together one unit test, I have added further tests to the test project to valid behaviors I want for my `MinimumDiscountHelper` class. You can see the additions in Listing 6-28, but th are so short and simple (which is generally a characteristic of unit tests) that I am not going to explain them in detail.

Listing 6-28. Defining the Remaining Tests in the UnitTest1.cs File

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;

namespace EssentialTools.Tests {

    [TestClass]
    public class UnitTest1 {

        private IDiscountHelper getTestObject() {
            return new MinimumDiscountHelper();
        }

        [TestMethod]
        public void Discount_Above_100() {
            // arrange
            IDiscountHelper target = getTestObject();
            decimal total = 200;

            // act
            var discountedTotal = target.ApplyDiscount(total);

            // assert
            Assert.AreEqual(total * 0.9M, discountedTotal);
        }

        [TestMethod]
        public void Discount_Between_10_And_100() {
            //arrange
            IDiscountHelper target = getTestObject();

            // act
            decimal TenDollarDiscount = target.ApplyDiscount(10);
            decimal HundredDollarDiscount = target.ApplyDiscount(1
            decimal FiftyDollarDiscount = target.ApplyDiscount(50)

            // assert
```

eBooks-IT.org

```
wrong");
            Assert.AreEqual(95, HundredDollarDiscount, "$100 dis
wrong");
              Assert.AreEqual(45, FiftyDollarDiscount, "$50 dis
wrong");
        }

        [TestMethod]
        public void Discount_Less_Than_10() {
            //arrange
            IDiscountHelper target = getTestObject();

            // act
            decimal discount5 = target.ApplyDiscount(5);
            decimal discount0 = target.ApplyDiscount(0);

            // assert
            Assert.AreEqual(5, discount5);
            Assert.AreEqual(0, discount0);

        }
        [TestMethod]
        [ExpectedException(typeof(ArgumentOutOfRangeException))]
        public void Discount_Negative_Total() {
            //arrange
            IDiscountHelper target = getTestObject();

            // act
            target.ApplyDiscount(-1);
        }
    }
}
```

Running the Unit Tests (and Failing)

Visual Studio provides the `Test Explorer` window for managing and running tests. Select `Windows` ➤ `T` `Explorer` from the Visual Studio `Test` menu to see the window and click the `Run All` button near the top. You will see results similar to the ones shown in Figure 6-6.

eBooks-IT.org

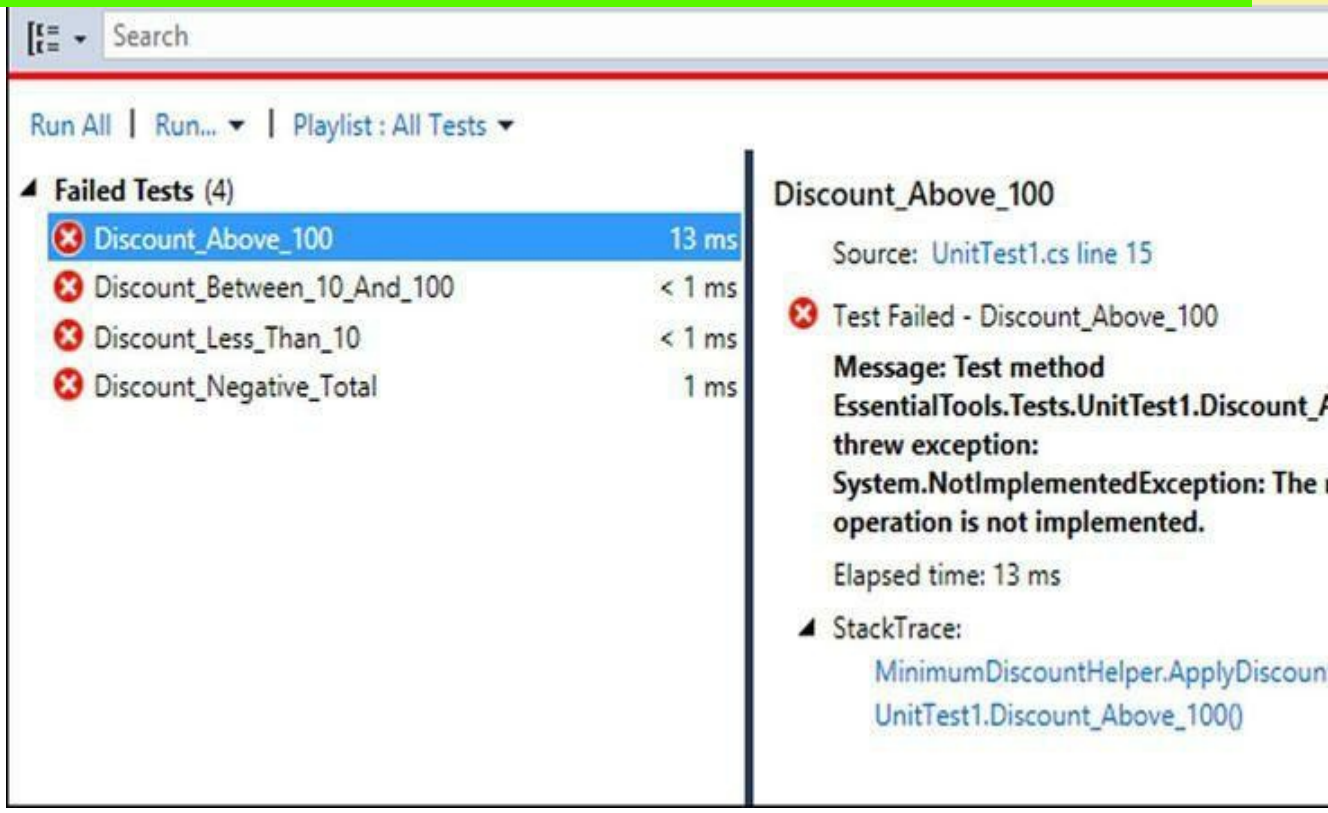


Figure 6-6. Running the tests in the project

You can see the list of tests I defined in the left-hand panel of the `Test Explorer` window. All of the tests course, because I have yet to implement the method I am testing. You can click any of the tests in the window to se why it has failed. The `Test Explorer` window provides a range of different ways to select and filter unit test choose which tests to run. For my simple example project, however, I will just run all of the tests by clicking `Run`

Implementing the Feature

I have reached the point where I can implement the feature, safe in the knowledge that I will be able to check that th as expected when I am finished. For all of my preparation, the implementation of the `MinimumDiscountHel` simple, as shown by Listing 6-29.

Listing 6-29. The Contents of the MinimumDiscountHelper.cs File

```
using System;

namespace EssentialTools.Models {
    public class MinimumDiscountHelper : IDiscountHelper {

        public decimal ApplyDiscount(decimal totalParam) {
            if (totalParam < 0) {
                throw new ArgumentOutOfRangeException();
            } else if (totalParam > 100) {
                return totalParam * 0.9M;
            } else if (totalParam > 10 && totalParam <= 100) {
                return totalParam -5;
            } else {
                return totalParam;
            }
```

eBooks-IT.org

```
    }
}
```

Testing and Fixing the Code

I have left a deliberate error in the code to demonstrate how iterative unit testing with Visual Studio works and you c effect of the error if you click the `Run All` button in the `Test Explorer` window. You can see the test re 6-7.

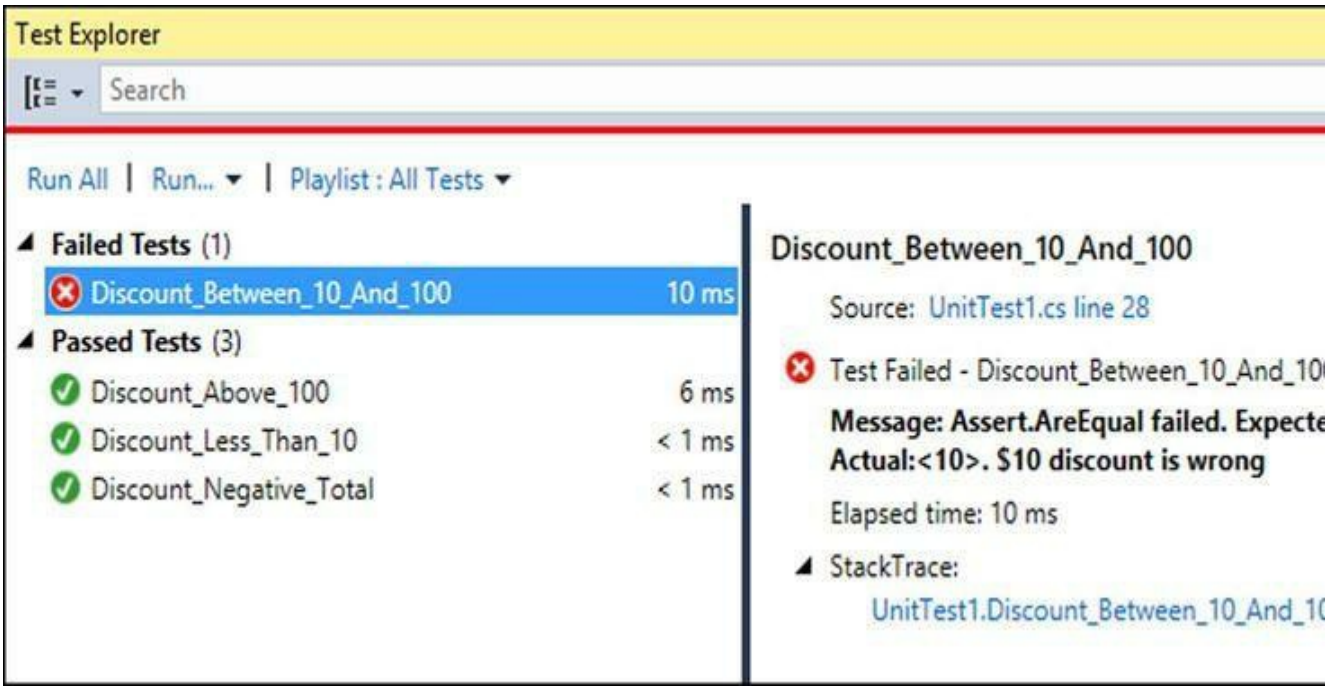


Figure 6-7. The effect of implementing the feature with a bug

Visual Studio always tries to promote the most useful information to the top of the `Test Explorer` window situation, this means that it displays failed tests before passed tests.

You can see that I passed three of the unit tests, but I have a problem that the `Discount_Between_10_` test method detected. When I click the failed test, I can see that my test expected a result of `5`, but actually got a val

At this point, I return to my code and see that I have not implemented my expected behaviors properly. Specificall handle the discounts for totals that are `10` or `100` properly. The problem is in this statement from the `MinumumDiscountHelper` class:

```
...
} else if (totalParam > 10 && totalParam < 100) {
...
```

The specification that I am working to implement sets out the behavior for values which are between $10 and $10 but my implementation excludes those values and only checks for values which are greater than $10, excluding totals exactly $10. The solution is simple and is shown in Listing 6-30. Only a single character needs to be added to change the `if` statement.

Listing 6-30. Fixing the Feature Code in the MinimumDiscountHelper.cs File

```
using System;

namespace EssentialTools.Models {
    public class MinimumDiscountHelper : IDiscountHelper {
```

eBooks-IT.org

```
            if (totalParam < 0) {
                throw new ArgumentOutOfRangeException();
            } else if (totalParam > 100) {
                return totalParam * 0.9M;
            } else if (totalParam >= 10 && totalParam <= 100) {
                return totalParam -5;
            } else {
                return totalParam;
            }
        }
    }
}
```

When I click the `Run All` button in the `Test Explorer` window, the results show that I have fixed the that my code passed all of the tests (see Figure 6-8).

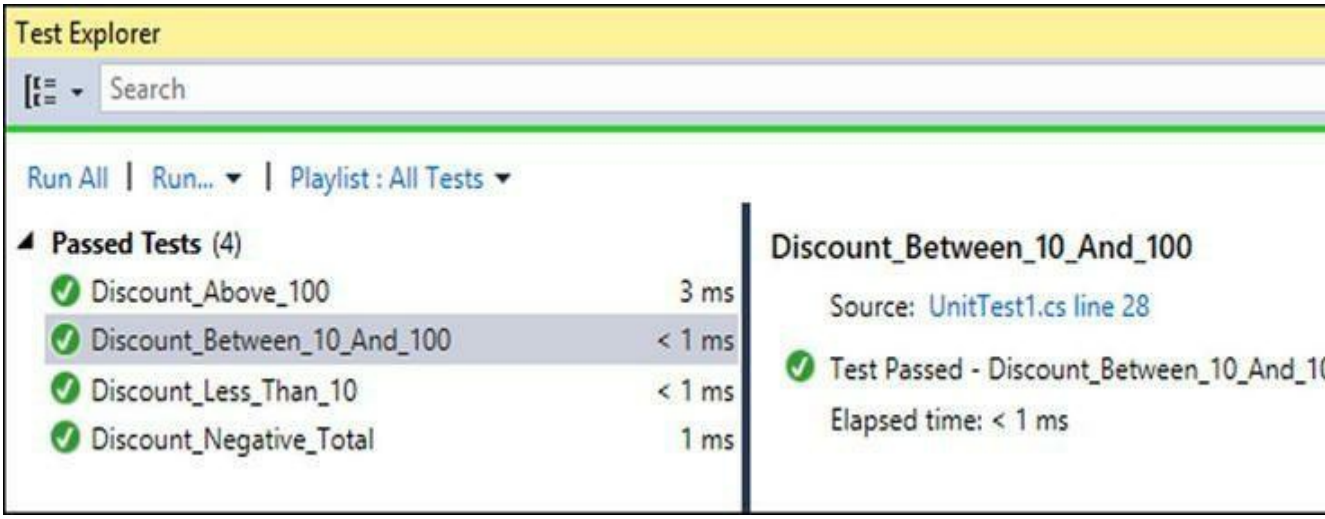


Figure 6-8. Passing all of the unit tests

This is just a quick introduction to unit testing, and I will further demonstrate unit tests in later chapters as well. T support in Visual Studio is pretty good and I recommend you explore the unit testing documentation on MSDN, whic find at `http://msdn.microsoft.com/en-us/library/dd264975.aspx`.

Using Moq

One of the reasons that I am able to keep my tests so simple in the previous section was because I am testing a singl depends on no other class to function. Such objects exist in real projects, of course, but you will also need to test ob cannot function in isolation. In these situations, you need to be able to focus on the class or method you are intereste you are not implicitly testing the dependencies as well.

One useful approach is to use *mock objects*, which simulate the functionality of real objects from your project, but and controlled way. Mock objects let you narrow the focus of your tests so that you only examine the functionality i are interested.

The paid-for versions of Visual Studio include support for creating mock objects through a feature called *fakes*, bu use a library called Moq, which is simple, easy-to-use and you can use with all Visual Studio editions, including the f

Understanding the Problem

Before I get into the using Moq, I want to demonstrate the problem that I am trying to fix. In this section, I am going the `LinqValueCalculator` class, which I defined in the `Models` folder of the example project. As a ren 6-31 shows the definition of the `LinqValueCalculator` class.

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

namespace EssentialTools.Models {

    public class LinqValueCalculator : IValueCalculator {
        private IDiscountHelper discounter;
        private static int counter = 0;

        public LinqValueCalculator(IDiscountHelper discountParam)
            discounter = discountParam;
            System.Diagnostics.Debug.WriteLine(
                 string.Format("Instance {0} created", ++counter))
        }

        public decimal ValueProducts(IEnumerable<Product> products
            return discounter.ApplyDiscount(products.Sum(p => p.Pr
        }
    }
}
```

To test this class, I added a new unit test class to the test project. You do this by right-clicking the test project in t Explorer and selecting Add ➤ Unit Test from the pop-up menu. If your Add menu does not have a Unit select New Item instead and use the Basic Unit Test template. You can see the changes I made to the which Visual Studio names UnitTest2.cs by default, in Listing 6-32.

Listing 6-32. Adding a Unit Test for the ShoppingCart Class in the UnitTest2.cs File

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;
using System.Linq;

namespace EssentialTools.Tests {
    [TestClass]
    public class UnitTest2 {

        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
              new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
             new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
             new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
        };

        [TestMethod]
        public void Sum_Products_Correctly() {

            // arrange
            var discounter = new MinimumDiscountHelper();
            var target = new LinqValueCalculator(discounter);
```

```
            // act
            var result = target.ValueProducts(products);

            // assert
            Assert.AreEqual(goalTotal, result);
        }
    }
}
```

The problem I face is that the `LinqValueCalculator` class depends on an implementation of the `IDiscountHelper` interface to operate. In the example, I used the `MinimumDiscountHelper` class presents two different issues.

First, I have made my unit test complex and brittle. In order to create a unit test that works, I need to take into acc discount logic in the `IDiscountHelper` implementation to figure out the expected value from the `ValueP` method. The brittleness comes from the fact that my tests will fail if the discount logic in the implementation change though the `LinqValueCalculator` class may well be working properly.

Second, and most troubling, I have extended the scope of my unit test so that it implicitly includes the `MinimumDiscountHelper` class. When my unit test fails, I will not know if the problem is in the `LinqValueCalculator` or `MinimumDiscountHelper` class.

The best unit tests are simple and focused, and my current setup does not allow for either of these characteristics. sections that follow, I show you how to add and apply Moq in your MVC project so that you can avoid these problem

Adding Moq to the Visual Studio Project

Just like with Ninject earlier in the chapter, the easiest way to add Moq to an MVC project is to use the integrated Vis support for *NuGet*. Open the NuGet console and enter the following command:

```
Install-Package Moq -version 4.1.1309.1617 -pro
EssentialTools.Tests
```

The `projectname` argument allows me to tell NuGet that I want the Moq package installed in my unit test pr than in the main application.

Adding a Mock Object to a Unit Test

Adding a mock object to a unit test means telling Moq what kind of object you want to work with, configuring its be then applying the object to the test target. You can see how I added a mock object to my unit test for the `LinqValueCalculator` in Listing 6-33.

Listing 6-33. Using a Mock Object in a Unit Test in the UnitTest2.cs File

```
using EssentialTools.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;
using Moq;

namespace EssentialTools.Tests {
    [TestClass]
    public class UnitTest2 {

        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
              new Product {Name = "Lifejacket", Category = "Water
```

eBooks-IT.org

```
            new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
        };

        [TestMethod]
        public void Sum_Products_Correctly() {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            var target = new LinqValueCalculator(mock.Object);

            // act
            var result = target.ValueProducts(products);

            // assert
            Assert.AreEqual(products.Sum(e => e.Price), result);
        }
    }
}
```

The syntax for using Moq is a little odd when you first see it, so I will walk through each stage of the process.

Tip Bear in mind that there are a number of different mocking libraries available, so the chances are good that yo alternative to suit you if you do not like the way that Moq works, although Moq is actually an easy library to use. Yo some of the other popular libraries to have manuals hundreds of pages long.

Creating a Mock Object

The first step is to tell Moq what kind of mock object you want to work with. Moq relies heavily on type parameters see this in the way that I tell Moq I want to create a mock `IDiscountHelper` implementation:

```
...
Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
...
```

I create a strongly typed `Mock<IDiscountHelper>` object, which tells the `Moq` library the type it will b This is the `IDiscountHelper` interface for my unit test, but it can be any type that you want to isolate to imp focus of your unit tests.

Selecting a Method

In addition to creating the strongly typed `Mock` object, I also need to specify the way that it behaves. This is at the mocking process and it allows you to ensure that you establish a baseline behavior in the mock object, which you ca the functionality of your target object in the unit test. Here is the statement from the unit test that sets up the behavio

```
...
mock.Setup (m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<
(total => total);
...
```

I use the `Setup` method to add a method to my mock object. Moq works using LINQ and lambda expressions.

eBooks-IT.org

that I am not going to get into here. This allows me to select the method I want to configure by using a lambda expre
my unit test, I want to define the behavior of the `ApplyDiscount` method, which is the only method in the `IDiscountHelper` interface, and the method I need to test the `LinqValueCalculator` class.

I also have to tell Moq what parameter values I am interested in, which I do using the `It` class, which I have high
follows:

```
...
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<
(total => total);
...
```

The `It` class defines a number of methods that are used with generic type parameters. In this case, I have called
method using `decimal` as the generic type. This tells Moq to apply the behavior I am defining whenever I call the
`ApplyDiscount` method any decimal value. Table 6-5 shows the methods that the `It` class provides, all of w
static.

***Table 6-5**. The Methods of the It Class*

Method	Description
`Is<T>(predicate)`	Specifies values of type `T` for which the predicate will return `true`. See Listing 6-34 for an example).
`IsAny<T>()`	Specifies any value of the type `T`.
`IsInRange<T>(min, max, kind)`	Matches if the parameter is between the defined values and of type `T`. The final parameter is a value from t enumeration and can be either `Inclusive` or `Exclusive`.
`IsRegex(expr)`	Matches a string parameter if it matches the specified regular expression.

I will show you a more complex example later that uses some other `It` methods, but for the moment I will stick
`IsAny<decimal>` method which allows me to respond to any decimal value.

Defining the Result

The `Returns` method allows me to specify the result that Moq will return when I call my mocked method. I spec
of the result using a type parameter and specify the result itself using a lambda expression. You can see how I have c
the example:

```
...
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<
(total => total) ;
...
```

By calling `Returns` method with a `decimal` type parameter (i.e., `Returns<decimal>`), I tell Moq t
going to return a `decimal` value. For the lambda expression, Moq passes me a value of the type I receive in the
`ApplyDiscount` method. I create a *pass-through* method in the example, in which I return the value that is pas
mock `ApplyDiscount` method without performing any operations on it. This is the simplest kind of mock me
will show you more sophisticated examples shortly.

Using the Mock Object

The last step is to use the mock object in the unit test, which I do by reading the value of the `Object` property of
`Mock<IDiscountHelper>` object:

```
...
var target = new LinqValueCalculator(mock.Object);
...
```

eBooks-IT.org

the ApplyDiscount method returns the value of the decimal parameter it is passed.

This makes it easy to perform my unit test because I can sum the prices of my test Product objects myself an I get the same value back from the LinqValueCalculator object:

```
...
Assert.AreEqual(products.Sum(e => e.Price), result);
...
```

The benefit of using Moq in this way is that my unit test only checks the behavior of the LinqValueCalcu object and does not depend on any of the real implementations of the IDiscountHelper interface in the Moc This means that when my tests fail, I know that the problem is either in the LinqValueCalculator implen the way I set up mock object, and solving a problem from either of these sources is simpler and easier than dealing v real objects and the interactions between them.

Creating a More Complex Mock Object

I showed you a simple mock object in the last section, but part of the beauty of Moq is the ability to build up comple test different situations. In Listing 6-34, I added a new unit test to the UnitTest2.cs file that mocks a more c implementation of the IDiscountHelper interface. In fact, I used Moq to model the behavior of the MinimumDiscountHelper class.

Listing 6-34. Mocking the Behavior of the MinimumDiscountHelper Class in the UnitTest2.cs File

```
using EssentialTools.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Linq;

namespace EssentialTools.Tests {
    [TestClass]
    public class UnitTest2 {

        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports",
275M},
              new Product {Name = "Lifejacket", Category = "Water
Price = 48.95M},
             new Product {Name = "Soccer ball", Category = "Soccer
= 19.50M},
             new Product {Name = "Corner flag", Category = "Soccer
= 34.95M}
        };

        [TestMethod]
        public void Sum_Products_Correctly() {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            var target = new LinqValueCalculator(mock.Object);

            // act
            var result = target.ValueProducts(products);

            // assert
```

eBooks-IT.org

```
        }

        private Product[] createProduct(decimal value) {
            return new[] { new Product { Price = value } };
        }

        [TestMethod]
        [ExpectedException(typeof(System.ArgumentOutOfRangeExcepti
        public void Pass_Through_Variable_Discounts() {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v
                .Throws<System.ArgumentOutOfRangeException>();
            mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v
                .Returns<decimal>(total => (total * 0.9M));
            mock.Setup(m => m.ApplyDiscount(It.IsInRange<decimal>(
                Range.Inclusive))).Returns<decimal>(total => total
            var target = new LinqValueCalculator(mock.Object);

            // act
                                                decimal    FiveDollarDisco
target.ValueProducts(createProduct(5));
                                                 decimal    TenDollarDisco
target.ValueProducts(createProduct(10));
                                               decimal    FiftyDollarDisco
target.ValueProducts(createProduct(50));
                                             decimal    HundredDollarDisco
target.ValueProducts(createProduct(100));
                                           decimal    FiveHundredDollarDisc
target.ValueProducts(createProduct(500));

            // assert
            Assert.AreEqual(5, FiveDollarDiscount, "$5 Fail");
            Assert.AreEqual(5, TenDollarDiscount, "$10 Fail");
            Assert.AreEqual(45, FiftyDollarDiscount, "$50 Fail");
            Assert.AreEqual(95, HundredDollarDiscount, "$100 Fail"
            Assert.AreEqual(450, FiveHundredDollarDiscount, "$500
            target.ValueProducts(createProduct(0));
        }
    }
}
```

In unit test terms, replicating the expected behavior of one of the other model classes would be an odd thing to do, perfect demonstration of some of the different Moq features.

I have defined four different behaviors for the `ApplyDiscount` method based on the value of the parameter The simplest is the catch-all, which returns a value for any `decimal` value, like this:

```
...
mock.Setup(m    =>    m.ApplyDiscount(It.IsAny<decimal>())).Returns<
(total => total);
...
```

eBooks-IT.org

Setup method affects the behavior of the mock object. Moq evaluates the behaviors in reverse order, so that it co ich you most recent calls to the Setup method first. This means that you have to take care to create your mock behaviors the most general to the most specific. The It.IsAny<decimal> condition is the most general condition I de example and so I apply it first. If I reversed the order of my Setup calls, this behavior would capture all of the cal the ApplyDiscount method and generate the wrong mock results.

Mocking For Specific Values (and Throwing an Exception)

For the second call to the Setup method, I have used the It.Is method:

```
...
mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v == 0)))
    .Throws<System.ArgumentOutOfRangeException>();
...
```

The predicate I have passed to the Is method returns true if the value passed to the ApplyDiscount m Rather than return a result, I used the Throws method, which causes Moq to throw a new instance of the excepti with the type parameter.

I also use the Is method to capture values that are greater than 100, like this:

```
...
mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v > 100)))
    .Returns<decimal>(total => (total * 0.9M));
...
```

The It.Is method is the most flexible way of setting up specific behaviors for different parameter values becau use any predicate that returns true or false. This is the method I use most often when creating complex mock

Mocking For a Range of Values

My final use of the It object is with the IsInRange method, which allows me to capture a range of parameter

```
...
mock.Setup(m => m.ApplyDiscount(It.IsInRange<decimal>(10,
Range.Inclusive)))
    .Returns<decimal>(total => total - 5);
...
```

I have included this for completeness, but in my own projects I tend to use the Is method and a predicate that do thing, like this:

```
...
mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v >= 10 && v <
    .Returns<decimal>(total => total - 5);
...
```

The effect is the same, but I find the predicate approach more flexible. Moq has a range of extremely useful featur can see how to apply them by reading the quick start provided at http://code.google.com/p/moq/wiki/QuickStart.

Summary

In this chapter, I looked at the three tools I find essential for effective MVC development: Ninject, the built-in Visual support for unit testing, and Moq. There are many alternatives, both open source and commercial, for all three tools

eBooks-IT.org

You may find that you do not like TDD or unit testing in general, or that you are happy performing DI and mockin That, of course, is entirely your choice. However, I think there are some substantial benefits in using all three tools i development cycle. If you are hesitant to adopt them because you have never tried them, I encourage you to suspen give them a go, at least for the duration of this book.

eBooks-IT.org

SportsStore: A Real Application

In the previous chapters, I built quick and simple MVC applications. I described the MVC pattern, the essential C# fe kinds of tools that good MVC developers require. Now it is time to put everything together and build a simple but rea commerce application.

My application, called *SportsStore*, will follow the classic approach taken by online stores everywhere. I will create product catalog that customers can browse by category and page, a shopping cart where users can add and remove a checkout where customers can enter their shipping details. I will also create an administration area that includes cre update, and delete (CRUD) facilities for managing the catalog; and I will protect it so that only logged-in administrato changes.

My goal in this chapter and those that follow is to give you a sense of what real MVC Framework development is creating as realistic an example as possible. I want to focus on the MVC Framework, of course, and so I have simpl integration with external systems, such as the database, and omitted others entirely, such as payment processing.

You might find the going a little slow as I build up the levels of infrastructure I need. Certainly, you *would* get the functionality built more quickly with Web Forms, just by dragging and dropping controls bound directly to a databas initial investment in an MVC application pays dividends, resulting in maintainable, extensible, well-structured code wi support for unit testing.

UNIT TESTING

I have made quite a big deal about the ease of unit testing in MVC, and about my belief that unit testing is an imp the development process. You will see this demonstrated throughout this part of the book because I have includ unit tests and techniques as they relate to key MVC features.

I know this is not a universal opinion. If you do not want to unit test, that is fine with me. To that end, when I h something to say that is purely about testing, I put it in a sidebar like this one. If you are not interested in unit te skip right over these sections, and the SportsStore application will work just fine. You do not need to do any kin testing to get the technology benefits of ASP.NET MVC, although, of course, support for testing is a key reason the MVC Framework.

Most of the MVC features I use for the SportsStore application have their own chapters later in the book. Rather t everything here, I tell you just enough to make sense for the example application and point you to the other chapter f information.

I will call out each step needed to build the application, so that you can see how the MVC features fit together. Yo particular attention when I create views. You will get some odd results if you do not follow the examples closely.

Getting Started

You will need to install Visual Studio if you are planning to code the SportsStore application on your own computer a through this part of the book. You can also download the SportsStore project as part of the code archive that accom book (available from Apress.com). You do not need to follow along, of course. I have tried to make the screen code listings as easy to follow as possible, just in case you are reading this book on a train, in a coffee shop, or the li

Creating the Visual Studio Solution and Projects

I am going to create a Visual Studio solution that contains three projects. One project will contain the domain model, the MVC application, and the third will contain the unit tests. To get started, I created a new Visual Studio solution ca

eBooks-IT.org

Types/Visual Studio Solutions section of the New Project dialog, as shown in Figure 7-1 OK button to create the solution.

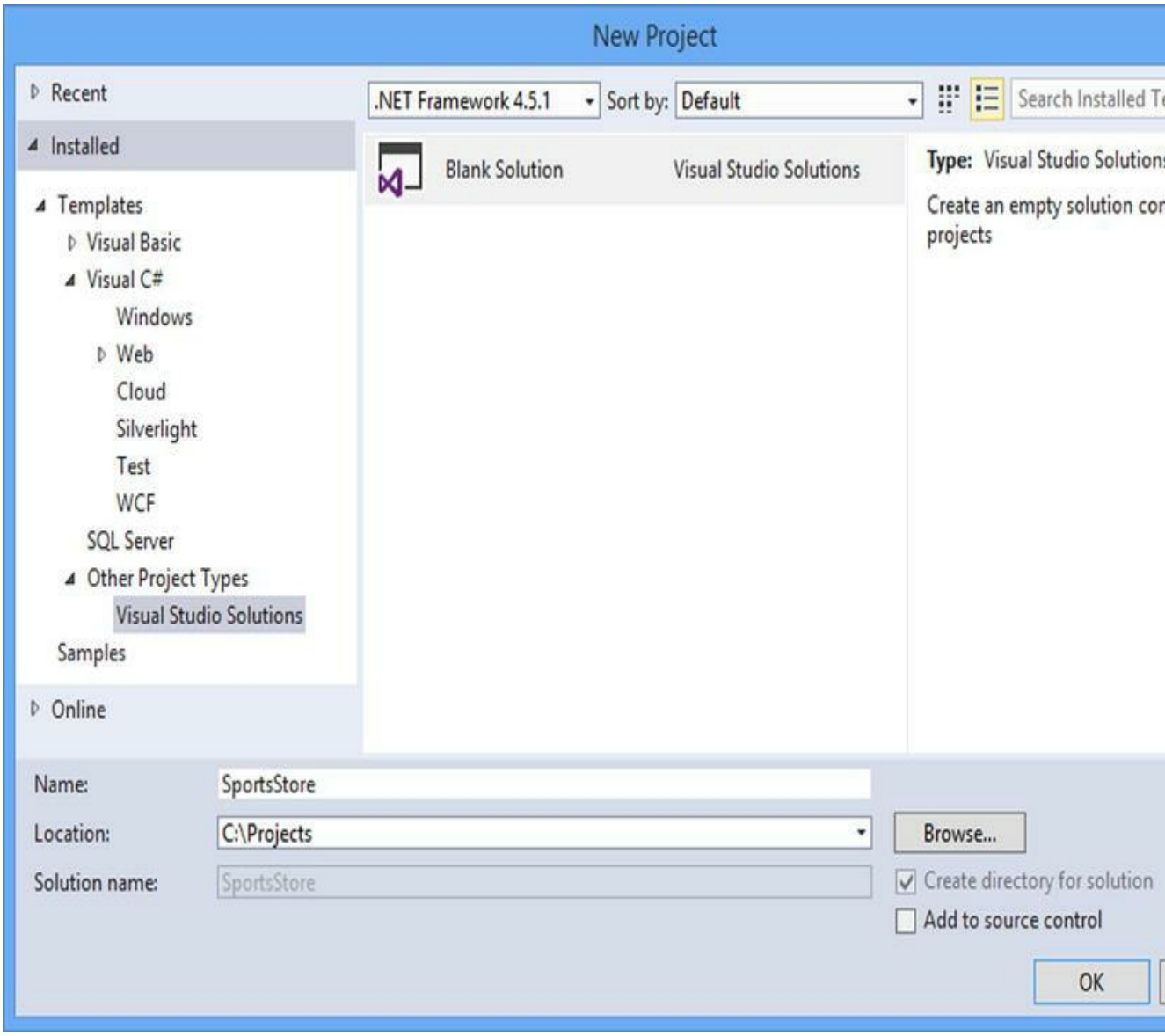


Figure 7-1. Creating a new Visual Studio solution

A Visual Studio solution is a container for one or more projects. I require three projects for my example app, whic described in Table 7-1. You add a project by right-clicking the `Solution` entry in the Solution Explorer and selec `New Project` from the pop-up menus.

***Table 7-1**. The Three SportsStore Projects*

Project Name	Visual Studio Project Template	Purpose
`SportsStore.Domain`	Class Library	Holds the domain entities and logic; set u via a repository created with the Entity F
`SportsStore.WebUI`	ASP.NET MVC Web Application (choose `Empty` when prompted to choose a project template and check the `MVC` option)	Holds the controllers and views; acts as th SportsStore application.
`SportsStore.UnitTests`	Unit Test Project	Holds the unit tests for the other two pro

I always use the `Empty` option for the `ASP.NET MVC Web Application` template. The other optio initial setup to the project that includes JavaScript libraries, CSS style sheets, and C# classes that configure applicatio security and routing. None of this is inherently bad–and some of the open-source libraries that Microsoft has recently

eBooks-IT.org

more about the workings of the MVC Framework.

When you have created the three projects, the Solution Explorer should look like Figure 7-2. I have deleted the `Cl` file that Visual Studio adds to the `SportsStore.Domain` project. I will not be using it.

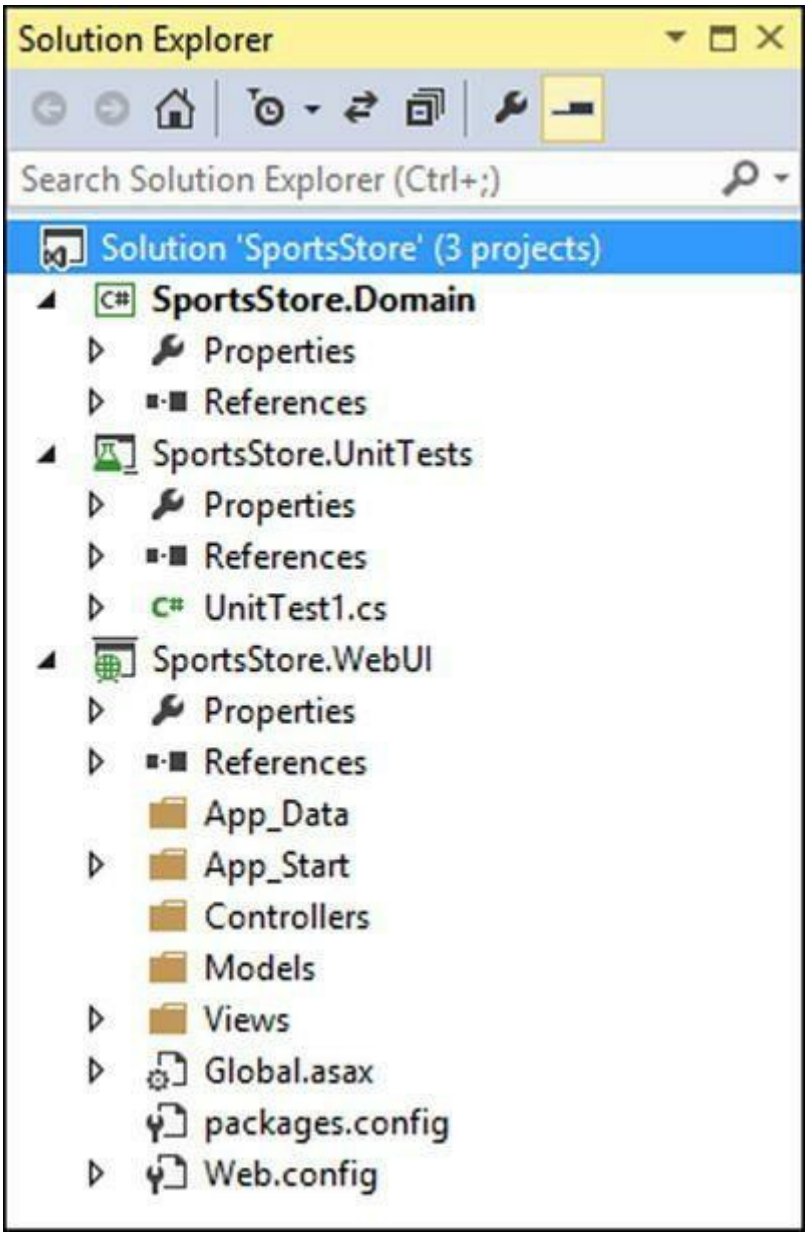


Figure 7-2. The projects shown in the Solution Explorer window

To make debugging easier, right-click the `SportsStore.WebUI` project and select `Set as Startu` `Project` from the pop-up menu (you will see the name turn bold). This means that when you select `Start D` or `Start without Debugging` from the `Debug` menu, it is this project that will be started.

Visual Studio will try to navigate to individual view files if you are editing them when you start the debugger, so rig `SportsStore.WebUI` project in the Solution Explorer and select `Properties` from the pop-up menu. C to open the web-related properties and select the `Specific Page` option. There is no need to enter a value into `Specific Page` text field. Just selecting the option is enough to stop Visual Studio from trying to guess the U to view and ensure that the browser requests the root URL for the application when you start the debugger.

Installing the Tool Packages

I will be using Ninject and Moq in this chapter. Select `Tools` ➤ `Library Package Manger` ➤ `Pack` `Manager Console` in Visual Studio to open the NuGet command line and enter the following commands:

eBooks-IT.org

```
Install-Package Ninject.Web.Common -Version 3.0.0.7 -pro
SportsStore.WebUI
Install-Package Ninject.MVC3 -Version 3.0.0.6 -pro
SportsStore.WebUI
Install-Package Ninject -version 3.0.1.10 -pro
SportsStore.UnitTests
Install-Package Ninject.Web.Common -version 3.0.0.7 -pro
SportsStore.UnitTests
Install-Package Ninject.MVC3 -Version 3.0.0.6 -pro
SportsStore.UnitTests
Install-Package Moq -version 4.1.1309.1617 -projectname SportsStor
Install-Package Moq -version 4.1.1309.1617 -pro
SportsStore.UnitTests
Install-Package Microsoft.Aspnet.Mvc -version 5.0.0 -pro
SportsStore.Domain
Install-Package Microsoft.Aspnet.Mvc -version 5.0.0 -pro
SportsStore.UnitTests
```

There are many NuGet commands to enter because I am being selective about which packages NuGet installs into projects and, as in previous chapters, I am specifying particular versions of the packages to download and install.

Adding References Between Projects

I need to set up dependencies between projects and to some of the Microsoft assemblies. Right-click each project in `Solution Explorer` window, select `Add Reference`, and add the references shown in Table 7-2 fr `Assemblies` ➤ `Framework`, `Assemblies` ➤ `Extensions` or `Solution` sections.

***Table 7-2**. Required Project Dependencies*

Project Name	Solution Dependencies	Assemblies References
SportsStore.Domain	None	System.ComponentModel.DataAnnotations
SportsStore.WebUI	SportsStore.Domain	None
SportsStore.UnitTests	SportsStore.Domain SportsStore.WebUI	System.Web Microsoft.CSharp

Caution Take the time to set these relationships up properly. If you do not have the right libraries and project ref will get into trouble when trying to build the project.

Setting Up the DI Container

In Chapter 6, I showed you how to use Ninject to create a custom dependency resolver that the MVC Framework w instantiate objects across the application. I am going to repeat that process, starting with adding an `Infrastruc` to the `SportsStore.WebUI` project and adding a class file called `NinjectDependencyResolve` You can see the contents of the new file in Listing 7-1.

***Listing 7-1**.* The Contents of the NinjectDependencyResolver.cs File

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

namespace SportsStore.WebUI.Infrastructure {
```

eBooks-IT.org

```
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            // put bindings here
        }
    }
}
```

As you may recall from Chapter 6, the next step is to create a bridge between the `NinjectDependencyR` class and the MVC support for dependency injection in the `App_Start/NinjectWebCommon.cs` file, w the Ninject NuGet packages added to the project, as shown in Listing 7-2.

Listing 7-2. Integrating Ninject in the NinjectWebCommon.cs File

```
...
private static void RegisterServices(IKernel kernel) {
    System.Web.Mvc.DependencyResolver.SetResolver(new
        SportsStore.WebUI.Infrastructure.NinjectDependencyResolver
}
...
```

Running the Application

If you select `Start Debugging` from the `Debug` menu, you will see an error page as shown in Figure 7-3. because you have requested a URL associated with a non-existent controller.

eBooks-IT.org

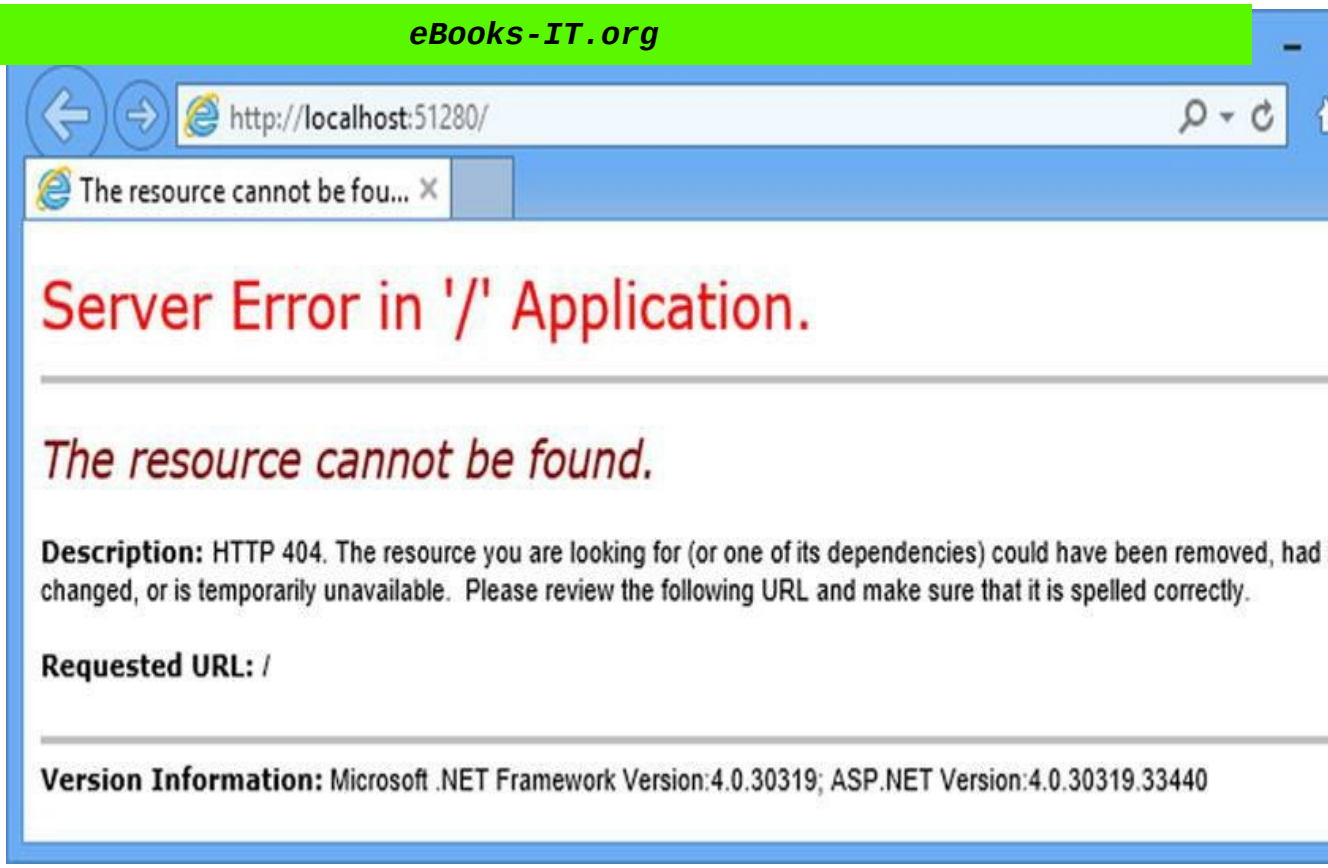
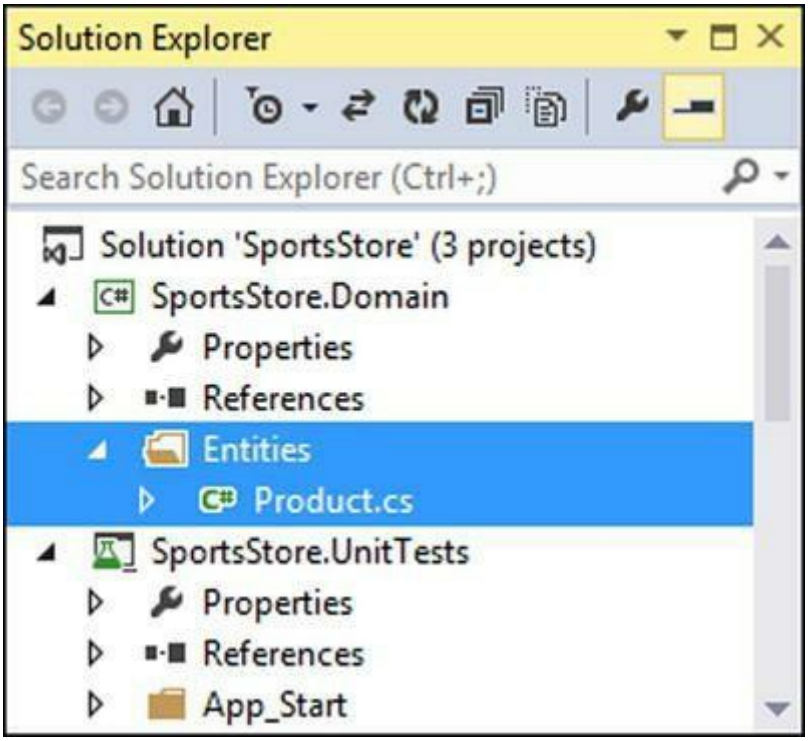


Figure 7-3. The error page

Starting the Domain Model

All MVC Framework projects start with the domain model because everything in an MVC Framework application rev
it. Since this is an e-commerce application, the most obvious domain entity I need is a product. Create a new folder c
`Entities` inside the `SportsStore.Domain` project and then a new C# class file called `Product.cs`
You can see the structure in Figure 7-4.



eBooks-IT.org

You are already familiar with the definition of the Product class, as I am going to use the one you saw in the p chapters. Edit the Product.cs class file so that it matches Listing 7-3.

Listing 7-3. The Contents of the Product.cs File

```
namespace SportsStore.Domain.Entities {

    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

I am following the technique of defining my domain model in a separate Visual Studio project, which means that th be marked as public. You do not need to follow this convention, but I find that it helps keep the model separate f controllers, which is useful in large and complex projects.

Creating an Abstract Repository

I need some way of getting Product entities from a database. As I explained in Chapter 3, the model includes the logic for storing and retrieving the data from the persistent data store, but even within the model, I want to keep a de separation between the data model entities and the storage and retrieval logic, which I achieve using the *repository pa* not worry about how I am going to implement data persistence for the moment, but I *will* start the process of defini interface for it.

Create a new top-level folder inside the SportsStore.Domain project called Abstract and, within th a new interface file called IProductsRepository.cs, the contents of which Listing 7-4 shows. You can interface by right-clicking the Abstract folder, selecting Add ➤ New Item, and selecting the Interfa

Listing 7-4. The Contents of the IProductRepository.cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract {
    public interface IProductRepository {

        IEnumerable<Product> Products { get; }
    }
}
```

This interface uses IEnumerable<T> to allow a caller to obtain a sequence of Product objects, without or where the data is stored or retrieved. A class that depends on the IProductRepository interface can ob Product objects without needing to know anything about where they are coming from or how the implementatio deliver them. This is the essence of the repository pattern. I will revisit the IProductRepository interface the development process to add features.

Making a Mock Repository

Now that I have defined an abstract interface, I could implement the persistence mechanism and hook it up to a datal want to add some of the other parts of the application first. In order to do this, I am going to create a mock impleme IProductRepository interface that will stand in until I return to the topic of data storage.

eBooks-IT.org

of the NinjectDependencyResolver class in the SportsStore.WebUI project, as illustrated b

Listing 7-5. Adding the Mock IProductRepository Implementation in the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Moq;
using Ninject;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Infrastructure {

    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
             Mock<IProductRepository> mock = new Mock<IProductRep
();
            mock.Setup(m => m.Products).Returns(new List<Product>
                new Product { Name = "Football", Price = 25 },
                new Product { Name = "Surf board", Price = 179 },
                new Product { Name = "Running shoes", Price = 95 }
            });

            kernel.Bind<IProductRepository>().ToConstant(mock.Obje
        }
    }
}
```

I had to add a number of namespaces to the file for this addition, but the process I used to create the mock reposi implementation uses the same Moq techniques I introduced in Chapter 6. I want Ninject to return the same mock obj it gets a request for an implementation of the `IProductRepository` interface, which is why I used the `To` method to set the Ninject scope, like this:

```
...
kernel.Bind<IProductRepository>().ToConstant(mock.Object) ;
...
```

Rather than create a new instance of the implementation object each time, Ninject will always satisfy requests for

Displaying a List of Products

I could spend the rest of this chapter building out the domain model and the repository, and not touch the UI project you would find that boring, though, so I am going to switch tracks and start using the MVC Framework in earnest. I model and repository features as I need them.

In this section, I am going to create a controller and an action method that can display details of the products in th For the moment, this will be for only the data in the mock repository, but I will sort that out later. I will also set up a *routing configuration* so that MVC knows how to map requests for the application to the controller I create.

Adding a Controller

Right-click the `Controllers` folder in the `SportsStore.WebUI` project and select `Add` ➤ `Contro` the pop-up menu. Select the `MVC 5 Controller - Empty` option, click the `Add` button and set the nan `ProductController`. Click the `Add` button and Visual Studio will create a new class file called `ProductController.cs`, which you should edit to match Listing 7-6.

Listing 7-6. The Initial Contents of the Product Controller.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;

        public ProductController(IProductRepository productReposit
            this.repository = productRepository;
        }
    }
}
```

In addition to removing the `Index` action method, I added a constructor that declares a dependency on the `IProductRepository` interface, which will lead Ninject to inject the dependency for the product repository instantiates the controller class. I also imported the `SportsStore.Domain` namespaces so that I can refer to repository and model classes without having to qualify their names. Next, I have added an action method, called `Lis` will render a view showing the complete list of products, as shown in Listing 7-7.

Listing 7-7. Adding an Action Method to the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
```

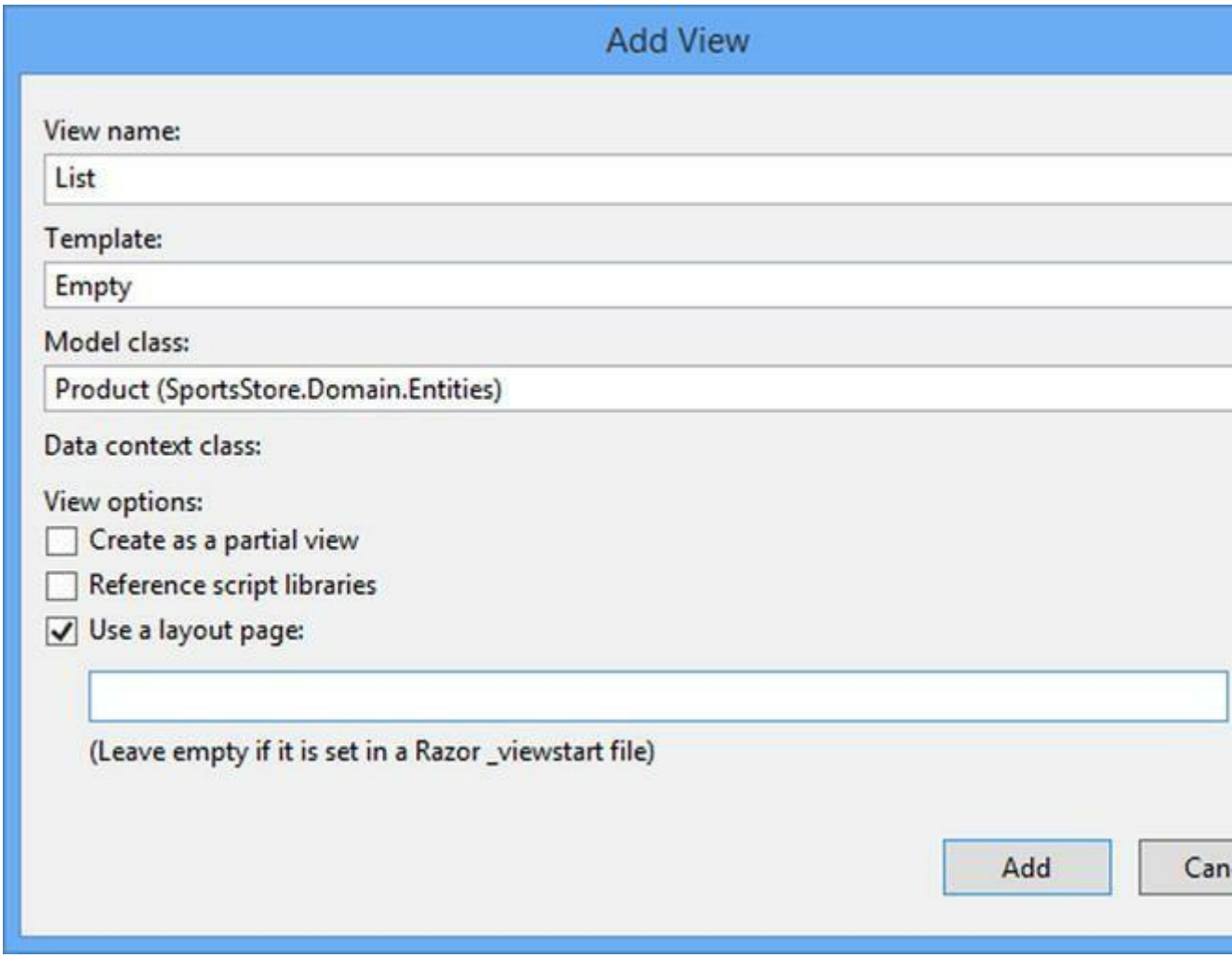
eBooks-IT.org

```
namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;

        public ProductController(IProductRepository productReposit
            this.repository = productRepository;
        }

        public ViewResult List() {
            return View(repository.Products);
        }
    }
}
```

Calling the `View` method like this (without specifying a view name) tells the framework to render the default view action method. Passing a `List` of `Product` objects to the `View` method, provides the framework with the dat to populate the `Model` object in a strongly typed view.

Adding the Layout, View Start File and View

Now I need to add the default view for the `List` action method. Right-click on the `List` action method in the `HomeController` class and select `Add View` from the pop-up menu. Set `View Name` to `List`, set T to `Empty`, and select `Product` for the `Model Class`, as shown in Figure 7-5. Ensure that the `Use A L Page` box is checked and click the `Add` button to create the view.



When you click the Add button, Visual Studio will create the List.cshtml file, but it will also create a _ViewStart.cshtml file and a Shared/_Layout.cshtml file. This is a helpful feature, but in kee Microsoft approach to default content, the _Layout.cshtml file contains template content that I do not want the layout so that it matches the content shown in Listing 7-8.

Listing 7-8. Editing the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-sca
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Rendering the View Data

Although I set the model type of the view to be the Product class, I actually want to work with an IEnumerable<Product>, which is what the Product controller obtains from the repository and passes In Listing 7-9, you can see that I have edited the @model expression and added some HTML and Razor ex display details of the products.

Listing 7-9. Editing the List.cshtml File

```
@using SportsStore.Domain.Entities
@model IEnumerable<Product>

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

I also changed the title of the page. Notice that I do not need to use the Razor @: expression to display the view d because each of the content lines in the code body either is a Razor directive or starts with an HTML element.

Tip I converted the Price property to a string using the ToString("c") method, which renders numer currency according to the culture settings that are in effect on your server. For example, if the server is set up as en (1002.3).ToString("c") will return $1,002.30, but if the server is set to en-GB, then the same return £1,002.30. You can change the culture setting for your server by adding a section to the <system.w the Web.config file like this: <globalization culture="en-GB" uiCulture="en-GB

I need to tell the MVC Framework that it should send requests that arrive for the root URL of my application (http://mysite/) to the List action method in the ProductController class. I do this by editin statement in the RegisterRoutes method in the App_Start/RouteConfig.cs file, as shown in

Listing 7-10. Adding the Default Route in the RouteConfig.cs File

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

namespace SportsStore.WebUI {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Product", action = "
                    id = UrlParameter.Optional }
            );
        }
    }
}
```

You can see the changes in bold. Change Home to Product and Index to List, as shown in the listing. ASP.NET routing feature in detail in Chapters 15 and 16. For now, it is enough to know that this change directs requ default URL to the List action method in the Product controller.

Tip Notice that I have set the value of the controller in Listing 7-10 to be Product and not ProductCon which is the name of the class. This is part of the ASP.NET MVC naming scheme, in which controller classes *alwa* Controller but you omit this part of the name when referring to the class.

Running the Application

I have all the basics in place. I have a controller with an action method that the MVC Framework will call when the c requested. That action method relies on a mock implementation of the repository interface, which generates some si The controller passes the test data to the view that I associated with the action method, and the view displays a simp details for each product. You can see the result by running the application, as shown in Figure 7-6. If you don't get t the figure, check that you have navigated to the root URL and not one that targets another action.

eBooks-IT.org

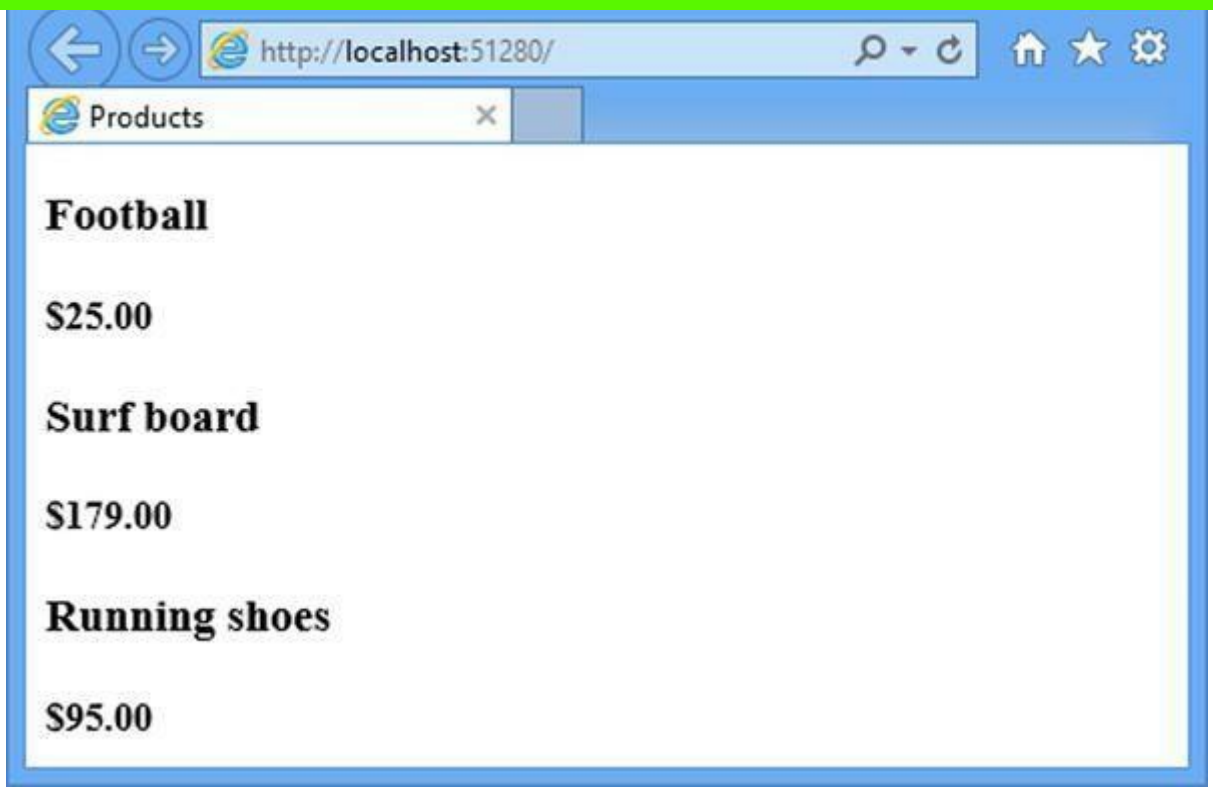


Figure 7-6. Viewing the basic application functionality

This is the typical pattern of development for the ASP.NET MVC Framework. An initial investment of time setting up is necessary, and then the basic features of the application snap together quickly.

EASIER DEBUGGING

When you run the project from the `Debug` menu, Visual Studio will create a new browser window to display application, which can take a few seconds. There are some tricks that you can use to speed the process up.

If you are editing view files and not classes, then you can make changes in Visual Studio while the debugger is r Reload the browser window when you want to see the effect of your changes. ASP.NET will recompile your vi classes and display the changes immediately. Visual Studio won't let you edit class files when the debugger is ru make some kinds of change to the project in the Solution Explorer, so this technique is most useful when you ar the fit and finish of the HTML that your application generates.

Visual Studio 2013 includes a new feature called *browser link* that lets you have multiple browser windows oper them from the Visual Studio menu bar. I demonstrate this feature in Chapter 14.

As a final alternative, you can keep your application open in a stand-alone browser window. To do this (assumir launched the debugger at least once already), right-click the IIS Express icon in the system tray and select the U app from the pop-up menu. After you have made your changes, compile the solution in Visual Studio by pressin choosing `Build` ➤ `Build Solution`, and then switch to your browser window and reload the Web p

Preparing a Database

I can already display simple views that contain details of the products, but I am displaying the test data that the mock `IproductRepository` returns. Before I can implement a real repository, I need to set up a database and po some data.

I am going to use SQL Server as the database, and I will access the database using the Entity Framework (EF), w Microsoft.NET ORM framework. An ORM framework presents the tables, columns, and rows of a relational databa regular C# objects. I mentioned in Chapter 6 that LINQ can work with different sources of data, and one of these is Framework. You will see how this simplifies things in a little while.

eBooks-IT.org

re diffe
databases available, but you can also work with object repositories, document stores, and some esoteric alternatives.
other .NET ORM frameworks as well, each of which takes a slightly different approach: variations that may give you
for your projects.

I am using the Entity Framework for a several reasons: it is simple and easy to get it up and working; the integratio
is first rate (and I like using LINQ); and it is good. The earlier releases were a bit hit-and-miss, but the current versio
and feature-rich.

Creating the Database

A nice feature of Visual Studio and SQL Server is the *LocalDB* feature, which is an administration-free implementatio
SQL Server features specifically designed for developers. Using this feature, I can skip the process of setting up a da
build my project and then deploy to a full SQL Server instance. Most MVC applications are deployed to hosted enviro
are run by professional administrators and so the LocalDB feature means that database configuration can be left in th
DBAs and developers can get on with coding. The LocalDB feature is installed automatically with Visual Studio Expre
Web, but you can download it directly from `www.microsoft.com/sqlserver` if you prefer.

The first step is to create the database connection in Visual Studio. Open the `Server Explorer` window f
`View` menu and click the `Connect to Database` button (it looks like a power cable with a green plus sig

You will see the `Choose Data Source` dialog. Select the `Microsoft SQL Server` option, as s
Figure 7-7, and click the `Continue` button. (Visual Studio remembers the selection you make, so you will not see
if you already created a database connection in another project).

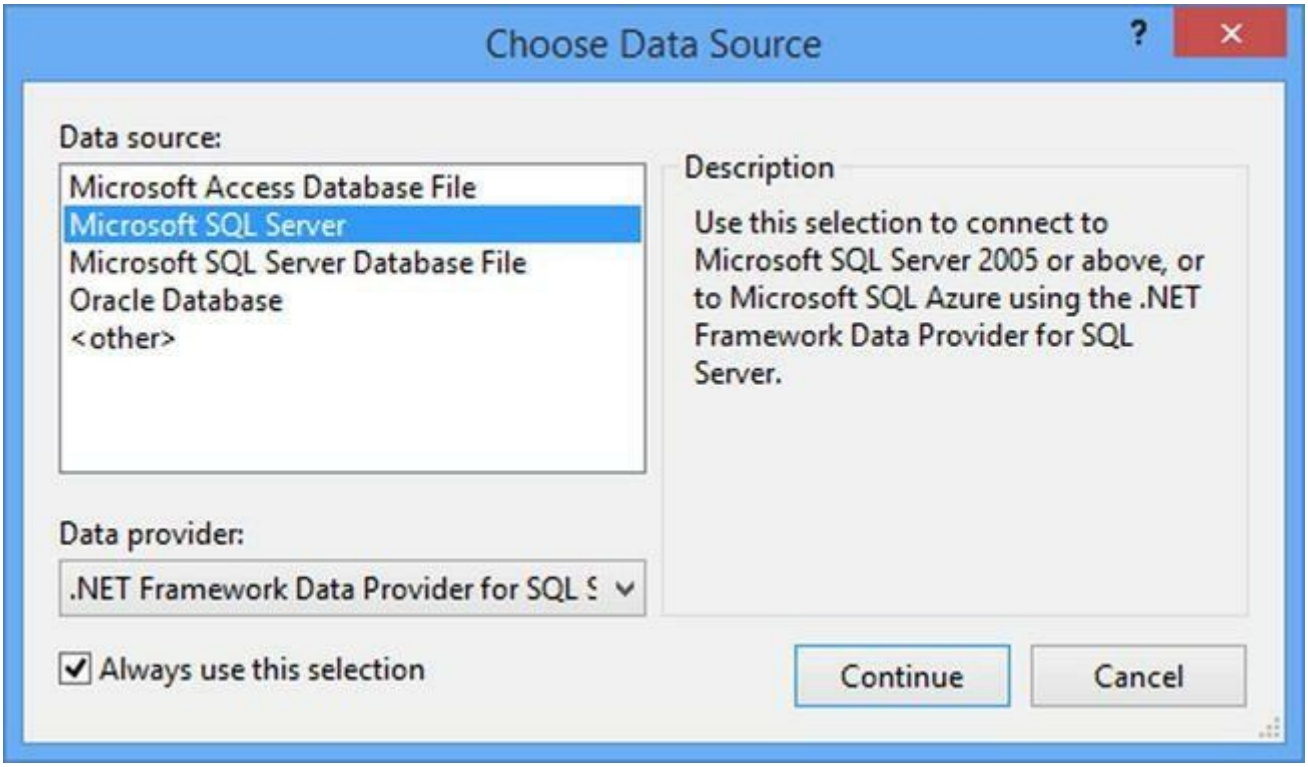


Figure 7-7. Selecting the Data Source

Next, you will see the `Add Connection` dialog. Set the server name to `(localdb)\v11.0`. This is a
that indicates that you want to use the LocalDB feature. Check the `Use Windows Authentication` opt
the database name to `SportsStore`, as shown by Figure 7-8.

eBooks-IT.org

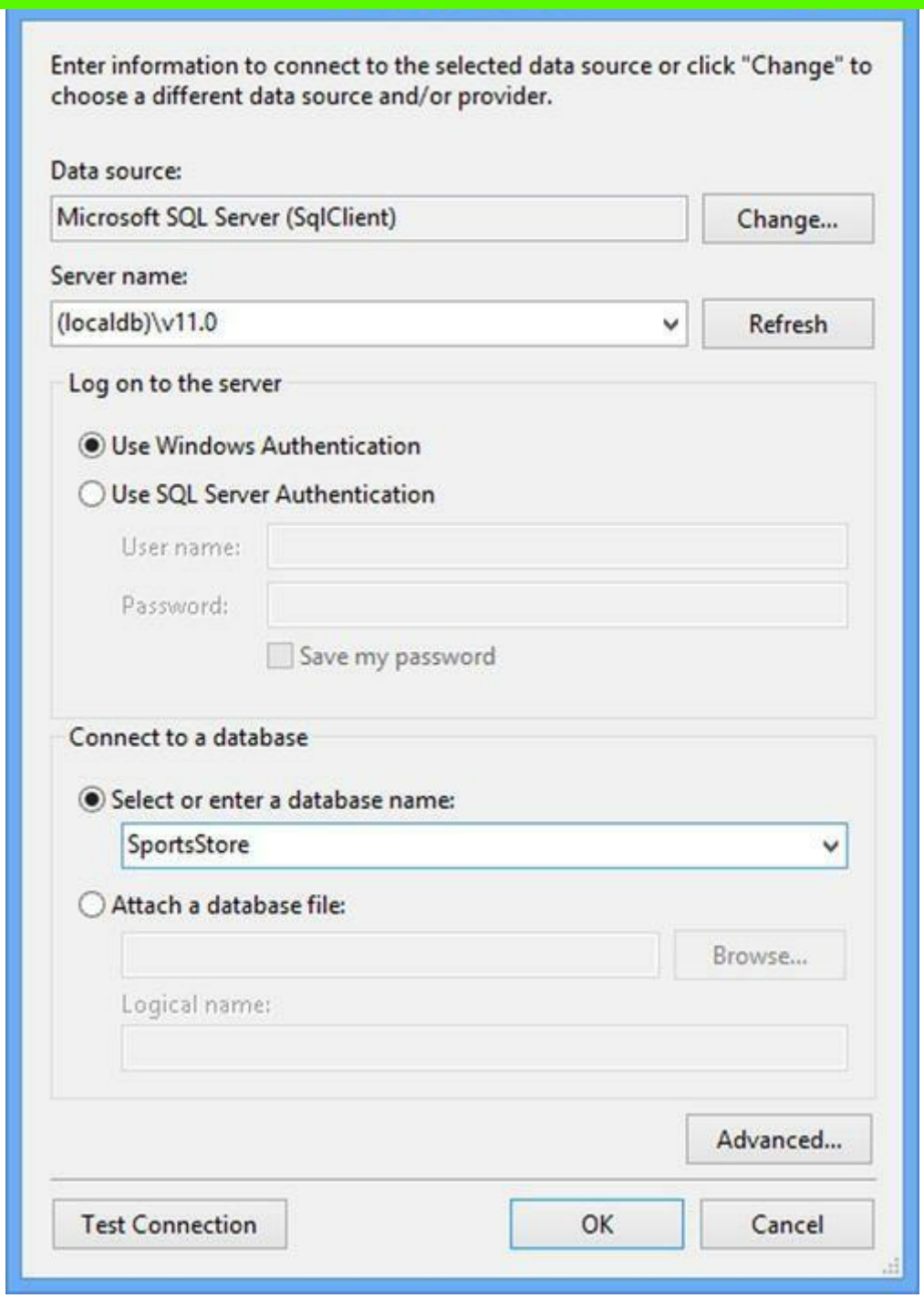


Figure 7-8. Setting up the SportsStore database

Tip If you did not see the Choose Data Source dialog, you can click the `Change` button in the top right of the Connection dialog.

Click the `OK` button and Visual Studio will prompt you to create the new database: click the `Yes` to go ahead. A n appear in the `Data Connections` section of the Server Explorer window, which you can expand to see the c facets of the database, as shown in Figure 7-9. You should see something similar, but the name of the database conn different because it will include the local PC name (the name of my workstation is `tiny`).

eBooks-IT.org

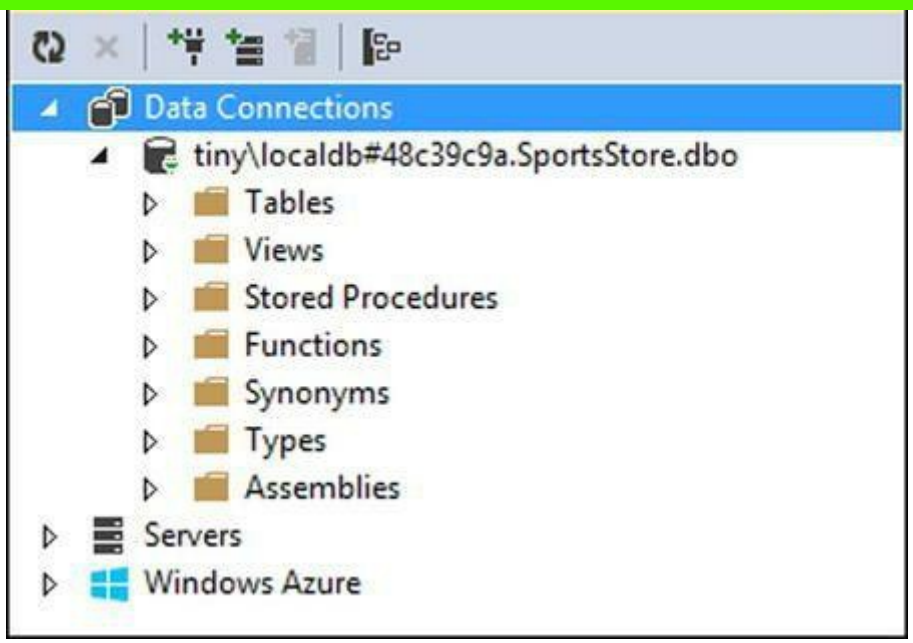


Figure 7-9. The LocalDB database as shown in the Server Explorer window

Defining the Database Schema

As I explained at the start of the chapter, my focus with the SportsStore application is to focus on the MVC Framew development process, and that means keeping the other components that the application relies on as simple as possib want to get into the topics of database design and the in-depth details of the Entity Framework, beyond what I need t demonstrate how to get data in and out of an application. These are topics in their own right and they are not part of the MVC Framework.

With this in mind, I am going to use a database that contains only one table. This is not how real e-commerce sites structure their data, of course, but the important lesson in this section is about the repository pattern and how I use i retrieve data, not the structure of the database.

To create the database table, right-click the `Tables` item for the new SportsStore database in the Server Explor and select `Add New Table`, as shown in Figure 7-10.

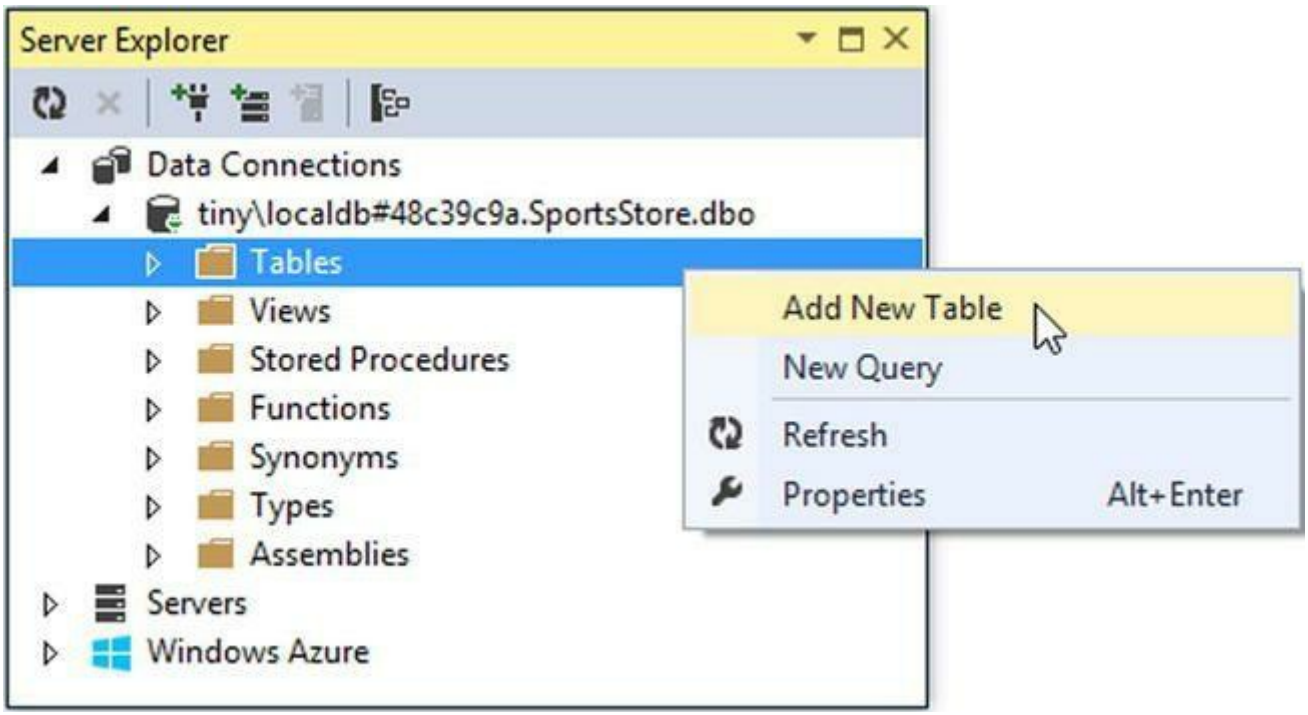


Figure 7-10. Adding a new table

eBooks-IT.org

of the designer, but I am going to use the T-SQL section because it is a more concise and accurate way of describing the specification I require in a book. Enter the SQL statement shown in Listing 7-11 and click the `Update` button in the corner of the table design window.

Listing 7-11. The SQL Statement to Create the Table in the SportsStore Database

```
CREATE TABLE Products
(
    [ProductID] INT NOT NULL PRIMARY KEY IDENTITY,
    [Name] NVARCHAR(100) NOT NULL,
    [Description] NVARCHAR(500) NOT NULL,
    [Category] NVARCHAR(50) NOT NULL,
    [Price] DECIMAL(16, 2) NOT NULL
)
```

This statement creates a table called `Products`, which has columns for the different properties I defined in the model class earlier in the chapter.

Tip Setting the `IDENTITY` property for the `ProductID` column means that SQL Server will generate a unique key value when I add data to this table. When using a database in a Web application, it can be difficult to generate unique keys because requests from users arrive concurrently. By enabling this feature, I can store new table rows and rely on SQL Server to sort out unique values.

When you click the `Update` button, Visual Studio will show a summary of the effect of the statement, as shown in Figure 7-11.

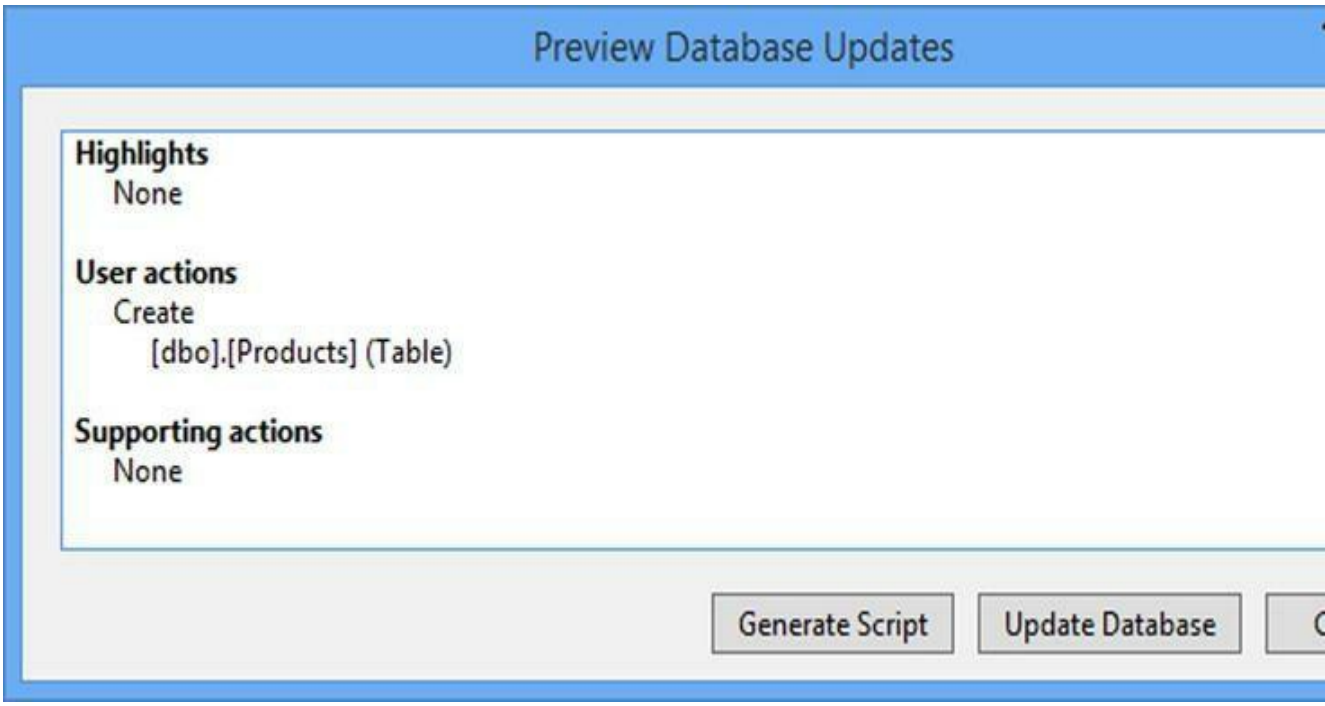


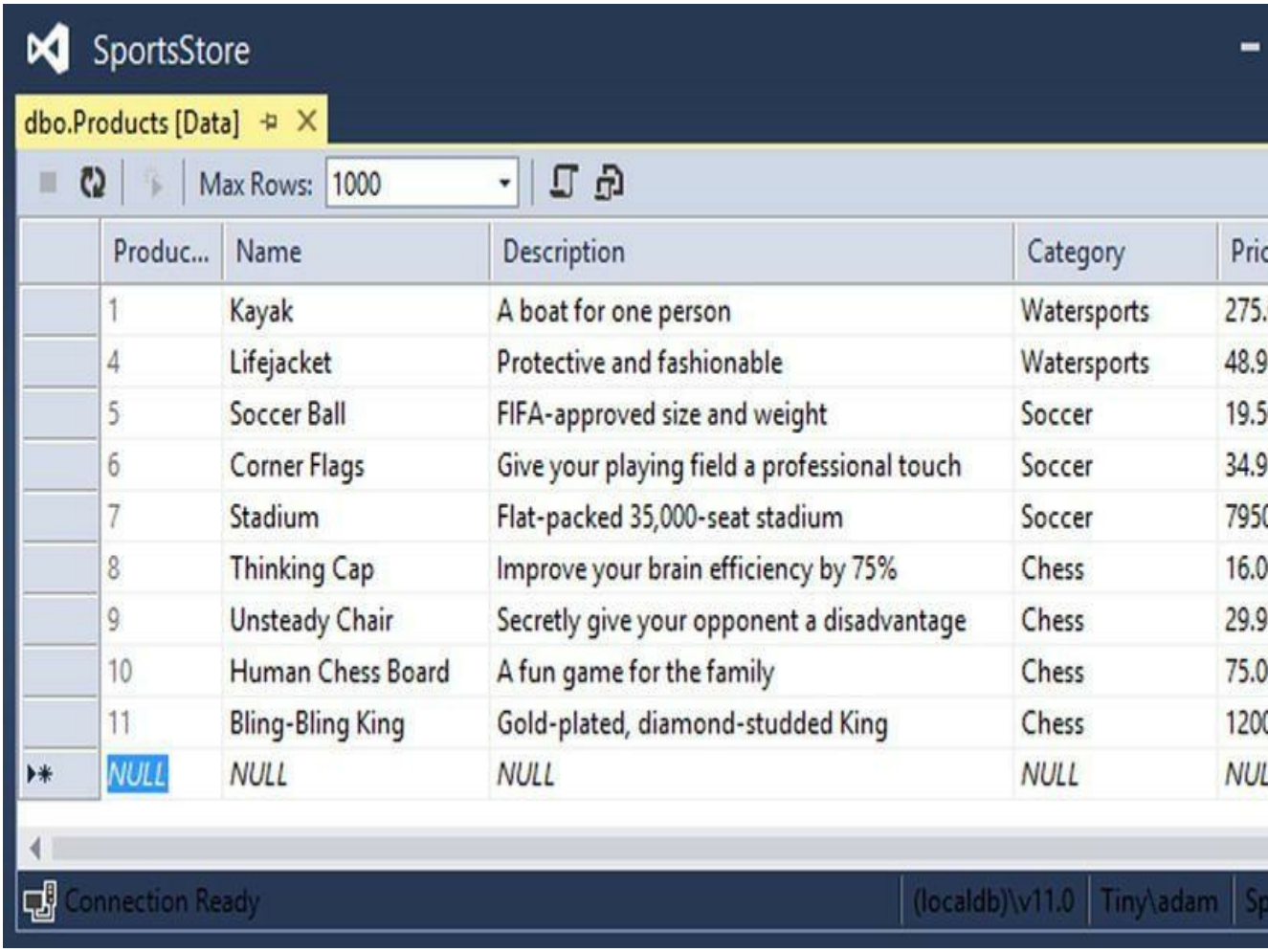
Figure 7-11. The summary of the effect of the SQL statement

Click the `Update Database` button to execute the SQL and create the `Products` table in the database. You are able to see the effect the update has if you click the `Refresh` button in the Server Explorer window. The `Table` window shows the new `Product` table and details of each of the rows.

Tip After you have updated the database, you can close the `dbo.Products` window. Visual Studio will offer you a chance to save the SQL script used to create the database. You do not need to save the script for this chapter, but it is useful in real projects if you need to configure multiple databases.

I am going to add data to the database so that I have something to work with until I add the catalog administration fe
Chapter 11.

In the `Server Explorer` window, expand the `Tables` item of the `SportsStore` database, right-c
`Products` table, and select `Show Table Data`. Enter the data shown in Figure 7-12. You can move from
by using the Tab key. At the end of each row, pressing tab will move to the next row and update the data in the datal



Produc...	Name	Description	Category	Pric
1	Kayak	A boat for one person	Watersports	275.
4	Lifejacket	Protective and fashionable	Watersports	48.9
5	Soccer Ball	FIFA-approved size and weight	Soccer	19.5
6	Corner Flags	Give your playing field a professional touch	Soccer	34.9
7	Stadium	Flat-packed 35,000-seat stadium	Soccer	7950
8	Thinking Cap	Improve your brain efficiency by 75%	Chess	16.0
9	Unsteady Chair	Secretly give your opponent a disadvantage	Chess	29.9
10	Human Chess Board	A fun game for the family	Chess	75.0
11	Bling-Bling King	Gold-plated, diamond-studded King	Chess	1200
NULL	NULL	NULL	NULL	NUL

Figure 7-12. Adding data to the Products table

Note You must leave the `ProductID` column empty. It is an identity column, so SQL Server will generate a
when you tab to the next row.

I listed the product details in Table 7-3 in case you cannot make out the detail from the figure. It doesn't matter if
enter all of the details exactly as I have, although you'll see different results from the ones I show as you work throu
process of creating the rest of the SportsStore application.

***Table 7-3**. The Data for the Products Table*

eBooks-IT.org

			Price
Kayak	A boat for one person	Watersports	275.00
Lifejacket	Protective and fashionable	Watersports	48.95
Soccer Ball	FIFA-approved size and weight	Soccer	19.50
Corner Flags	Give your playing field a professional touch	Soccer	34.95
Stadium	Flat-packed, 35,000-seat stadium	Soccer	79,500.
Thinking Cap	Improve your brain efficiency by 75%	Chess	16.00
Unsteady Chair	Secretly give your opponent a disadvantage	Chess	29.95
Human Chess Board	A fun game for the family	Chess	75.00
Bling-Bling King	Gold-plated, diamond-studded King	Chess	1,200.0

Creating the Entity Framework Context

Recent versions of the Entity Framework include a nice feature called *code-first*. The idea is that I can define classes and then generate a database from those classes.

This is great for green-field development projects, but these are few and far between. Instead, I am going to show variation on code-first, where I associate the model classes with an existing database. Select `Tools` ➤ `Librar` `Package Manager` ➤ `Package Manager Console` in Visual Studio to open the NuGet command enter the following command:

```
Install-Package EntityFramework -projectname SportsStore.Domain
Install-Package EntityFramework -projectname SportsStore.WebUI
```

Tip You may see errors in the `Package Manager Console` telling you that *binding redirects* cannot b You can safely ignore these warnings.

This command adds the Entity Framework package to the solution. I need to install the same package in the `Dom` `WebUI` projects so that I create the classes that will access the database in the `Domain` project and access the da `WebUI` project.

The next step is to create a *context* class that will associate the model with the database. Create a new folder in the `SportsStore.Domain` project called `Concrete` and add a new class file called `EFDbContext.cs` Edit the contents of the class file so they match Listing 7-12.

Listing 7-12. The Content of the EFDbContext.cs File

```
using SportsStore.Domain.Entities;
using System.Data.Entity;

namespace SportsStore.Domain.Concrete {

    public class EFDbContext : DbContext {
        public DbSet<Product> Products { get; set; }
    }
}
```

To take advantage of the code-first feature, I need to create a class that is derived from `System.Data.Entity.DbContext`. This class then automatically defines a property for each table in t that I want to work with.

The name of the property specifies the table, and the type parameter of the `DbSet` result specifies the model typ

eBooks-IT.org and th

parameter is `Product`, meaning that the Entity Framework should use the `Product` model type to represent ro
`Products` table.

Next, I need to tell the Entity Framework how to connect to the database, which I do by adding a database connec
the `Web.config` file in the `SportsStore.WebUI` project with the same name as the context class, as sh
Listing 7-13.

Listing 7-13. Adding a Database Connection in the Web.config File

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>

  <connectionStrings>
            <add   name="EFDbContext"   connectionString="Data
(localdb)\v11.0;Initial
        Catalog=SportsStore;Integrated Security=True"
        providerName="System.Data.SqlClient"/>
  </connectionStrings>

  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
     <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5.1" />
    <httpRuntime targetFramework="4.5.1" />
  </system.web>
</configuration>
```

Tip Notice that I have switched project here. I define the model and the repository logic in the `SportsStor`
project, but the database connection information is put in the `Web.config` file in the `SportsStore.WebU`

Caution I have had to split the value of the `connectionString` attribute across multiple lines to fit it on
it is important to put everything on a single line in the `Web.config` file.

There will be another `add` element in the `connectionsStrings` section of the `Web.config` file. Vi
creates this element by default and you can either ignore it or, as I have, delete it from the `Web.config` file.

Creating the Product Repository

All that remains is to add a class file to the `Concrete` folder of the `SportsStore.Domain` project called
`EFProductRepository.cs`. Edit your class file so it matches Listing 7-14.

Listing 7-14. The Contents of the EFProductRepostory.cs File

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Collections.Generic;

namespace SportsStore.Domain.Concrete {

    public class EFProductRepository : IProductRepository {
```

```
        public IEnumerable<Product> Products {
            get { return context.Products; }
        }
    }
}
```

This is the repository class. It implements the `IProductRepository` interface and uses an instance of `EFDbContext` to retrieve data from the database using the Entity Framework. You will see how I work with the Framework (and how simple it is) as I add features to the repository.

To use the new repository class, I need to edit the Ninject bindings and replace the mock repository with a binding one. Edit the `NinjectDependencyResolver.cs` class file in the `SportsStore.WebUI` project `AddBindings` method looks like Listing 7-15.

Listing 7-15. Adding the Real Repository Binding in the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Moq;
using Ninject;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Concrete;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Infrastructure {

    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            kernel.Bind<IProductRepository>().To<EFProductReposito
        }
    }
}
```

The new binding is in bold. It tells Ninject to create instances of the `EFProductRepository` class to ser for the `IProductRepository` interface. All that remains now is to run the application again. Figure 7-13 sh results, which demonstrate the application is getting its product data from the database, rather than the mock reposito

eBooks-IT.org

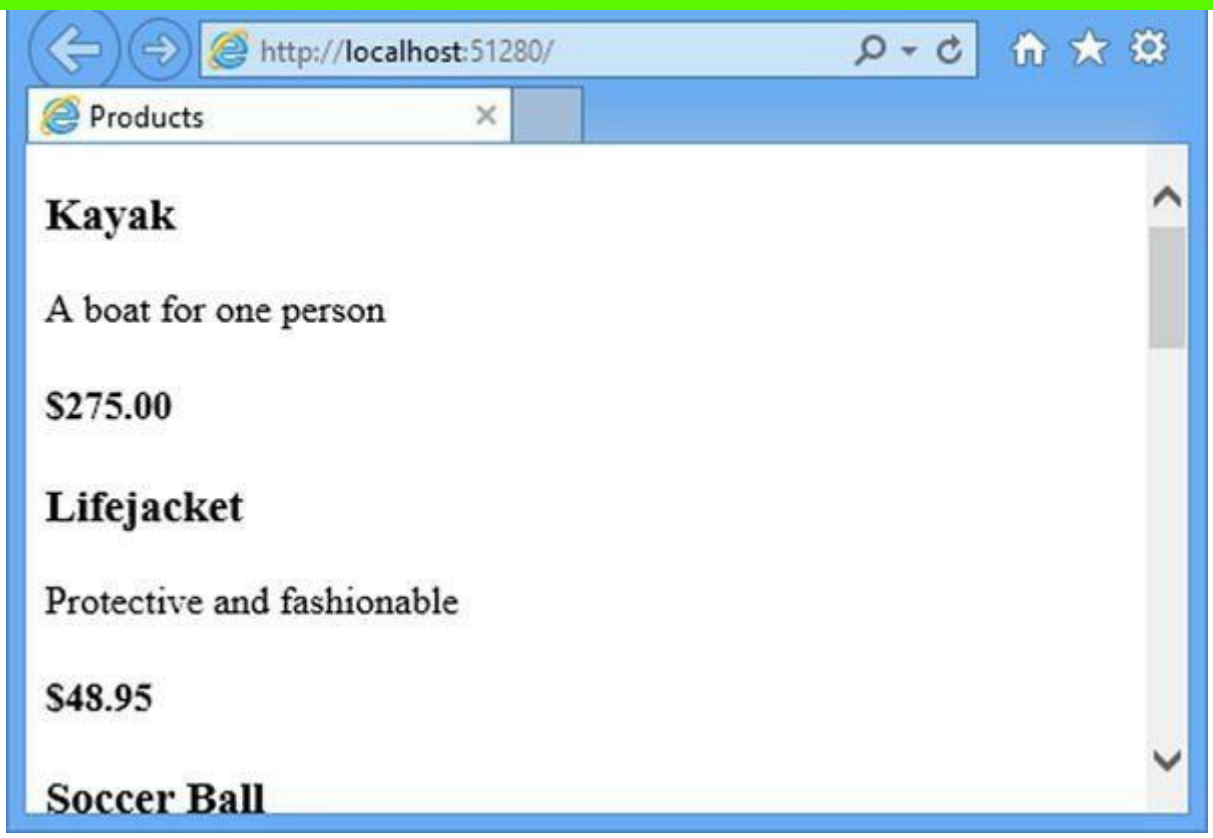


Figure 7-13. The result of implementing the real repository

Tip If you get a `System.ArgumentException` when you start the project, then you have split the de database connection over two lines in the `Web.config` file. See the previous section for details.

This approach to getting the Entity Framework to present a SQL Server database as a series of model objects is si to work with, and it allows me to keep my focus on the MVC Framework. Of course, I am skipping over a lot of the the Entity Framework operates and the huge number of configuration options that are available. I like the Entity Fram and I recommend that you spend some time getting to know it in detail. A good place to start is the Microsoft site fo Framework: http://msdn.microsoft.com/data/ef.

Adding Pagination

You can see from Figure 7-13 that the `List.cshtml` view displays all of the products in the database on a sing this section, I will add support for pagination so that the view displays a number of products on a page, and the user from page to page to view the overall catalog. To do this, I am going to add a parameter to the `List` method in the controller, as shown in Listing 7-16.

Listing 7-16. Adding Pagination Support to the List Action Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {
```

eBooks-IT.org

```
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productReposit
            this.repository = productRepository;
        }

        public ViewResult List(int page = 1) {
            return View(repository.Products
                .OrderBy(p => p.ProductID)
                .Skip((page - 1) * PageSize)
                .Take(PageSize));
        }
    }
}
```

The `PageSize` field specifies that I want four products per page. I will come back and replace this with a bett later on. I have added an *optional parameter* to the `List` method. This means that if I call the method without a pa (`List()`), my call is treated as though I had supplied the value specified in the parameter definition (`List(1)`) that the action method displays the first page of products when the MVC Framework invokes it without an argument body of the action method, I get the `Product` objects, order them by the primary key, skip over the products tha the start of the current page, and take the number of products specified by the `PageSize` field.

UNIT TEST: PAGINATION

I can unit test the pagination feature by creating a mock repository, injecting it into the constructor of the `ProductController` class, and then calling the `List` method to request a specific page. I can then c `Product` objects I get with what I would expect from the test data in the mock implementation. See Chapter of how to set up unit tests. Here is the unit test I created for this purpose, in the `UnitTest1.cs` file of the `SportsStore.UnitTests` project:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
namespace SportsStore.UnitTests {
  [TestClass]
  public class UnitTest1 {
    [TestMethod]
    public void Can_Paginate() {
      // Arrange
      Mock<IProductRepository> mock = new Mock<IProductRepository
      mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
      });
      ProductController controller = new ProductController(mock.
      controller.PageSize = 3;
      // Act
```

eBooks-IT.org

```
            (IEnumerable<Product>)controller.List(2).Model;
            // Assert
            Product[] prodArray = result.ToArray();
            Assert.IsTrue(prodArray.Length == 2);
            Assert.AreEqual(prodArray[0].Name, "P4");
            Assert.AreEqual(prodArray[1].Name, "P5");
        }
    }
}
```

Notice how easy it is to get the data that returned from a controller method. I call the `Model` property on the r the `IEnumerable<Product>` sequence generated in the `List` method. I then check that the data is v In this case, I converted the sequence to an array using the LINQ `ToArray` extension method and checked t the values of the individual objects.

Displaying Page Links

If you run the application, you will see that there are only four items shown on the page. If you want to view anothe can append query string parameters to the end of the URL, like this:

http://localhost:51280/?page=2

You will need to change the port part of the URL to match whatever port your ASP.NET development server is ru Using these query strings, you can navigate through the catalog of products.

Of course, there is no way for customers to figure out that these query string parameters exist, and even if there v not going to want to navigate this way. Instead, I need to render some page links at the bottom of the each list of pro customers can navigate between pages. To do this, I am going to implement a reusable HTML helper method, similar `Html.TextBoxFor` and `Html.BeginForm` methods I used in Chapter 2. The helper will generate the HT for the navigation links I require.

Adding the View Model

To support the HTML helper, I am going to pass information to the view about the number of pages available, the cu and the total number of products in the repository. The easiest way to do this is to create a view model, which I intro in Chapter 3. Add the class shown in Listing 7-17, called `PagingInfo`, to the `Models` folder in the `SportsStore.WebUI` project.

Listing 7-17. The Contents of the PagingInfo.cs File

```
using System;

namespace SportsStore.WebUI.Models {

    public class PagingInfo {
        public int TotalItems { get; set; }
        public int ItemsPerPage { get; set; }
        public int CurrentPage { get; set; }

        public int TotalPages {
                    get { return (int)Math.Ceiling((decimal)Total
ItemsPerPage); }
        }
    }
}
```

eBooks-IT.org

To emphasize this, I put this class in the SportsStore.WebUI project to keep it separate from the domain m

Adding the HTML Helper Method

Now that I have a view model, I can implement the HTML helper method, which I am going to call PageLinks. folder in the SportsStore.WebUI project called HtmlHelpers and add a new class file called PagingHelpers.cs, the contents of which Listing 7-18 shows.

Listing 7-18. The Contents of the PagingHelpers.cs Class File

```
using System;
using System.Text;
using System.Web.Mvc;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.HtmlHelpers {

    public static class PagingHelpers {

        public static MvcHtmlString PageLinks(this HtmlHelper html
                                              PagingInfo pagingInf
                                              Func<int, string> pa

            StringBuilder result = new StringBuilder();
            for (int i = 1; i <= pagingInfo.TotalPages; i++) {
                TagBuilder tag = new TagBuilder("a");
                tag.MergeAttribute("href", pageUrl(i));
                tag.InnerHtml = i.ToString();
                if (i == pagingInfo.CurrentPage) {
                    tag.AddCssClass("selected");
                    tag.AddCssClass("btn-primary");
                }
                tag.AddCssClass("btn btn-default");
                result.Append(tag.ToString());
            }
            return MvcHtmlString.Create(result.ToString());
        }
    }
}
```

The PageLinks extension method generates the HTML for a set of page links using the information provided PagingInfo object. The Func parameter accepts a delegate that it uses to generate the links to view other pag

UNIT TEST: CREATING PAGE LINKS

To test the PageLinks helper method, I call the method with test data and compare the results to the expec The unit test method is as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
```

eBooks-IT.org

```
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using SportsStore.WebUI.Models;
using SportsStore.WebUI.HtmlHelpers;

namespace SportsStore.UnitTests {
  [TestClass]
  public class UnitTest1 {

    [TestMethod]
    public void Can_Paginate() {
      // ...statements removed for brevity...
    }

    [TestMethod]
    public void Can_Generate_Page_Links() {
      // Arrange - define an HTML helper - we need to do this
      // in order to apply the extension method
      HtmlHelper myHelper = null;
      // Arrange - create PagingInfo data
      PagingInfo pagingInfo = new PagingInfo {
        CurrentPage = 2,
        TotalItems = 28,
        ItemsPerPage = 10
      };

      // Arrange - set up the delegate using a lambda expression
      Func<int, string> pageUrlDelegate = i => "Page" + i;

      // Act
      MvcHtmlString result = myHelper.PageLinks(pagingInfo, pageUrlDelegate);

      // Assert
      Assert.AreEqual(@"<a class=""btn btn-default"" href=""Page1""">1</a>"
    + @"<a class=""btn btn-default btn-primary selected"" href=""Page2""">2</a>"
        + @"<a class=""btn btn-default"" href=""Page3""">3</a>",
        result.ToString());
    }
  }
}
```

This test verifies the helper method output by using a literal string value that contains double quotes. C# is perfe of working with such strings, as long as the string is prefixed with `@` and use two sets of double quotes (`""`) i set of double quotes. You must remember not to break the literal string into separate lines, unless the string you comparing to is similarly broken. For example, the literal I use in the test method has wrapped onto two lines be width of a printed page is narrow. I have not added a newline character; if I did, the test would fail.

An extension method is available for use only when the namespace that contains it is in scope. In a code file, this i `using` statement; but for a Razor view, you must add a configuration entry to the `Web.config` file, or add a statement to the view itself. There are, confusingly, two `Web.config` files in a Razor MVC project: the main on resides in the root directory of the application project, and the view-specific one, which is in the `Views` folder. Th want to make is to the `Views/web.config` file as shown in Listing 7-19.

Listing 7-19. Adding the HTML Helper Method Namespace to the Views/web.config File

eBooks-IT.org

```
  <system.web.webPages.razor>
              <host    factoryType="System.Web.Mvc.MvcWebRazorHost
System.Web.Mvc,             Version=5.0.0.0,             Culture=
PublicKeyToken=31BF3856AD364E35" />
    <pages pageBaseType="System.Web.Mvc.WebViewPage">
      <namespaces>
        <add namespace="System.Web.Mvc" />
        <add namespace="System.Web.Mvc.Ajax" />
        <add namespace="System.Web.Mvc.Html" />
        <add namespace="System.Web.Routing" />
        <add namespace="SportsStore.WebUI" />
        <add namespace="SportsStore.WebUI.HtmlHelpers"/>
      </namespaces>
    </pages>
  </system.web.webPages.razor>
...
```

Every namespace that I refer to in a Razor view needs to be used explicitly, declared in the `web.config` file o a `@using` expression.

Adding the View Model Data

I am not quite ready to use the HTML helper method. I have yet to provide an instance of the `PagingInfo` view to the view. I could do this using the view bag feature, but I would rather wrap all of the data I am going to send fro controller to the view in a single view model class. To do this, I added a class file called `ProductsListView` to the `Models` folder of the `SportsStore.WebUI` project. Listing 7-20 shows the contents of the new file

Listing 7-20. The Contents of the ProductsListViewModel.cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models {
    public class ProductsListViewModel {

        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
    }
}
```

I can update the `List` action method in the `ProductController` class to use the `ProductsListViewModel` class to provide the view with details of the products to display on the page and pagination, as shown in Listing 7-21.

Listing 7-21. Updating the List Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
```

```
namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productReposit
            this.repository = productRepository;
        }

        public ViewResult List(int page = 1) {
            ProductsListViewModel model = new ProductsListViewMode
                Products = repository.Products
                .OrderBy(p => p.ProductID)
                .Skip((page - 1) * PageSize)
                .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                }
            };
            return View(model);
        }
    }
}
```

These changes pass a `ProductsListViewModel` object as the model data to the view.

UNIT TEST: PAGE MODEL VIEW DATA

I need to ensure that the controller sends the correct pagination data to the view. Here is the unit test I added to project to test this:

```
...
[TestMethod]
public void Can_Send_Pagination_View_Model() {

  // Arrange
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  });

  // Arrange
  ProductController controller = new ProductController(mock.Obje
  controller.PageSize = 3;

  // Act
  ProductsListViewModel result =
```

```
    // Assert
    PagingInfo pageInfo = result.PagingInfo;
    Assert.AreEqual(pageInfo.CurrentPage, 2);
    Assert.AreEqual(pageInfo.ItemsPerPage, 3);
    Assert.AreEqual(pageInfo.TotalItems, 5);
    Assert.AreEqual(pageInfo.TotalPages, 2);
  }
  ...
```

I also need to modify the earlier pagination unit test, contained in the `Can_Paginate` method. It relies on t action method returning a `ViewResult` whose `Model` property is a sequence of `Product` objects, bu wrapped that data inside another view model type. Here is the revised test:

```
  ...
  [TestMethod]
  public void Can_Paginate() {

    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRepository>()
    mock.Setup(m => m.Products).Returns(new Product[] {
      new Product {ProductID = 1, Name = "P1"},
      new Product {ProductID = 2, Name = "P2"},
      new Product {ProductID = 3, Name = "P3"},
      new Product {ProductID = 4, Name = "P4"},
      new Product {ProductID = 5, Name = "P5"}
    });
    ProductController controller = new ProductController(mock.Obje
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result = (ProductsListViewModel)controller.List(2).Model;

    // Assert
    Product[] prodArray = result. Products .ToArray();
    Assert.IsTrue(prodArray.Length == 2);
    Assert.AreEqual(prodArray[0].Name, "P4");
    Assert.AreEqual(prodArray[1].Name, "P5");
  }
  ...
```

I would usually create a common setup method, given the degree of duplication between these two test methods since I am delivering the unit tests in individual sidebars like this one, I am going to keep everything separate so each test on its own.

The view is currently expecting a sequence of `Product` objects, so I need to update the `List.cshtml` fil Listing 7-22, to deal with the new view model type.

Listing 7-22. Updating the List.cshtml File

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}
```

eBooks-IT.org

```
@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

I have changed the `@model` directive to tell Razor that I am now working with a different data type. I updated t `foreach` loop so that the data source is the `Products` property of the model data.

Displaying the Page Links

I have everything in place to add the page links to the `List` view. I created the view model that contains the paging updated the controller so that it passes this information to the view, and changed the `@model` directive to match th view type. All that remains is to call the HTML helper method from the view, which you can see in Listing 7-23.

Listing 7-23. Calling the HTML Helper Method in the List.cshtml File

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

<div>
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List", new
x }))
</div>
```

If you run the application, you will see the new page links, as illustrated in Figure 7-14. The style is still basic, whi later in the chapter. What is important for the moment is that the links take the user from page to page in the catalog explore the products for sale.

eBooks-IT.org

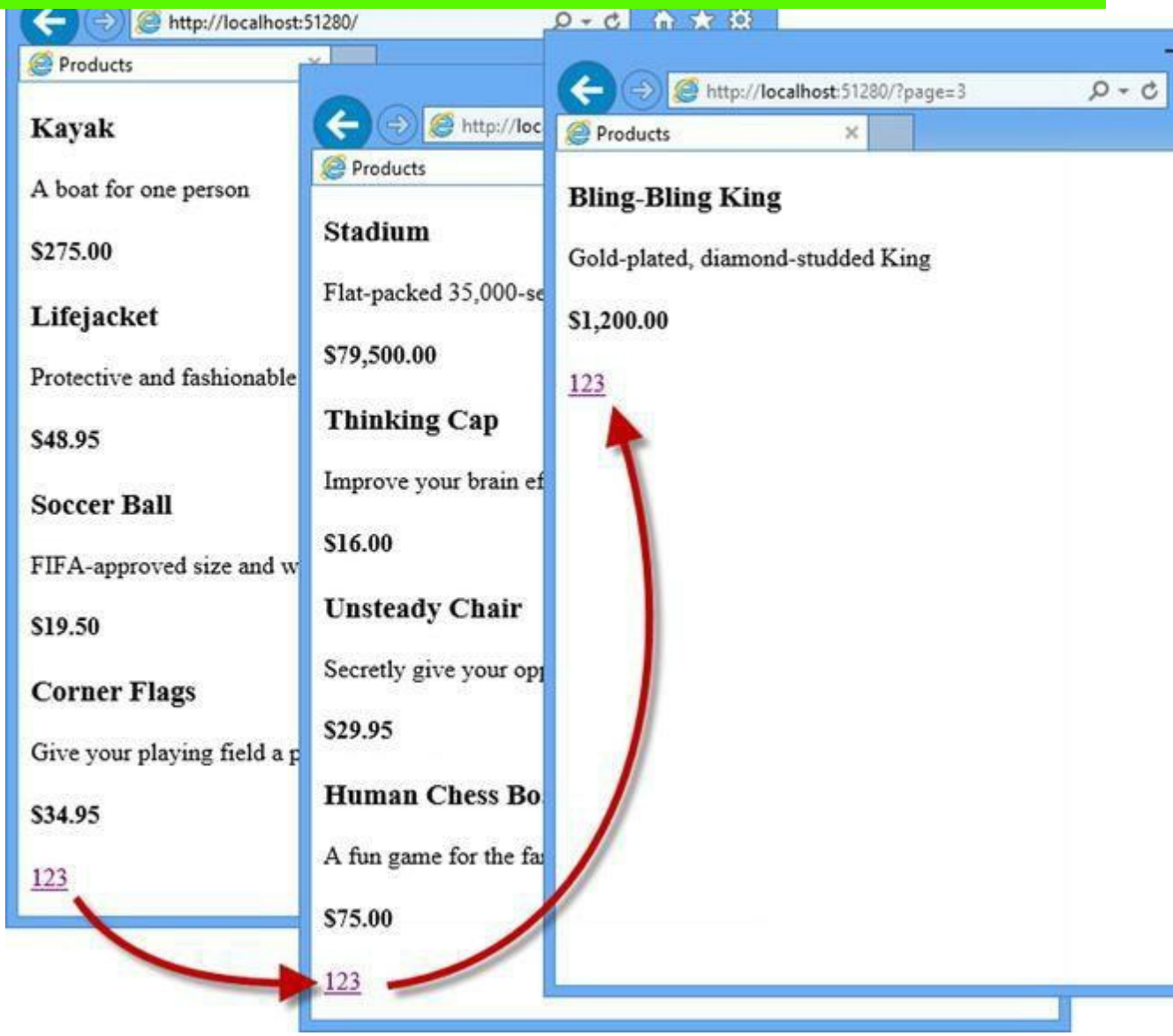


Figure 7-14. Displaying page navigation links

WHY NOT JUST USE A GRIDVIEW?

If you have worked with ASP.NET before, you might think that was a lot of work for an unimpressive result. I me pages and pages just to get a page list. If I were using Web Forms, I could have done the same thing using t Web Forms `GridView` or `ListView` controls, right out of the box, by hooking it up directly to the `Pro` database table.

What I have accomplished in this chapter may not look like much, but it is different from dragging a control ont surface. First, I am building an application with a sound and maintainable architecture that involves proper separ concerns. Unlike the simplest use of the `ListView` control, I have not directly coupled the UI and the datab approach that gives quick results but that causes pain and misery over time. Second, I have been creating unit t and these allow me to validate the behavior of the application in a natural way that is nearly impossible with a co Forms control. Finally, bear in mind that I have given over a lot of this chapter to creating the underlying infrast which I am building the application. I need to define and implement the repository only once, for example, and r have, I will be able to build and test new features quickly and easily, as the following chapters will demonstrate.

None of this detracts from the immediate results that Web Forms can deliver, of course, but as I explained in C immediacy comes with a cost that can be expensive and painful in large and complex projects.

Improving the URLs

eBooks-IT.org

is:

http://localhost/?page=2

I create URLs that are more appealing by creating a scheme that follows the pattern of *composable URLs*. A comp one that makes sense to the user, like this one:

http://localhost/Page2

MVC makes it easy to change the URL scheme in application because it uses the ASP.NET *routing* feature. All I n a new route to the `RegisterRoutes` method in the `RouteConfig.cs` file, which you will find in the A folder of the `SportsStore.WebUI` project. You can see the change I made to this file in Listing 7-24.

Listing 7-24. Adding a New Route to the RouteConfig.cs File

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

namespace SportsStore.WebUI {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: null,
                url: "Page{page}",
                defaults: new { Controller = "Product", action = "
            );

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Product", action = "
                    id = UrlParameter.Optional }
            );
        }
    }
}
```

It is important that you add this route before the `Default` one that is already in the file. As you will see in Chap routing system processes routes in the order they are listed, and I need the new route to take precedence over the ex

This is the only alteration required to change the URL scheme for product pagination. The MVC Framework and t function are tightly integrated, and so the application automatically reflects a change like this in the result produced b `Url.Action` method (which is what I used in the `List.cshtml` view to generate the page links). Do not v routing does not make sense to you now. I explain it in detail in Chapters 15 and 16.

If you run the application and navigate to a page, you will see the new URL scheme in action, as illustrated in Figu

eBooks-IT.org

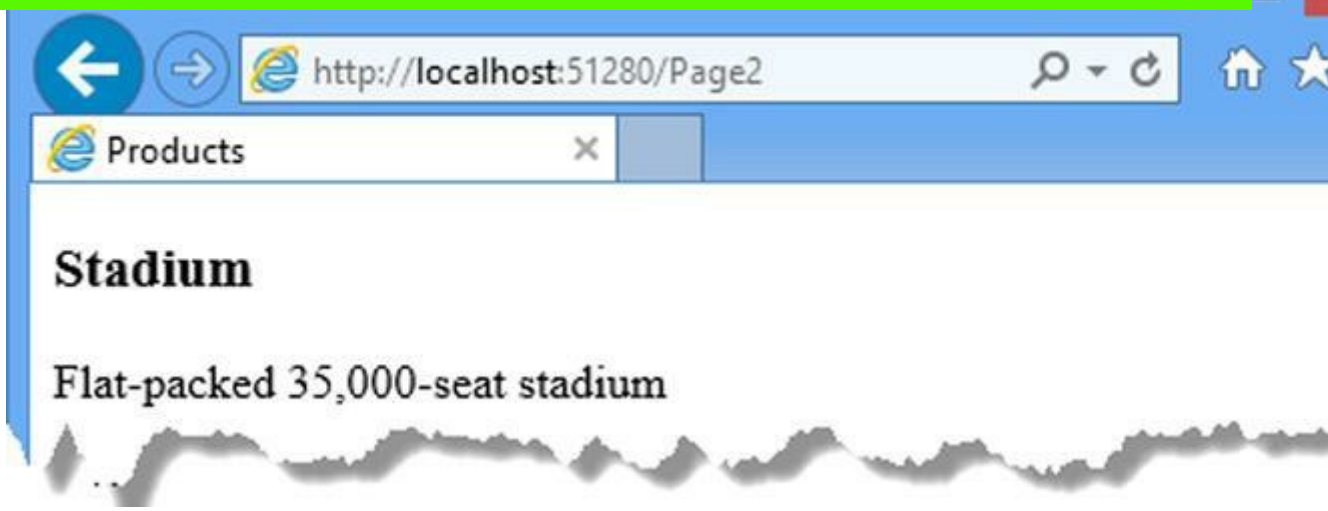


Figure 7-15. The new URL scheme displayed in the browser

Styling the Content

I have built a great deal of infrastructure and the application is really starting to come together, but I have not paid an its appearance. Even though this book is not about design or CSS, the SportsStore application design is so miserably undermines its technical strengths. In this section, I will put some of that right. I am going to implement a classic tw layout with a header, as shown in Figure 7-16.

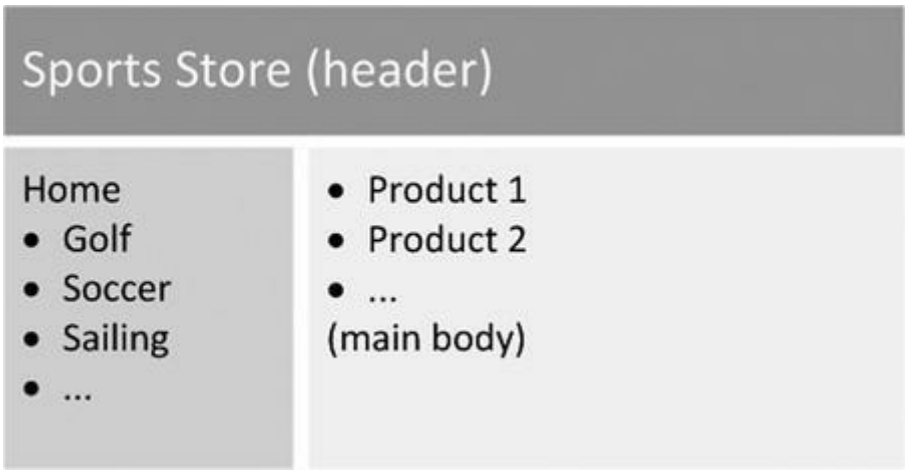


Figure 7-16. The design goal for the SportsStore application

Installing the Bootstrap Package

I am going to use the Bootstrap package to provide the CSS styles I will apply to the application. To install the Boots select `Library Package Manager`➤`Package Manager Console` from the Visual Studio T Visual Studio will open the NuGet command line. Enter the following command and hit return:

```
Install-Package -version 3.0.0 bootstrap -projectname SportsStore.
```

This is the same basic NuGet command I used in Chapter 2, with the addition of the `projectname` argument NuGet adds the files to the right project.

Note Once again, I am going to use Bootstrap without going into the details of the features that the package prov details of Bootstrap and the other client-side libraries that Microsoft has blessed for use with the MVC Framework, s ASP.NET MVC 5 Client book, published by Apress in 2014.

In Chapter 5, I explained how Razor layouts work and how you apply them. When I created the `List.cshtml` `Product` controller, I asked you to check the option to use a layout, but leave the box that specifies a layout blanl the effect of using the layout specified by the `Views/_ViewStart.cshtml` file, which Visual Studio crea automatically along the view. You can see the contents of the view start file in Listing 7-25.

Listing 7-25. The Contents of the _ViewStart.cshtml File

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

The value of the `Layout` property specifies that views will use the `Views/Shared/_Layout.csht` layout, unless they explicitly specify an alternative. I reset the content of the `_Layout.cshtml` file earlier in th remove the template content that Visual Studio adds and in Listing 7-26 you can see how I have returned to this file t Bootstrap CSS file and apply some of the CSS styles it defines.

Listing 7-26. Applying Bootstrap CSS to the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-sca
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="navbar navbar-inverse" role="navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            Put something useful here later
        </div>
        <div class="col-xs-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

I have added the `bootstrap.css` and `bootstrap-theme.css` files to the layout using `link` eler applied various Bootstrap classes to create a simple layout. I also need to change the `List.cshtml` file, as show 7-27.

Listing 7-27. Applying Bootstrap to Style the List.cshtml File

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
```

```
@foreach (var p in Model.Products) {
    <div class="well">
        <h3>
            <strong>@p.Name</strong>
                                <span   class="pull-right   label
primary">@p.Price.ToString("c")</span>
        </h3>
        <span class="lead"> @p.Description</span>
    </div>
}

<div class="btn-group pull-right">
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List", new
x }))
</div>
```

THE PROBLEM WITH STYLING ELEMENTS

The HTML elements generated by an MVC application come from a variety of sources (static content, Razor ex HTML helper methods, etc.), so the style classes become diffused throughout the project. If this makes you fe uncomfortable, then you are not alone. Mixing the CSS styles in with element generation is not a great idea and to the idea of separating out unrelated functionality that pervades MVC. You can improve on this situation by as Bootstrap classes to elements based on their role in the application and then use a library like jQuery or LESS to your custom classes and the Bootstrap ones.

I am going to keep things simple for this application and accept that I have embedded the Bootstrap classes thro application, even though it complicates the process of changing styles in the future. I would not do this in a real know this example application is not going to enter into a maintenance phase.

If you run the application, you will see that I have improved the appearance—at least a little, anyway—as illustrate 17.

eBooks-IT.org

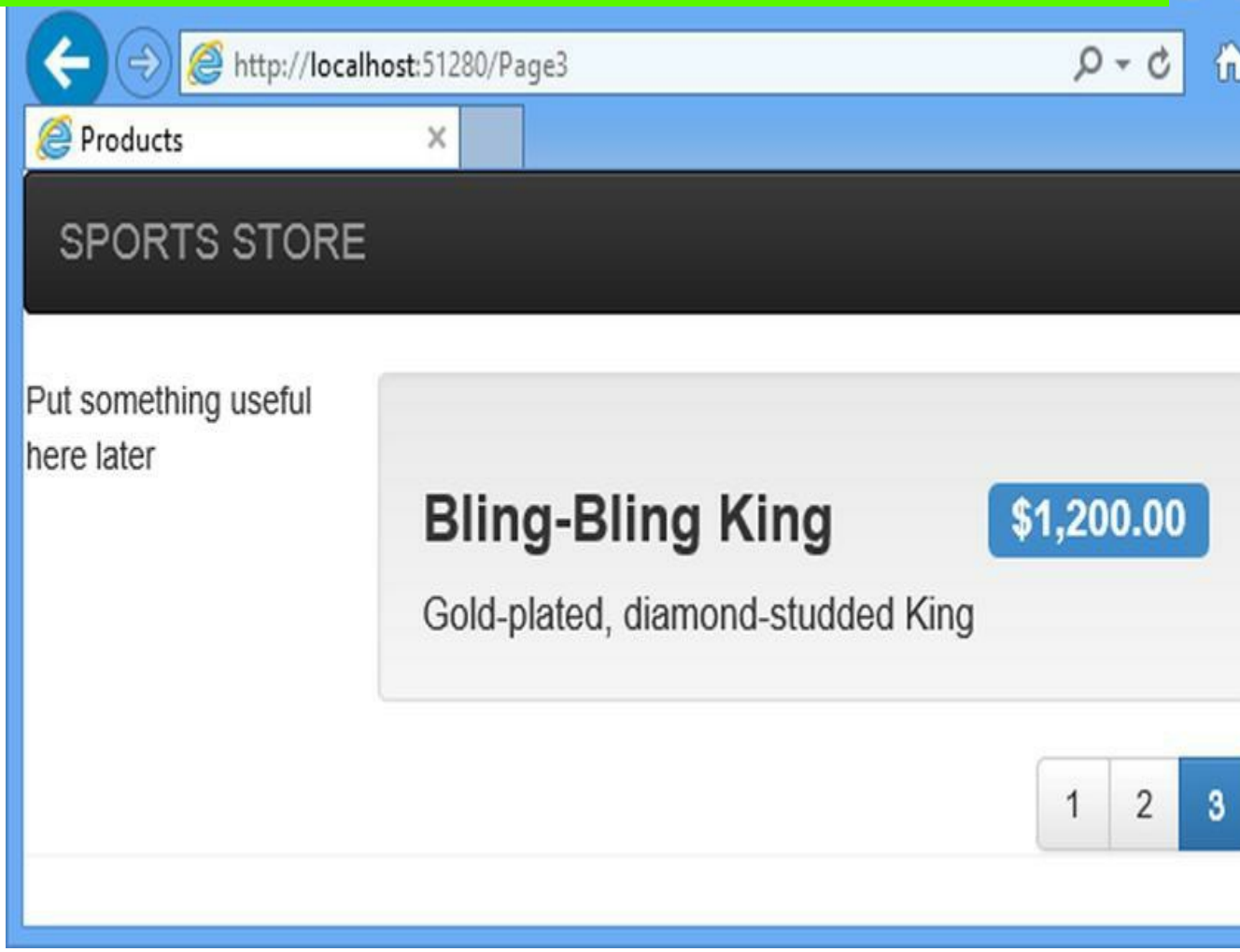


Figure 7-17. The design-enhanced SportsStore application

Creating a Partial View

As a finishing trick for this chapter, I am going to refactor the application to simplify the `List.cshtml` view. I create a *partial view*, which is a fragment of content that you can embed into another view, rather like a template. Pa are contained within their own files and are reusable across multiple views, which can help reduce duplication if you render the same kind of data in several places in your application.

To add the partial view, right-click the `/Views/Shared` folder in the `SportsStore.WebUI` project a `Add` ➤ `View` from the pop-up menu. Set `View Name` to `ProductSummary`, set `Template` to `Emp` `Product` from the `Model Class` drop-down list and check the `Create As A Partial View` b in Figure 7-18.

eBooks-IT.org

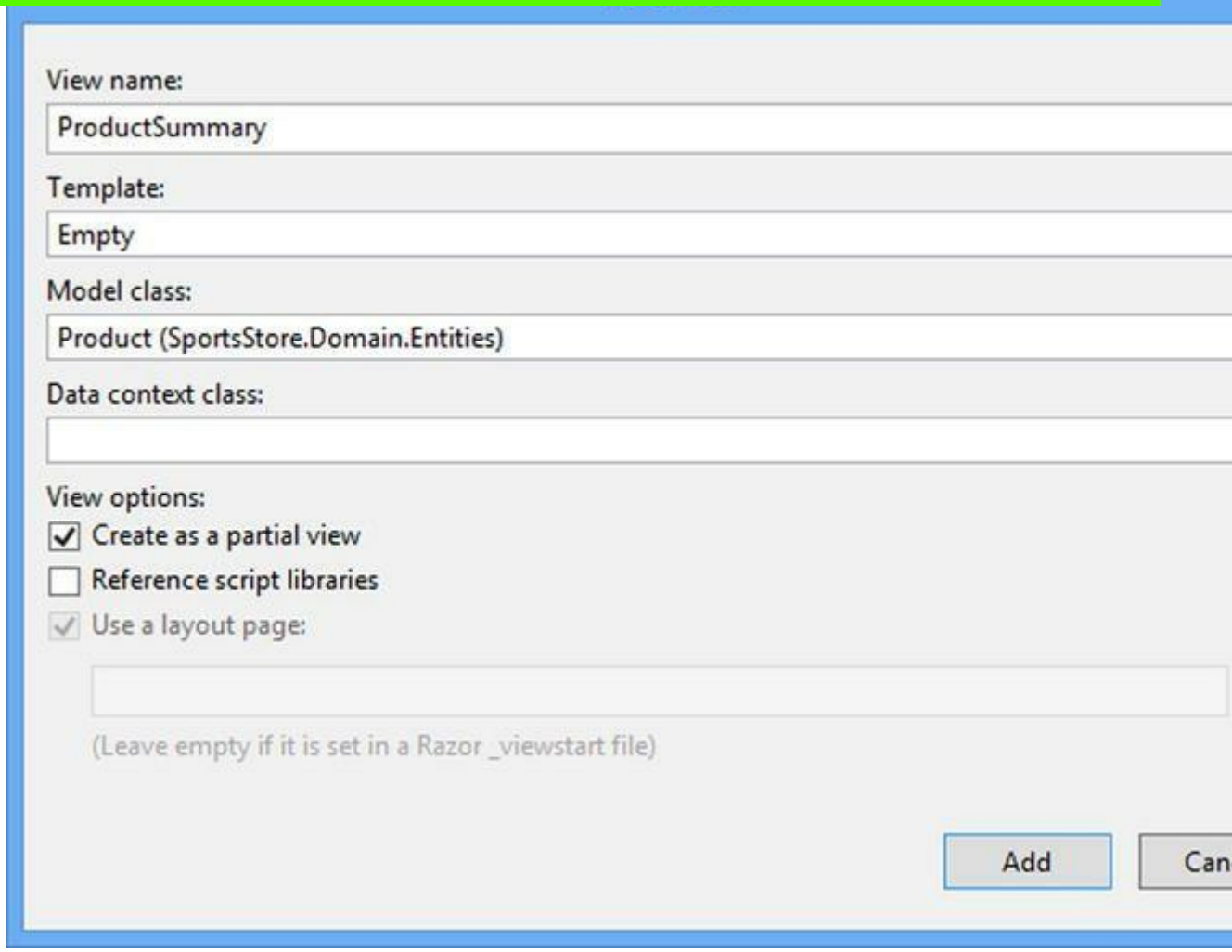


Figure 7-18. Creating a partial view

Click the `Add` button, and Visual Studio will create a partial view file called `Views/Shared/ProductSummary.cshtml`. A partial view is similar to a regular view, except that it fragment of HTML, rather than a full HTML document. If you open the `ProductSummary` view, you will see contains only the model view directive, which is set to the `Product` domain model class. Apply the changes show 7-28.

Listing 7-28. Adding Markup to the ProductSummary.cshtml File

```
@model SportsStore.Domain.Entities.Product

<div class="well">
    <h3>
        <strong>@Model.Name</strong>
                                <span    class="pull-right    label
primary">@Model.Price.ToString("c")</span>
    </h3>
    <span class="lead"> @Model.Description</span>
</div>
```

Now I need to update `Views/Products/List.cshtml` so that it uses the partial view. You can see th Listing 7-29.

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products) {
    @Html.Partial("ProductSummary", p)
}

<div class="pager">
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List", new
x}))
</div>
```

I have taken the markup that was previously in the `foreach` loop in the `List.cshtml` view and moved it partial view. I call the partial view using the `Html.Partial` helper method. The parameters are the name of the view model object. Switching to a partial view like this is good practice, but it does not change the appearance of the as Figure 7-19 shows.

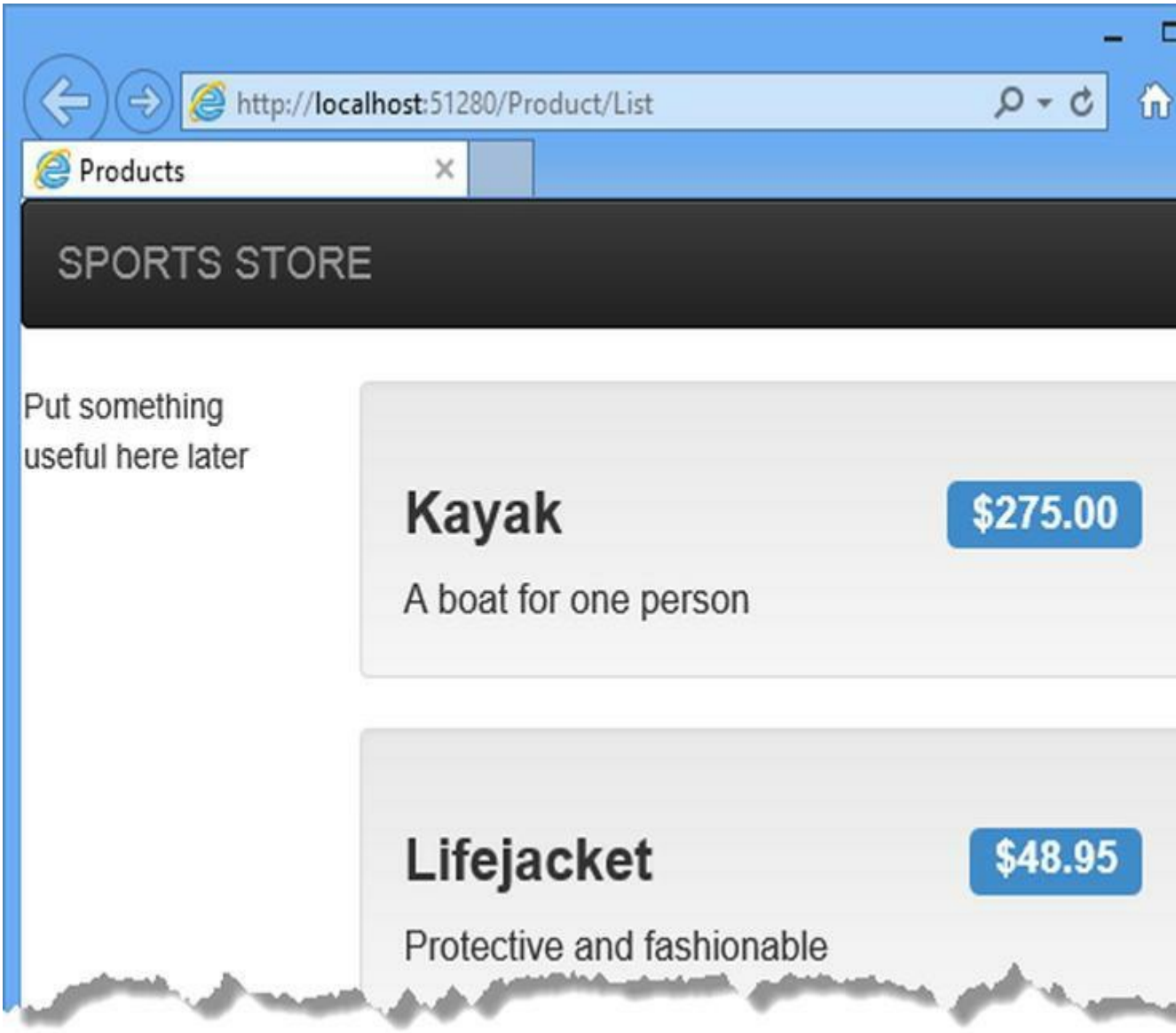


Figure 7-19. Applying a partial view

eBooks-IT.org

In this chapter, I built most of the core infrastructure for the SportsStore application. It does not have many features could demonstrate to a client at this point, but behind the scenes, there are the beginnings of a domain model with a p repository backed by SQL Server and the Entity Framework. There is a single controller, `ProductControlle` produce paginated lists of products, and I have set up DI and defined a clean and friendly URL scheme.

If this chapter felt like a lot of setup for little benefit, then the next chapter will balance the equation. Now that the structure is in place, we can forge ahead and add all of the customer-facing features: navigation by category, a shopp a checkout process.

eBooks-IT.org

SportsStore: Navigation

In the previous chapter, I set up the core infrastructure of the SportsStore application. Now I will use that infrastruc features to the application and you will start to see how the investment in the basic plumbing pays off. I will be able t important customer-facing features simply and easily and, along the way, you will see some additional functionality t Framework provides.

Adding Navigation Controls

The SportsStore application will be more usable if customers can navigate products by category. I will do this in thre

- Enhance the `List` action model in the `ProductController` class so that it is able to filter th objects in the repository.
- Revisit and enhance the URL scheme and revise the routing strategy.
- Create a category list that will go into the sidebar of the site, highlighting the current category and linking to o

Filtering the Product List

I am going to start by enhancing the view model class, `ProductsListViewModel`, which I added to the `SportsStore.WebUI` project in the last chapter. I need to communicate the current category to the view in render the sidebar, and this is as good a place to start as any. Listing 8-1 shows the changes I made to the `ProductsListView.cs` file.

Listing 8-1. Enhancing the ProductsListView .cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models {
    public class ProductsListViewModel {

        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
        public string CurrentCategory { get; set; }
    }
}
```

I added a property called `CurrentCategory`. The next step is to update the `Product` controller so that action method will filter `Product` objects by category and use the new property I added to the view model to indi category has been selected. The changes are shown in Listing 8-2.

Listing 8-2. Adding Category Support to the List Action Method in the ProductController.cs File

```
using System;
```

eBooks-IT.org

```
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productReposit
            this.repository = productRepository;
        }

        public ViewResult List(string category , int page = 1) {
            ProductsListViewModel model = new ProductsListViewMode
                Products = repository.Products
                           .Where(p => category == null || p.Cate
category)
                    .OrderBy(p => p.ProductID)
                    .Skip((page - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                },
                CurrentCategory = category
            };
            return View(model);
        }
    }
}
```

I made three changes to the action method. First, I added a parameter called `category`. This `category` pa used by the second change in the listing, which is an enhancement to the LINQ query. If `category` is not `null` `Product` objects with a matching `Category` property are selected. The last change is to set the value of the `CurrentCategory` property I added to the `ProductsListViewModel` class. However, these chang the value of `PagingInfo.TotalItems` is incorrectly calculated. I will fix this in a while.

UNIT TEST: UPDATING EXISTING UNIT TESTS

I changed the signature of the `List` action method, which will prevent some of the existing unit test methods compiling. To address this, I need to pass `null` as the first parameter to the `List` method in those unit tests with the controller. For example, in the `Can_Paginate` test, the action section of the unit test becomes as

```
...
[TestMethod]
public void Can_Paginate() {

  // Arrange
```

eBooks-IT.org

```
  mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  });

  // create a controller and make the page size 3 items
  ProductController controller = new ProductController(mock.Obje
  controller.PageSize = 3;

  // Act
  ProductsListViewModel result
    = (ProductsListViewModel)controller.List(null, 2).Model;

  // Assert
  Product[] prodArray = result.Products.ToArray();
  Assert.IsTrue(prodArray.Length == 2);
  Assert.AreEqual(prodArray[0].Name, "P4");
  Assert.AreEqual(prodArray[1].Name, "P5");
}
...
```

By using `null`, I receive all of the `Product` objects that the controller gets from the repository, which is th situation I had before adding the new parameter. I need to make the same kind of change to the `Can_Send_Pagination_View_Model` test as well:

```
...
[TestMethod]
public void Can_Send_Pagination_View_Model() {

  // Arrange
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  });

  // Arrange
  ProductController controller = new ProductController(mock.Obje
  controller.PageSize = 3;

  // Act
  ProductsListViewModel result
    = (ProductsListViewModel)controller.List(null, 2).Model;

  // Assert
  PagingInfo pageInfo = result.PagingInfo;
  Assert.AreEqual(pageInfo.CurrentPage, 2);
  Assert.AreEqual(pageInfo.ItemsPerPage, 3);
  Assert.AreEqual(pageInfo.TotalItems, 5);
```

```
}
...
```

Keeping your unit tests synchronized with your code changes quickly becomes second nature when you get into mind-set.

The effect of the category filtering is evident, even with these small changes. Start the application and select a cate the follow query string, changing the port to match the one that Visual Studio assigned for your project:

http://localhost:51280/?category=Soccer

You will see only the products in the `Soccer` category, as shown in Figure 8-1.

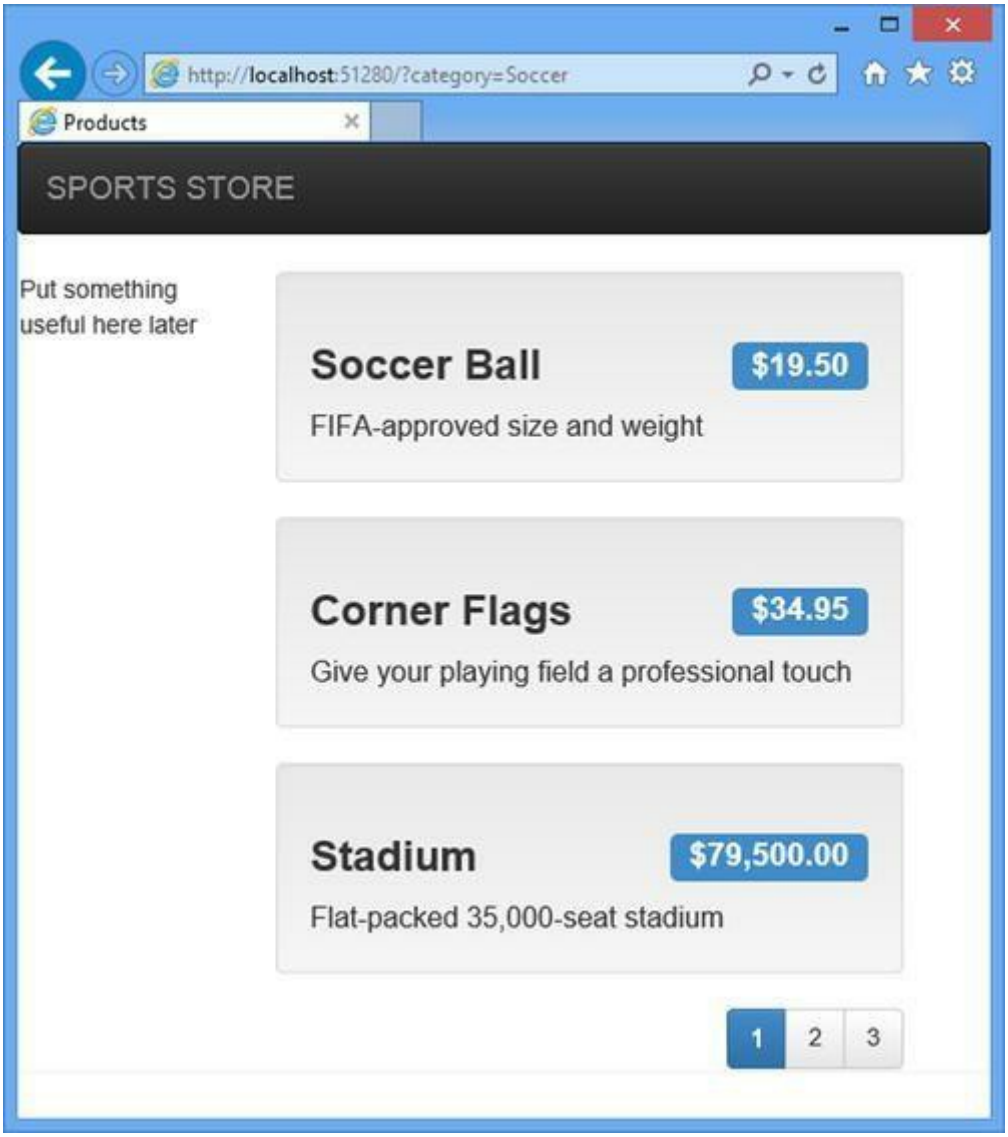


Figure 8-1. Using the query string to filter by category

Obviously, users won't want to navigate to categories using URLs, but you can see how small changes can have a an MVC Framework application once the basic structure is in place.

UNIT TEST: CATEGORY FILTERING

I need a unit test to properly test the category filtering function, to ensure that the filter can correctly generate p specified category. Here is the test:

```
...
```

eBooks-IT.org

```
public void Can_Filter_Products() {
  // Arrange
  // - create the mock repository
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
    new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
    new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
    new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
    new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
  });
  // Arrange - create a controller and make the page size 3 item
  ProductController controller = new ProductController(mock.Obje
  controller.PageSize = 3;
  // Action
  Product[] result = ((ProductsListViewModel)controller.List("Ca
1).Model)
    .Products.ToArray();
  // Assert
  Assert.AreEqual(result.Length, 2);
  Assert.IsTrue(result[0].Name == "P2" && result[0].Category ==
"Cat2");
  Assert.IsTrue(result[1].Name == "P4" && result[1].Category ==
"Cat2");
}
...
```

This test creates a mock repository containing `Product` objects that belong to a range of categories. One sp category is requested using the `Action` method, and the results are checked to ensure that the results are the in the right order.

Refining the URL Scheme

No one wants to see or use ugly URLs such as `/?category=Soccer`. To address this, I am going to revisit scheme to create an approach to URLs that better suits me and my customers. To implement the new scheme, I cha `RegisterRoutes` method in the `App_Start/RouteConfig.cs` file, as shown in Listing 8-3.

Listing 8-3. The New URL Scheme in the RouteConfig.cs File

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

namespace SportsStore.WebUI {

    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(null,
```

eBooks-IT.org

```
                    new {
                        controller = "Product", action = "List",
                        category = (string)null, page = 1
                    }
                );

                routes.MapRoute(null,
                    "Page{page}",
                     new { controller = "Product", action = "List", ca
(string)null },
                    new { page = @"\d+" }
                );

                routes.MapRoute(null,
                    "{category}",
                    new { controller = "Product", action = "List", pag
                );

                routes.MapRoute(null,
                    "{category}/Page{page}",
                    new { controller = "Product", action = "List" },
                    new { page = @"\d+" }
                );

                routes.MapRoute(null, "{controller}/{action}");
            }
        }
}
```

Caution It is important to add the new routes in Listing 8-3 in the order they are shown. Routes are applied in th which they are defined, and you will get some odd effects if you change the order.

Table 8-1 describes the URL scheme that these routes represent. I explain the routing system in detail in Chapters

Table 8-1. *Route Summary*

URL	Leads To
/	Lists the first page of products from all categories
/Page2	Lists the specified page (in this case, page 2), showing items from all categories
/Soccer	Shows the first page of items from a specific category (in this case, the Soccer category)
/Soccer/Page2	Shows the specified page (in this case, page 2) of items from the specified category (in this case, Soccer)

The ASP.NET routing system is used by MVC to handle *incoming* requests from clients, but it also generates *outg* that conform to the URL scheme and that can be embedded in Web pages. By using the routing system to handle inc requests and generate outgoing URLs, I can ensure that all of the URLs in the application are consistent.

Note I show you how to unit test routing configurations in Chapter 15.

The `Url.Action` method is the most convenient way of generating outgoing links. In the previous chapter, I helper method in the `List` view in order to display the page links. Now that I have added support for category filte go back and pass this information to the helper method, as shown in Listing 8-4.

Listing 8-4. Adding Category Information to the Pagination Links in the List.cshtml File

```
@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products) {
    @Html.Partial("ProductSummary", p)
}

<div class="btn-group pull-right">
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List",
        new { page = x, category = Model.CurrentCategory }))
</div>
```

Prior to this change, the links generated for the pagination links were like this:

http://<myserver>:<port>/Page1

If the user clicked a page link like this, the category filter he applied would be lost, and he would be presented with containing products from all categories. By adding the current category, taken from the view model, I generate URLs instead:

http:// <myserver> : <port> /Chess/Page1

When the user clicks this kind of link, the current category will be passed to the `List` action method, and the fil preserved. After you have made this change, you can visit a URL such as `/Chess` or `/Soccer`, and you will se page link at the bottom of the page correctly includes the category.

Building a Category Navigation Menu

I need to provide customers with a way to select a category that does not involve typing in URLs. This means preser with a list of the categories available and indicating which, if any, is currently selected. As I build out the application, list of categories in more than one controller, so I need something that is self-contained and reusable.

The ASP.NET MVC Framework has the concept of *child actions*, which are perfect for creating items such as a r navigation control. A child action relies on the HTML helper method called `Html.Action`, which lets you inclu from an arbitrary action method in the current view. In this case, I can create a new controller (I will call it `NavController`) with an action method (which I will call `Menu`) that renders a navigation menu. I will then `Html.Action` helper method to inject the output from that method into the layout.

This approach gives me a real controller that can contain whatever application logic I need and that can be unit tes other controller. It is a nice way of creating smaller segments of an application while preserving the overall MVC Fra approach.

Creating the Navigation Controller

Right-click the `Controllers` folder in the `SportsStore.WebUI` project and select `Add` ➤ `Contro` the pop-up menu. Select `MVC 5 Controller - Empty` from the list, click the `Add` button, set the cont `NavController` and click the `Add` button to create the `NavController.cs` class file. Remove the `I` method that Visual Studio adds to new controllers by default and add a new action method called `Menu`, as shown i

Listing 8-5. Adding The Menu Action Method to the NavController.cs File

```
using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers {
    public class NavController : Controller {
```

eBooks-IT.org

```
        public string Menu() {
            return "Hello from NavController";
        }
    }
}
```

This method returns a static message string but it is enough to get me started while I integrate the child action into the application. I want the category list to appear on all pages, so I am going to render the child action in the layout, r a specific view. Edit the `Views/Shared/_Layout.cshtml` file so that it calls the `Html.Action` h as shown in Listing 8-6.

Listing 8-6. Adding the RenderAction Call to the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-sca
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="navbar navbar-inverse" role="navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

I removed the placeholder text and replaced it with a call to the `Html.Action` method. The parameters to this the name of the action method I want to call (`Menu`) and the controller that contains it (`Nav`). If you run the applic will see that the output of the `Menu` action method is included in the response sent to the browser, as shown in Figu

eBooks-IT.org

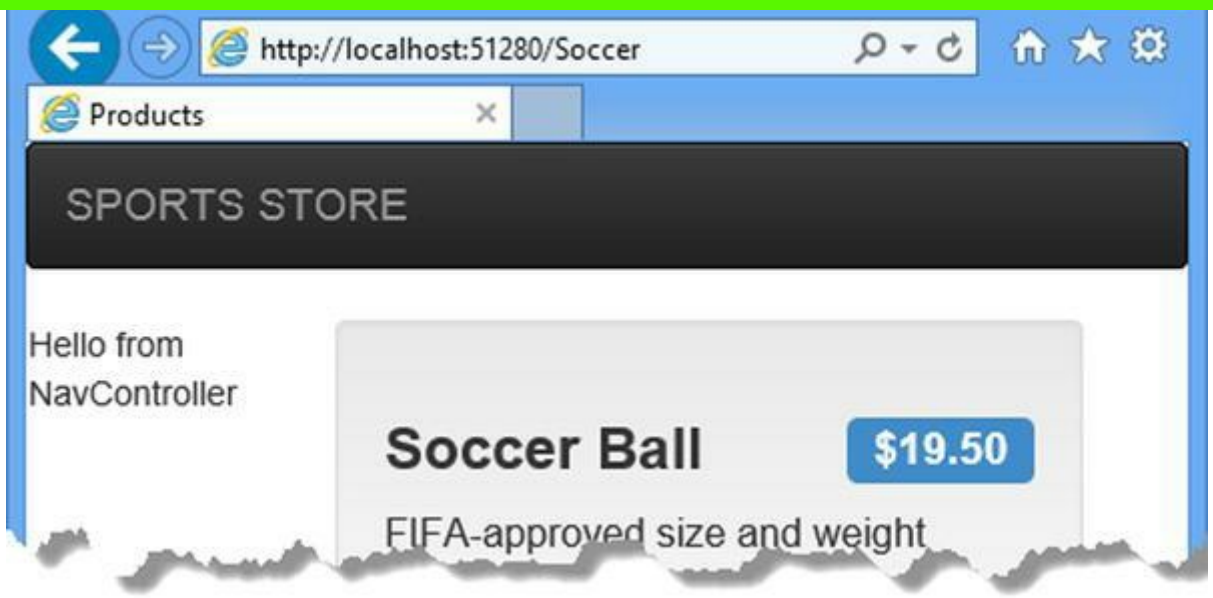


Figure 8-2. Displaying the output from the Menu action method

Generating Category Lists

I can now return to the `Nav` controller and generate a real set of categories. I do not want to generate the category l controller. Instead, I am going to use a helper method in the view to do that. All I am going to do in the `Menu` actio create the list of categories, which I have done in Listing 8-7.

Listing 8-7. Implementing the Menu Method in the NavController.cs File

```
using System.Collections.Generic;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using System.Linq;

namespace SportsStore.WebUI.Controllers {

    public class NavController : Controller {
        private IProductRepository repository;

        public NavController(IProductRepository repo) {
            repository = repo;
        }

        public PartialViewResult Menu() {
            IEnumerable<string> categories = repository.Products
                                    .Select(x => x.Category)
                                    .Distinct()
                                    .OrderBy(x => x);

            return PartialView(categories);
        }
    }
}
```

The first change is to add a constructor that accepts an `IProductRepository` implementation as its argu

eBooks-IT.org

The second change is to the Menu action method, which now uses a LINQ query to obtain a list of categories from repository and passes them to the view. Notice that, since I am working with a partial view in this controller, I call th PartialView method in the action method and that the result is a PartialViewResult object.

UNIT TEST: GENERATING THE CATEGORY LIST

The unit test for my ability to produce a category list is relatively simple. My goal is to create a list that is sorted alphabetical order and contains no duplicates. The simplest way to do this is to supply some test data that *does* h categories and that is *not* in order, pass this to the NavController, and assert that the data has been prop up. Here is the unit test:

```
...
[TestMethod]
public void Can_Create_Categories() {

  // Arrange
  // - create the mock repository
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Apples"
    new Product {ProductID = 2, Name = "P2", Category = "Apples"
    new Product {ProductID = 3, Name = "P3", Category = "Plums"}
    new Product {ProductID = 4, Name = "P4", Category = "Oranges
  });

  // Arrange - create the controller
  NavController target = new NavController(mock.Object);

  // Act = get the set of categories
  string[] results =
((IEnumerable<string>)target.Menu().Model).ToArray();

  // Assert
  Assert.AreEqual(results.Length, 3);
  Assert.AreEqual(results[0], "Apples");
  Assert.AreEqual(results[1], "Oranges");
  Assert.AreEqual(results[2], "Plums");
}
...
```

I created a mock repository implementation that contains repeating categories and categories that are not in orde that the duplicates are removed and that alphabetical ordering is imposed.

Creating the View

To create the view for the Menu action method, right-click on the Views/Nav folder and select Add ➤ MVC Page (Razor) from the pop-up menu. Set the name to Menu and click the OK button to create the Menu. file. Remove the contents that Visual Studio adds to new views and set the content to match Listing 8-8.

Listing 8-8. The Contents of the Menu.cshtml File

```
@model IEnumerable<string>

@Html.ActionLink("Home", "List", "Product", null,
    new { @class = "btn btn-block btn-default btn-lg" })
```

eBooks-IT.org

```
@foreach (var link in Model) {
    @Html.RouteLink(link, new {
        controller = "Product",
        action = "List",
        category = link,
        page = 1
    }, new {
        @class = "btn btn-block btn-default btn-lg"
    })
}
```

I added a link called `Home` that will appear at the top of the category list and will list all of the products with no c I did this using the `ActionLink` helper method, which generates an HTML anchor element using the routing inf configured earlier.

I then enumerated the category names and created links for each of them using the `RouteLink` method. This `ActionLink`, but it lets me supply a set of name/value pairs that are taken into account when generating the UR routing configuration. Do not worry if all this talk of routing does not make sense yet. I explain everything in depth i and 16.

The links I generate will look pretty ugly by default, so I have supplied an object to both the `ActionLink` and `RouteLink` helper methods that specifies values for attributes on the elements that are created. The objects I cre the `class` attribute (prefixed with a `@` because `class` is a reserved C# keyword) and apply Bootstrap classes to links as large buttons.

You can see the category links if you run the application, as shown in Figure 8-3. If you click a category, the list c updated to show only items from the selected category.

eBooks-IT.org

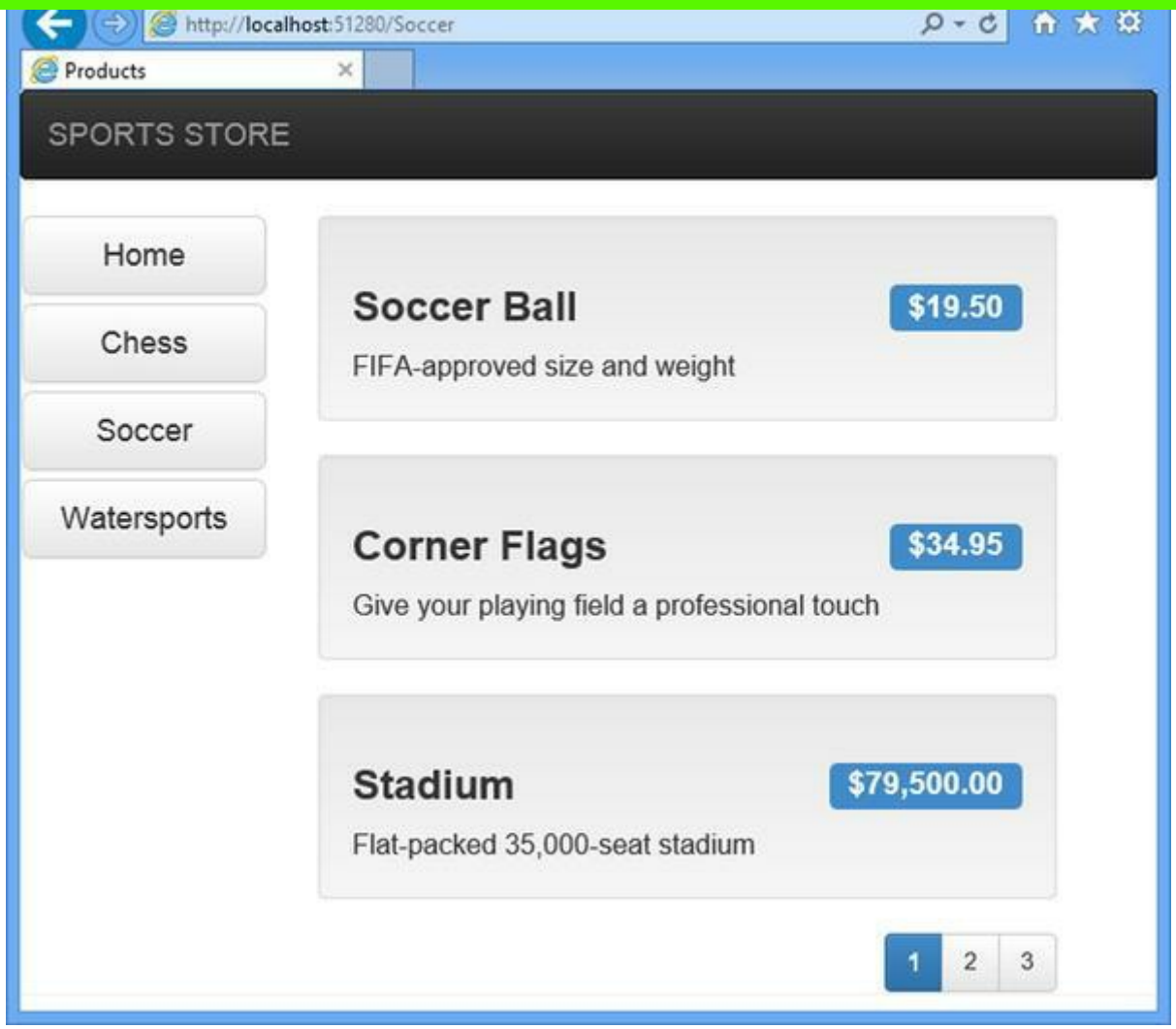


Figure 8-3. The category links

Highlighting the Current Category

At present, I do not indicate to users which category they are viewing. It might be something that the customer coul the items in the list, but I would prefer to provide solid visual feedback. I could do this by creating a view model that list of categories and the selected category, and in fact, this is exactly what I would usually do. But for variety I am g the view bag feature I introduced in Chapter 2. This feature allows me to pass data from the controller to the view w view model. Listing 8-9 shows the changes to the `Menu` action method in the `Nav` controller.

Listing 8-9. Using the View Bag Feature in the NavController.cs File

```
using System.Collections.Generic;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using System.Linq;

namespace SportsStore.WebUI.Controllers {

    public class NavController : Controller {
        private IProductRepository repository;

        public NavController(IProductRepository repo) {
```

```
        }

        public PartialViewResult Menu(string category = null) {

            ViewBag.SelectedCategory = category;

            IEnumerable<string> categories = repository.Products
                                    .Select(x => x.Category)
                                    .Distinct()
                                    .OrderBy(x => x);

            return PartialView(categories);
        }
    }
}
```

I added a parameter to the `Menu` action method called `category`. The value for this parameter will be provid automatically by the routing configuration. Inside the method body, I have dynamically assigned a `SelectedCa` property to the `ViewBag` object and set its value to be the current category. As I explained in Chapter 2, the `Vie` dynamic object and I create new properties simply by setting values for them.

UNIT TEST: REPORTING THE SELECTED CATEGORY

I can test that the `Menu` action method correctly adds details of the selected category by reading the value of t `ViewBag` property in a unit test, which is available through the `ViewResult` class. Here is the test:

```
...
[TestMethod]
public void Indicates_Selected_Category() {

  // Arrange
  // - create the mock repository
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Apples"
    new Product {ProductID = 4, Name = "P2", Category = "Oranges
  });

  // Arrange - create the controller
  NavController target = new NavController(mock.Object);

  // Arrange - define the category to selected
  string categoryToSelect = "Apples";

  // Action
  string result =
target.Menu(categoryToSelect).ViewBag.SelectedCategory;

  // Assert
  Assert.AreEqual(categoryToSelect, result);
}
...
```

This unit test will not compile unless you add a reference to the `Microsoft.CSharp` assembly, as descr previous chapter.

eBooks-IT.org

tage of

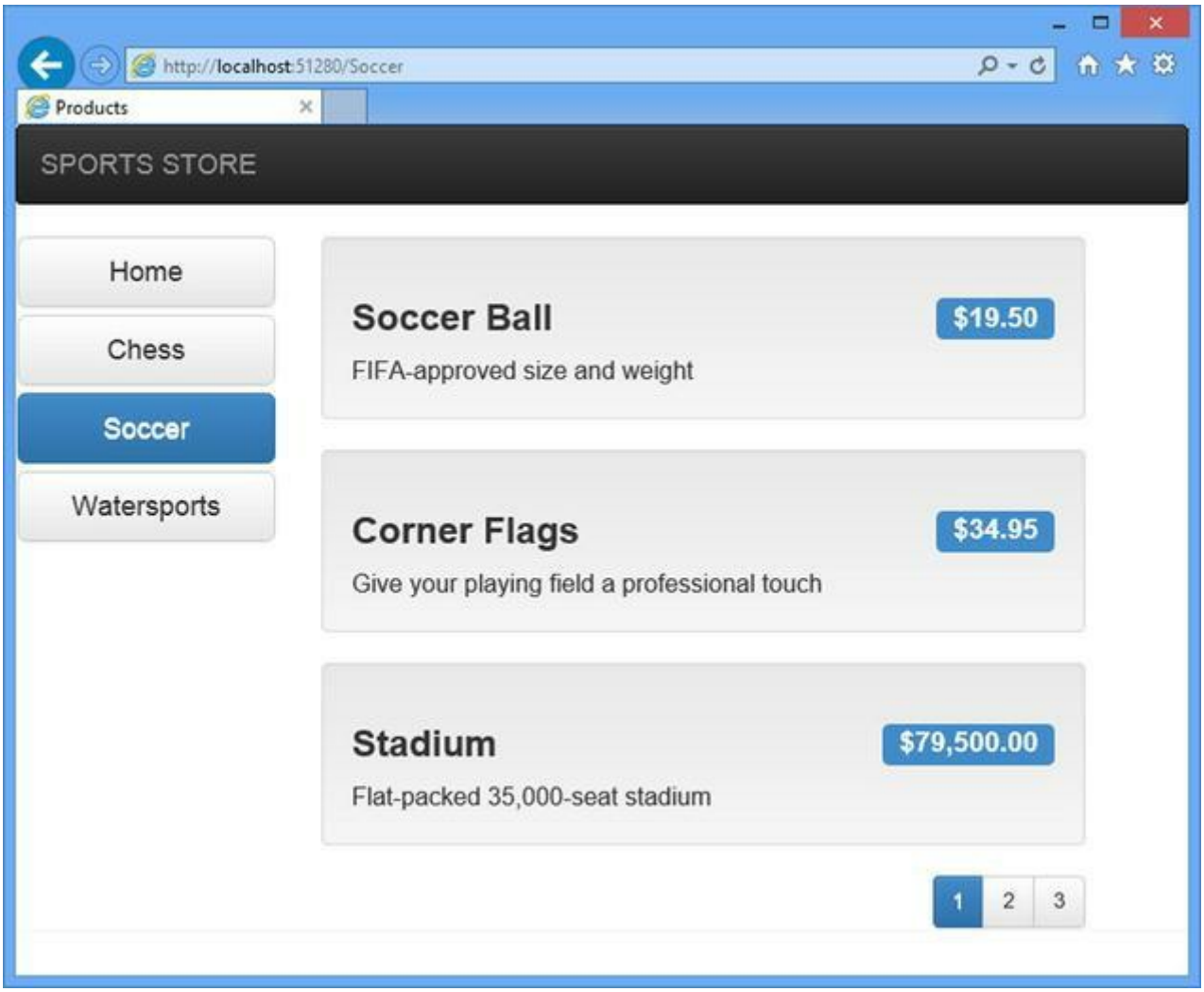
a CSS class to the HTML anchor element that represents the selected category. Listing 8-10 shows the changes to th

Menu.cshtml file.

Listing 8-10. Highlighting the Selected Category in the Menu.cshtml File

```
@model IEnumerable<string>

@Html.ActionLink("Home", "List", "Product", null,
    new { @class = "btn btn-block btn-default btn-lg" })

@foreach (var link in Model) {
    @Html.RouteLink(link, new {
        controller = "Product",
        action = "List",
        category = link,
        page = 1
    }, new {
        @class = "btn btn-block btn-default btn-lg"
            + (link == ViewBag.SelectedCategory ? " btn-primary" :
    })
}
```

The change is simple. If the current `link` value matches the `SelectedCategory` value, then I add the el creating to another Bootstrap class, which will cause the button to be highlighted. Running the application shows the category highlighting, which you can also see in Figure 8-4.



eBooks-IT.org

Correcting the Page Count

I need to correct the page links so that they work correctly when a category is selected. Currently, the number of pa determined by the total number of products in the repository and not the number of products in the selected category that the customer can click the link for page 2 of the `Chess` category and end up with an empty page because ther enough chess products to fill two pages. You can see the problem in Figure 8-5.

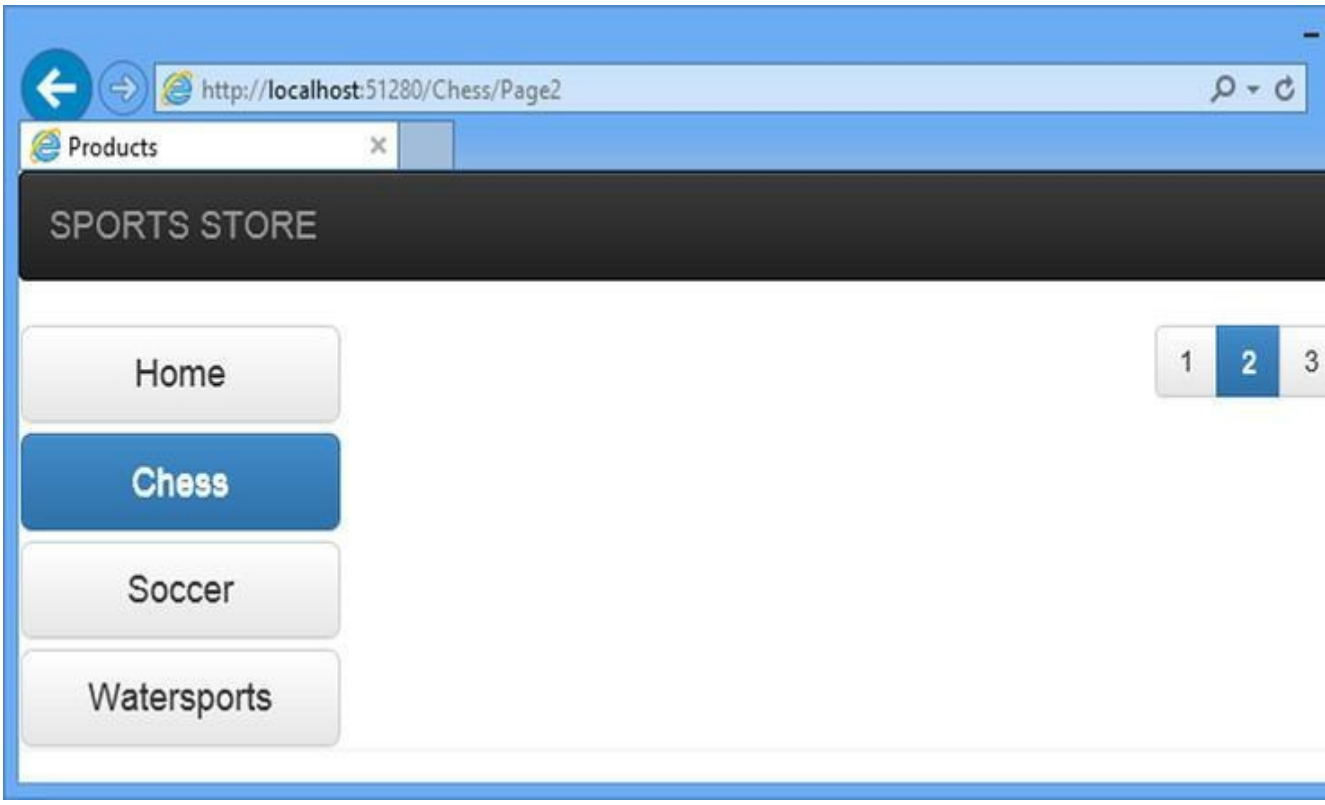


Figure 8-5. Displaying the wrong page links when a category is selected

I can fix this by updating the `List` action method in the `Product` controller so that the pagination informatio categories into account. You can see the required changes in Listing 8-11.

Listing 8-11. Creating Category-Aware Pagination Data in the ProductController.cs File

```
...
public ViewResult List(string category, int page = 1) {

    ProductsListViewModel viewModel = new ProductsListViewModel {
        Products = repository.Products
            .Where(p => category == null || p.Category == category
            .OrderBy(p => p.ProductID)
            .Skip((page - 1) * PageSize)
            .Take(PageSize),
        PagingInfo = new PagingInfo {
            CurrentPage = page,
            ItemsPerPage = PageSize,
            TotalItems = category == null ?
                repository.Products.Count() :
                                repository.Products.Where(e => e.Cate
category).Count()
        },
```

```
    };
    return View(viewModel);
}
...
```

If a category has been selected, I return the number of items in that category; if not, I return the total number of p
when I view a category, the links at the bottom of the page correctly reflect the number of products in the category,
Figure 8-6.

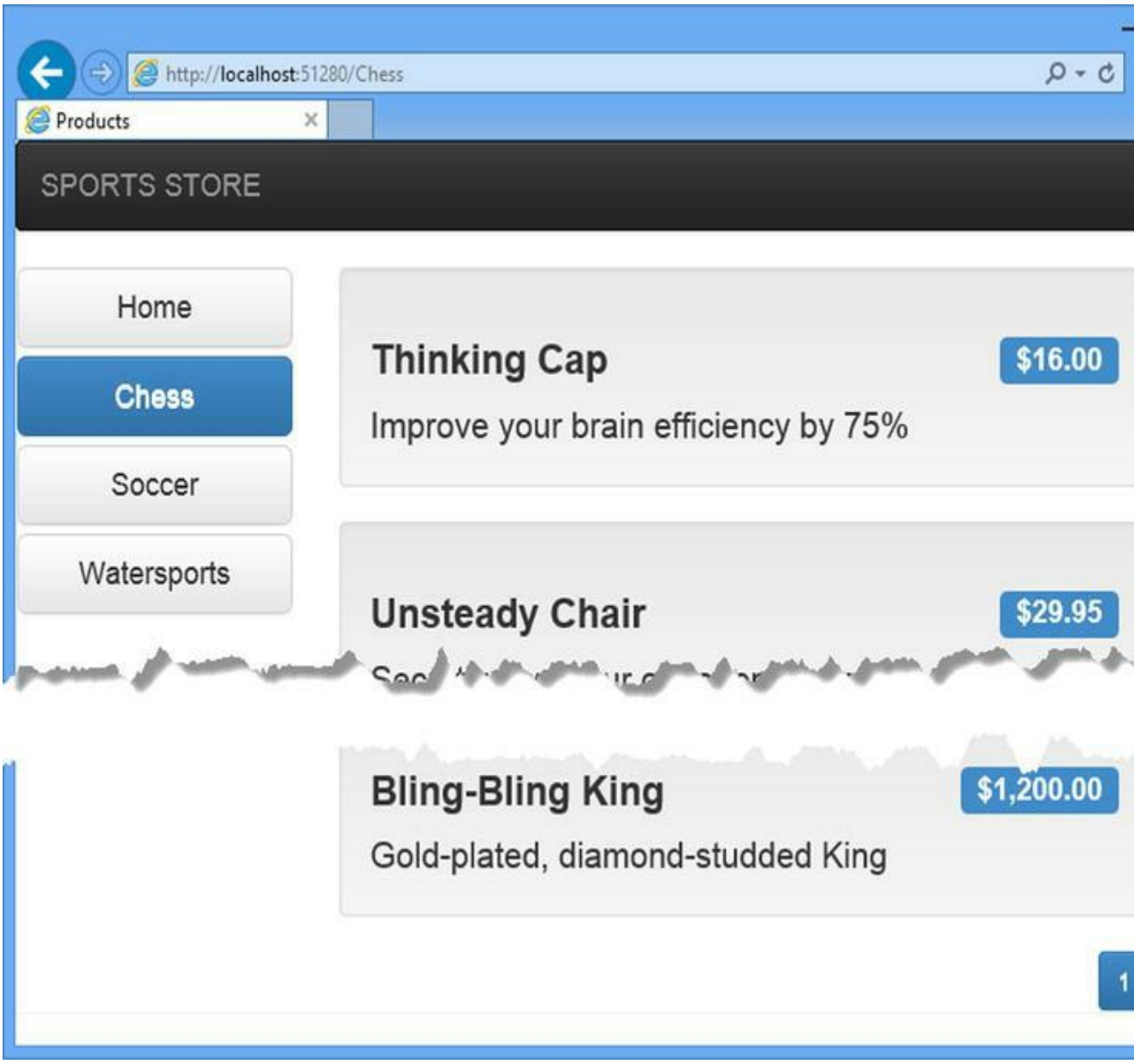


Figure 8-6. Displaying category-specific page counts

UNIT TEST: CATEGORY-SPECIFIC PRODUCT COUNTS

Testing that I am able to generate the current product count for different categories is simple. I create a mock r
contains known data in a range of categories and then call the `List` action method requesting each category i
the unit test:

```
...
[TestMethod]
public void Generate_Category_Specific_Product_Count() {
```

eBooks-IT.org

```
    // Create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>()
    mock.Setup(m => m.Products).Returns(new Product[] {
      new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
      new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
      new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
      new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
      new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
    });

    // Arrange - create a controller and make the page size 3 item
    ProductController target = new ProductController(mock.Object);
    target.PageSize = 3;

    // Action - test the product counts for different categories
    int res1 = ((ProductsListViewModel)target
      .List("Cat1").Model).PagingInfo.TotalItems;
    int res2 = ((ProductsListViewModel)target
      .List("Cat2").Model).PagingInfo.TotalItems;
    int res3 = ((ProductsListViewModel)target
      .List("Cat3").Model).PagingInfo.TotalItems;
    int resAll = ((ProductsListViewModel)target
      .List(null).Model).PagingInfo.TotalItems;

    // Assert
    Assert.AreEqual(res1, 2);
    Assert.AreEqual(res2, 2);
    Assert.AreEqual(res3, 1);
    Assert.AreEqual(resAll, 5);
  }
  ...
```

Notice that I also call the `List` method, specifying no category, to make sure I get the right total count as wel

Building the Shopping Cart

The application is progressing nicely, but I cannot sell any products until I implement a shopping cart. In this section the shopping cart experience shown in Figure 8-7. This will be familiar to anyone who has ever made a purchase onl

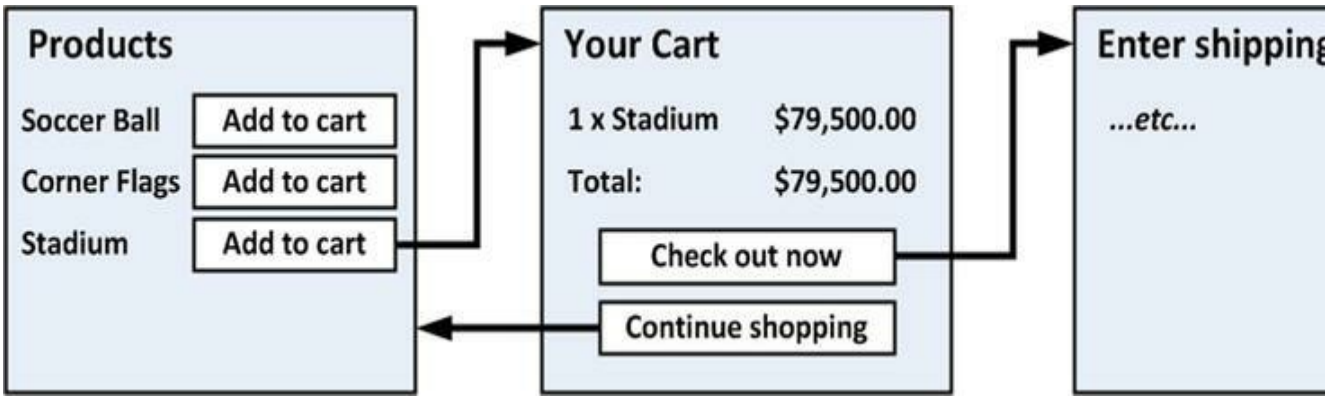


Figure 8-7. The basic shopping cart flow

An `Add to Cart` button will be displayed alongside each of the products in the catalog. Clicking this button summary of the products the customer has selected so far, including the total cost. At this point, the user can click t

eBooks-IT.org

order and finish the shopping session.

Defining the Cart Entity

A shopping cart is part of the business domain, so it makes sense to represent a cart by creating an entity in the dom
Add a class file called `Cart.cs` to the `Entities` folder in the `SportsStore.Domain` project and use
the classes shown in Listing 8-12.

Listing 8-12. The Cart and CartLine Classes in the Cart.cs File

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

namespace SportsStore.Domain.Entities {

    public class Cart {
        private List<CartLine> lineCollection = new List<CartLine>

        public void AddItem(Product product, int quantity) {
            CartLine line = lineCollection
                .Where(p => p.Product.ProductID == product.Product
                .FirstOrDefault();

            if (line == null) {
                lineCollection.Add(new CartLine { Product = produc
                    Quantity = quantity });
            } else {
                line.Quantity += quantity;
            }
        }

        public void RemoveLine(Product product) {
                    lineCollection.RemoveAll(l  =>  l.Product.Produ
product.ProductID);
        }

        public decimal ComputeTotalValue() {
            return lineCollection.Sum(e => e.Product.Price * e.Qua

        }
        public void Clear() {
            lineCollection.Clear();
        }

        public IEnumerable<CartLine> Lines {
            get { return lineCollection; }
        }
    }

    public class CartLine {
        public Product Product { get; set; }
        public int Quantity { get; set; }
    }
}
```

eBooks-IT.org

quantity the user wants to buy. I defined methods to add an item to the cart, remove a previously added item from the cus calculate the total cost of the items in the cart, and reset the cart by removing all of the items. I also provided a prope access to the contents of the cart using an `IEnumerable<CartLine>`. This is all straightforward stuff, eas implemented in C# with the help of a little LINQ.

UNIT TEST: TESTING THE CART

The `Cart` class is relatively simple, but it has a range of important behaviors that must work properly. A poorl cart would undermine the entire SportsStore application. I have broken down the features and tested them indiv created a new unit test file in the `SportsStore.UnitTests` project called `CartTests.cs` to co tests.

The first behavior relates to when I add an item to the cart. If this is the first time that a given `Product` has the cart, I want a new `CartLine` to be added. Here is the test, including the unit test class definition:

```
using System. Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using SportsStore.Domain.Entities;

namespace SportsStore.UnitTests {
  [TestClass]
  public class CartTests {

    [TestMethod]
    public void Can_Add_New_Lines() {

      // Arrange - create some test products
      Product p1 = new Product { ProductID = 1, Name = "P1" };
      Product p2 = new Product { ProductID = 2, Name = "P2" };

      // Arrange - create a new cart
      Cart target = new Cart();

      // Act
      target.AddItem(p1, 1);
      target.AddItem(p2, 1);
      CartLine[] results = target.Lines.ToArray();

      // Assert
      Assert.AreEqual(results.Length, 2);
      Assert.AreEqual(results[0].Product, p1);
      Assert.AreEqual(results[1].Product, p2);
    }
  }
}
```

However, if the customer has already added a `Product` to the cart, I want to increment the quantity of the c `CartLine` and not create a new one. Here is the test:

```
..
[TestMethod]
public void Can_Add_Quantity_For_Existing_Lines() {

  // Arrange - create some test products
  Product p1 = new Product { ProductID = 1, Name = "P1" };
  Product p2 = new Product { ProductID = 2, Name = "P2" };
```

eBooks-IT.org

```
  // Arrange - create a new cart
  Cart target = new Cart();

  // Act
  target.AddItem(p1, 1);
  target.AddItem(p2, 1);
  target.AddItem(p1, 10);
  CartLine[] results = target.Lines.OrderBy(c =>
c.Product.ProductID).ToArray();

  // Assert
  Assert.AreEqual(results.Length, 2);
  Assert.AreEqual(results[0].Quantity, 11);
  Assert.AreEqual(results[1].Quantity, 1);
}
...
```

I also need to check that users can change their mind and remove products from the cart. This feature is impler
`RemoveLine` method. Here is the test:

```
...
[TestMethod]
public void Can_Remove_Line() {

  // Arrange - create some test products
  Product p1 = new Product { ProductID = 1, Name = "P1" };
  Product p2 = new Product { ProductID = 2, Name = "P2" };
  Product p3 = new Product { ProductID = 3, Name = "P3" };

  // Arrange - create a new cart
  Cart target = new Cart();

  // Arrange - add some products to the cart
  target.AddItem(p1, 1);
  target.AddItem(p2, 3);
  target.AddItem(p3, 5);
  target.AddItem(p2, 1);

  // Act
  target.RemoveLine(p2);

  // Assert
  Assert.AreEqual(target.Lines.Where(c => c.Product == p2).Count
  Assert.AreEqual(target.Lines.Count(), 2);
}
...
```

The next behavior I want to test is the ability to calculate the total cost of the items in the cart. Here's the test fo
behavior:

```
...
[TestMethod]
public void Calculate_Cart_Total() {

  // Arrange - create some test products
  Product p1 = new Product { ProductID = 1, Name = "P1", Price =
```

```
    // Arrange - create a new cart
    Cart target = new Cart();

    // Act
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);
    target.AddItem(p1, 3);
    decimal result = target.ComputeTotalValue();

    // Assert
    Assert.AreEqual(result, 450M);
  }
  ...
```

The final test is simple. I want to ensure that the contents of the cart are properly removed when reset. Here is t

```
  ...
  [TestMethod]
  public void Can_Clear_Contents() {

    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1", Price =
  };
    Product p2 = new Product { ProductID = 2, Name = "P2", Price =

    // Arrange - create a new cart
    Cart target = new Cart();

    // Arrange - add some items
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);

    // Act - reset the cart
    target.Clear();

    // Assert
    Assert.AreEqual(target.Lines.Count(), 0);
  }
  ...
```

Sometimes, as in this case, the code required to test the functionality of a type is longer and more complex than itself. Do not let that put you off writing the unit tests. Defects in simple classes can have huge impacts, especia play such an important role as `Cart` does in the example application.

Adding the Add to Cart Buttons

I need to edit the `Views/Shared/ProductSummary.cshtml` view to add the buttons to the product changes are shown in Listing 8-13.

Listing 8-13. Adding the Buttons to the Product Summary.cshtml File View

```
@model SportsStore.Domain.Entities.Product

<div class="well">
```

eBooks-IT.org

```
        <strong>@Model.Name</strong>
                                <span    class="pull-right    label
primary">@Model.Price.ToString("c")</span>
    </h3>

    @using (Html.BeginForm("AddToCart", "Cart")) {
        <div class="pull-right">
            @Html.HiddenFor(x => x.ProductID)
            @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
               <input type="submit" class="btn btn-success" value=
cart" />
        </div>
    }

    <span class="lead"> @Model.Description</span>
</div>
```

I added a Razor block that creates a small HTML form for each product in the listing. When this form is submitted invoke the `AddToCart` action method in the `Cart` controller (which I will implement in just a moment).

Note By default, the `BeginForm` helper method creates a form that uses the HTTP `POST` method. You can so that forms use the `GET` method, but you should think carefully about doing so. The HTTP specification requires requests must be *idempotent*, meaning that they must not cause changes, and adding a product to a cart is definitely have more to say on this topic in Chapter 16, including an explanation of what can happen if you ignore the need for `GET` requests.

CREATING MULTIPLE HTML FORMS IN A PAGE

Using the `Html.BeginForm` helper in each product listing means that every `Add to cart` button is its own separate HTML `form` element. This may be surprising if you have been developing with ASP.NET We which imposes a limit of one form per page if you want to use the view state feature or complex controls (whic on view state). Since ASP.NET MVC does not use view state, there is no limit the number of forms you can cr

Equally, there is no requirement to create a form for each button. However, since each form will post back to th controller method, but with a different set of parameter values, it is a nice and simple way to deal with the butto

Implementing the Cart Controller

I need a controller to handle the `Add to cart` button presses. Create a new controller called `CartContro` `SportsStore.WebUI` project and edit the content so that it matches Listing 8-14.

Listing 8-14. The Contents of the CartController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

        public CartController(IProductRepository repo) {
            repository = repo;
```

```
            public RedirectToRouteResult AddToCart(int productId,
returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null) {
                GetCart().AddItem(product, 1);
            }
            return RedirectToAction("Index", new { returnUrl });
        }

         public RedirectToRouteResult RemoveFromCart(int productId
returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null) {
                GetCart().RemoveLine(product);
            }
            return RedirectToAction("Index", new { returnUrl });
        }

        private Cart GetCart() {

            Cart cart = (Cart)Session["Cart"];
            if (cart == null) {
                cart = new Cart();
                Session["Cart"] = cart;
            }
            return cart;
        }
    }
}
```

There are a few points to note about this controller. The first is that I use the ASP.NET session state feature to sto retrieve `Cart` objects. This is the purpose of the `GetCart` method. ASP.NET has a nice session feature that use URL rewriting to associate multiple requests from a user together to form a single browsing session. A related featur state, which associates data with a session. This is an ideal fit for the `Cart` class. I want each user to have their ow want the cart to be persistent between requests. Data associated with a session is deleted when a session expires (typ because a user has not made a request for a while), which means that I do not need to manage the storage or life cyc `Cart` objects. To add an object to the session state, I set the value for a key on the `Session` object, like this:

```
...
Session["Cart"] = cart;
...
```

To retrieve an object again, I simply read the same key, like this:

```
...
Cart cart = (Cart)Session["Cart"];
...
```

Tip Session state objects are stored in the memory of the ASP.NET server by default, but you can configure a ra different storage approaches, including using a SQL database. See my book, *Pro ASP.NET MVC 5 Platform*, publish in 2014, for details.

eBooks-IT.org

in the HTML forms created in the ProductSummary.cshtml view. This allows the MVC Framework to a e inp incoming form POST variables with those parameters, meaning I do not need to process the form myself.

Displaying the Contents of the Cart

The final point to note about the Cart controller is that both the AddToCart and RemoveFromCart metl RedirectToAction method. This has the effect of sending an HTTP redirect instruction to the client brows browser to request a new URL. In this case, I have asked the browser to request a URL that will call the Index ac of the Cart controller.

I am going to implement the Index method and use it to display the contents of the Cart. If you refer back to you will see that this is the workflow when the user clicks the Add to cart button.

I need to pass two pieces of information to the view that will display the contents of the cart: the Cart object an display if the user clicks the Continue shopping button. I created a new class file called CartIndexViewModel.cs in the Models folder of the SportsStore.WebUI project. The cont file are shown in Listing 8-15.

Listing 8-15. The Contents of the CartIndexViewModel.cs File

```
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models {
    public class CartIndexViewModel {
        public Cart Cart { get; set; }
        public string ReturnUrl { get; set; }
    }
}
```

Now that I have the view model, I can implement the Index action method in the Cart controller class, as sh 8-16.

Listing 8-16. The Index Action Method in the CartController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

        public CartController(IProductRepository repo) {
            repository = repo;
        }

        public ViewResult Index(string returnUrl) {
            return View(new CartIndexViewModel {
                Cart = GetCart(),
                ReturnUrl = returnUrl
            });
        }
```

```
    }

}
```

The last step to display the contents of the cart is to create the new view. Right-click on the `Index` action meth
`Add View` from the pop-up menu. Set the name to `Index` and click the `OK` button to create the `Index.cs`
file. Edit the view to match the contents shown in Listing 8-17.

Listing 8-17. The Contents of the Index.cshtml File

```
@model SportsStore.WebUI.Models.CartIndexViewModel

@{
    ViewBag.Title = "Sports Store: Your Cart";
}

<h2>Your cart</h2>
<table class="table">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.Lines) {
            <tr>
                <td class="text-center">@line.Quantity</td>
                <td class="text-left">@line.Product.Name</td>
                  <td class="text-right">@line.Product.Price.ToStr
</td>
                <td class="text-right">
                    @((line.Quantity * line.Product.Price).ToStrin
                </td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3" class="text-right">Total:</td>
            <td class="text-right">
                @Model.Cart.ComputeTotalValue().ToString("c")
            </td>
        </tr>
    </tfoot>
</table>

<div class="text-center">
          <a   class="btn   btn-primary"   href="@Model.ReturnUrl">
shopping</a>
</div>
```

The view enumerates the lines in the cart and adds rows for each of them to an HTML table, along with the total c

eBooks-IT.org

tables a
alignment. I now have the basic functions of the shopping cart in place. First, products are listed along with a buttor
to the cart, as shown in Figure 8-8.

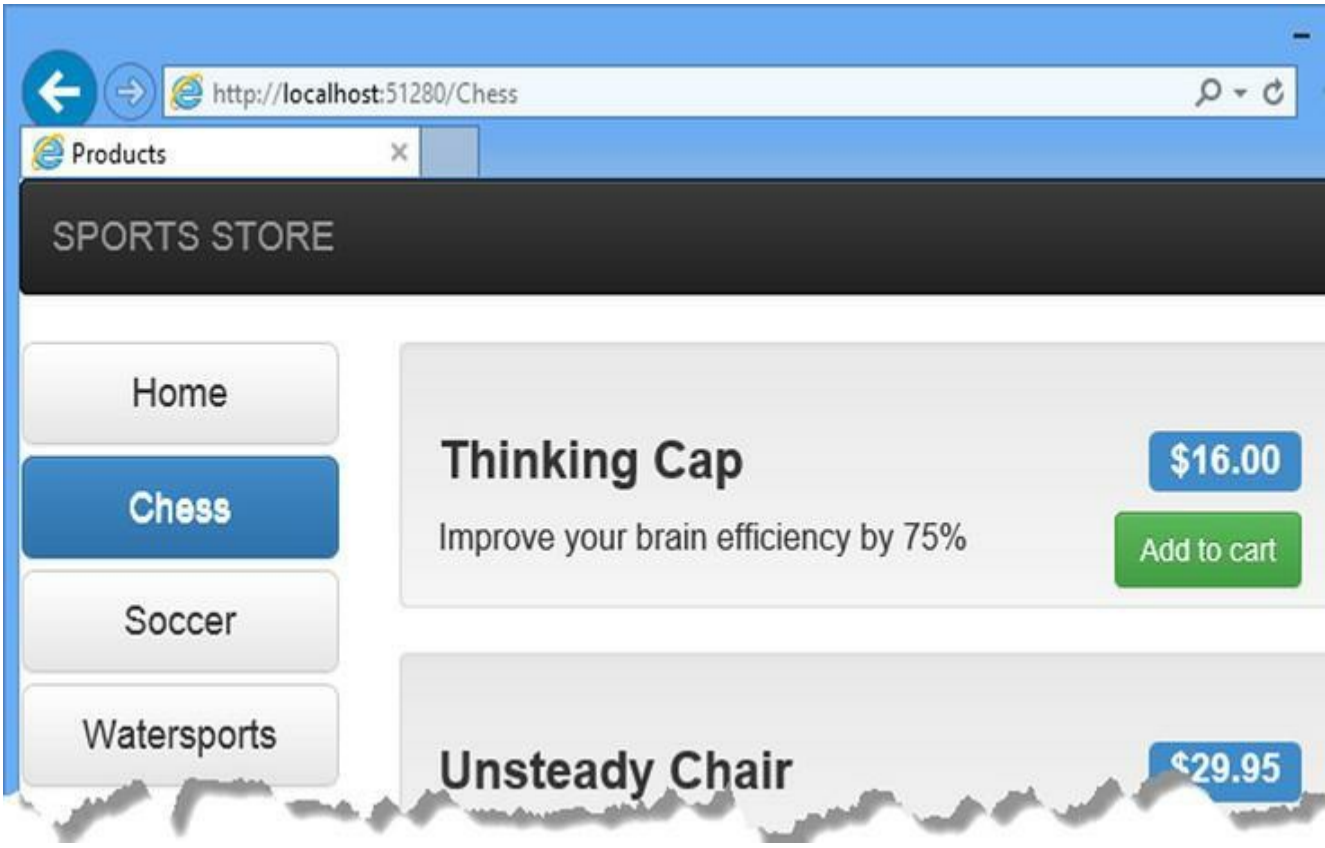


Figure 8-8. The Add to cart button

And second, when the user clicks the `Add to cart button`, the appropriate product is added to their ca
summary of the cart is displayed, as shown in Figure 8-9. And clicking the `Continue shopping` button retu
to the product page they came from.

eBooks-IT.org

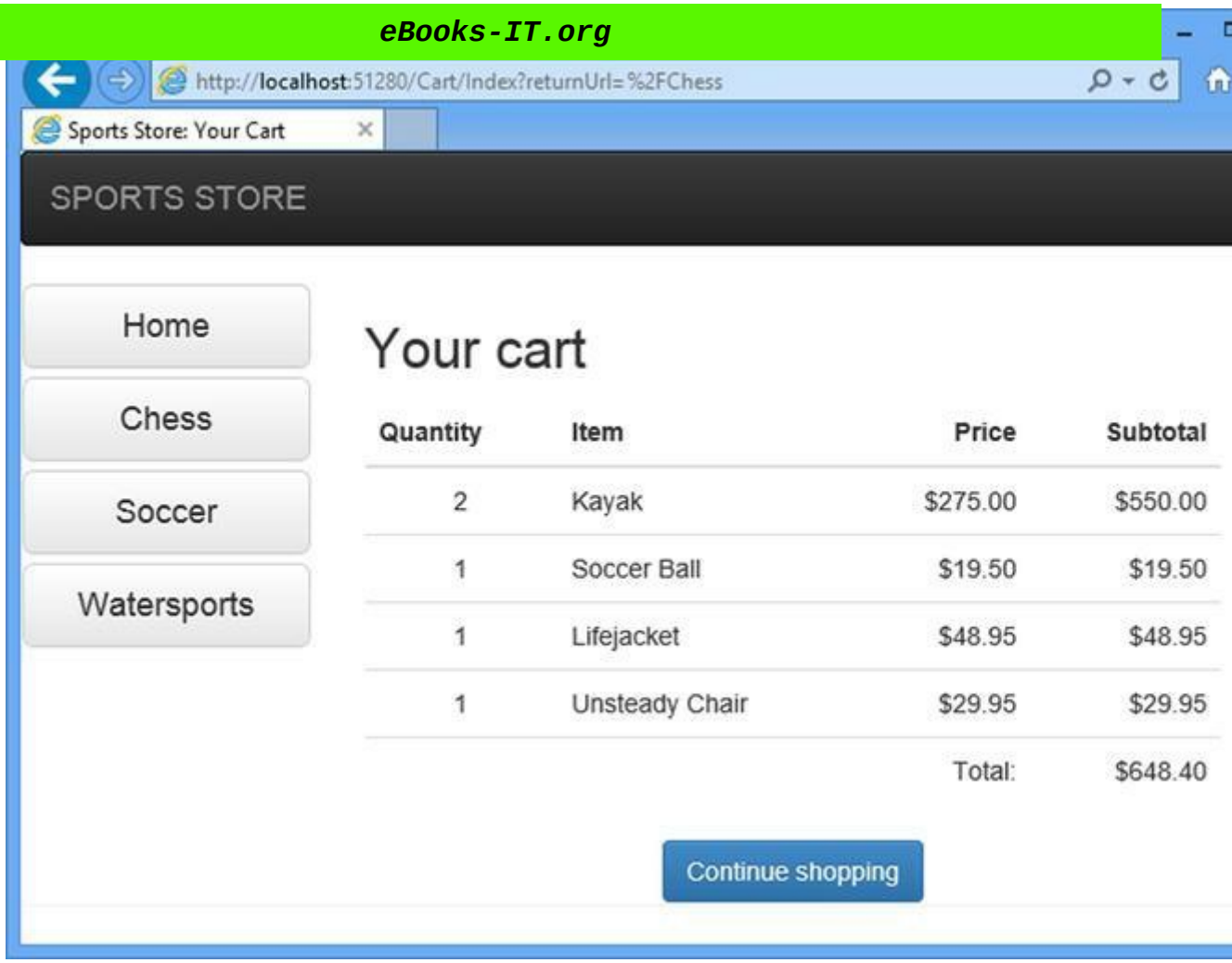


Figure 8-9. Displaying the contents of the shopping cart

Summary

In this chapter, I started to flesh out the customer-facing parts of the SportsStore app. I provided the means by whic can navigate by category and put the basic building blocks in place for adding items to a shopping cart. I have more and I continue the development of the application in the next chapter.

eBooks-IT.org

SportsStore: Completing the Cart

In this chapter, I continue to build out the SportsStore example app. In the previous chapter, I added the basic suppo shopping cart and now I am going to improve on and complete that functionality.

Using Model Binding

The MVC Framework uses a system called *model binding* to create C# objects from HTTP requests in order to pass parameter values to action methods. This is how the MVC framework processes forms, for example: it looks at the p the action method that has been targeted and uses a *model binder* to get the form values sent by the browser and con the type of the parameter with the same name before passing them to the action method.

Model binders can create C# types from any information that is available in the request. This is one of the central f MVC Framework. I am going to create a custom model binder to improve the `CartController` class.

I like using the session state feature in the `Cart` controller to store and manage the `Cart` objects that I set up i but I do not like the way I have to go about it. It does not fit the rest of the application model, which is based around method parameters. I cannot properly unit test the `CartController` class unless I mock the `Session` par base class, and that means mocking a whole bunch of other stuff I would rather not deal with.

To solve this problem, I am going to create a custom model binder that obtains the `Cart` object contained in the The MVC Framework will then be able to create `Cart` objects and pass them as parameters to the action methods `CartController` class. The model binding feature is powerful and flexible. I go into a lot more depth about t Chapter 24, but this is a nice example to get started with.

Creating a Custom Model Binder

I create a custom model binder by implementing the `System.Web.Mvc.IModelBinder` interface. To cr implementation, I added a new folder in the `SportsStore.WebUI` project called `Infrastructure/B` and created a `CartModelBinder.cs` class file inside it. Listing 9-1 shows the contents of the new file.

Listing 9-1. The Contents of the CartModelBinder.cs File

```
using System.Web.Mvc;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Infrastructure.Binders {

    public class CartModelBinder : IModelBinder {
        private const string sessionKey = "Cart";

        public object BindModel(ControllerContext controllerContex
            ModelBindingContext bindingContext) {

            // get the Cart from the session

            Cart cart = null;
            if (controllerContext.HttpContext.Session != null) {
```

eBooks-IT.org

```
(Cart)controllerContext.HttpContext.Session[sessionKey];
            }
            // create the Cart if there wasn't one in the session
            if (cart == null) {
                cart = new Cart();
                if (controllerContext.HttpContext.Session != null)
                        controllerContext.HttpContext.Session[sessi
cart;
                }
            }
            // return the cart
            return cart;
        }
    }
}
```

The `IModelBinder` interface defines one method: `BindModel`. The two parameters are provided to mak domain model object possible. The `ControllerContext` provides access to all the information that the cont has, which includes details of the request from the client. The `ModelBindingContext` gives you informatio model object you are being asked to build and some tools for making the binding process easier.

For my purposes, the `ControllerContext` class is the one I am interested in. It has an `HttpContex` which in turn has a `Session` property that lets me get and set session data. I can obtain the `Cart` object associa user's session by reading a value from the session data, and create a `Cart` if there is not one there already.

I need to tell the MVC Framework that it can use the `CartModelBinder` class to create instances of `Car` the `Application_Start` method of `Global.asax`, as shown in Listing 9-2.

Listing 9-2. Registering the CartModelBinder Class in the Global.asax.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Infrastructure.Binders;

namespace SportsStore.WebUI {

    public class MvcApplication : System.Web.HttpApplication {

        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ModelBinders.Binders.Add(typeof(Cart), new CartModelBi
        }
    }
}
```

I can now update the `Cart` controller to remove the `GetCart` method and rely on the model binder to provide controller with `Cart` objects. Listing 9-3 shows the changes.

Listing 9-3. Relying on the Model Binder in the CartController.cs File

eBooks-IT.org

```
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

        public CartController(IProductRepository repo) {
            repository = repo;
        }

        public ViewResult Index(Cart cart, string returnUrl) {
            return View(new CartIndexViewModel {
                ReturnUrl = returnUrl,
                Cart = cart
            });
        }

        public RedirectToRouteResult AddToCart(Cart cart, int prod
                string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null) {
                cart.AddItem(product, 1);
            }
            return RedirectToAction("Index", new { returnUrl });
        }

              public RedirectToRouteResult RemoveFromCart(Cart ca
productId,
                string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null) {
                cart.RemoveLine(product);
            }
            return RedirectToAction("Index", new { returnUrl });
        }
    }
}
```

I have removed the `GetCart` method and added a `Cart` parameter to each of the action methods. When the M
Framework receives a request that requires, say, the `AddToCart` method to be invoked, it begins by looking at th
for the action method. It looks at the list of binders available and tries to find one that can create instances of each pa
The custom binder is asked to create a `Cart` object, and it does so by working with the session state feature. Betw
custom binder and the default binder, the MVC Framework is able to create the set of parameters required to call the
method, allowing me to refactor the controller so that it has no knowledge of how `Cart` objects are created when r
received.

There are several benefits to using a custom model binder like this. The first is that I have separated the logic used
`Cart` from that of the controller, which allows me to change the way I store `Cart` objects without needing to ch

eBooks-IT.org

hem as
method parameters and take advantage of the custom model binder. The third benefit, and the one I think is most imp
I can now unit test the `Cart` controller without needing to mock a lot of ASP.NET plumbing.

UNIT TEST: THE CART CONTROLLER

I can unit test the `CartController` class by creating `Cart` objects and passing them to the action meth
to test three different aspects of this controller:

- The `AddToCart` method should add the selected product to the customer's cart.
- After adding a product to the cart, the user should be redirected to the `Index` view.
- The URL that the user can follow to return to the catalog should be correctly passed to the `Index` action m

Here are the unit tests I added to the `CartTests.cs` file in the `SportsStore.UnitTests` proje

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using SportsStore.Domain.Entities;
using System.Linq;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.WebUI.Controllers;
using System.Web.Mvc;

using SportsStore.WebUI.Models;
namespace SportsStore.UnitTests {
  [TestClass]
  public class CartTests {

    // . . . existing test methods omitted for brevity. . .

    [TestMethod]
    public void Can_Add_To_Cart() {

      // Arrange - create the mock repository
      Mock<IProductRepository> mock = new Mock<IProductRepository>();
      mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Apples"},
      }.AsQueryable());

      // Arrange - create a Cart
      Cart cart = new Cart();

      // Arrange - create the controller
      CartController target = new CartController(mock.Object);

      // Act - add a product to the cart
      target.AddToCart(cart, 1, null);

      // Assert
      Assert.AreEqual(cart.Lines.Count(), 1);
      Assert.AreEqual(cart.Lines.ToArray()[0].Product.ProductID, 1);
    }

    [TestMethod]
```

```
            // Arrange - create the mock repository
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new Product[] {
                new Product {ProductID = 1, Name = "P1", Category = "Apples"},
            }.AsQueryable());

            // Arrange - create a Cart
            Cart cart = new Cart();

            // Arrange - create the controller
            CartController target = new CartController(mock.Object);

            // Act - add a product to the cart
            RedirectToRouteResult result = target.AddToCart(cart, 2, "myUrl");

            // Assert
            Assert.AreEqual(result.RouteValues["action"], "Index");
            Assert.AreEqual(result.RouteValues["returnUrl"], "myUrl");

        }

        [TestMethod]
        public void Can_View_Cart_Contents() {

            // Arrange - create a Cart
            Cart cart = new Cart();

            // Arrange - create the controller
            CartController target = new CartController(null);

            // Act - call the Index action method
            CartIndexViewModel result
                = (CartIndexViewModel)target.Index(cart, "myUrl").ViewData.Model;

            // Assert
            Assert.AreSame(result.Cart, cart);
            Assert.AreEqual(result.ReturnUrl, "myUrl");
        }
    }
}
```

Completing the Cart

Now that I have introduced the custom model binder, it is time to complete the cart functionality by adding two new first will allow the customer to remove an item from the cart. The second feature will display a summary of the cart the page.

Removing Items from the Cart

I already defined and tested the `RemoveFromCart` action method in the controller, so letting the customer rem just a matter of exposing this method in a view, which I are going to do by adding a `Remove` button in each row o summary. The changes to `Views/Cart/Index.cshtml` are shown in Listing 9-4.

```
@model SportsStore.WebUI.Models.CartIndexViewModel

@{
    ViewBag.Title = "Sports Store: Your Cart";
}

<style>
    #cartTable td { vertical-align: middle; }
</style>

<h2>Your cart</h2>
<table id="cartTable" class="table">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.Lines) {
            <tr>
                <td class="text-center">@line.Quantity</td>
                <td class="text-left">@line.Product.Name</td>
                  <td class="text-right">@line.Product.Price.ToStr
</td>
                <td class="text-right">
                    @((line.Quantity * line.Product.Price).ToStrin
                </td>
                <td>
                    @using (Html.BeginForm("RemoveFromCart", "Cart
                        @Html.Hidden("ProductId", line.Product.Pro
                        @Html.HiddenFor(x => x.ReturnUrl)
                        <input class="btn btn-sm btn-warning"
                            type="submit" value="Remove" />
                    }
                </td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3" class="text-right">Total:</td>
            <td class="text-right">
                @Model.Cart.ComputeTotalValue().ToString("c")
            </td>
        </tr>
    </tfoot>
</table>

<div class="text-center">
          <a   class="btn   btn-primary"   href="@Model.ReturnUrl">
```

eBooks-IT.org

```
</div>
```

I added a new column to each row of the table that contains a `form` with an `input` element. I styled the `inp` as a button with Bootstrap and added a `style` element and an `id` to the `table` element to ensure that the butto content of the other columns are properly aligned.

Note I used the strongly typed `Html.HiddenFor` helper method to create a hidden field for the `Return` property, but I had to use the string-based `Html.Hidden` helper to do the same for the `ProductId` field. If `Html.HiddenFor(x => line.Product.ProductID)`, the helper would render a hidden field w `line.Product.ProductID`. The name of the field would not match the names of the parameters for the `CartController.RemoveFromCart` action method, which would prevent the default model binders fr so the MVC Framework would not be able to call the method.

You can see the `Remove` buttons at work by running the application and adding items to the shopping cart. Rem the cart already contains the functionality to remove it, which you can test by clicking one of the new buttons, as sh 9-1.

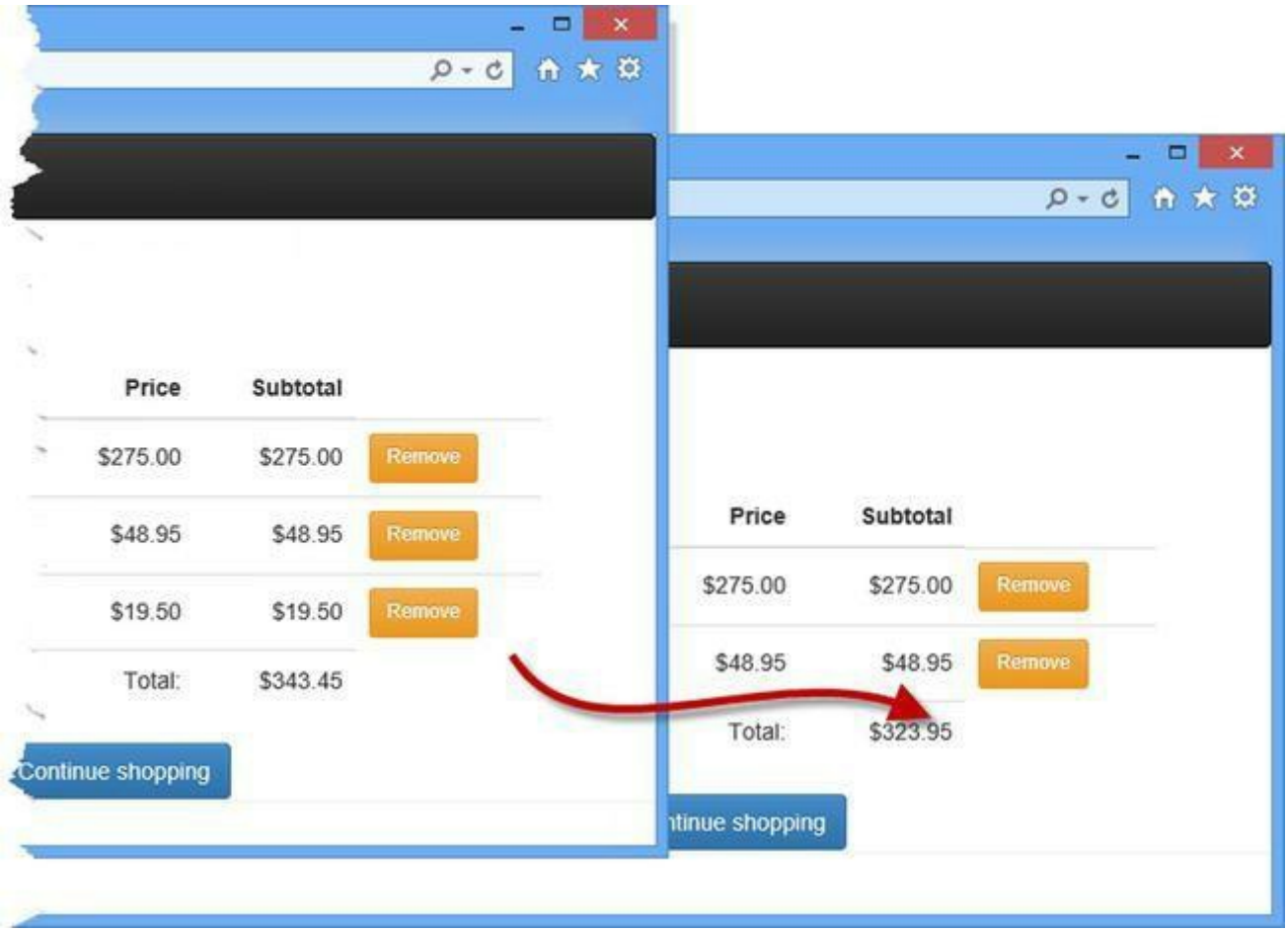


Figure 9-1. Removing an item from the shopping cart

Adding the Cart Summary

I may have a functioning cart, but there is an issue with the way it is integrated into the interface. Customers can tell their cart only by viewing the cart summary screen. And they can view the cart summary screen only by adding a ne to the cart.

To solve this problem, I am going to add a widget that summarizes the contents of the cart and that can be clicked the cart contents throughout the application. I will do this in much the same way that I added the navigation widget— whose output I will inject into the Razor layout. To start, I need to add the simple method, shown in Listing 9-5, to t `CartController` class.

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

        public CartController(IProductRepository repo) {
            repository = repo;
        }

        // . . . other action methods omitted for brevity . . .

        public PartialViewResult Summary(Cart cart) {
            return PartialView(cart);
        }
    }
}
```

This simple method needs to render a view, supplying the current `Cart` (which will be obtained using the custon binder) as view data. To create the view, right-click the `Summary` action method and select `Add View` from th menu. Set the name of the view to `Summary` and click the `OK` button to create the `Views/Cart/Summary` file. Edit the view so that it matches Listing 9-6.

Listing 9-6. The Contents of the Summary.cshtml File

```
@model SportsStore.Domain.Entities.Cart

<div class="navbar-right">
    @Html.ActionLink("Checkout", "Index", "Cart",
        new { returnUrl = Request.Url.PathAndQuery },
        new { @class = "btn btn-default navbar-btn" })
</div>

<div class="navbar-text navbar-right">
    <b>Your cart:</b>
    @Model.Lines.Sum(x => x.Quantity) item(s),
    @Model.ComputeTotalValue().ToString("c")
</div>
```

The view displays the number of items in the cart, the total cost of those items, and a link that shows the details o the user (and, as you will have expected by now, I have assigned the elements that the view contains to classes defin Bootstrap). Now that I have created the view that is returned by the `Summary` action method, I can call the `Sum` method in the `_Layout.cshtml` file to display the cart summary, as shown in Listing 9-7.

Listing 9-7. Adding the Summary Partial View to the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
```

eBooks-IT.org

```
    <meta name="viewport" content="width=device-width, initial-sca
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="navbar navbar-inverse" role="navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

You can see the cart summary by running the application. The item count and total increase as you add items to th shown by Figure 9-2.

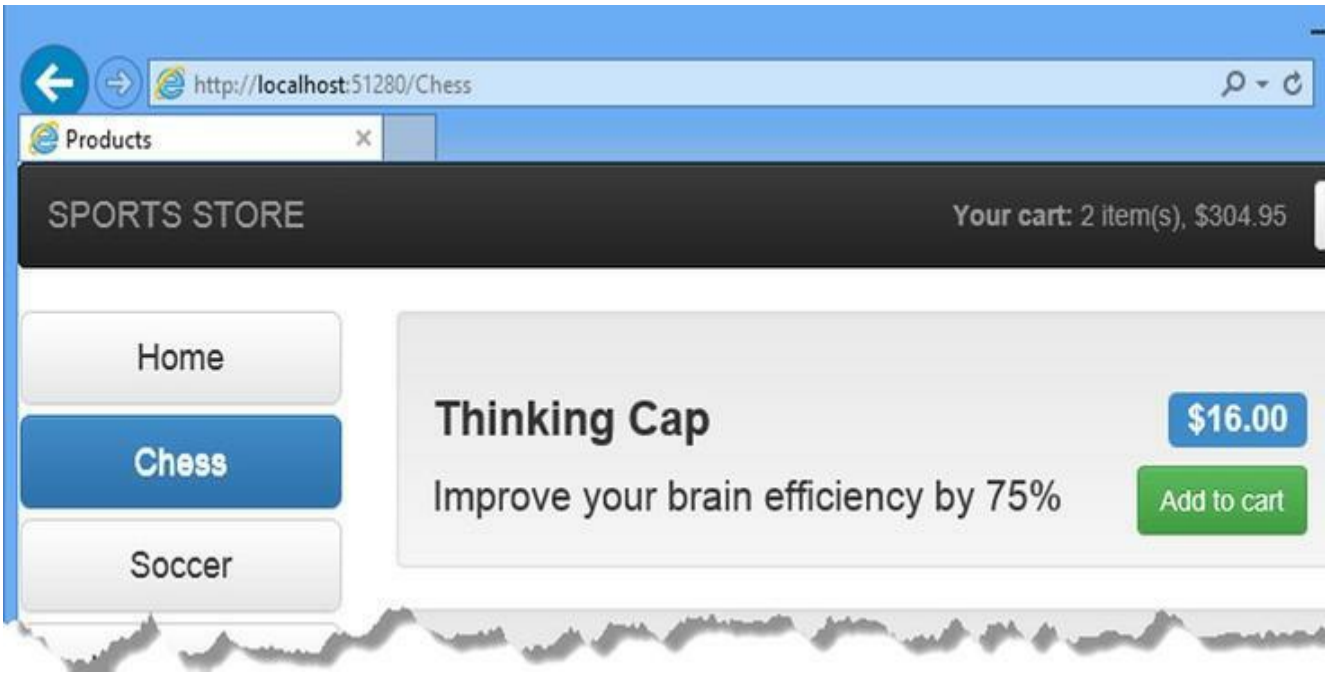


Figure 9-2. The cart summary widget

With this addition, customers know what is in their cart and have an obvious way to check out from the store. Yo once again, how easy it is to use the `Html.Action` helper method to incorporate the output from an action meth view. This is a nice technique for breaking down the functionality of an application into distinct, reusable blocks.

Submitting Orders

I have now reached the final customer feature in SportsStore: the ability to check out and complete an order. In the f sections, I will extend the domain model to provide support for capturing the shipping details from a user and add the support to process those details.

Add a class file called `ShippingDetails.cs` to the `Entities` folder of the `SportsStore.Doma` and edit it to match the contents shown in Listing 9-8. This is the class I will use to represent the shipping details for

Listing 9-8. The Contents of the ShippingDetails.cs File

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Domain.Entities {

    public class ShippingDetails {
        [Required(ErrorMessage = "Please enter a name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the first address l
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string Line3 { get; set; }

        [Required(ErrorMessage = "Please enter a city name")]
        public string City { get; set; }

        [Required(ErrorMessage = "Please enter a state name")]
        public string State { get; set; }

        public string Zip { get; set; }

        [Required(ErrorMessage = "Please enter a country name")]
        public string Country { get; set; }

        public bool GiftWrap { get; set; }
    }
}
```

You can see that I am using the validation attributes from the `System.ComponentModel.DataAnno` namespace, just as I did in Chapter 2. I explore validation further in Chapter 25.

Note The `ShippingDetails` class does not have any functionality, so there is nothing that that can be se tested.

Adding the Checkout Process

The goal is to reach the point where users are able to enter their shipping details and submit their order. To start this add a `Checkout now` button to the cart summary view. Listing 9-9 shows the change I applied to the `Views/Cart/Index.cshtml` file.

Listing 9-9. Adding the Checkout Now Button to the Index.cshtml File

```
. . .
<div class="text-center">
          <a   class="btn   btn-primary"   href="@Model.ReturnUrl">
shopping</a>
    @Html.ActionLink("Checkout now", "Checkout", null, new { @clas
btn-primary"})
```

. . .

This change generates a link that I have styled as a button and that, when clicked, calls the `Checkout` action m
`Cart` controller. You can see how this button appears in Figure 9-3.

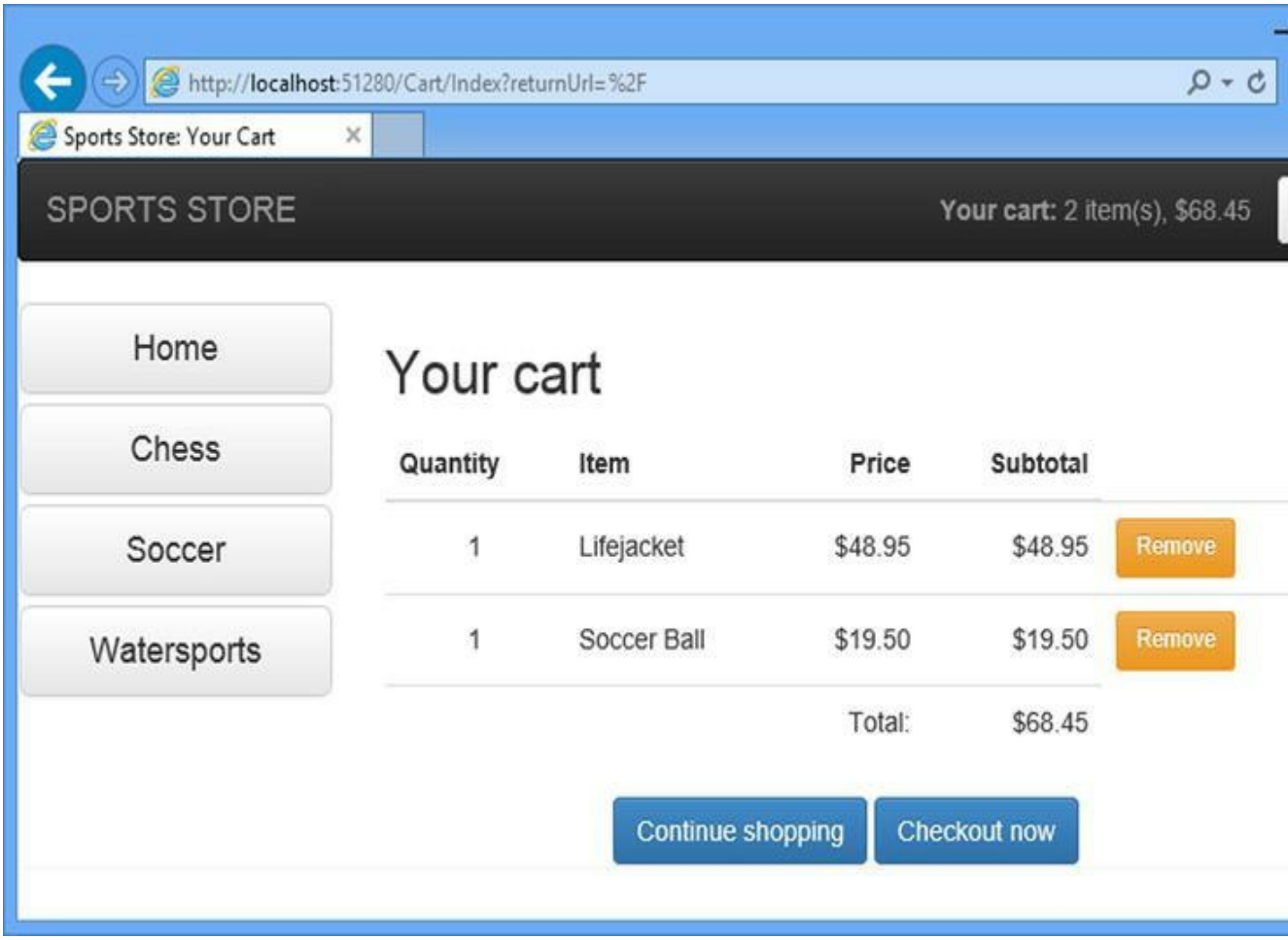


Figure 9-3. The Checkout now button

As you might expect, I now need to define the `Checkout` method in the `CartController` class, as sho
9-10.

Listing 9-10. The Checkout Action Method in the CartController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

        public CartController(IProductRepository repo) {
            repository = repo;
        }

        // . . . other action methods omitted for brevity . . .
```

eBooks-IT.org

```
        public ViewResult Checkout() {
            return View(new ShippingDetails());
        }
    }
}
```

The `Checkout` method returns the default view and passes a new `ShippingDetails` object as the view create the view for the action method, right-click on the `Checkout` action method and select `Add View` from menu. Set the name to `Checkout` and click the `OK` button. Visual Studio will create the `Views/Cart/Checkout.cshtml` file, which you should edit to match Listing 9-11.

Listing 9-11. The Contents of the Checkout.cshtml File

```
@model SportsStore.Domain.Entities.ShippingDetails

@{
    ViewBag.Title = "SportStore: Checkout";
}

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away

@using (Html.BeginForm()) {

    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name:</label>
        @Html.TextBoxFor(x => x.Name, new {@class = "form-control"
    </div>

    <h3>Address</h3>

    <div class="form-group">
        <label>Line 1:</label>
        @Html.TextBoxFor(x => x.Line1, new {@class = "form-control
    </div>
    <div class="form-group">
        <label>Line 2:</label>
        @Html.TextBoxFor(x => x.Line2, new {@class = "form-control
    </div>
    <div class="form-group">
        <label>Line 3:</label>
        @Html.TextBoxFor(x => x.Line3, new {@class = "form-control
    </div>
    <div class="form-group">
        <label>City:</label>
        @Html.TextBoxFor(x => x.City, new {@class = "form-control"
    </div>
    <div class="form-group">
        <label>State:</label>
        @Html.TextBoxFor(x => x.State, new {@class = "form-control
    </div>
    <div class="form-group">
        <label>Zip:</label>
```

eBooks-IT.org

```
    </div>
    <div class="form-group">
        <label>Country:</label>
        @Html.TextBoxFor(x => x.Country, new {@class = "form-contr
    </div>

    <h3>Options</h3>
    <div class="checkbox">
        <label>
            @Html.EditorFor(x => x.GiftWrap)
            Gift wrap these items
        </label>
    </div>

    <div class="text-center">
            <input class="btn btn-primary" type="submit" value="
order" />
    </div>
}
```

For each of the properties in the model, I have created a `label` and `input` element formatted with Bootstrap user input. You can see the effect I have created by starting the application and clicking the `Checkout` button at t page and then clicking `Checkout now`, as shown by Figure 9-4. (You can also reach this view by navigating to `/Cart/Checkout` URL).

eBooks-IT.org

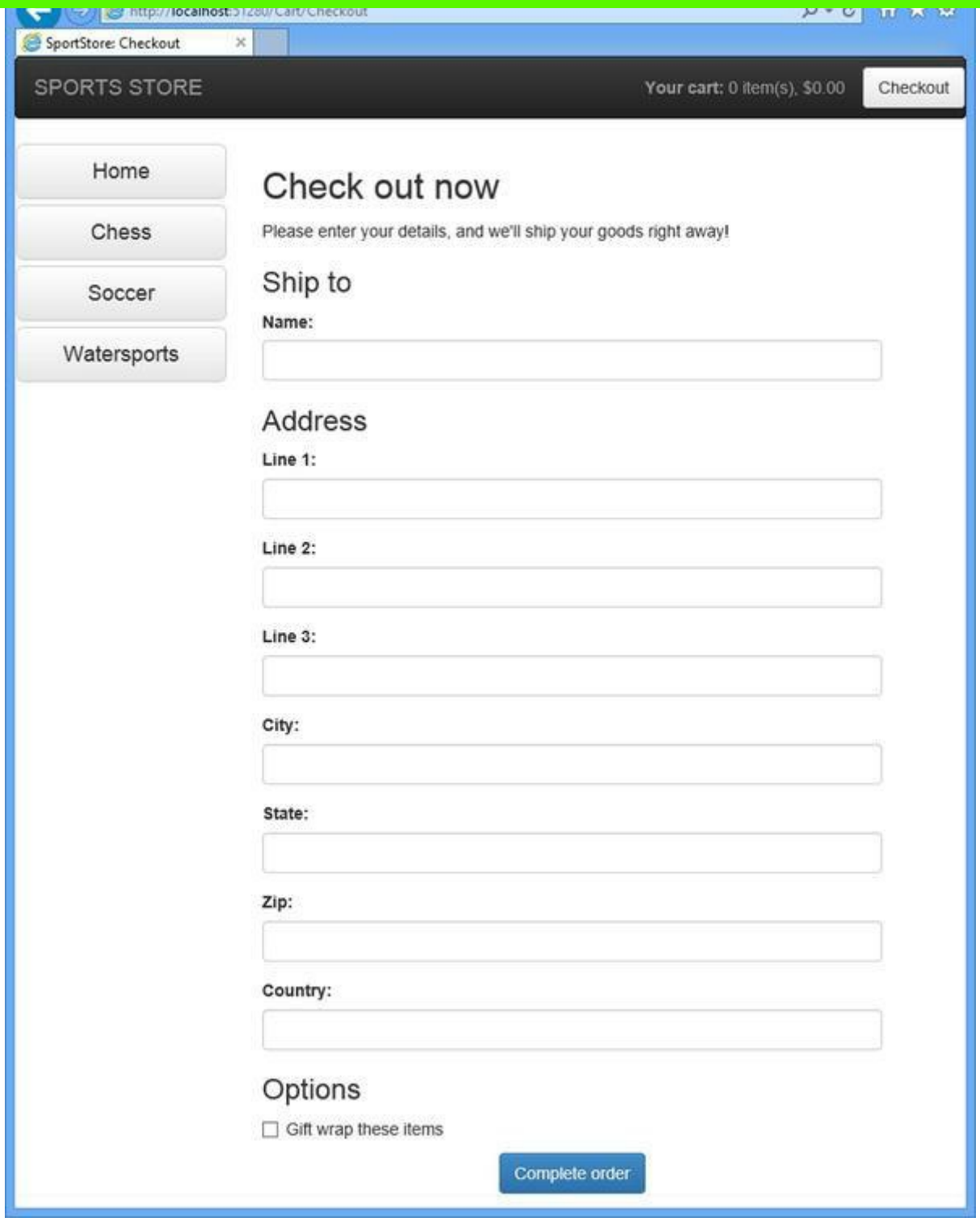


Figure 9-4. The shipping details form

The problem with this view is that it contains a lot of repeated markup. There are MVC Framework HTML helpers reduce the duplication, but they make it hard to structure and style the content in the way that I want. Instead, I am handy feature to get metadata about the view model object and combine it with a mix of C# and Razor expressions. what I have done in Listing 9-12.

Listing 9-12. Reducing Duplication in the Checkout.cshtml File

```
@model SportsStore.Domain.Entities.ShippingDetails

@{
```

```
}

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away

@using (Html.BeginForm()) {

    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name</label>
        @Html.TextBoxFor(x => x.Name, new {@class = "form-control"
    </div>

    <h3>Address</h3>

    foreach (var property in ViewData.ModelMetadata.Properties) {
          if (property.PropertyName != "Name" && property.Property
"GiftWrap") {
            <div class="form-group">
                  <label>@(property.DisplayName ?? property.Prope
</label>
                 @Html.TextBox(property.PropertyName, null, new {
"form-control"})
            </div>
        }
    }

    <h3>Options</h3>
    <div class="checkbox">
        <label>
            @Html.EditorFor(x => x.GiftWrap)
            Gift wrap these items
        </label>
    </div>

    <div class="text-center">
            <input class="btn btn-primary" type="submit" value="
order" />
    </div>
}
```

The static `ViewData.ModelMetadata` property returns a `System.Web.Mvc.ModelMetaDa` that provides information about the model type for the view. The `Properties` property I use in the `foreach` a collection of `ModelMetaData` objects, each of which represents a property defined by the model type. I use `PropertyName` property to ensure that I don't generate content for the `Name` or `GiftWrap` properties (wh with elsewhere in the view) and generate a set of elements, complete with Bootstrap classes, for all of the other prop

Tip The `for` and `if` keywords I have used are within the scope of a Razor expression (the `@using` express creates the form) and so I don't need to prefix them with the `@` character. In fact, were I to do so, Razor would rep It can take a little while to get used to when the `@` character is required with Razor, but it becomes second nature for programmers. For those that can't quite get it right first time (which includes me), the Razor error message displayed browser provides specific instructions to correct any mistakes.

I am not quite done, however. If you run the example and look at the output generated by the view, you will see th the labels are not quite correct, as Figure 9-5 illustrates.

eBooks-IT.org

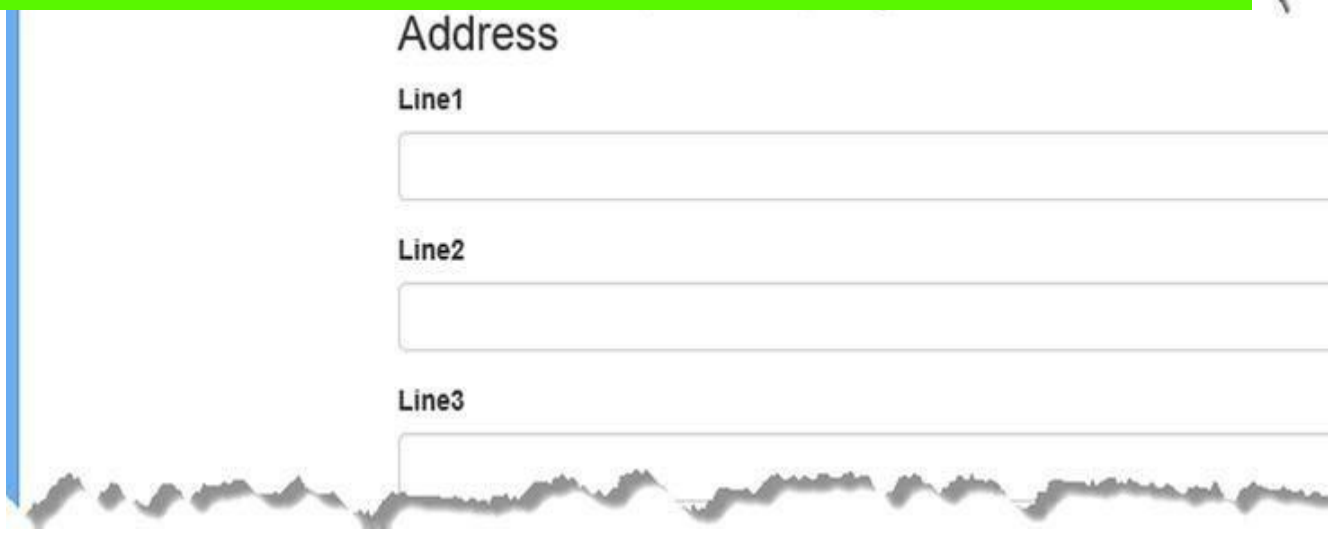


Figure 9-5. The problem with generating labels from property names

The issue is that the property names don't always make for good labels. This is why I check to see if there is a `DisplayName` value available when I generate the form elements, like this:

```
. . .
<label>@(property.DisplayName ?? property.PropertyName)</label>
. . .
```

To take advantage of the `DisplayName` property, I need to apply the `Display` attribute to the model class Listing 9-13.

Listing 9-13. Applying th e `Display` attribute to the ShippingDetails.cs File

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Domain.Entities {

    public class ShippingDetails {
        [Required(ErrorMessage = "Please enter a name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the first address l
        [Display(Name="Line 1")]
        public string Line1 { get; set; }
        [Display(Name = "Line 2")]
        public string Line2 { get; set; }
        [Display(Name = "Line 3")]
        public string Line3 { get; set; }

        [Required(ErrorMessage = "Please enter a city name")]
        public string City { get; set; }

        [Required(ErrorMessage = "Please enter a state name")]
        public string State { get; set; }

        public string Zip { get; set; }

        [Required(ErrorMessage = "Please enter a country name")]
```

```
        public bool GiftWrap { get; set; }
    }
}
```

Setting the `Name` value for the `Display` attribute allows me to set up a value that will be read by the `Displ` property in the view. You can see the effect by starting the application and viewing the checkout page, as shown in I

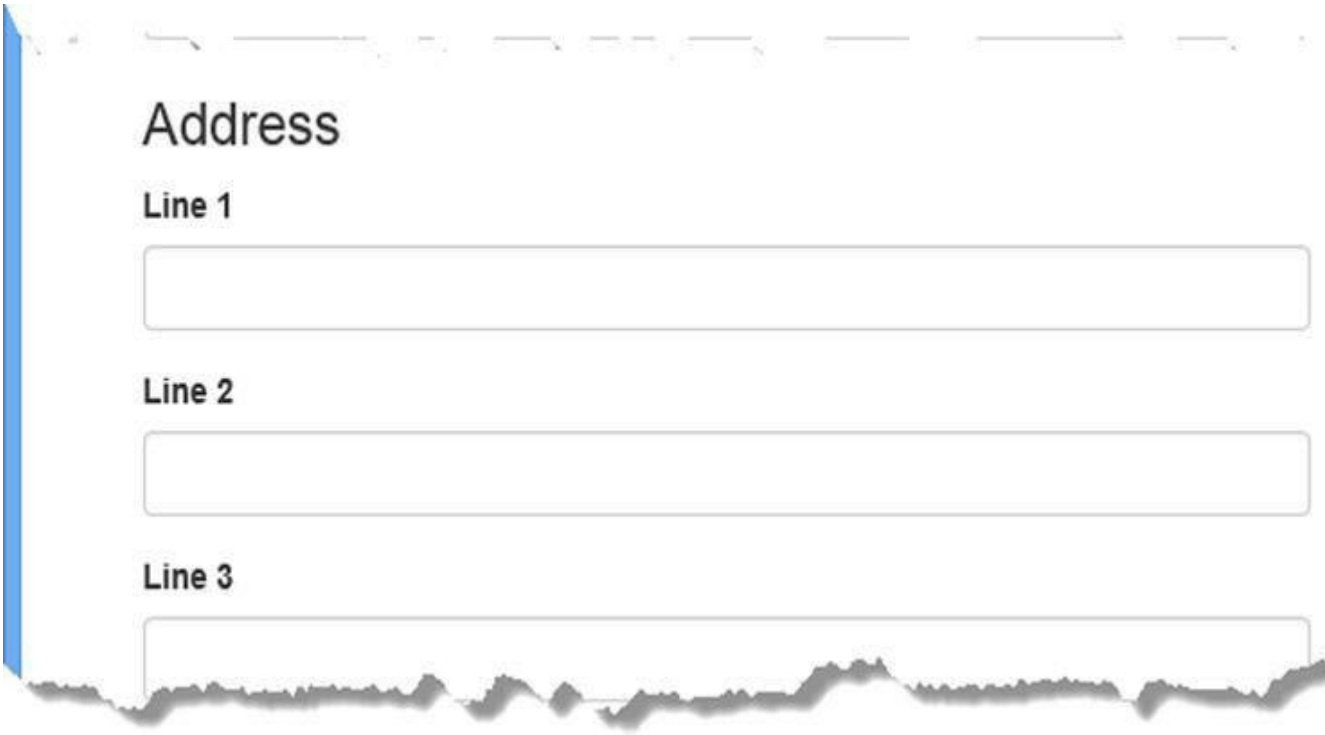


Figure 9-6. The effect of the Display attribute on the model type

This example shows two different aspects of working with the MVC Framework. The first is that you can work a situation to simplify your markup or code. The second is that even though the role that views play in the MVC patter to displaying data and markup, the tools that Razor and C# provide for this task are rich and flexible, even to the exte with type metadata.

Implementing the Order Processor

I need a component in the application to which I can hand details of an order for processing. In keeping with the prin MVC model, I am going to define an interface for this functionality, write an implementation of the interface, and the the two using the DI container, Ninject.

Defining the Interface

Add a new interface called `IOrderProcessor` to the `Abstract` folder of the `SportsStore.Doma` and edit the contents so that they match Listing 9-14.

Listing 9-14. The Contents of the IOrderProcessor.cs File

```
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract {

    public interface IOrderProcessor {
```

```
    }
}
```

Implementing the Interface

The implementation of `IOrderProcessor` is going to deal with orders by e-mailing them to the site administra simplifying the sales process, of course. Most e-commerce sites would not simply e-mail an order, and I have not pr support for processing credit cards or other forms of payment. But I want to keep things focused on MVC, and so e

Create a new class file called `EmailOrderProcessor.cs` in the `Concrete` folder of the `SportsStore.Domain` project and edit the contents so that they match Listing 9-15. This class uses the bui support included in the .NET Framework library to send an e-mail.

Listing 9-15. The Contents of the EmailOrderProcessor.cs File

```
using System.Net;
using System.Net.Mail;
using System.Text;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Concrete {

    public class EmailSettings {
        public string MailToAddress = "orders@example.com";
        public string MailFromAddress = "sportsstore@example.com";
        public bool UseSsl = true;
        public string Username = "MySmtpUsername";
        public string Password = "MySmtpPassword";
        public string ServerName = "smtp.example.com";
        public int ServerPort = 587;
        public bool WriteAsFile = false;
        public string FileLocation = @"c:\sports_store_emails";
    }

    public class EmailOrderProcessor :IOrderProcessor {
        private EmailSettings emailSettings;

        public EmailOrderProcessor(EmailSettings settings) {
            emailSettings = settings;
        }

         public void ProcessOrder(Cart cart, ShippingDetails shipp
{

            using (var smtpClient = new SmtpClient()) {

                smtpClient.EnableSsl = emailSettings.UseSsl;
                smtpClient.Host = emailSettings.ServerName;
                smtpClient.Port = emailSettings.ServerPort;
                smtpClient.UseDefaultCredentials = false;
                smtpClient.Credentials
                    = new NetworkCredential(emailSettings.Username
                          emailSettings.Password);
```

eBooks-IT.org

```
                    smtpClient.DeliveryMethod
                        = SmtpDeliveryMethod.SpecifiedPickupDirect
                                    smtpClient.PickupDirectoryLoc
emailSettings.FileLocation;
                    smtpClient.EnableSsl = false;
                }

                StringBuilder body = new StringBuilder()
                    .AppendLine("A new order has been submitted")
                    .AppendLine("---")
                    .AppendLine("Items:");

                foreach (var line in cart.Lines) {
                    var subtotal = line.Product.Price * line.Quant
                          body.AppendFormat("{0} x {1} (subtotal:
line.Quantity,
                                      line.Product.Name,
                                      subtotal);
                }

                           body.AppendFormat("Total order value:
cart.ComputeTotalValue())
                    .AppendLine("---")
                    .AppendLine("Ship to:")
                    .AppendLine(shippingInfo.Name)
                    .AppendLine(shippingInfo.Line1)
                    .AppendLine(shippingInfo.Line2 ?? "")
                    .AppendLine(shippingInfo.Line3 ?? "")
                    .AppendLine(shippingInfo.City)
                    .AppendLine(shippingInfo.State ?? "")
                    .AppendLine(shippingInfo.Country)
                    .AppendLine(shippingInfo.Zip)
                    .AppendLine("---")
                    .AppendFormat("Gift wrap: {0}",
                        shippingInfo.GiftWrap ? "Yes" : "No");

                MailMessage mailMessage = new MailMessage(
                                        emailSettings.MailFromAddre
From
                                        emailSettings.MailToAddress
To
                                        "New order submitted!",
Subject
                                        body.ToString());
Body

                if (emailSettings.WriteAsFile) {
                    mailMessage.BodyEncoding = Encoding.ASCII;
                }

                smtpClient.Send(mailMessage);
            }
        }
    }
}
```

eBooks-IT.org

demanded by the EmailOrderProcessor constructor and contains all the settings that are required to conf
.NET e-mail classes.

Tip Do not worry if you do not have an SMTP server available. If you set the EmailSettings.WriteA
property to true, the e-mail messages will be written as files to the directory specified by the FileLocatior
This directory must exist and be writable. The files will be written with the .eml extension, but they can be read v
editor. The location I have set in the listing is c:\sports_store_emails.

Registering the Implementation

Now that I have an implementation of the IOrderProcessor interface and the means to configure it, I can us
create instances of it. Edit the NinjectDependencyResolver.cs file in the Infrastructure f
SportsStore.WebUI project and make the changes shown in Listing 9-16 to the AddBindings metho

Listing 9-16. Adding Ninject Bindings for IOrderProcessor to the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Web.Mvc;
using Moq;
using Ninject;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Concrete;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Infrastructure {

    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            kernel.Bind<IProductRepository>().To<EFProductReposito

            EmailSettings emailSettings = new EmailSettings {
                WriteAsFile = bool.Parse(ConfigurationManager
                    .AppSettings["Email.WriteAsFile"] ?? "false")
            };

            kernel.Bind<IOrderProcessor>().To<EmailOrderProcessor>
                .WithConstructorArgument("settings", emailSettings
```

eBooks-IT.org

```
    }
}
```

I created an `EmailSettings` object, which I use with the Ninject `WithConstructorArgument` that I can inject it into the `EmailOrderProcessor` constructor when new instances are created to service r the `IOrderProcessor` interface. In Listing 9-16, I specified a value for only one of the `EmailSetting` `WriteAsFile`. I read the value of this property using the `ConfigurationManager.AppSettin` which provides access to application settings defined in the `Web.config` file (the one in the root project folder), shown in Listing 9-17.

Listing 9-17. Application Settings in the Web.config File

```
. . .
<appSettings>
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  <add key="Email.WriteAsFile" value="true"/>
</appSettings>
. . .
```

Completing the Cart Controller

To complete the `CartController` class, I need to modify the constructor so that it demands an implementati `IOrderProcessor` interface and add a new action method that will handle the HTTP form `POST` request wl clicks the `Complete order` button. Listing 9-18 shows both changes.

Listing 9-18. Completing the Controller in the CartController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;
        private IOrderProcessor orderProcessor;

           public CartController(IProductRepository repo, IOrderP
proc) {
            repository = repo;
            orderProcessor = proc;
        }

        // . . . other action methods omitted for brevity . . .

        public ViewResult Checkout() {
            return View(new ShippingDetails());
        }
```

eBooks-IT.org

```
                public ViewResult Checkout(Cart cart, Shippin
shippingDetails) {
            if (cart.Lines.Count() == 0) {
                        ModelState.AddModelError("", "Sorry, your
empty!");
            }

            if (ModelState.IsValid) {
                orderProcessor.ProcessOrder(cart, shippingDetails)
                cart.Clear();
                return View("Completed");
            } else {
                return View(shippingDetails);
            }
        }
    }
}
```

You can see that the `Checkout` action method I added is decorated with the `HttpPost` attribute, which me will be invoked for a `POST` request—in this case, when the user submits the form. Once again, I am relying on the system, both for the `ShippingDetails` parameter (which is created automatically using the HTTP form data `Cart` parameter (which is created using the custom binder).

Note The change in constructor forces me to update the unit tests I created for the `CartController` clas `null` for the new constructor parameter will let the unit tests compile.

The MVC Framework checks the validation constraints that I applied to `ShippingDetails` using the data attributes, and any validation problems violations are passed to the action method through the `ModelState` prop if there are any problems by checking the `ModelState.IsValid` property. Notice that I call the `ModelState.AddModelError` method to register an error message if there are no items in the cart. I wil to display such errors shortly, and I have much more to say about model binding and validation in Chapters 24 and 2

UNIT TEST: ORDER PROCESSING

To complete the unit testing for the `CartController` class, I need to test the behavior of the new overlo of the `Checkout` method. Although the method looks short and simple, the use of MVC Framework model b that there is a lot going on behind the scenes that needs to be tested.

I want to process an order only if there are items in the cart *and* the customer has provided valid shipping detail other circumstances, the customer should be shown an error. Here is the first test method:

```
. . .
[TestMethod]
public void Cannot_Checkout_Empty_Cart() {

  // Arrange - create a mock order processor
  Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();
  // Arrange - create an empty cart
  Cart cart = new Cart();
  // Arrange - create shipping details
  ShippingDetails shippingDetails = new ShippingDetails();
  // Arrange - create an instance of the controller
  CartController target = new CartController(null, mock.Object);

  // Act
```

eBooks-IT.org

```
  // Assert - check that the order hasn't been passed on to the
processor
  mock.Verify(m => m.ProcessOrder(It.IsAny<Cart>(),
It.IsAny<ShippingDetails>()),
    Times.Never());
  // Assert - check that the method is returning the default vie
  Assert.AreEqual("", result.ViewName);
  // Assert - check that I am passing an invalid model to the vi
  Assert.AreEqual(false, result.ViewData.ModelState.IsValid);
}
. . .
```

This test ensures that I cannot check out with an empty cart. I check this by ensuring that the `ProcessOrd` mock `IOrderProcessor` implementation is never called, that the view that the method returns is the def (which will redisplay the data entered by customers and give them a chance to correct it), and that the model st passed to the view has been marked as invalid. This may seem like a belt-and-braces set of assertions, but I nee be sure that I have the right behavior. The next test method works in much the same way, but injects an error i model to simulate a problem reported by the model binder (which would happen in production when the custom invalid shipping data):

```
. . .
[TestMethod]
public void Cannot_Checkout_Invalid_ShippingDetails() {

  // Arrange - create a mock order processor
  Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();

  // Arrange - create a cart with an item
  Cart cart = new Cart();
  cart.AddItem(new Product(), 1);

  // Arrange - create an instance of the controller
  CartController target = new CartController(null, mock.Object);

  // Arrange - add an error to the model
  target.ModelState.AddModelError("error", "error");

  // Act - try to checkout
  ViewResult result = target.Checkout(cart, new ShippingDetails(

  // Assert - check that the order hasn't been passed on to the
processor
  mock.Verify(m => m.ProcessOrder(It.IsAny<Cart>(),
It.IsAny<ShippingDetails>()),
    Times.Never());
  // Assert - check that the method is returning the default vie
  Assert.AreEqual("", result.ViewName);
  // Assert - check that I am passing an invalid model to the vi
  Assert.AreEqual(false, result.ViewData.ModelState.IsValid);
}
. . .
```

Having established that an empty cart or invalid details will prevent an order from being processed, I need to ens process orders when appropriate. Here is the test:

```
. . .
```

eBooks-IT.org

```
public void Can_Checkout_And_Submit_Order() {

    // Arrange - create a mock order processor
    Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();

    // Arrange - create a cart with an item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);

    // Arrange - create an instance of the controller
    CartController target = new CartController(null, mock.Object);

    // Act - try to checkout
    ViewResult result = target.Checkout(cart, new ShippingDetails(

    // Assert - check that the order has been passed on to the pro
    mock.Verify(m => m.ProcessOrder(It.IsAny<Cart>(),
It.IsAny<ShippingDetails>()),
      Times.Once());
    // Assert - check that the method is returning the Completed v
    Assert.AreEqual("Completed", result.ViewName);
    // Assert - check that I am passing a valid model to the view
    Assert.AreEqual(true, result.ViewData.ModelState.IsValid);
}
. . .
```

Notice that I did not need to test that I can identify valid shipping details. This is handled for me automatically b binder using the attributes applied to the properties of the `ShippingDetails` class.

Displaying Validation Errors

The MVC Framework will use the validation attributes applied to the `ShippingDetails` class to validate user However, I need to make a couple of changes to display any problems to the user. The first issue is that I need to pr summary of any problems to the user. This is especially important when dealing with problems that are not related to fields, such as when the user tries to check out with no items in the cart.

To display a useful summary of the validation errors, I can use the `Html.ValidationSummary` helper r as I did in Chapter 2. Listing 9-19 shows the addition to `Checkout.cshtml` view.

Listing 9-19. Adding a Validation Summary to the Checkout.cshtml File

```
. . .
@using (Html.BeginForm()) {

    @Html.ValidationSummary()
    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name</label>
        @Html.TextBoxFor(x => x.Name, new {@class = "form-control"
    </div>

    <h3>Address</h3>
. . .
```

The next step is to create some CSS styles that target the classes used by the validation summary and those to wh Framework adds invalid elements. I created a new `Style Sheet` called `ErrorStyles.css` in the Con

eBooks-IT.org

in Chapter 2.

Listing 9-20. The Contents of the ErrorStyles.css File

```
.field-validation-error    {color: #f00;}
.field-validation-valid    { display: none;}
.input-validation-error      { border: 1px solid #f00; backgroun
#fee; }
.validation-summary-errors { font-weight: bold; color: #f00;}
.validation-summary-valid  { display: none;}
```

In order to use apply the styles, I updated the `_Layout.cshtml` file to add a link element for the `ErrorStyles.css` file, as shown in Listing 9-21.

Listing 9-21. Adding a Link Element in the _Layout.cshtml File

```
. . .
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-sca
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head
. . .
```

With these changes, validation errors are reported through highlighting problematic fields and by showing a summa problems, as Figure 9-7 illustrates.

eBooks-IT.org

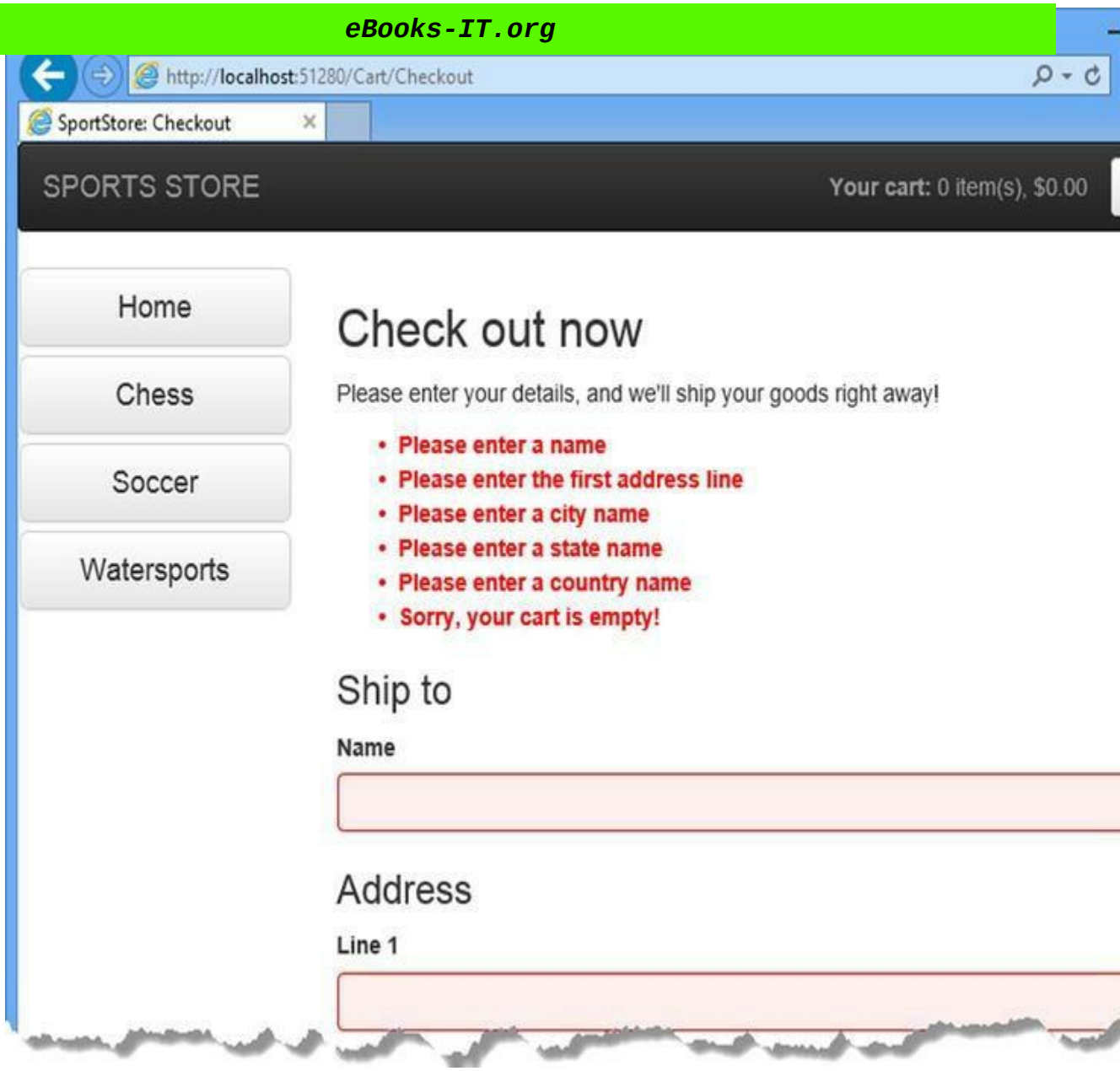


Figure 9-7. Displaying validation messages

Tip The data submitted by the user is sent to the server before it is validated, which is known as server-side valid which the MVC Framework has excellent support. The problem with server-side validation is that the user isn't told until after the data has been sent to the server, processed and the result page generated—something that can take a fe on a busy server. For this reason, server-side validation is usually complemented by client-side validation, where Java to check the values that the user has entered before the form data is sent to the server. I describe client-side validatio 25.

Displaying a Summary Page

To complete the checkout process, I will show customers a page that confirms the order has been processed and th their business. Create a new view called `Completed.cshtml` in the `Views/Cart` folder and edit the con Listing 9-22.

Listing 9-22. The Contents of the Completed.cshtml File

```
@{
    ViewBag.Title = "SportsStore: Order Submitted";
}
```

eBooks-IT.org

```
<h2>Thanks!</h2>
Thanks for placing your order. We'll ship your goods as soon as po
```

I don't need to make any code changes to integrate this view into the application because I already added the requi when I defined the `Checkout` action method back in Listing 9-18. Now customers can go through the entire pro selecting products to checking out. If they provide valid shipping details (and have items in their cart), they will see t page when they click the `Complete order` button, as shown in Figure 9-8.

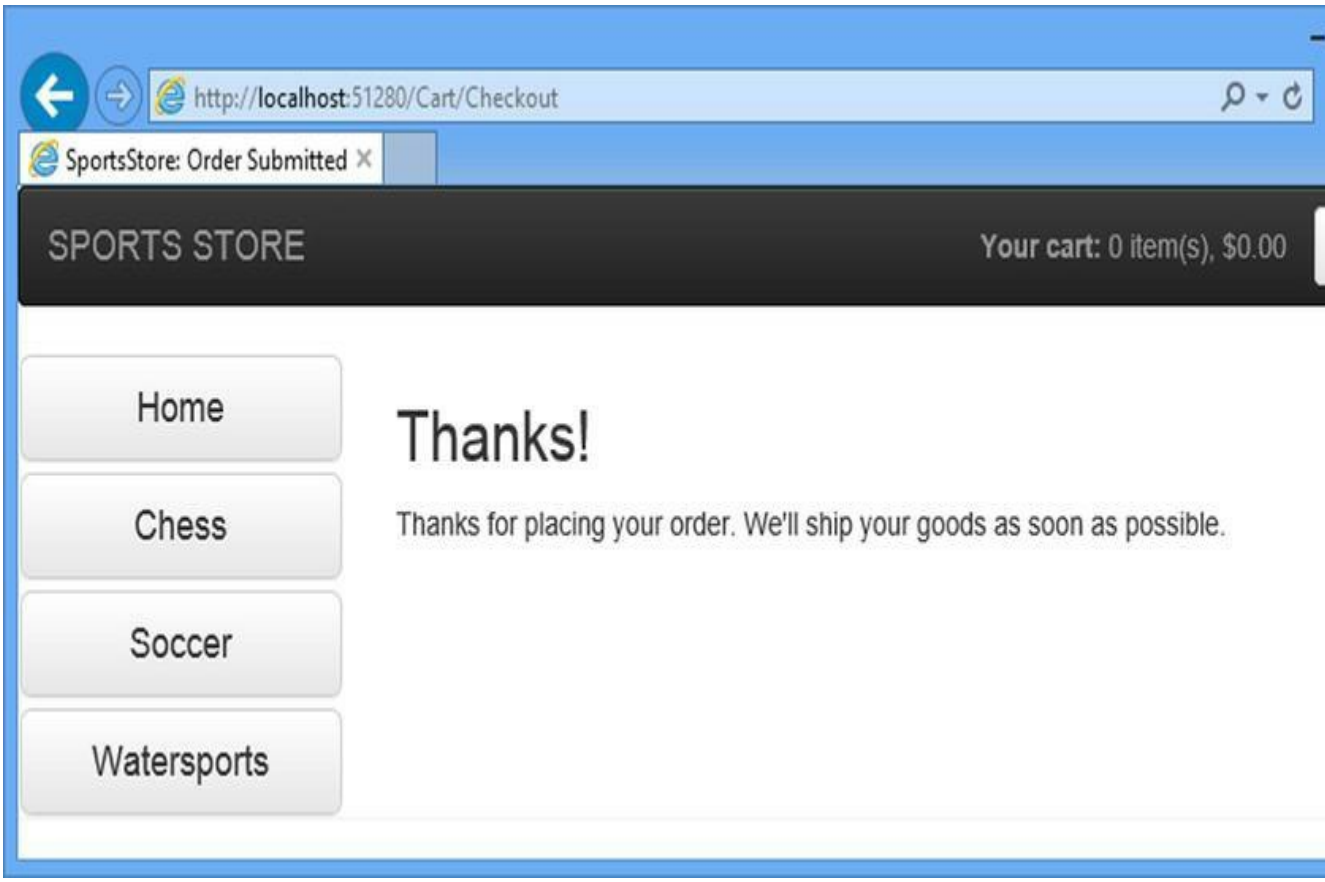


Figure 9-8. The thank-you page

Summary

I have completed all the major parts of the customer-facing portion of SportsStore. It might not be enough to worry I have a product catalog that can be browsed by category and page, a neat shopping cart, and a simple checkout pro

The well-separated architecture means I can easily change the behavior of any piece of the application without wo causing problems or inconsistencies elsewhere. For example, I could process orders by storing them in a database, a not have any impact on the shopping cart, the product catalog, or any other area of the application. In the next chapt different techniques to create a mobile version of the SportsStore application.

eBooks-IT.org

SportsStore: Mobile

There is no escaping the popularity of devices such as smartphones and tablets, and if you want to deliver your appli widest possible audience, you will have to embrace the world of mobile web browsers. If I sound less than enthusias because the phrase *mobile web browsers* runs the gamut from fast, capable, and modern browsers that can rival a de browser right through to the slow, inconsistent, and outdated.

The bottom line is that delivering a good experience to mobile users is *hard*—much harder than delivering content desktops. It takes careful planning, design and an enormous amount of testing—and even then it is easy to be caught smartphone or tablet.

Putting Mobile Web Development in Context

The MVC Framework does have some features that can help with mobile development. But the MVC Framework is framework that receives HTTP requests and generates HTML responses and there is a limited amount it can do to wide variation in capabilities that you will encounter when targeting mobile clients. The degree to which The MVC F help depends on the mobile strategy that you have adopted. There are three basic mobile web strategies you can fo describe the following sections.

Note There is a fourth option, which is to create a native application, but I don't discuss that here since it doesn' involve the MVC Framework or, for that matter, web applications.

Doing Nothing (Or As Little As Possible)

It may seem like an odd idea to do nothing, but some mobile devices are capable of handling content that has been de desktop clients. Many—admittedly the most recent—have high-resolution, high-density displays with plenty of memo browsers that can render HTML and run JavaScript quickly. If your app isn't too demanding, you may find that man devices won't encounter any problems at all displaying your application content. As an example, Figure 10-1 shows displays the SportsStore application without any modifications.

eBooks-IT.org

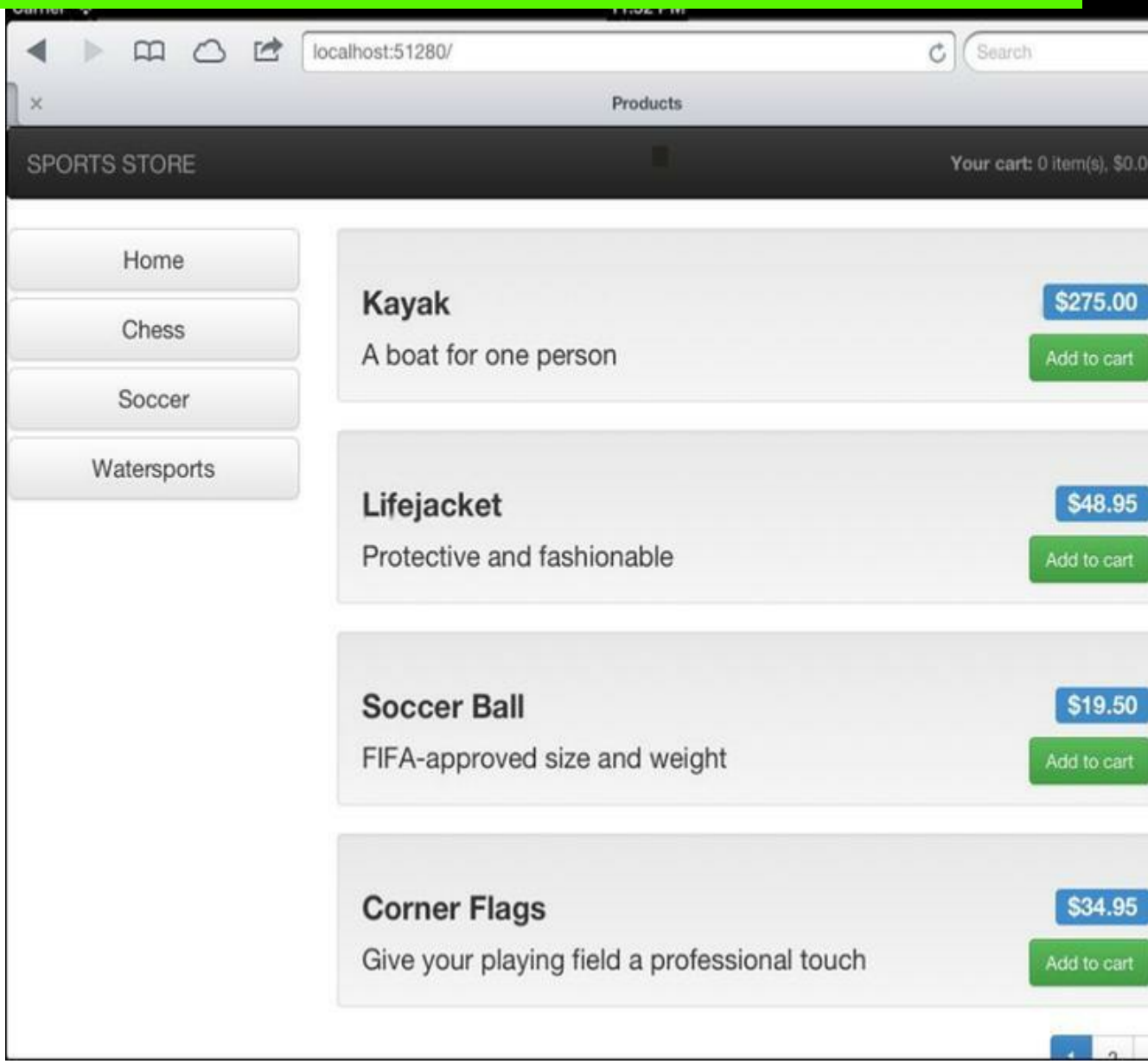


Figure 10-1. Displaying the SportsStore application on a tablet

It does pretty well. The only issue is that the pagination links are off the bottom of the page, which is easily adjust changing the page layout or changing the number of products that are displayed on a page.

Note The screenshots I show in this chapter are all obtained using browserstack.com, which is the cro browser testing service I use for my own projects. It isn't a perfect service. It is often slow to use at peak times, ca from outside of the US, and the mobile devices are emulated. I use it mainly for the desktop browser support, which robust, but I get decent results and I don't have to maintain my own set of emulators. You can get a free trial to follo examples in this chapter and there are plenty of competitors out there if you want to go elsewhere. Note that I don't relationship with Browser Stack other than as a regular customer, for which I pay full price and receive no special tr

Using Responsive Design

The next strategy is to create content that adapts to the capabilities of the device on which it is being displayed, know *responsive design*. The CSS standard has features that let you change the styling applied to elements based on the ca the device, a technique that is most frequently used to alter the layout of content based on screen width.

Responsive design is something that is handled by the client using CSS and not directly managed by the MVC Fran into the topic of responsive design in depth in my *Pro ASP.NET MVC 5 Client* book, but to give a demonstration ho

eBooks-IT.org

features that are included in the Bootstrap library, which I have been using to style the SportsStore application (and some r
become one of the libraries Microsoft includes in the MVC 5 project templates for Visual Studio 2013).

My goal will be to adjust the layout of the main part of the application so that it is visible on an iPhone. My "do not
doesn't pay off for this kind of device because it has a narrow screen, as Figure 10-2 shows.

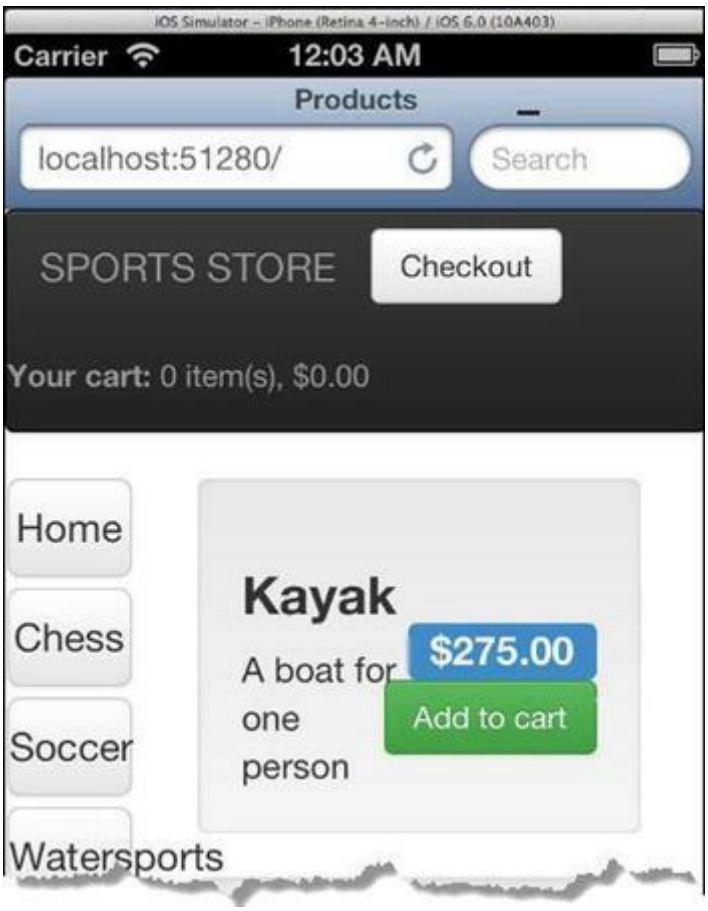


Figure 10-2. Displaying the SportsStore application on a smartphone

I am going to tackle this problem in sections, focusing on different aspects of the layout. My goal is to preserve al
functionality of the application, but presented in a different way.

Note The MVC Framework isn't an active participant in the responsive design. It sends all browsers the same co
them figure out which bits they need to display. This means that there is no sensible way to add unit tests for respon
a Visual Studio project. It is a technique that requires careful testing in the client and is difficult to automate.

Creating a Responsive Header

I am going to start with the page header, which contains the SportsStore name, the cart summary, and the `Checko`
Although the simplest solution would be to remove the SportsStore name and free up enough space for the rest of th
am going to keep it there (see the *Accepting the Realities of Branding* sidebar) and rearrange everything across two l

ACCEPTING THE REALITIES OF BRANDING

One of the easiest ways to free up screen real-estate is to remove your branding from the application. I am only
SportsStore name as text, but you can see how much of the screen it occupies. What was a modest degree of b
the desktop becomes a space hog on a smart phone.

Removing branding is difficult, however. Not for technical reasons, but because most branding teams are obses
slathering branding on everything. It is the reason that there are company-branded pens in the boardroom, comp
cups in the break room, and why you get business cards with new logos every 18 months. Companies rebrand
because people who work in branding know, deep down, that they don't have real jobs and the constant empha
and color schemes creates a frenzy of activity that distracts them from the creeping existential dread that haunts

My advice is to accept that a certain amount of screen space will always be given over to branding, even on the least capable device. You might try to fight against the idea, but the branding team is usually part of the marketin department, marketing usually reports to the VP of Sales, and the VP of Sales has a hot line to the CEO because that the board cares about. Some arguments just can't be won.

In Listing 10-1, you can see how I have adjusted the content of the header in the `_Layout.cshtml` file in th `SportsStore.WebUI` project.

Listing 10-1. Adding Responsive Content to the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-sca
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
    <style>
        .navbar-right {
            float: right !important;
            margin-right: 15px; margin-left: 15px;
        }
    </style>
</head>
<body>
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">
            <span class="hidden-xs">SPORTS STORE</span>
            <div class="visible-xs">SPORTS</div>
            <div class="visible-xs">STORE</div>
        </a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

Bootstrap defines a set of classes that can be used to show or hide elements based on the width of the device scre something you can do manually using CSS *media queries*, but the Bootstrap classes are integrated into the other style

For the SportsStore branding, I have used the `visible-xs` and `hidden-xs` classes to switch to text on t will be shown vertically when the window size is below 768 pixels. Bootstrap provides pairs of classes that show an elements at different browser window sizes, the names of which start with `visible-` or `hidden-`. The `*-x` (i.e., `visible-xs` and `hidden-xs`) are the ones I used in the example. The `*-sm` classes work on windo 768 pixels, the `*-md` classes work on windows wider than 992 pixels, and the `*-lg` classes work on windows w

Caution Responsive CSS features like the ones that Bootstrap provide are based on the size of the browser wind device screen. Mobile device browsers are usually displayed full-screen, which means that the window size and the s the same, but you can't always rely on this being the case. As ever, you need to test against the devices you are targ ensure that you have not made assumptions that catch you out.

You can see the effect of these changes by starting the application and viewing the product listing in a regular desk which has the advantage of letting you change the size of the window. Make the window smaller (less than 786 pixel will see the SportsStore text break into two lines, as illustrated by Figure 10-3.

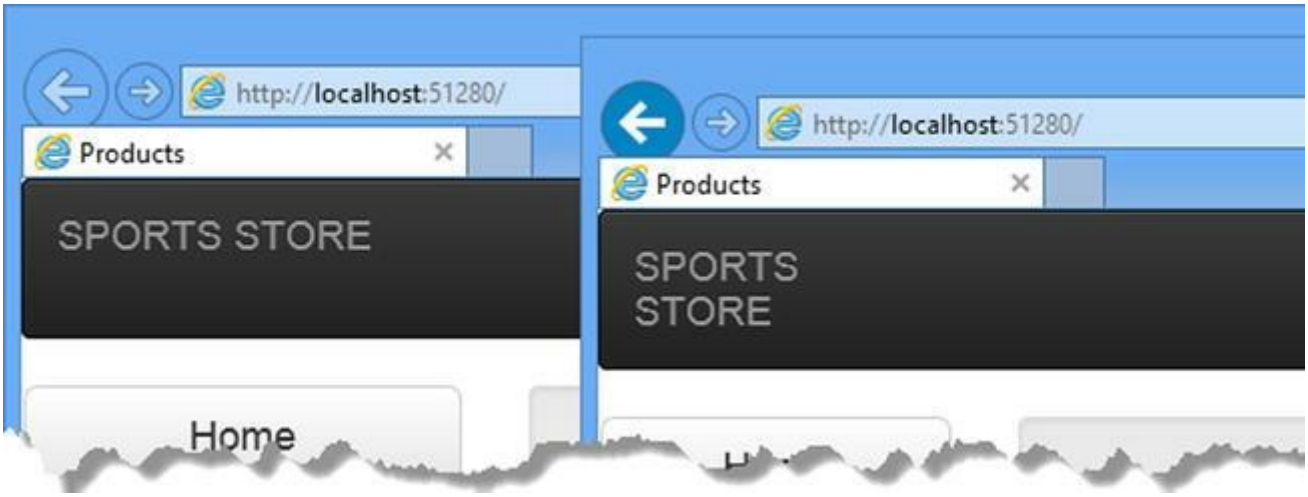


Figure 10-3. Using Bootstrap responsive design features to adjust the application branding

This may seem like a small change but it has a big impact on smaller screens, especially when combined with the c made to the `Views/Cart/Summary.cshtml` file, which is the view that provides the summary of the car contents. You can see the changes I made in Listing 10-2.

Listing 10-2. Adding Responsive Content to the Summary.cshtml File

```
@model SportsStore.Domain.Entities.Cart

<div class="navbar-right hidden-xs">
    @Html.ActionLink("Checkout", "Index", "Cart",
    new { returnUrl = Request.Url.PathAndQuery },
    new { @class = "btn btn-default navbar-btn" })
</div>

<div class="navbar-right visible-xs">
          <a   href=@Url.Action("Index",   "Cart",   new   {   retur
Request.Url.PathAndQuery })
       class="btn btn-default navbar-btn">
        <span class="glyphicon glyphicon-shopping-cart"></span>
    </a>
</div>

<div class="navbar-text navbar-right">
    <b class="hidden-xs">Your cart:</b>
    @Model.Lines.Sum(x => x.Quantity) item(s),
    @Model.ComputeTotalValue().ToString("c")
</div>
```

This is the same technique that I applied to the `_Layout.cshtml` file, where I selectively show and hide co

eBooks-IT.org

utton, u

the icons that is included in the Bootstrap package.

The Bootstrap icons are applied through a `span` element, which means that I can't use the `Html.ActionL` method because it doesn't present me with the ability to set the contents of the element it creates. Instead, I define th directly and use the `Url.Action` helper method (which I describe properly in Chapter 23) to generate a URL fo attribute. The result is an `a` element with the same attributes that would be created by the `Html.ActionLink` that contains a `span` element. You can see the effect of the changes I made to both files in Figure 10-4, which sho content displayed on the iPhone.

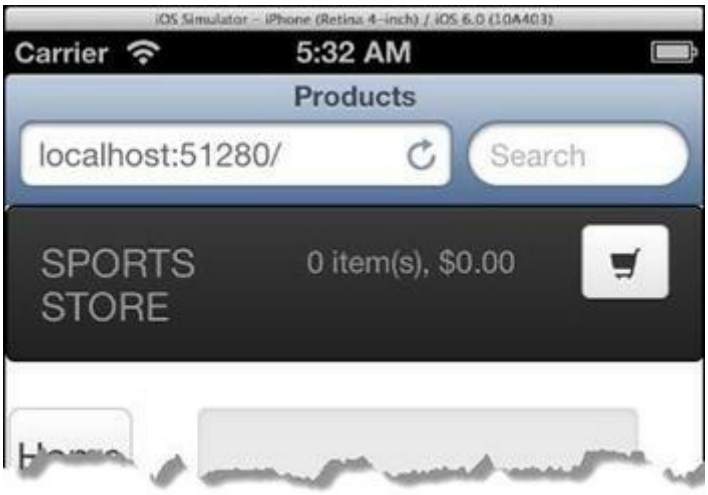


Figure 10-4. The modified SportsStore header displayed on an iPhone simulator

MOBILE FIRST VERSUS DESKTOP FIRST

Most web application projects start with desktop clients and then add support for mobile clients, just as I am do book. This is known as *desktop first* design/development and a common problem is that the server-side develop largely finished before work on the mobile client begins, resulting in an awkward mobile experience which is ha features designed for more capable desktop clients.

There is an alternative philosophy called *mobile first* design/development which, as the name suggests, starts wi client as the foundation for the application and adds features to take advantage of more capable desktop browse

Or to put it another way, desktop first tends to start with a full set of features and degrade gracefully for less ca devices, while mobile first tends to start with a smaller set of features that are gracefully enhanced for more cap

Both approaches have their merits and I tend to favor desktop first development because it is easy to get deskto load content from a local development workstation, something that can be surprisingly hard when working with hardware. I tend to work on a tight cycle of write-compile-check (which means reloading URLs into the brows and I get frustrated with the hoops one has to go through to get the same cycle going on a mobile device.

The danger in putting any group of users first is that you create a substandard experience for another group, jus pain around. Proponents of mobile first design often argue that this can't happen when you start with the basic then scale up, but that has not been my experience.

It is important to have a solid plan for what functionality and layout you are going to deliver to *all* devices *befor* developing for *any* of them. When you have such a plan, it doesn't matter what kind of device you begin with a the server-side parts of the application will be built from the ground up to support a full range of clients.

Creating a Responsive Product List

To complete my responsive adaptations, I need a product list that will display on narrow devices. The biggest proble is caused by the horizontal space that is taken up by the category buttons. I am going to move the buttons out of the individual product descriptions the whole width of the display. In Listing 10-3, you can see that I have further modif `_Layout.cshtml` file.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-sca
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
    <style>
        .navbar-right {
            float: right !important;
            margin-right: 15px; margin-left: 15px;
        }
    </style>
</head>
<body>
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">
            <span class="hidden-xs">SPORTS STORE</span>
            <div class="visible-xs">SPORTS</div>
            <div class="visible-xs">STORE</div>
        </a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="row panel">
        <div class="col-sm-3 hidden-xs">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-12 col-sm-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

There can only be one call to the `RenderBody` method in a layout. I get into the details of layouts in Chapter 2 effect of this limit is that I can't have duplicate sets of elements that I show and hide, each containing a `RenderB` Instead, I need to change the layout of the grid that contains the `RenderBody` method call so that the elements i adapt around the content from the view.

One of the reasons that I used the Bootstrap grid to structure the content in the `_Layout.cshtml` file in Ch it includes some responsive design features that let me work around the `RenderBody` limitation. The Bootstrap g supports 12 columns and you specify how many an element will occupy by applying a class, like this, which is how Bootstrap classes in Chapter 7:

```
...
<div class="col-xs-8" >
    @RenderBody()
</div>
...
```

Much like the `hidden-*` and `visible-*` classes that I described earlier, Bootstrap provides a set of class number of columns that an element occupies in the grid based on the width of the window.

The `col-xs-*` classes are fixed and don't change based on the width of the screen. My use of the `col-xs`

eBooks-IT.org

based on the width of the window. The col-sm-* classes set the columns when the window is 768 pixels or wi
col-md-* classes work on windows that are 992 pixels or wider and, finally, the col-lg-* classes work on
that are 1200 pixels or wider. With this in mind, here are the classes that I applied to the div element that surrounds
RenderBody method call from Listing 10-3:

```
...
<div class="col-xs-12 col-sm-8">
    @RenderBody()
</div>
...
```

The effect of applying both classes is that the div element will occupy all 12 columns in the grid by default and 8
when the screen is 768 pixels or wider. The other columns in the grid contain the category buttons, as follows:

```
...
<div class="col-sm-3 hidden-xs" >
    @Html.Action("Menu", "Nav")
</div>
...
```

This element will occupy 3 columns when the screen is wider than 768 pixels and be hidden otherwise. Combined
other classes I applied, the effect is that the product descriptions fill small windows and share the available space wit
category buttons for larger windows. You can see both layouts in Figure 10-5. I used a desktop browser to for this f
I am able to easily vary the width of the window.

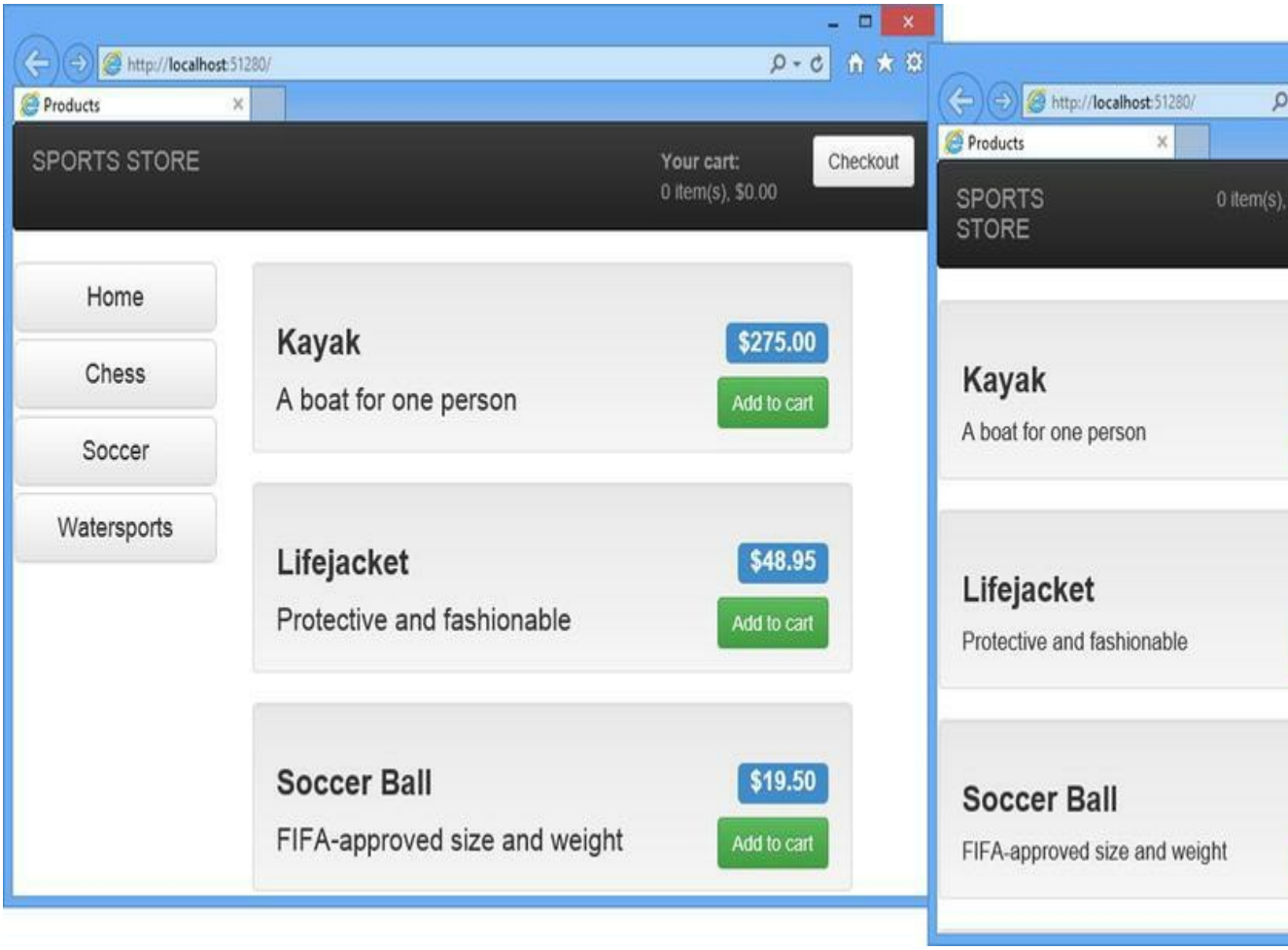


Figure 10-5. Using a responsive grid in the product layout

I don't want to leave mobile users without the ability to filter products, which means that I need to present the catego
different way. To do this, I created a new view called `MenuHorizontal.cshtml` in the `Views/Nav` fo
content shown in Listing 10-4.

Listing 10-4. The Contents of the MenuHorizontal.cshtml File

```
@model IEnumerable<string>

<div class="btn-group btn-group-sm btn-group-justified">
    @Html.ActionLink("Home", "List", "Product", new { @class = "b
default btn-sm" })

    @foreach (var link in Model) {
        @Html.RouteLink(link, new {
           controller = "Product",
           action = "List",
           category = link,
           page = 1
        }, new {
           @class = "btn btn-default btn-sm"
                + (link == ViewBag.SelectedCategory ? " btn-primar
        })
    }
</div>
```

This is a variation on the original `Menu.cshtml` layout, but with a container `div` element and some Bootstra
create a horizontal layout of the buttons. The basic functionality, however, is the same. I generate a set of links whic
the products by category.

The set of category buttons is generated through the `Menu` action method of the `Nav` controller, which I need to
that it selects the right view file based on the required orientation of the buttons, as shown in Listing 10-5.

Listing 10-5. Updating the Menu Action Method in the NavController.cs File

```
using System.Collections.Generic;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using System.Linq;

namespace SportsStore.WebUI.Controllers {

    public class NavController : Controller {
        private IProductRepository repository;

        public NavController(IProductRepository repo) {
            repository = repo;
        }

        public PartialViewResult Menu(string category = null,
            bool horizontalLayout = false) {

            ViewBag.SelectedCategory = category;

            IEnumerable<string> categories = repository.Products
```

```
                                                .Distinct()
                                                .OrderBy(x => x);

                    string viewName = horizontalLayout ? "MenuHoriz
"Menu";
                return PartialView(viewName, categories);
            }
        }
}
```

I have defined a new parameter for the action method that specifies the orientation, which I use to select the name file passed to the `PartialView` method. To set the value of this parameter, I need to return to the `_Layout` file, as shown in Listing 10-6.

Listing 10-6. Updating the _Layout.cshtml File to Include the Horizontal Buttons

```
...
<body>
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">
            <span class="hidden-xs">SPORTS STORE</span>
            <div class="visible-xs">SPORTS</div>
            <div class="visible-xs">STORE</div>
        </a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="visible-xs">
        @Html.Action("Menu", "Nav", new { horizontalLayout = true
    </div>
    <div class="row panel">
        <div class="col-sm-3 hidden-xs">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-12 col-sm-8">
            @RenderBody()
            </div>
    </div>
</body>
...
```

The optional third argument to the `Html.Action` method is an object that lets me set values for the routing s I explain in Chapters 15 and 16. I use this feature to signal which view the controller should select. The overall effec changes is shown in Figure 10-6.

eBooks-IT.org

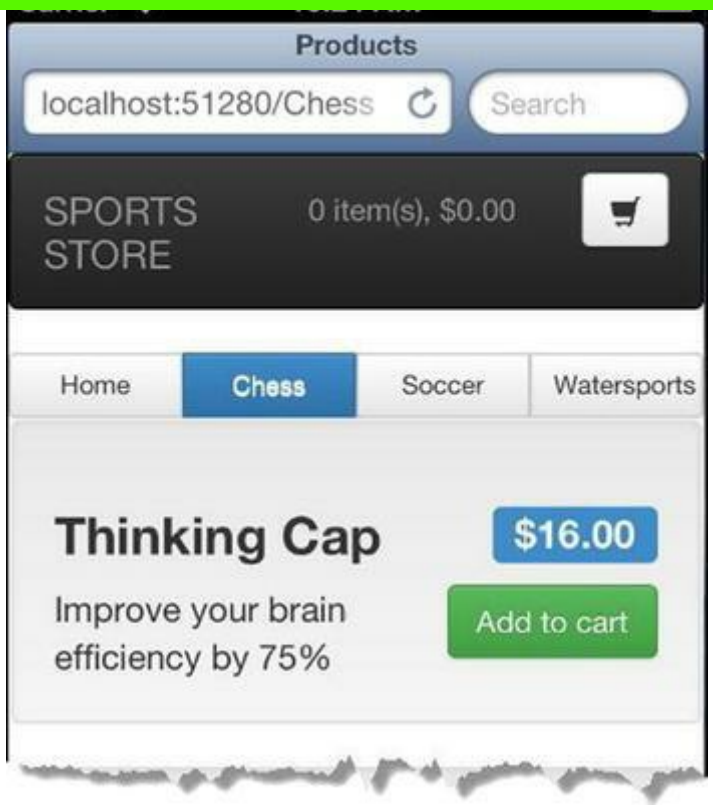


Figure 10-6. The revised product listing for small screens

You can see that moving the buttons to the top of the product listing creates enough space for each product to be properly. I could continue improving the fit and finish of the views, but you get the idea. Aside from a brief demonst responsive CSS classes can be used, I wanted to touch upon some of the limitations that the MVC Framework impo the `RenderBody` method limit) and some of the facilities it can provide to assist generating content in different w passing data from a view to a controller via the routing system and the `Html.Action` helper method).

Note I have focused on one orientation for the iPhone, but don't forget that most mobile devices allow for multip orientations and that you will have to cater for them all in a real project.

Removing View Duplication

In the previous example, I wanted to show you how you can have a controller select a view based on routing inform by a call to the `Html.Action` helper method. It is an important and useful feature, but I would not have used it project because it leaves me with two views, `Menu.cshtml` and `MenuHorizontal.cshtml`, that con similar markup and Razor expressions. This is a maintenance risk because any changes that I require to the category will have to be applied in two places. To resolve this I am going to consolidate the views. I created a new view file c `FlexMenu.cshtml` in the `Views/Nav` folder with the content shown in Listing 10-7.

Listing 10-7. The Contents of the FlexMenu.cshtml File

```
@model IEnumerable<string>

@{
bool horizontal = ((bool)(ViewContext.RouteData.Values["horizontal
?? false));
string wrapperClasses = horizontal ? "btn-group btn-group-sm bt
justified" : null;
}
```

```
    @Html.ActionLink("Home", "List", "Product",
    new { @class =  horizontal ? "btn btn-default btn-sm" :
        "btn btn-block btn-default btn-lg"
     })

    @foreach (var link in Model) {
        @Html.RouteLink(link, new {
            controller = "Product",
            action = "List",
            category = link,
            page = 1
        }, new {
            @class = (horizontal ? "btn btn-default btn-sm"
                : "btn btn-block btn-default btn-lg" )
                + (link == ViewBag.SelectedCategory ? " btn-primar
        })
    }

</div>
```

The cost of removing duplication is a more complex view that can generate both orientations of buttons and it is a personal preference as to which approach you take. If you are like me and prefer to avoid duplication, then this listin several useful features you can apply to views.

The first is the ability to access routing information directly from the view. The `ViewContext` property prov information about the current state of the request that is being processed, including details of the routing information,

```
...
bool horizontal = ((bool)(ViewContext.RouteData.Values["horizontal
?? false));
...
```

The second feature is the ability to create local variables within a view. This is possible because of the way that Ra compiled into classes (which I describe in Chapter 20), and I have created a local variable called `horizontal` th don't have to check the route data throughout the listing to figure out which orientation the view is being used for.

Caution Local variables should be used sparingly because it is the slippery slope into creating views that are harc and hard to test, but I sometimes use them in situations like this where I see them as an acceptable cost of simplifyin

A related feature is the way that Razor will conditionally set attributes based on variables. I defined a string of clas local variable in the view like this:

```
...
string wrapperClasses = horizontal ? "btn-group btn-group-sm bt
justified" : null;
...
```

The value of the `wrapperClasses` variable is either the string of class names that I used for horizontal layo `null`. I apply this variable to the class attribute like this:

```
...
<div class="@wrapperClasses">
...
```

When the variable is `null`, Razor is smart enough to remove the `class` attribute from the `div` element entire an element like this:

When the variable is not null, Razor will insert the value and leave the class attribute intact, producing a res

```
<div class="btn-group btn-group-sm btn-group-justified">
```

This is a nice way of matching the characteristics of the C# with the semantics of HTML and is a feature that is e useful when writing complex views because it won't insert null values into attributes and it won't generate empty which can cause problems with CSS selectors (and JavaScript libraries that use attributes to select elements, such as

Note Conditional attributes will work on any variable, not just the ones you have defined in the view. This means apply this feature to model properties and the view bag.

To use my consolidate view, I need to revise the Menu action method in the Nav controller, as shown in Listing

Listing 10-8. Updating the Menu Action in the NavController.cs File

```
...
public PartialViewResult Menu(string category = null) {

    ViewBag.SelectedCategory = category;

    IEnumerable<string> categories = repository.Products
                    .Select(x => x.Category)
                    .Distinct()
                    .OrderBy(x => x);

    return PartialView("FlexMenu", categories);
}
...
```

I removed the parameter that receives the orientation and changed the call to the PartialView method so th FlexMenu view is always selected. The result of these changes doesn't alter the layout of the content or the effe responsive design, but it does remove the duplication in the views and means that you can delete the Menu.csht MenuHorizontal.cshtml views from the Visual Studio project. Both orientations of category filter buttor produced by the FlexMenu.cshtml view.

THE LIMITATIONS OF RESPONSIVE DESIGN

There are some problems with responsive design as a way to support mobile clients. The first is that you end up lot of content and sending it to the server so that it can be displayed in different scenarios. You saw this in the p section when the HTML generated by the layout contained multiple sets of elements for the page header and the filter buttons. The extra elements don't amount to much on a per-request basis, but the overall effect for a busy a sharp increase in the amount of bandwidth you will need to provision, with the corresponding increase in runn

The second problem is that responsive design can be fiddly and it requires endless testing to get right. Not all de the underlying CSS features that enable responsive design (known as *media queries*) properly and, unless you ar and careful, you will end up with an application that delivers an adequate experience on every device without ex of them, a kind of blandness that comes from averaging out all of the device quirks.

Responsive design can be useful when applied thoughtfully, but it can easily result in an application that is riddle compromises that don't deliver a good experience for any of your target users.

Creating Mobile Specific Content

Responsive design delivers the same content to all devices and uses CSS to figure out how that content should be pre process that doesn't involve the server-side part of the application and which assumes that you want to treat all devic

eBooks-IT.org

send different HTML to different kinds of client. This works well if you want to present a completely different aspec application on the desktop to, say, a tablet.

Note You don't have to choose between responsive design and mobile-specific content and, in most projects, yo use both to get a good result on the devices you target. As an example, you may decide to create content specifically and use responsive design to create the horizontal and vertical orientations that most tablets support.

The MVC Framework supports a feature called *display modes*, which allows you to create different views that are based on the client that has made the request, a feature provided by the ASP.NET Framework. I explain how you ca manage display modes in depth in my *Pro ASP.NET MVC 5 Platform* book, but for the SportsStore application, I an the simplest form of display modes, which is to treat all mobile devices as being the same. My goal will be to deliver to mobile devices using the popular jQuery Mobile library, while keeping the existing content for desktop devices.

Note I am not going to go into any detail about jQuery Mobile in this book, other than to demonstrate how it can deliver mobile-specific content. For full details of jQuery Mobile, see my *Pro jQuery 2.0* book, published by Apress.

Creating a Mobile Layout

All I have to do to create mobile-specific content is to create views and layouts that have a `.Mobile.cshtml` created a new layout called `_Layout.Mobile.cshtml` in the `Views/Shared` folder with the content Listing 10-9.

Note Because the name of the view contains an additional period, you will need to create the view by right-clicki `Shared` folder and selecting `Add` ➤ `MVC 5 Layout Page (Razor)` from the pop-up menu.

Listing 10-9. The Contents of the _Layout.Mobile.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-sca
    <link rel="stylesheet"
                  href="http://code.jquery.com/mobile/1.3.2/jquery
1.3.2.min.css" />
    <script src="http://code.jquery.com/jquery-1.9.1.min.js"></scr
    <script
                   src="http://code.jquery.com/mobile/1.3.2/jquery
1.3.2.min.js"></script>
    <title>@ViewBag.Title</title>
</head>
<body>
    <div data-role="page" id="page1">
        <div data-theme="a" data-role="header" data-position="fixe
            <h3>SportsStore</h3>
            @Html.Action("Menu", "Nav")
        </div>
        <div data-role="content">
                    <ul  data-role="listview"  data-divider-theme="b
inset="false">
                @RenderBody()
            </ul>
        </div>
    </div>
</body>
```

This layout uses jQuery Mobile, which I obtain using a content delivery network (CDN) so that I don't have to ins package for the JavaScript and CSS files I need.

Note I am just scratching the surface by creating mobile-specific views, because I am using the same controller methods that are used for desktop clients. Having separate views allows you to introduce different controllers, which to a set of clients, and this can be used to create totally different features and functionality for different types of clier

The MVC Framework will automatically identify mobile clients and use the `_Layout.Mobile.cshtml` f rendering views, seamlessly replacing the `_Layout.cshtml` file which is used for other clients. You can see t the change in Figure 10-7.

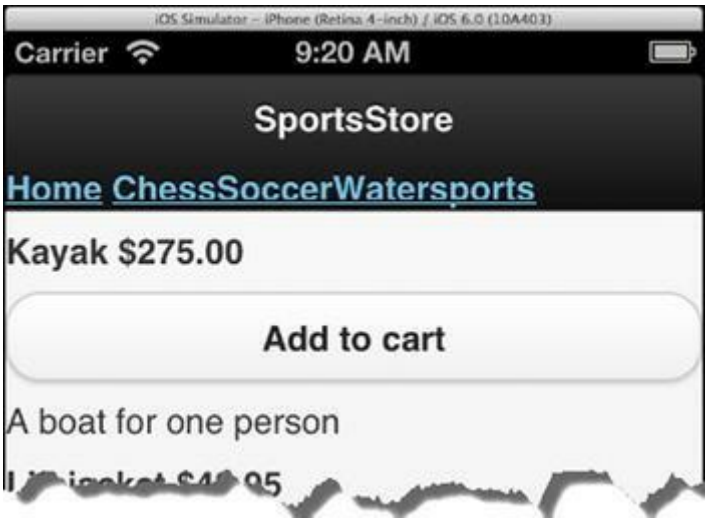


Figure 10-7. The effect of creating a mobile layout in the SportsStore application

You can see that the layout is different, but the overall effect is a mess, and that's because I need to create mobile the main view that is handling the request and the partial view that is used for the category filtering buttons.

Creating the Mobile Views

I am going to start with the category filtering, which means creating a view called `FlexMenu.Mobile.csh` `Views/Nav` folder with the content shown in Listing 10-10.

Listing 10-10. The Contents of the FlexMenu.Mobile.html File

```
@model IEnumerable<string>

<div data-role="navbar">
    <ul>
        @foreach (var link in Model) {
            <li>
                @Html.RouteLink(link, new {
                    controller = "Product",
                    action = "List",
                    category = link,
                    page = 1
                }, new {
                    data_transition = "fade",
                    @class = (link == ViewBag.SelectedCategory
                        ? "ui-btn-active" : null)
                })
```

eBooks-IT.org

```
        }
    </ul>
</div>
```

This view uses a Razor `foreach` expression to generate `li` elements for the product categories, producing ele
are organized in the way that jQuery Mobile expects for a navigation bar at the top of the page. You can see the effec
10-8.

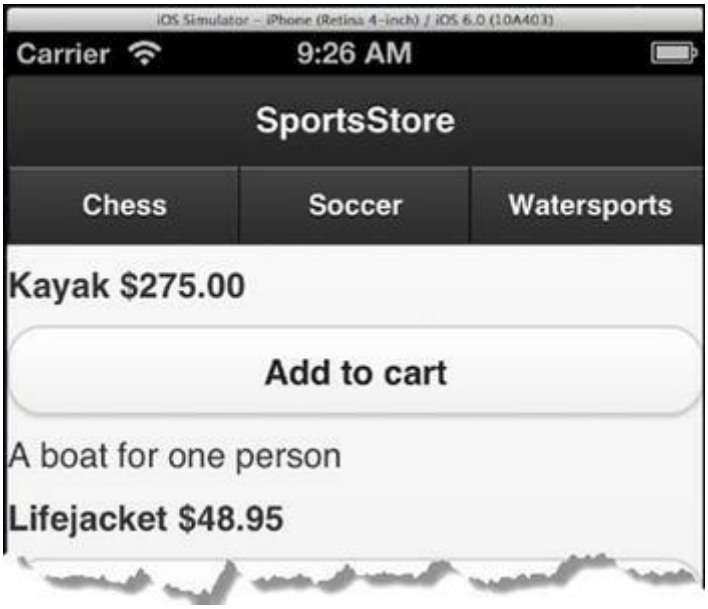


Figure 10-8. The effect of creating a mobile-specific view

Note jQuery Mobile relies on the use of data attributes to format elements. Data attributes are prefixed with `dat`
an unofficial way of defining custom attributes for years before becoming an official part of the HTML5 standard. In
needed to add a `data-transition` attribute to the `li` elements, but I can't use `data-transition` a
name for the anonymous object because this would be a C# expression. The problem is the hyphen and Razor works
by translating underscores in the property names to hyphens in attribute names, such that I was able to use
`data_transition` in the listing and get a `data-transition` attribute on the elements I generate.

The product information is still a mess, but the category buttons are now being generated by the new mobile-spec
worth taking a moment to reflect on what the MVC Framework is doing to render the content in Figure 10-8.

The HTTP request from the browser targets the `List` action method in the `Product` controller, which tells t
Framework to render the `List.cshtml` view file. The MVC Framework knows that the request came from a m
browser and so it starts looking for mobile-specific views. There is no `List.Mobile.cshtml`, and so the
`List.cshtml` file is processed instead. This view relies on the `_Layout.cshtml` file, but the MVC Fram
notices that there is a mobile-specific version available and so it uses `_Layout.Mobile.cshtml` instead. T
requires `the FlexMobile.cshtml` file but there is a mobile version of that as well, and so on.

The result is that the response to the browser is generated from a mix of mobile-specific and general views, with t
Framework using the most specific view file available, but seamlessly falling back when needed.

THE TWO PROBLEMS IN THE EXAMPLE

The example in this chapter is intended to demonstrate the way that the MVC Framework can deliver mobile-sp
but I would be remiss if I didn't point out two potentially serious problems that this example introduces to the S
application.

The first is that I have provided less functionality in my mobile views than in the desktop ones. There is no cart
the page header, for example. I left some features out to simplify the changes I had to make, but I recommend a
delivering reduced functionality to *any* device unless there is a technical limitation that prevents the device from
support it. Mobile devices are increasingly capable and many users will only access your application with a mob
The days when you could assume that mobile access was a supplement to desktop use have passed.

eBooks-IT.org

the number of users that would prefer to have the desktop layout on a mobile device, even though it might be a Don't u and require some zooming and scrolling on smaller screens. Some mobile devices allow larger monitors to be co example, and this is rarely detected by the mechanism that the ASP.NET Framework uses to identify mobile dev should always give mobile users the choice about which layout they receive.

Neither of these issues prevents me from deploying my application, but they are the kind of frustration that plag web application users. Mobile devices will be a big part of any modern web application and you should take eve to deliver a good user experience to this important category of users.

My last change is to create a mobile-specific version of the view that generates the product summary. I created a called `ProductSummary.Mobile.cshtml` in the `Views/Shared` folder with the contents shown 11.

Listing 10-11. The Contents of the ProductSummary.Mobile.cshtml File

```
@model SportsStore.Domain.Entities.Product

<div    data-role="collapsible"    data-collapsed="false"    data-
theme="c">
    <h2>
        @Model.Name
    </h2>
    <div class="ui-grid-b">
        <div class="ui-block-a">
            @Model.Description
        </div>
        <div class="ui-block-b">
            <strong>(@Model.Price.ToString("c"))</strong>
        </div>
        <div class="ui-block-c">
            @using (Html.BeginForm("AddToCart", "Cart")) {
                @Html.HiddenFor(x => x.ProductID)
                @Html.Hidden("returnUrl", Request.Url.PathAndQuery
                <input type="submit" data-inline="true"
                       data-mini="true" value="Add to cart" />
            }
        </div>
    </div>
</div>
```

This view uses a jQuery Mobile widget that allows users to open and collapse regions of content. It isn't an ideal v presenting product information, but it is simple and my emphasis in this section is on mobile-specific content rather t jQuery Mobile library. You can see the effect of this new view in Figure 10-9.

eBooks-IT.org

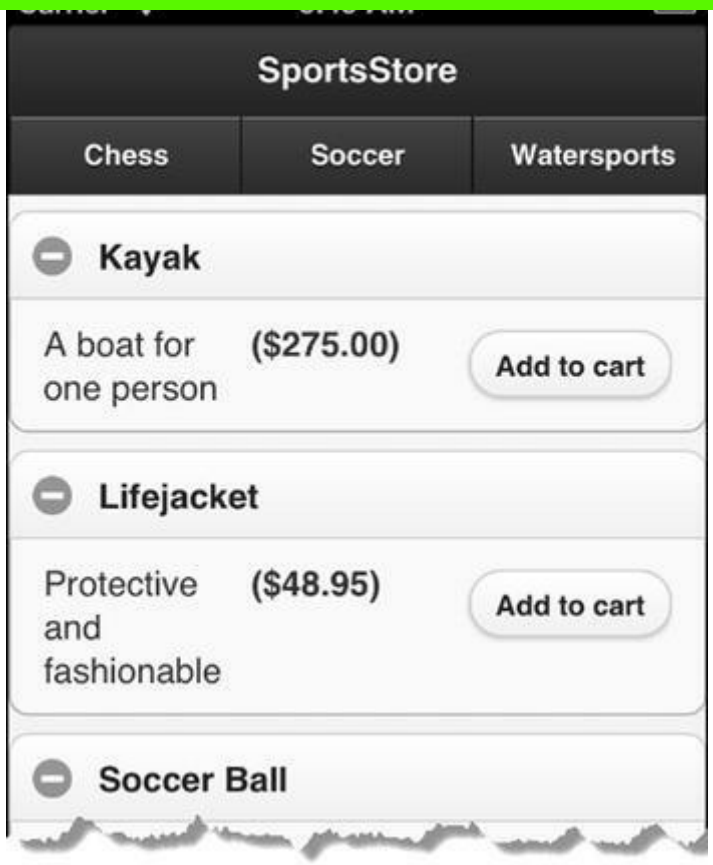


Figure 10-9. The effect of mobile-specific views

In a real project, I would carry on, of course, and create mobile-specific versions of the views that display the pag the shopping cart and the checkout form. I am not going to because you have already seen how the MVC Framewor target mobile devices.

Summary

In this chapter I have shown you two techniques for handling mobile devices: responsive design and mobile-specific Responsive design isn't directly related to the MVC Framework, which sends the same content to all browsers and l figure out what to do with it. But as I demonstrated, there are some limitations in the way that views work that requi thought and some nice Razor features that can ease the overall process.

Creating mobile-specific content is something that the MVC Framework does actively participate in by automatical mobile-specific views and layouts if they are available and blending them seamlessly into the process that renders HT clients. In the next chapter, I add the basic features required to administer the SportsStore product catalog.

eBooks-IT.org

SportsStore: Administration

In this chapter, I continue to build the SportsStore application in order to give the site administrator a way of managi product catalog. I will add support for creating, editing, and removing items from the product repository, as well as and displaying images alongside products in the catalog.

Adding Catalog Management

The convention for managing collections of items is to present the user with two types of pages: a *list* page and an *e* shown in Figure 11-1.

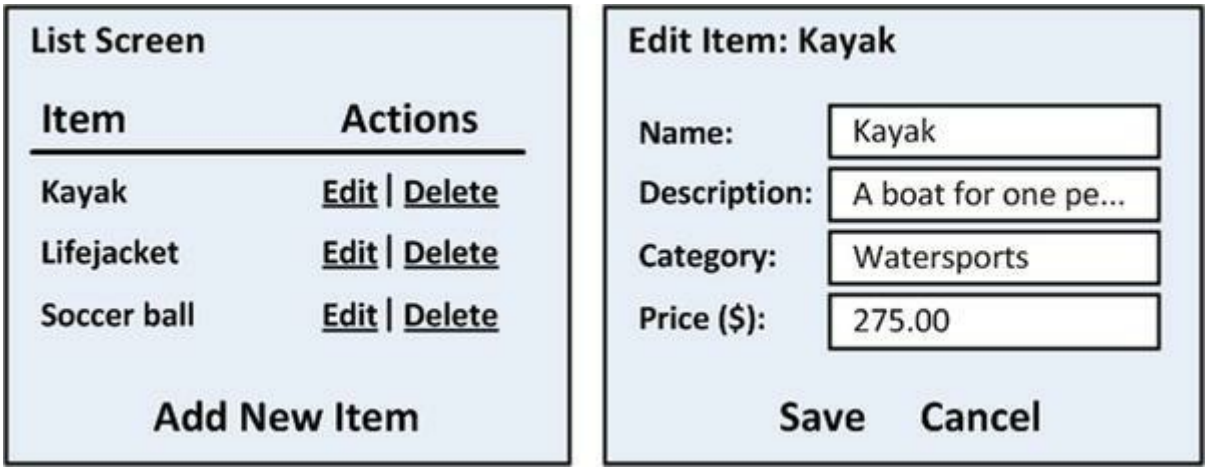


Figure 11-1. Sketch of a CRUD UI for the product catalog

Together, these pages allow a user to create, read, update, and delete items in the collection. Collectively, these act known as *CRUD*. Developers need to implement CRUD so often that Visual Studio tries to help by offering to genera controllers that have action methods for CRUD operations and view templates that support them. But like all of the V templates, I think it is better to learn how to use the features of the MVC Framework directly.

Creating a CRUD Controller

I am going to create a new controller for the SportsStore administration features. Right-click on the `Controlle` the `SportsStore.WebUI` project in the Solution Explorer and select `Add` ➤ `Controller` from the po Select `MVC 5 Controller - Empty` from the list of options, click the `Add` button, set the name to `AdminController` and click the `Add` button to create the `Controllers/AdminController.` the contents of the controller class to match Listing 11-1.

Listing 11-1. The Contents of the AdminController.cs File

```
using System.Web.Mvc;
using SportsStore.Domain.Abstract;

namespace SportsStore.WebUI.Controllers {
```

eBooks-IT.org

```
    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo) {
            repository = repo;
        }

        public ViewResult Index() {
            return View(repository.Products);
        }
    }
}
```

The controller constructor declares a dependency on the `IProductRepository` interface, which Ninject when instances are created. The controller defines a single action method, `Index`, that calls the `View` method to default view for the action, passing the set of products in the database as the view model.

UNIT TEST: THE INDEX ACTION

The behavior that I care about for the `Index` method of the `Admin` controller is that it correctly returns the objects that are in the repository. I can test this by creating a mock repository implementation and comparing th with the data returned by the action method. Here is the unit test, which I placed into a new unit test file called `AdminTests.cs` in the `SportsStore.UnitTests` project:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;

namespace SportsStore.UnitTests {
  [TestClass]
  public class AdminTests {

    [TestMethod]
    public void Index_Contains_All_Products() {
      // Arrange - create the mock repository
      Mock<IProductRepository> mock = new Mock<IProductRepository
      mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
      });

      // Arrange - create a controller
      AdminController target = new AdminController(mock.Object);

      // Action
      Product[] result = ((IEnumerable<Product>)target.Index().
        ViewData.Model).ToArray();

      // Assert
```

eBooks-IT.org

```
            Assert.AreEqual("P1", result[0].Name);
            Assert.AreEqual("P2", result[1].Name);
            Assert.AreEqual("P3", result[2].Name);
        }
    }
}
```

Creating a New Layout

I am going to create a new layout to use with the SportsStore administration views. It will be a simple layout that pro point where I can apply changes to all the administration views.

Create the new layout, by right-clicking the `Views/Shared` folder in the `SportsStore.WebUI` proje `Add > MVC 5 Layout Page (Razor)` from the pop-up menu. Set the name to `_AdminLayout` (don't forget the underscore) and click the `OK` button to create the `Views/Shared/_AdminLayout.cs` Set the contents of the new view to match Listing 11-2.

Note As I explained previously, the convention is to start the layout name with an underscore (_). Razor is also another Microsoft technology called WebMatrix, which uses the underscore to prevent layout pages from being serv browsers. MVC does not need this protection, but the convention for naming layouts is carried over to MVC applicat

Listing 11-2. The Contents of the _AdminLayout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title></title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

I have added a call to the `RenderBody` method, so that the contents of the view that the layout is being used f inserted into the response to the server. (I would not have had to do this if I had used the `Add ➤ New Item` me used the Visual Studio layout template, but I took a shortcut to create the view directly, which meant I had to edit the get the content I require.) I also added link elements for the Bootstrap files and for the CSS file that contains the style highlight validation errors to the user.

Implementing the List View

Now that I have the new layout, I can add a view to the project for the `Index` action method of the `Admin` cont though I am not a fan of the Visual Studio scaffold and template features, I am going to create the view for the `Ind` using the scaffold system so you can see how it works. Just because I don't like pre-cut code, doesn't mean that yo use it.

eBooks-IT.org

menu. Set View Name to Index, select the List for the Template option (this is where I usually select Empty options), select Product as the Model Class, check the option to use a layout page, and select th _AdminLayout.cshtml file from the Views/Shared folder. You can see all of the configuration opti 11-2.

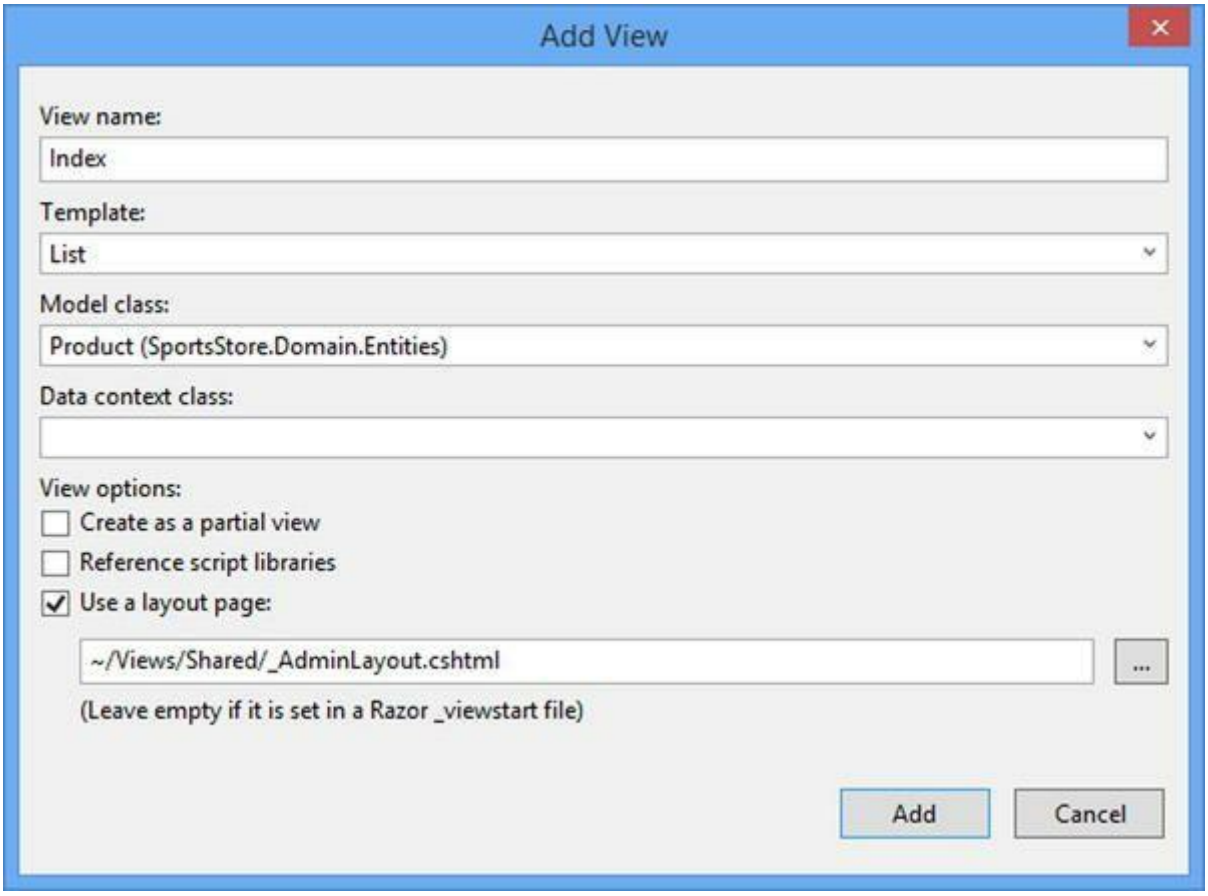


Figure 11-2. Configuring a scaffold view

Note When using the List scaffold, Visual Studio assumes you are working with an IEnumerable sequ model view type, so you can just select the singular form of the class from the list.

Click the Add button to create the new view, which will have the contents shown in Listing 11-3. (I have format markup so that it doesn't require so much space on the page.)

Listing 11-3. The Contents of the Views/Admin/Index.cshtml File

```
@model IEnumerable<SportsStore.Domain.Entities.Product>

@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>@Html.DisplayNameFor(model => model.Name)</th>
```

eBooks-IT.org

```
            <th>@Html.DisplayNameFor(model => model.Price)</th>
            <th>@Html.DisplayNameFor(model => model.Category)</th>
            <th></th>
        </tr>

    @foreach (var item in Model) {
        <tr>
            <td>@Html.DisplayFor(modelItem => item.Name)</td>
            <td>@Html.DisplayFor(modelItem => item.Description)</td>
            <td>@Html.DisplayFor(modelItem => item.Price)</td>
            <td>@Html.DisplayFor(modelItem => item.Category)</td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.Product
                @Html.ActionLink("Details", "Details", new { id=item.P
})  |
                  @Html.ActionLink("Delete", "Delete", new { id=item.P
})
            </td>
        </tr>
    }

    </table>
```

Visual Studio looks at the type of view model object and generates elements in a table that correspond to the prope by the model type. You can see how this view is rendered by starting the application and navigating to the `/Admin` URL. Figure 11-3 shows the results.

eBooks-IT.org

http://localhost:39534/Admin/Index

localhost

Index

Create New

Name	Description	Price	Category	
Kayak	A boat for one person	275.00	Watersports	Edit | Deta
Lifejacket	Protective and fashionable	48.95	Watersports	Edit | Deta
Soccer Ball	FIFA-approved size and weight	19.50	Soccer	Edit | Deta
Corner Flags	Give your playing field a professional touch	34.95	Soccer	Edit | Deta
Stadium	Flat-packed 35,000-seat stadium	79500.00	Soccer	Edit | Deta
Thinking Cap	Improve your brain efficiency by 75%	16.00	Chess	Edit | Deta
Unsteady Chair	Secretly give your opponent a disadvantage	29.95	Chess	Edit | Deta
Human Chess Board	A fun game for the family	75.00	Chess	Edit | Deta
Bling-Bling King	Gold-plated, diamond-studded King	1200.00	Chess	Edit | Deta

Figure 11-3. Rendering the scaffold List view

The scaffold view makes a reasonable attempt at setting up a sensible baseline for the view. I have columns for ea properties in the `Product` class and links for the other CRUD operations that target action methods in the `Admi` (although, since I created that controller without using scaffolding, the action methods do not exist).

The scaffolding is clever, but the views that it generates are bland and so general as to be worthless in a project of complexity. My advice is to start with empty controllers, views and layouts and add the functionality you need as an need it.

Returning to the do-it-yourself approach, edit the `Index.cshtml` file so that it corresponds to Listing 11-4.

Listing 11-4. Modifying the Index.cshtml View

```
@model IEnumerable<SportsStore.Domain.Entities.Product>

@{
    ViewBag.Title = "Admin: All Products";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
```

```
<div class="panel panel-default">

    <div class="panel-heading">
        <h3>All Products</h3>
    </div>
    <div class="panel-body">
        <table class="table table-striped table-condensed table-bo
            <tr>
                <th class="text-right">ID</th>
                <th>Name</th>
                <th class="text-right">Price</th>
                <th class="text-center">Actions</th>
            </tr>
            @foreach (var item in Model) {
                <tr>
                    <td class="text-right">@item.ProductID</td>
                            <td>@Html.ActionLink(item.Name,  "Edit"
item.ProductID })</td>
                    <td class="text-right">@item.Price.ToString("c
                    <td class="text-center">
                        @using (Html.BeginForm("Delete", "Admin"))
                            @Html.Hidden("ProductID", item.Product
                            <input type="submit"
                                   class="btn btn-default btn-xs"
                                   value="Delete" />
                        }
                    </td>
                </tr>
            }
        </table>
    </div>
    <div class="panel-footer">
        @Html.ActionLink("Add a new product", "Create", null,
            new { @class = "btn btn-default" })
    </div>
</div>
```

This view presents the information in a more compact form, omitting some of the properties from the `Product` using Bootstrap to apply styling. You can see how this view renders in Figure 11-4.

eBooks-IT.org

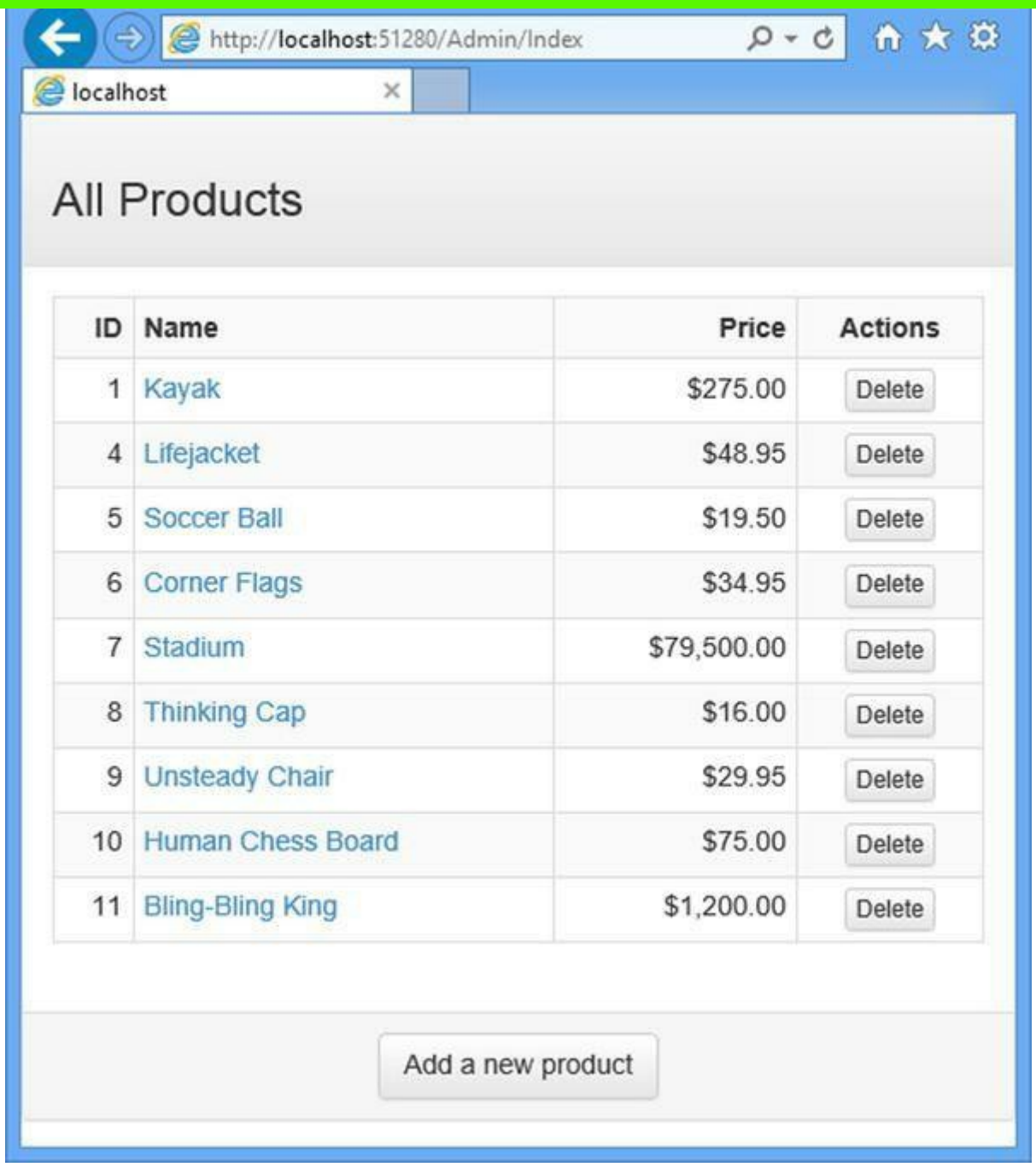


Figure 11-4. Rendering the modified Index view

Now I have a nice list page. The administrator can see the products in the catalog and there are links or buttons to and inspect items. In the following sections, I will add the functionality to support each of these actions.

Editing Products

To provide create and update features, I will add a product-editing page similar to the one shown in Figure 11-1. The parts to this job:

- Display a page that will allow the administrator to change values for the properties of a product
- Add an action method that can process those changes when they are submitted

Creating the Edit Action Method

eBooks-IT.org

Html.ActionLink helper method in the Index view.

Listing 11-5. Adding the Edit Action Method in the AdminController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo) {
            repository = repo;
        }

        public ViewResult Index() {
            return View(repository.Products);
        }

        public ViewResult Edit(int productId) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            return View(product);
        }
    }
}
```

This simple method finds the product with the ID that corresponds to the `productId` parameter and passes it model object to the `View` method.

UNIT TEST: THE EDIT ACTION METHOD

I want to test for two behaviors in the `Edit` action method. The first is that I get the product I ask for when I valid ID value. Obviously, I want to make sure that I am editing the product I expected. The second behavior is get any product at all when I request an ID value that is not in the repository. Here are the test methods I added `AdminTests.cs` unit test file:

```
...
[TestMethod]
public void Can_Edit_Product() {

  // Arrange - create the mock repository
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
  });

  // Arrange - create the controller
  AdminController target = new AdminController(mock.Object);
```

```
      Product p1 = target.Edit(1).ViewData.Model as Product;
      Product p2 = target.Edit(2).ViewData.Model as Product;
      Product p3 = target.Edit(3).ViewData.Model as Product;

      // Assert
      Assert.AreEqual(1, p1.ProductID);
      Assert.AreEqual(2, p2.ProductID);
      Assert.AreEqual(3, p3.ProductID);
    }

    [TestMethod]
    public void Cannot_Edit_Nonexistent_Product() {

      // Arrange - create the mock repository
      Mock<IProductRepository> mock = new Mock<IProductRepository>()
      mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
      });

      // Arrange - create the controller
      AdminController target = new AdminController(mock.Object);

      // Act
      Product result = (Product)target.Edit(4).ViewData.Model;

      // Assert
      Assert.IsNull(result);
    }
    ...
```

Creating the Edit View

Now that I have an action method, I can create a view for it to render. Right-click on the `Views/Admin` folder Explorer and select `Add` ➤ `MVC 5 View Page (Razor)` from the menu. Set the name to `Edit.csh` the button to create the file and edit the contents to match Listing 11-6.

Listing 11-6. The Contents of the Edit .cshtml File

```
@model SportsStore.Domain.Entities.Product

@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<h1>Edit @Model.Name</h1>

@using (Html.BeginForm()) {
    @Html.EditorForModel()
    <input type="submit" value="Save" />
    @Html.ActionLink("Cancel and return to List", "Index")
}
```

eBooks-IT.org

rFor

helper method. This method asks the MVC Framework to create the editing interface for me, which it does by inspec model type—in this case, the `Product` class. To see the page that is generated from the `Edit` view, run the ap navigate to `/Admin/Index`. Click one of the product name links, and you will see the page shown in Figure 11-

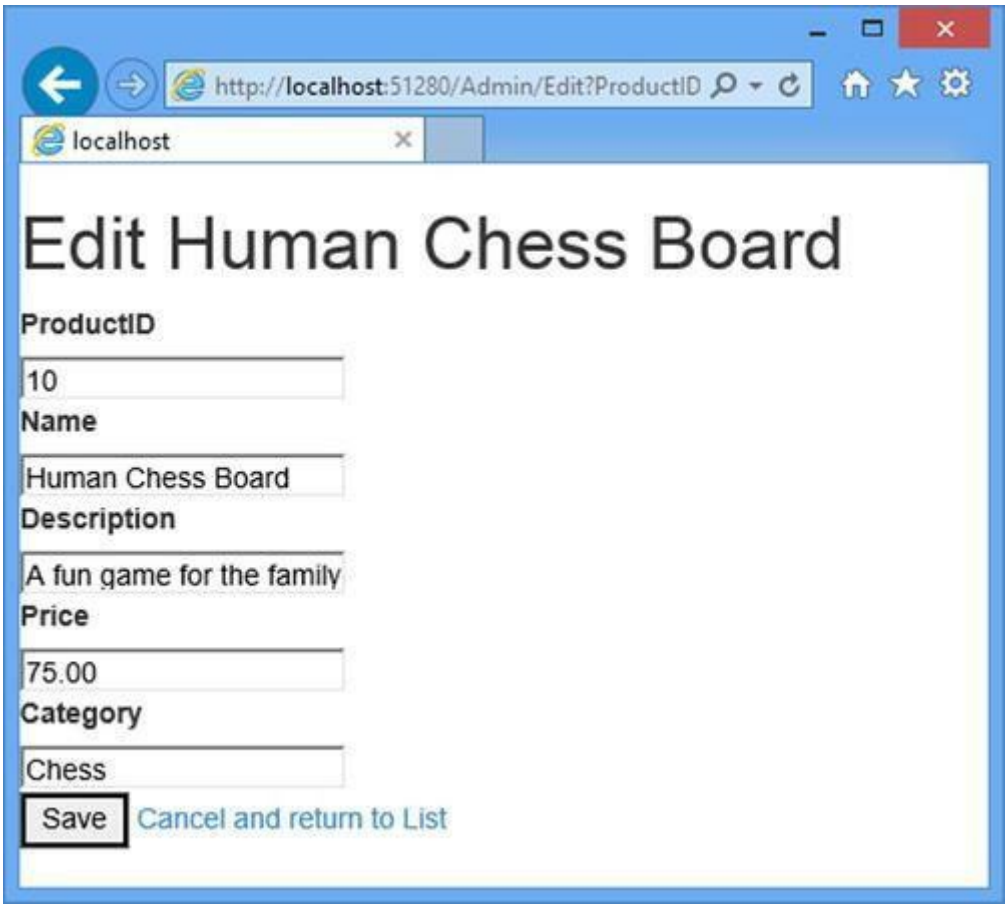


Figure 11-5. The page generated using the EditorForModel helper method

Let's be honest: the `EditorForModel` method is convenient, but it does not produce the most attractive res addition, I do not want the administrator to be able to see or edit the `ProductID` attribute, and the text box for th `Description` property is far too small.

I can give the MVC Framework directions about how to create editors for properties by using *model metadata*. Th to apply attributes to the properties of the new model class to influence the output of the `Html.EditorForMo` method. Listing 11-7 shows how to use metadata on the `Product` class in the `SportsStore.Domain` pr

Listing 11-7. Using Model Metadata in the Product.cs File

```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace SportsStore.Domain.Entities {

    public class Product {

        [HiddenInput(DisplayValue = false)]
        public int ProductID { get; set; }
        public string Name { get; set; }

        [DataType(DataType.MultilineText)]
        public string Description { get; set; }
        public decimal Price { get; set; }
```

```
    }
}
```

The `HiddenInput` attribute tells the MVC Framework to render the property as a hidden form element, and t
`DataType` attribute allows me to specify how a value is presented and edited. In this case, I have selected the
`MultilineText` option. The `HiddenInput` attribute is part of the `System.Web.Mvc` namespace a
`DataType` attribute is part of the `System.ComponentModel.DataAnnotations` namespace, w
had you add references to the assemblies for these namespace to the `SportsStore.Domain` project in Chap

Figure 11-6 shows the `Edit` page once the metadata has been applied. You can no longer see or edit the `Produ`
property, and there is a multiline text box for entering the description. However, the overall appearance is poor.

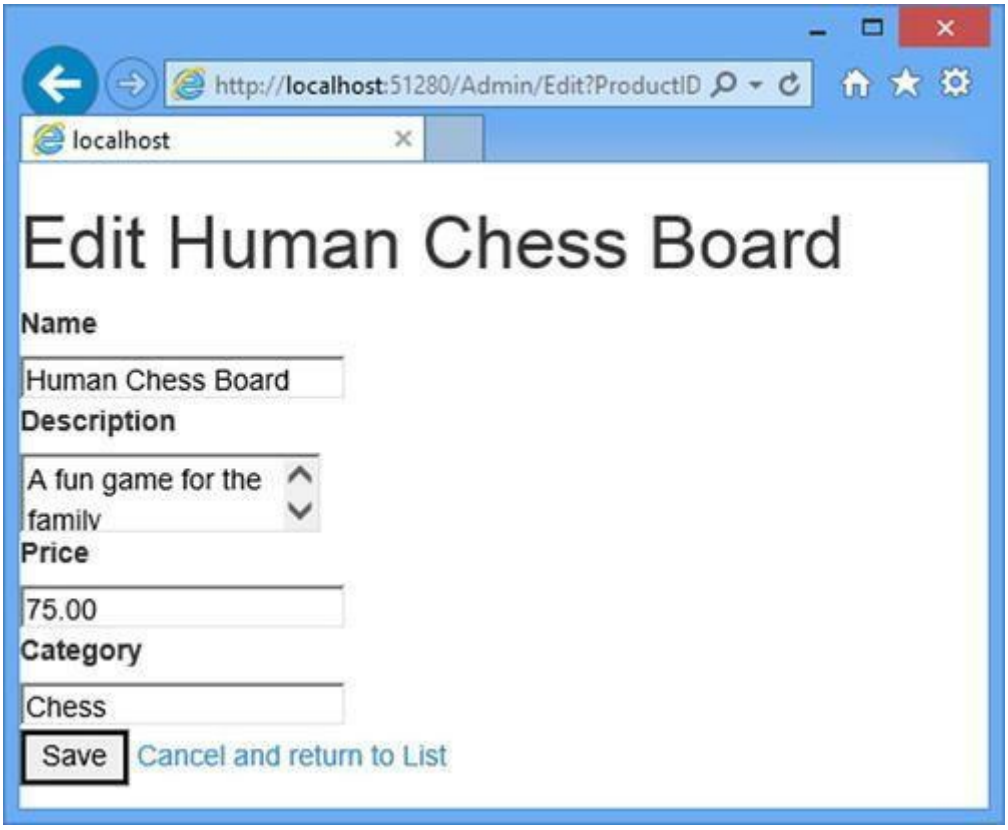


Figure 11-6. The effect of applying metadata

The problem is that the `Html.EditorForModel` helper doesn't know anything about the `Product` cla
generates some basic and safe HTML. There are three ways we can deal with this. The first is to define CSS styles f
that the helper generates, which is made easier by the classes that are automatically added to the HTML elements by
Framework.

If you look at the source for the page shown in Figure 11-6, you will see that the `textarea` element that has b
for the product description has been assigned to the `text-box-multi-line` CSS class:

```
<textarea class="text-box-multi-line" id="Description" name="Descr
```

Other HTML elements are assigned similar classes and you can create CSS styles for each of them. This approach
when you are creating custom styles, but doesn't make it easy to apply predefined classes like the ones that Bootstra

The second approach is to provide the helper with templates that it can use to generate the elements, including the
we require. I show you how to do this in Chapter 22.

The third approach is to create the elements needed directly, without using the model-level helper method. I like th
model helper, but I rarely use it, preferring to create the HTML myself and use helpers on the individual properties, a
Listing 11-8.

Listing 11-8. Updating the Edit.cshtml File

```
@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<div class="panel">
    <div class="panel-heading">
        <h3>Edit @Model.Name</h3>
    </div>

    @using (Html.BeginForm()) {
        <div class="panel-body">
        @Html.HiddenFor(m => m.ProductID)
        @foreach (var property in ViewData.ModelMetadata.Propertie
            if (property.PropertyName != "ProductID") {
                <div class="form-group">
                                        <label>@(property.Display
property.PropertyName)</label>
                    @if (property.PropertyName == "Description") {
                        @Html.TextArea(property.PropertyName, null
                                new { @class = "form-control", row
                    } else {
                        @Html.TextBox(property.PropertyName, null,
                                new { @class = "form-control" })
                    }
                </div>
            }
        }
        </div>

        <div class="panel-footer">
            <input type="submit" value="Save" class="btn btn-prima
             @Html.ActionLink("Cancel and return to List", "Index
new {
                @class = "btn btn-default"
            })
        </div>
    }
</div>
```

This is a variation on the metadata technique that I used in Chapter 9 and it is something that I find myself using o though I could achieve similar results through the HTML helper methods with the customization techniques I describ 22. There is something pleasing about this approach that gels with my development style but, as ever with the MVC there are different approaches you can use if processing metadata doesn't do it for you. You can see how this view i the browser in Figure 11-7.

eBooks-IT.org

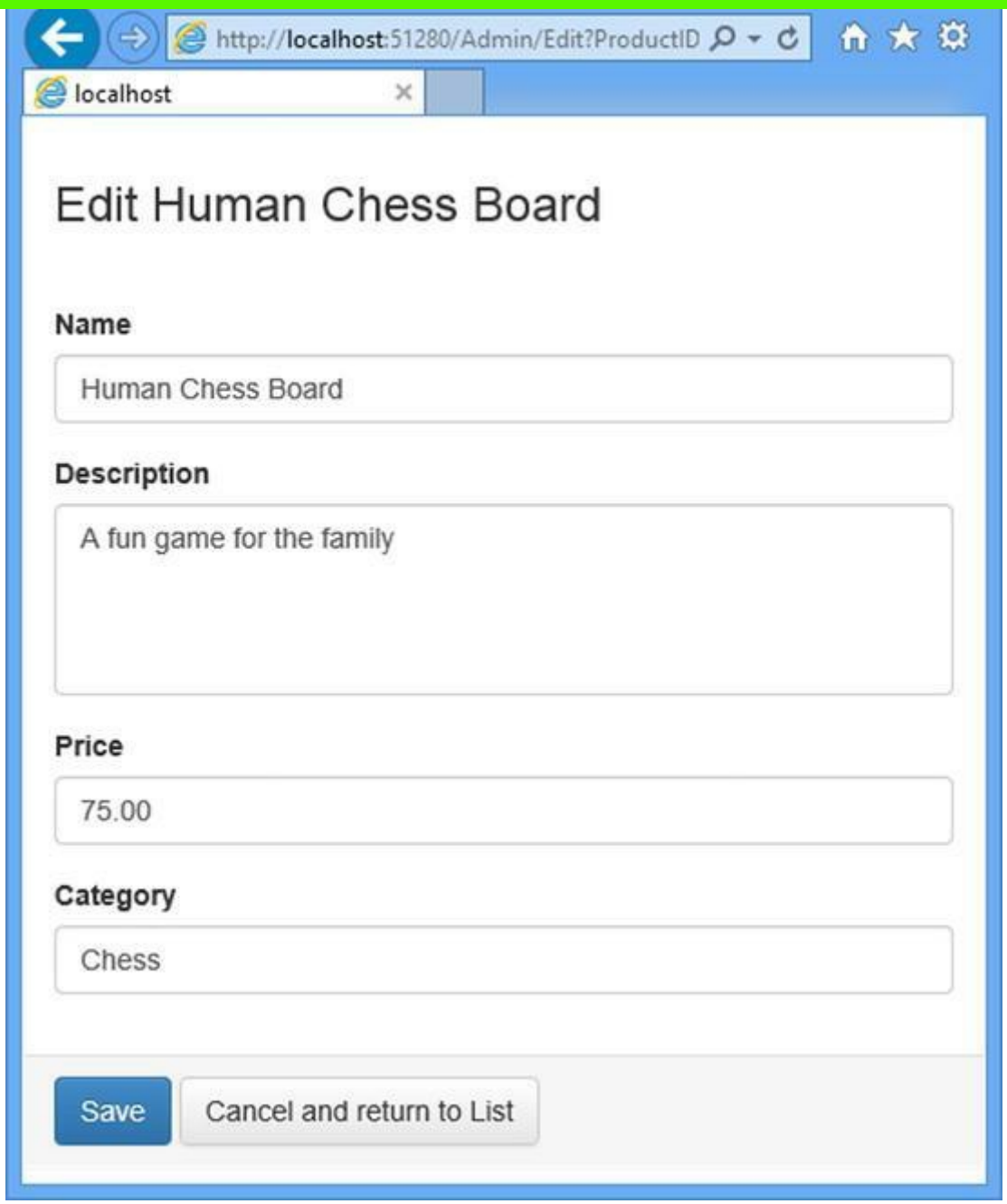


Figure 11-7. Displaying the editor page for products

Updating the Product Repository

Before I can process edits, I need to enhance the product repository so that it is able to save changes. First, I will ad method to the `IProductRepository` interface, as shown in Listing 11-9. (As a reminder, you will find this the `Abstract` folder of the `SportsStore.Domain` project.)

Listing 11-9. Adding a Method to the IProductRespository.cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract {
    public interface IProductRepository {
```

```
        void SaveProduct(Product product);
    }
}
```

I can then add the new method to the Entity Framework implementation of the repository, defined in the `Concrete/EFProductRepository.cs` file, as shown in Listing 11-10.

Listing 11-10. Implementing the SaveProduct Method in the EFProductRepository.cs File

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Collections.Generic;

namespace SportsStore.Domain.Concrete {

    public class EFProductRepository : IProductRepository {
        private EFDbContext context = new EFDbContext();

        public IEnumerable<Product> Products {
            get { return context.Products; }
        }

        public void SaveProduct(Product product) {

            if (product.ProductID == 0) {
                context.Products.Add(product);
            } else {
                                                                Product    dbE
context.Products.Find(product.ProductID);
                if (dbEntry != null) {
                    dbEntry.Name = product.Name;
                    dbEntry.Description = product.Description;
                    dbEntry.Price = product.Price;
                    dbEntry.Category = product.Category;
                }
            }
            context.SaveChanges();
        }
    }
}
```

The implementation of the `SaveChanges` method adds a product to the repository if the `ProductID` is 0 applies any changes to the existing entry in the database.

I do not want to go into details of the Entity Framework because, as I explained earlier, it is a topic in itself and no MVC Framework. But there is something in the `SaveProduct` method that has a bearing on the design of the M application.

I know I need to perform an update when I receive a `Product` parameter that has a `ProductID` that is not this by getting a `Product` object from the repository with the same `ProductID` and updating each of the pro match the parameter object.

I can do this because the Entity Framework keeps track of the objects that it creates from the database. The objec the `SaveChanges` method is created by the MVC Framework using the default model binder, which means that Framework does not know anything about the parameter object and will not apply an update to the database. There a ways of resolving this issue and I have taken the simplest one, which is to locate the corresponding object that the E

eBooks-IT.org

An alternative approach would be to create a custom model binder that only obtains objects from the repository. T like a more elegant approach, but it would require me to add a find capability to the repository interface so I could lo `Product` objects by `ProductID` values.

Handling Edit POST Requests

At this point, I am ready to implement an overload of the `Edit` action method in the `Admin` controller that will ha requests when the administrator clicks the `Save` button. The new method is shown in Listing 11-11.

Listing 11-11. Adding the POST-Handling Edit Action Method in the AdminController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo) {
            repository = repo;
        }

        public ViewResult Index() {
            return View(repository.Products);
        }

        public ViewResult Edit(int productId) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            return View(product);
        }

        [HttpPost]
        public ActionResult Edit(Product product) {
            if (ModelState.IsValid) {
                repository.SaveProduct(product);
                 TempData["message"] = string.Format("{0} has been
product.Name);
                return RedirectToAction("Index");
            } else {
                // there is something wrong with the data values
                return View(product);
            }
        }
    }
}
```

I check that the model binder has been able to validate the data submitted to the user by reading the value of the `ModelState.IsValid` property. If everything is OK, I save the changes to the repository, and invoke the I method to return the user to the list of products. If there is a problem with the data, I render the `Edit` view again s

eBooks-IT.org

After I have saved the changes in the repository, I store a message using the *TempData* feature. This is a key/valu similar to the session data and view bag features I used previously. The key difference from session data is that temp deleted at the end of the HTTP request.

Notice that I return the `ActionResult` type from the `Edit` method. I have been using the `ViewResul` now. `ViewResult` is derived from `ActionResult`, and it is used when you want the framework to rende However, other types of `ActionResults` are available, and one of them is returned by the `RedirectToA` method, which redirects the browser so that the `Index` action method is invoked. I describe the set of action resu 17.

I cannot use `ViewBag` in this situation because the user is being redirected. `ViewBag` passes data between th and view, and it cannot hold data for longer than the current HTTP request. I could have used the session data featu the message would be persistent until I explicitly removed it, which I would rather not have to do. So, the TempData perfect fit. The data is restricted to a single user's session (so that users do not see each other's `TempData`) and long enough for me to read it. I will read the data in the view rendered by the action method to which I have redirect which I define in the next section.

UNIT TEST: EDIT SUBMISSIONS

For the `POST`-processing `Edit` action method, I need to make sure that valid updates to the `Product` obj model binder has created are passed to the product repository to be saved. I also want to check that invalid upd model error exists) are not passed to the repository. Here are the test methods:

```
...
[TestMethod]
public void Can_Save_Valid_Changes() {


  // Arrange - create mock repository
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  // Arrange - create the controller
  AdminController target = new AdminController(mock.Object);
  // Arrange - create a product
  Product product = new Product {Name = "Test"};

  // Act - try to save the product
  ActionResult result = target.Edit(product);

  // Assert - check that the repository was called
  mock.Verify(m => m.SaveProduct(product));

  // Assert - check the method result type
  Assert.IsNotInstanceOfType(result, typeof(ViewResult));
}

[TestMethod]
public void Cannot_Save_Invalid_Changes() {

  // Arrange - create mock repository
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  // Arrange - create the controller
  AdminController target = new AdminController(mock.Object);
  // Arrange - create a product
  Product product = new Product { Name = "Test" };
  // Arrange - add an error to the model state
  target.ModelState.AddModelError("error", "error");

  // Act - try to save the product
```

```
        // Assert - check that the repository was not called
        mock.Verify(m => m.SaveProduct(It.IsAny<Product>()), Times.Nev
        // Assert - check the method result type
        Assert.IsInstanceOfType(result, typeof(ViewResult));
    }
    ...
```

Displaying a Confirmation Message

I am going to deal with the message I stored using `TempData` in the `_AdminLayout.cshtml` layout file. the message in the template, I can create messages in any view that uses the template without needing to create addit blocks. Listing 11-12 shows the change to the file.

Listing 11-12. Handling the ViewBag Message in the _AdminLayout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title></title>
</head>
<body>
    <div>
        @if (TempData["message"] != null) {
            <div class="alert alert-success">@TempData["message"]<
        }
        @RenderBody()
    </div>
</body>
</html>
```

Tip The benefit of dealing with the message in the template like this is that users will see it displayed on whatever rendered after they have saved a change. At the moment, I return them to the list of products, but I could change the render some other view, and the users will still see the message (as long as the next view also uses the same layout).

I now have all the pieces in place to edit products. To see how it all works, start the application, navigate to the `Admin/Index` URL, and make some edits. Click the `Save` button. You will be returned to the list view, and th `TempData` message will be displayed, as shown in Figure 11-8.

eBooks-IT.org

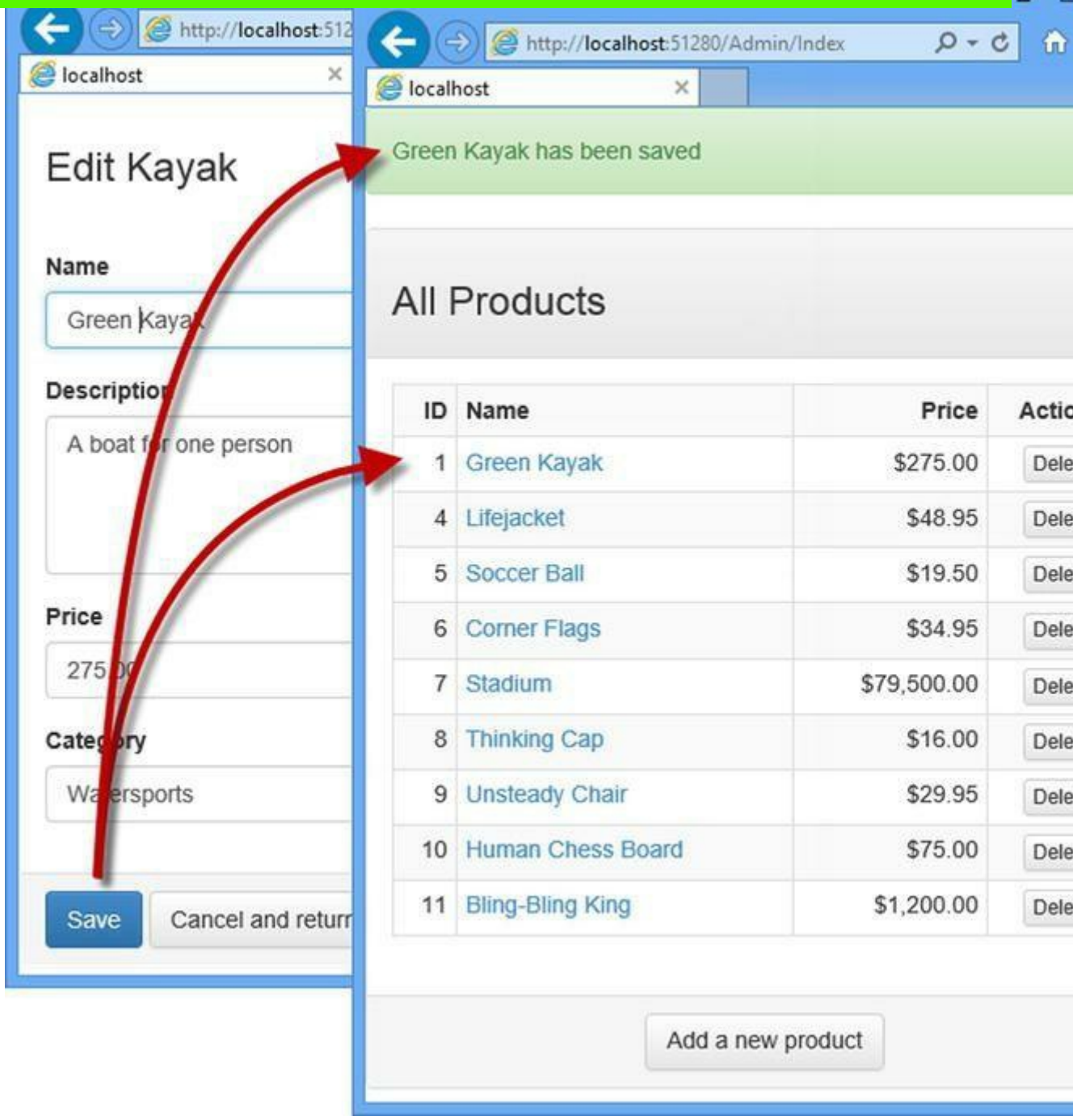


Figure 11-8. Editing a product and seeing the TempData message

The message will disappear if you reload the product list screen, because `TempData` is deleted when it is read. convenient, since I do not want old messages hanging around.

Adding Model Validation

As is the case for most projects, I need to add validation rules to the model entity. At the moment, the administrator negative prices or blank descriptions, and SportsStore would happily try and store that data in the database. Whether bad data would be stored would depend on whether it conformed to the constraints in the table definition I used to cr database in Chapter 7. Listing 11-13 shows how I have applied data annotations to the `Product` class, just as I di `ShippingDetails` class in the last chapter.

Listing 11-13. Applying Validation Attributes to the Product.cs File

eBooks-IT.org

```
using System.Web.Mvc;

namespace SportsStore.Domain.Entities {

    public class Product {

        [HiddenInput(DisplayValue = false)]
        public int ProductID { get; set; }

        [Required(ErrorMessage = "Please enter a product name")]
        public string Name { get; set; }
        [DataType(DataType.MultilineText)]

        [Required(ErrorMessage = "Please enter a description")]
        public string Description { get; set; }

        [Required]
           [Range(0.01, double.MaxValue, ErrorMessage = "Please
positive price")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Please specify a category")]
        public string Category { get; set; }
    }
}
```

The `Html.TextBox` and `Html.TextArea` helper methods that I used in the `Edit.cshtml` view t
`input` and `textarea` elements will be used by the MVC Framework to signal validation problems. These signa
using classes for which I defined styles in the `Content/ErrorStyles.css` file, which have the effect of
problems. I need to provide the user with details of any problems and you can see how I have done this in Listing 11

Listing 11-14. Adding Validation Messages to the Edit.cshtml File

```
...
<div class="panel-body">
@foreach (var property in ViewData.ModelMetadata.Properties) {
    if (property.PropertyName != "ProductID") {
        <div class="form-group">
                  <label>@(property.DisplayName ?? property.Prope
</label>
            @if (property.PropertyName == "Description") {
                @Html.TextArea(property.PropertyName, null,
                        new { @class = "form-control", rows = 5 })
            } else {
                @Html.TextBox(property.PropertyName, null,
                        new { @class = "form-control" })
            }
            @Html.ValidationMessage(property.PropertyName)
        </div>
    }
}
</div>
...
```

eBooks-IT.org

of all of
problems in the form. In this listing, I used the `Html.ValidationMessage` helper, which displays a mess
single model property. You can put the `Html.ValidationMessage` helper anywhere in a view, but it is co
(and sensible) to put it somewhere near the element that has the validation issue to give the user some context. Figure
how the validation messages appear when you edit a product and enter data that breaks the rules applied to the `Pro`

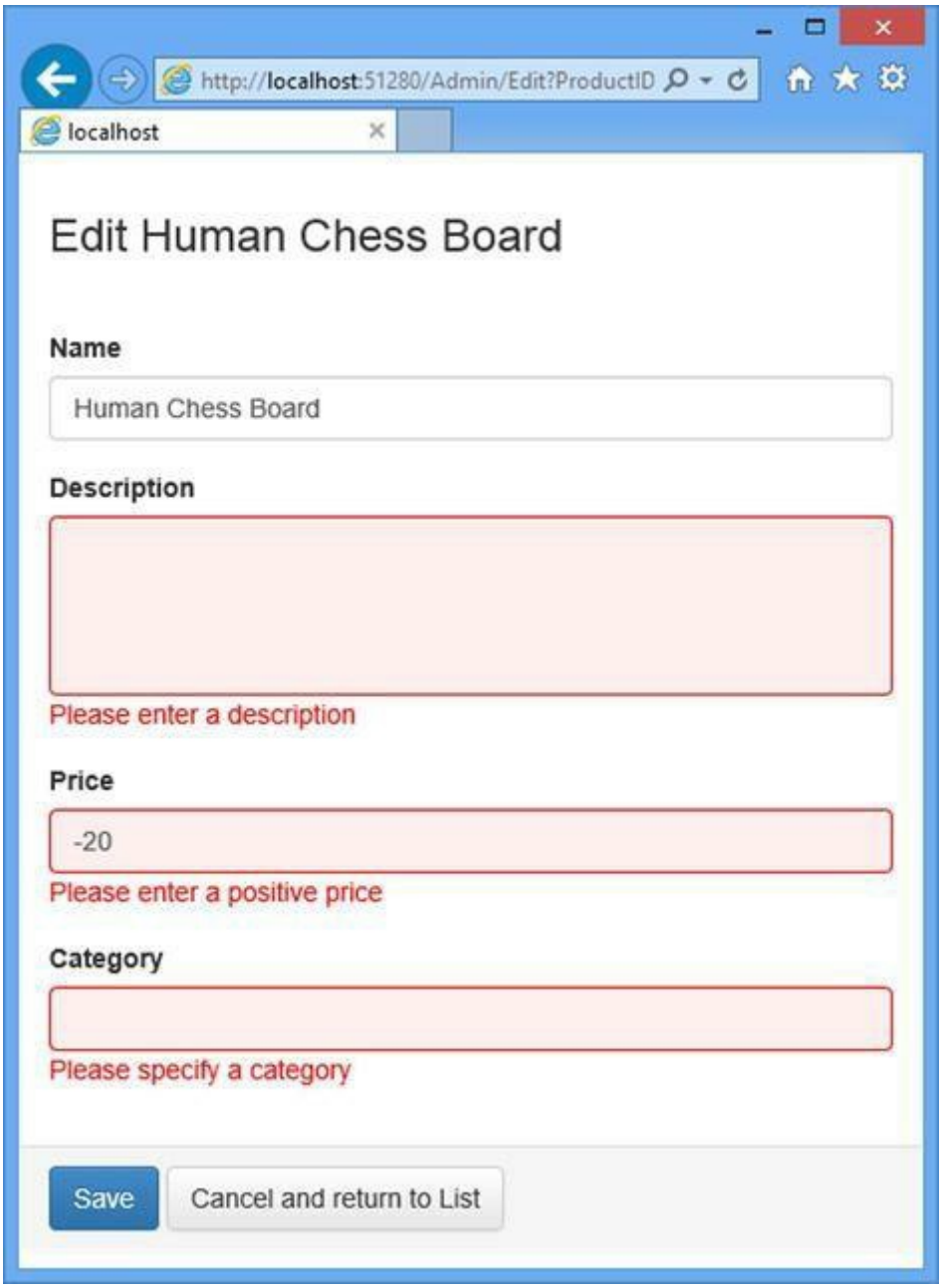


Figure 11-9. Data validation when editing products

Enabling Client-Side Validation

At present, data validation is applied only when the administration user submits edits to the server, but most users ex
immediate feedback if there are problems with the data they have entered. This is why developers often want to perf
side validation, where the data is checked in the browser using JavaScript. The MVC Framework can perform clien
validation based on the data annotations I applied to the domain model class.

This feature is enabled by default, but it has not been working because I have not added links to the required JavaS
libraries. Microsoft provides support for client-side validation based on the jQuery library and a popular jQuery plug-i
obviously enough, jQuery Validation. Microsoft extends these tools to add support for validation attributes.

The first step is to install the validation package. Select `Tools` ➤ `Library Package Manager` ➤ `P`
`Manager Console` in Visual Studio to open the NuGet command line and enter the following command:

eBooks-IT.org


```
projectname SportsStore.Web01
```

Tip Do not worry if you see a message telling you that the package is already installed. Visual Studio will silently package to the project if you accidentally checked the `Reference Script Libraries` option when us scaffolding feature to create a view.

Next, I need to add `script` elements to bring the JavaScript files in the package into the application HTML. Th place to add these links is in the `_AdminLayout.cshtml` file, so that client validation can work on any page layout. You can see the changes I made in Listing 11-15. (The order of the `script` elements is important.)

Listing 11-15. Importing JavaScript Files for Client-Side Validation into the _AdminLayout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <script src="~/Scripts/jquery-1.9.1.js"></script>
    <script src="~/Scripts/jquery.validate.js"></script>
    <script src="~/Scripts/jquery.validate.unobtrusive.js"></scrip
    <title></title>
</head>
<body>
    <div>
        @if (TempData["message"] != null) {
            <div class="alert alert-success">@TempData["message"]<
        }
        @RenderBody()
    </div>
</body>
</html>
```

These additions to the layout enable the client-side validation feature, providing feedback to the user about the valu entering before they submit the form. The appearance of error messages to the user is the same because the CSS cla used by the server validation are also used by the client-side validation, but the response is immediate and does not re request to be sent to the server. In most situations, client-side validation is a useful feature, but if for some reason yo to validate at the client, you need to add the following statements to the view:

```
...
@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
    HtmlHelper.ClientValidationEnabled = false;
    HtmlHelper.UnobtrusiveJavaScriptEnabled = false;
}
...
```

These statements disable client-side validation for the view to which they are added. You can disable client-side va

```
...
<configuration>
    <appSettings>
        <add key="ClientValidationEnabled" value="false"/>
        <add key="UnobtrusiveJavaScriptEnabled" value="false"/>
    </appSettings>
</configuration>
...
```

Creating New Products

Next, I will implement the `Create` action method, which is the one specified by the `Add a new product` product list page. This will allow the administrator to add new items to the product catalog. Adding the ability to crea products will require one small addition and one small change to the application. This is a great example of the power of a well-structured MVC application. First, add the `Create` method, shown in Listing 11-16, to the `Admin` con

Listing 11-16. Adding the Create Action Method to the AdminController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo) {
            repository = repo;
        }

        // ...other action methods omitted for brevity...

        public ViewResult Create() {
            return View("Edit", new Product());
        }
    }
}
```

The `Create` method does not render its default view. Instead, it specifies that the `Edit` view should be used. acceptable for one action method to use a view that is usually associated with another view. In this case, I inject a ne `Product` object as the view model so that the `Edit` view is populated with empty fields.

Note I have not added a unit test for this action method. Doing so would only be testing the MVC Framework ab process the `ViewResult` I return as the action method result, which is something we can take for granted. (On usually write tests for underlying frameworks unless there is a suspicion of a problem.)

This leads to the modification. I would usually expect a form to post back to the action that rendered it, and this is `Html.BeginForm` assumes by default when it generates an HTML form. However, this does not work for the method, because I want the form to be posted back to the `Edit` action so that I can save the newly created produc address this, I can use an overloaded version of the `Html.BeginForm` helper method to specify that the target generated in the `Edit` view is the `Edit` action method of the `Admin` controller, as shown in Listing 11-17, whi the change I have made to the `Views/Admin/Edit.cshtml` view file.

```
@model SportsStore.Domain.Entities.Product

@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<div class="panel">
    <div class="panel-heading">
        <h3>Edit @Model.Name</h3>
    </div>

    @using (Html.BeginForm("Edit", "Admin")) {
        <div class="panel-body">
        @foreach (var property in ViewData.ModelMetadata.Propertie
            if (property.PropertyName != "ProductID") {
                <div class="form-group">
                                                <label>@(property.Display
property.PropertyName)</label>
                    @if (property.PropertyName == "Description") {
                        @Html.TextArea(property.PropertyName, null
                                new { @class = "form-control", row
                    } else {
                        @Html.TextBox(property.PropertyName, null,
                                new { @class = "form-control" })
                    }
                    @Html.ValidationMessage(property.PropertyName)
                </div>
            }
        }
        </div>

        <div class="panel-footer">
            <input type="submit" value="Save" class="btn btn-prima
             @Html.ActionLink("Cancel and return to List", "Index
new {
                @class = "btn btn-default"
            })
        </div>
    }
</div>
```

Now the form will always be posted to the `Edit` action, regardless of which action rendered it. I can now create clicking the `Add a new product` link and filling in the details, as shown in Figure 11-10.

eBooks-IT.org

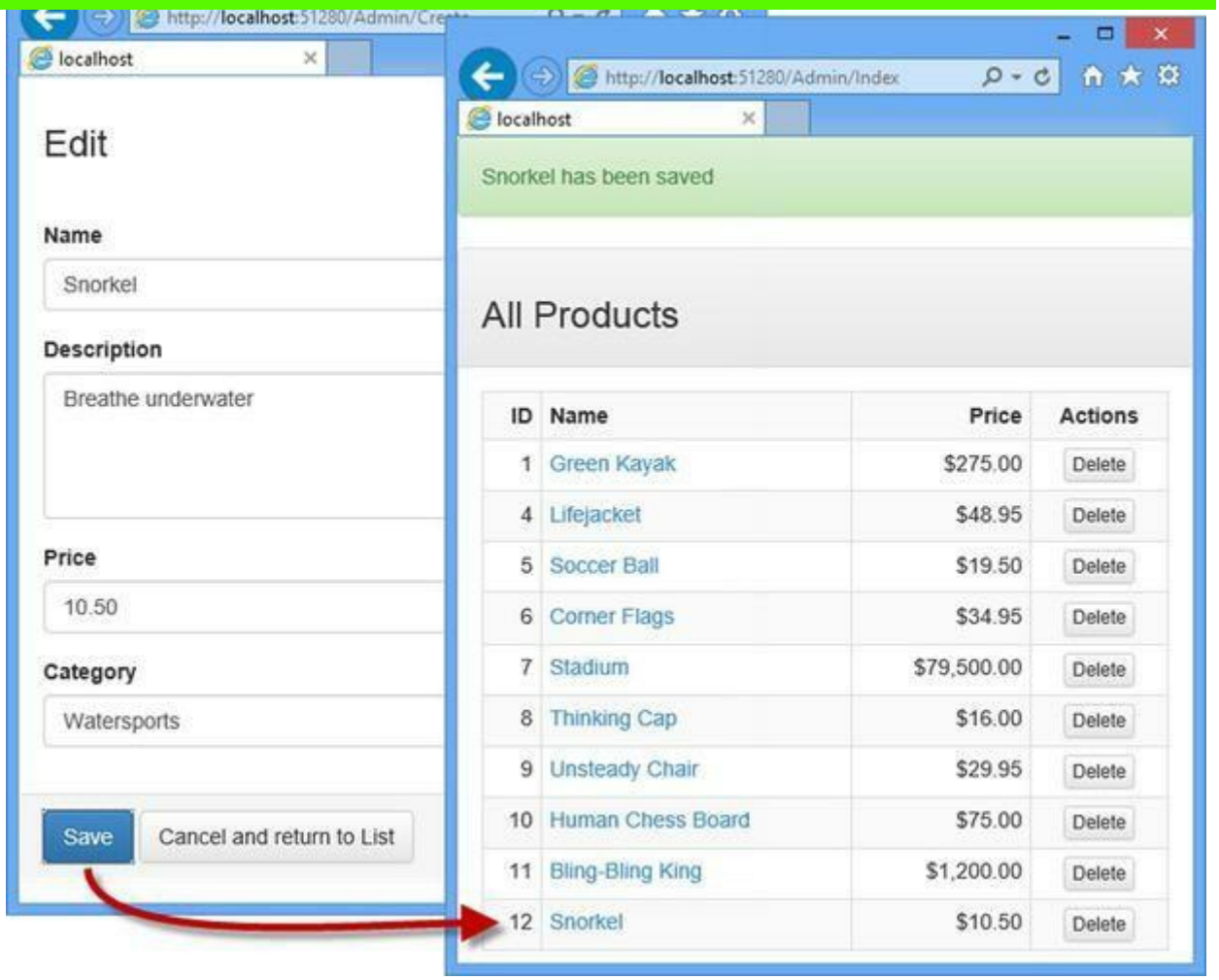


Figure 11-10. Adding a new product to the catalog

Deleting Products

Adding support for deleting items is also simple. First, I add a new method to the `IProductRepository` int
shown in Listing 11-18.

Listing 11-18. Adding a Method to Delete Products to the IProductRepository.cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract {
    public interface IProductRepository {

        IEnumerable<Product> Products { get; }

        void SaveProduct(Product product);

        Product DeleteProduct(int productID);
    }
}
```

Next, I implement this method in the Entity Framework repository class, `EFProductRepository`, as sho
11-19.

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Collections.Generic;

namespace SportsStore.Domain.Concrete {

    public class EFProductRepository : IProductRepository {
        private EFDbContext context = new EFDbContext();

        public IEnumerable<Product> Products {
            get { return context.Products; }
        }

        public void SaveProduct(Product product) {

            if (product.ProductID == 0) {
                context.Products.Add(product);
            } else {
                                                        Product    dbE
context.Products.Find(product.ProductID);
                if (dbEntry != null) {
                    dbEntry.Name = product.Name;
                    dbEntry.Description = product.Description;
                    dbEntry.Price = product.Price;
                    dbEntry.Category = product.Category;
                }
            }
            context.SaveChanges();
        }

        public Product DeleteProduct(int productID) {
            Product dbEntry = context.Products.Find(productID);
            if (dbEntry != null) {
                context.Products.Remove(dbEntry);
                context.SaveChanges();
            }
            return dbEntry;
        }
    }
}
```

The final step is to implement a `Delete` action method in the `Admin` controller. This action method should su `POST` requests, because deleting objects is not an idempotent operation. As I will explain in Chapter 16, browsers a free to make `GET` requests without the user's explicit consent, and so I must be careful to avoid making changes as consequence of `GET` requests. Listing 11-20 shows the new action method.

Listing 11-20. The Delete Action Method in the AdminController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
```

```
    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo) {
            repository = repo;
        }

        // ...other action methods omitted for brevity...

        [HttpPost]
        public ActionResult Delete(int productId) {
            Product deletedProduct = repository.DeleteProduct(prod
            if (deletedProduct != null) {
                TempData["message"] = string.Format("{0} was delet
                    deletedProduct.Name);
            }
            return RedirectToAction("Index");
        }
    }
}
```

UNIT TEST: DELETING PRODUCTS

I want to test the basic behavior of the `Delete` action method, which is that when a valid `ProductID` is parameter, the action method calls the `DeleteProduct` method of the repository and passes the correct `ProductID` value to be deleted. Here is the test:

```
...
[TestMethod]
public void Can_Delete_Valid_Products() {

  // Arrange - create a Product
  Product prod = new Product { ProductID = 2, Name = "Test" };

  // Arrange - create the mock repository
  Mock<IProductRepository> mock = new Mock<IProductRepository>()
  mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    prod,
    new Product {ProductID = 3, Name = "P3"},
  });

  // Arrange - create the controller
  AdminController target = new AdminController(mock.Object);

  // Act - delete the product
  target.Delete(prod.ProductID);

  // Assert - ensure that the repository delete method was
  // called with the correct Product
  mock.Verify(m => m.DeleteProduct(prod.ProductID));
}
...
```

eBooks-IT.org

n Figur

shown in the figure, I have taken advantage of the `TempData` variable to display a message when a product is del
catalog.

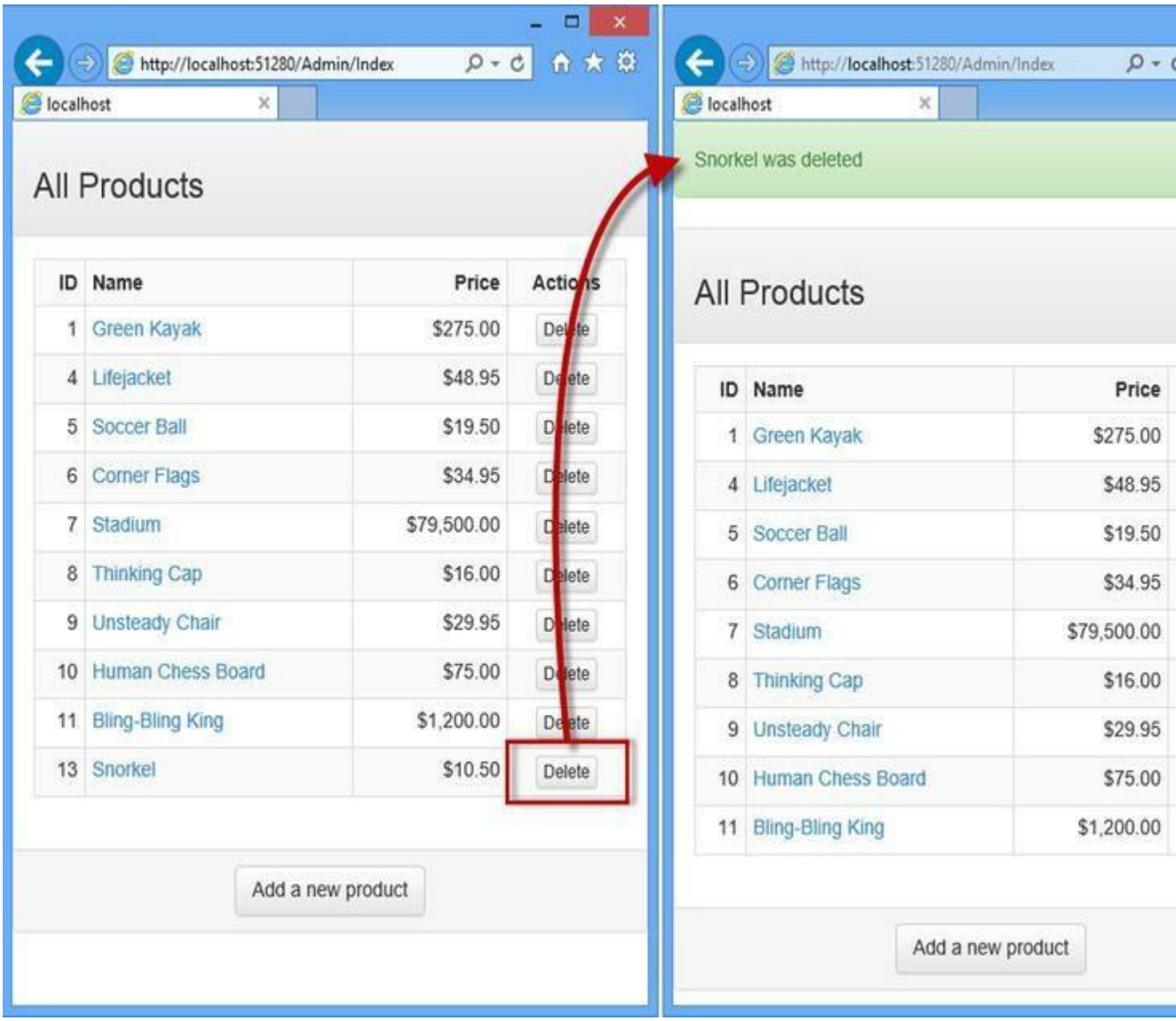


Figure 11-11. Deleting a product from the catalog

Summary

In this chapter, I introduced the administration capability and showed you how to implement CRUD operations that a
administrator to create, read, update, and delete products from the repository. In the next chapter, I show you how t
administration functions so that they are not available to all users, and I add the finishing touches to complete the Spo
functionality.

eBooks-IT.org

SportsStore: Security & Finishing Touches

In the previous chapter, I added support for administering the SportsStore application, and it will not have escaped y that anyone could modify the product catalog if I deployed the application as it is. All they would need to know is tha administration features are available using the `Admin/Index` URL. In this chapter, I am going to show you how random people from using the administration functions by password-protecting access to the entire `Admin` control have the security in place, I will complete the SportsStore app by adding support for product images. This may seem feature, but it requires some interesting MVC techniques.

Securing the Administration Controller

Because ASP.NET MVC is built on the core ASP.NET platform, I have access to the ASP.NET authentication and au features, which are packaged as a general-purpose system for keeping track of who is logged in.

DIGGING INTO THE ASP.NET SECURITY FEATURES

In this chapter, I only touch on the security features that are available. In part, this is because these features are ASP.NET platform rather than the MVC Framework and in part because there are several different approaches cover all the authentication and authorization features in detail in my *Pro ASP.NET MVC 5 Platform* book, whic publish in 2014. But I don't want you have to buy a second book to learn about something as fundamental as w security, and so Apress has kindly agreed to package up the key security chapters from the *Platform* book and them for free from `Apress.com`. They won't be available until I have written the book, of course, but it is project on my slate and it shouldn't be too far into 2014 before they are available for download.

Creating a Basic Security Policy

I am going to start by configuring *forms authentication*, which is one of the ways in which users can be authenticate ASP.NET application. In Listing 12-1, you can see the additions I have made to the `Web.config` file in the `SportsStore.WebUI` project (the one in the root of the project and not the one in the views folder).

Listing 12-1. Configuring Forms Authentication in the Web.config File

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
  </configSections>

  <connectionStrings>
              <add   name="EFDbContext"   connectionString="Data
(localdb)\v11.0;Initial
        Catalog=SportsStore;Integrated Security=True"
            providerName="System.Data.SqlClient" />
  </connectionStrings>

  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
```

eBooks-IT.org

```
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="Email.WriteAsFile" value="true"/>
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5.1" />
    <httpRuntime targetFramework="4.5.1" />
    <globalization uiCulture="en-US" culture="en-US" />
    <authentication mode="Forms">
      <forms loginUrl="~/Account/Login" timeout="2880" />
    </authentication>
  </system.web>
</configuration>
```

Authentication is set up using the `authentication` element and I have used the `mode` attribute to specify forms authentication, which is the kind most often used for Internet-facing web applications. In ASP.NET 4.5.1, Mi added support for a wider range of Internet-suitable authentication options, which I describe in the *Pro ASP.NET M Platform* book, as noted earlier in the chapter. I am going to stick with forms authentication because it works with lo credentials and is simple to set up and manage.

Note The main alternatives to forms authentication are *Windows authentication,* where the operating system cred used to identify users and *Organizational authentication*, where the user is authenticated using a cloud service such Azure. I am not going to get into either of these options because they are not widely used in Internet-facing applicatio

The `loginUrl` attribute tells ASP.NET where to redirect users when they need to authenticate themselves (in t `~/Account/Login` URL) and the `timeout` attribute specifies how long a user remains authenticated onc successfully logged in, expressed in minutes (2,880 minutes is 48 hours).

The next step is to tell ASP.NET where it will get details of the application users. I have broken this into a separate I am about to do something that should never, ever, be done in a real project: I am going to define a username and pa `Web.config` file. You can see the changes in Listing 12-2.

Listing 12-2. Defining a Username and Password in the Web.config File

```
...
<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" timeout="2880">
    <credentials passwordFormat="Clear">
      <user name="admin" password="secret" />
    </credentials>
  </forms>
</authentication>
...
```

I want to keep the example simple and focus on the way that the MVC Framework allows you to apply authentica authorization to a web application. But putting credentials in the `Web.config` file is a recipe for disaster, especia the `passwordFormat` attribute on the `credentials` element to `Clear`, meaning that passwords are st text.

Caution Don't store user credentials in the `Web.config` file and don't store passwords as plain text. See th chapters excepted from my *Pro ASP.NET MVC 5 Platform* book (as described at the start of the section) for details users via a database.

Despite being unsuitable for real projects, using the `Web.config` file to store credentials lets me focus on MV without getting sidetracked into aspects of the core ASP.NET platform. The result of the additions to the `Web.co` that I have a hard-coded username (`admin`) and password (`secret`).

The MVC Framework has a powerful feature called *filters*. These are .NET attributes that you can apply to an action controller class and they introduce additional logic when a request is processed to change the behavior of the MVC F

There are different kinds of filters available and you can create your own custom filters, as I explain in Chapter 18 that interests me at the moment is the default *authorization filter*, `Authorize`. In Listing 12-3, you can see how this filter to the `Admin` controller.

Listing 12-3. Adding the Authorize Attribute in the AdminController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    [Authorize]
    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo) {
            repository = repo;
        }

        // ...action methods omitted for brevity...
    }
}
```

When applied without parameters, the `Authorize` attribute grants access to the controller action methods to a authenticated users. This means that if you are authenticated, you are automatically authorized to use the administrati This is fine for SportsStore, where there is only one set of restricted action methods and only one user.

Note You can apply filters to an individual action method or to a controller. When you apply a filter to a controlle though you had applied it to every action method in the controller class. In Listing 12-3, I applied the `Authorize` class, so all of the action methods in the `Admin` controller are available only to authenticated users.

You can see the effect that the `Authorize` filter has by running the application and navigating to the `/Admi` URL. You will see an error similar to the one shown in Figure 12-1.

eBooks-IT.org

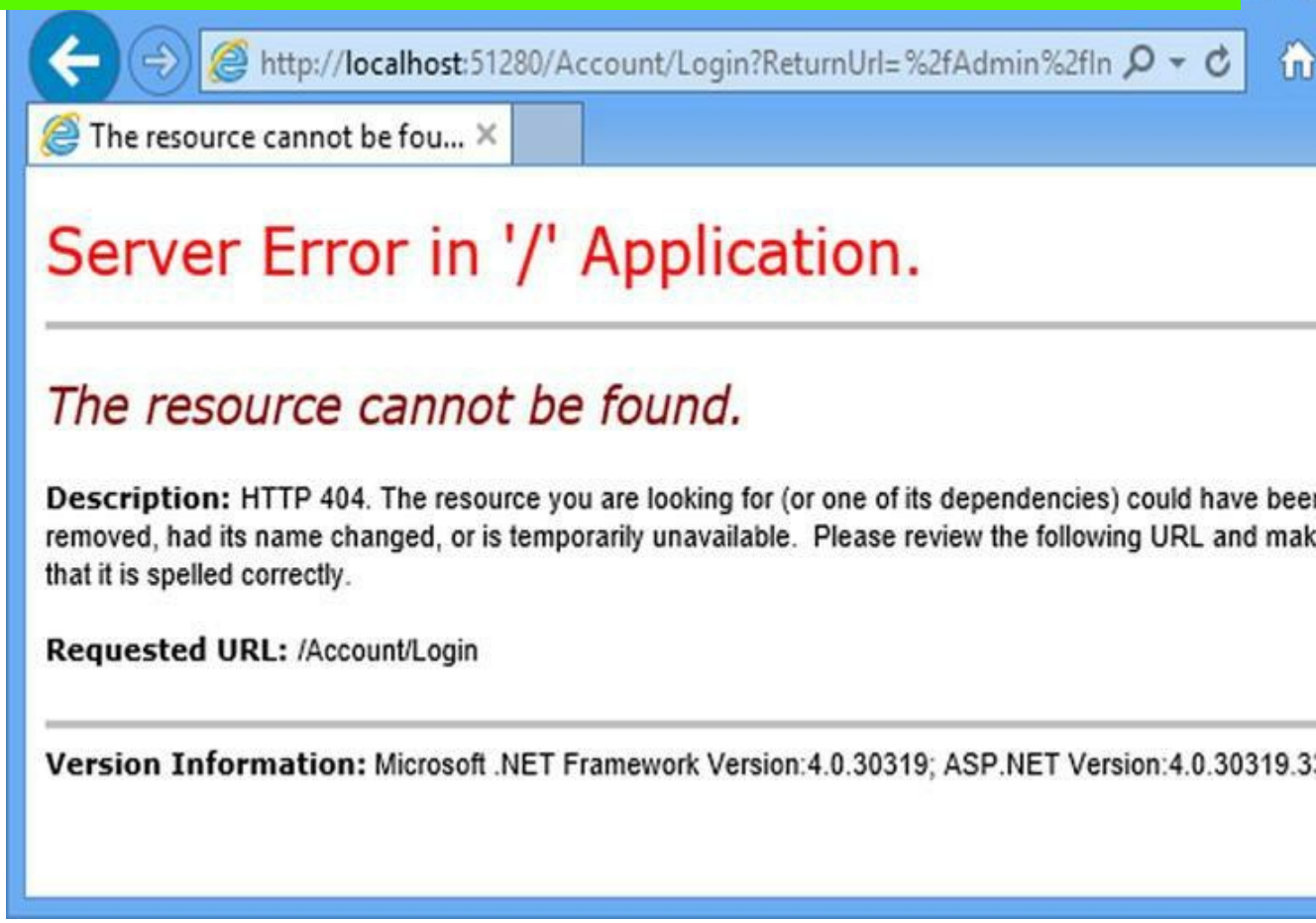


Figure 12-1. The effect of the Authorize filter

When you try to access the `Index` action method of the `Admin` controller, the MVC Framework detects the `Authorize` filter. Because you have not been authenticated, you are redirected to the URL specified in the `Web` forms authentication section: `/Account/Login`. I have not created the `Account` controller yet (which is w the error shown in the figure), but the fact that the MVC Framework has tried to redirect the request shows that the `Authorize` filter is working.

Creating the Authentication Provider

Using the forms authentication feature requires calls to two static methods of the `System.Web.Security.FormsAuthentication` class:

- The `Authenticate` method validates credentials supplied by the user.
- The `SetAuthCookie` method adds a cookie to the response to the browser, so that users do not need t every time they make a request.

The problem with calling static methods from within action methods is that it makes unit testing the controller diff frameworks typically mock only instance members. The classes that comprise the MVC Framework have been desig testing in mind, but the `FormsAuthentication` class predates the unit testing-friendly design of MVC.

The best way to address the problem is to decouple the controller from the static methods using an interface, whic additional benefit that this fits in with the broader MVC design pattern and makes it easier to switch to a different aut system later.

I start by defining the authentication provider interface. Create a new folder called `Abstract` in the `Infras` folder of the `SportsStore.WebUI` project and add a new interface called `IAuthProvider`. The conte interface are shown in Listing 12-4.

```
namespace SportsStore.WebUI.Infrastructure.Abstract {

    public interface IAuthProvider {
        bool Authenticate(string username, string password);
    }
}
```

I can now create an implementation of this interface that acts as a wrapper around the static methods of the `FormsAuthentication` class. Create another new folder in `Infrastructure`—this time called `Co` and create a new class called `FormsAuthProvider`. The contents of this class are shown in Listing 12-5.

Listing 12-5. The Contents of the FormsAuthProvider.cs File

```
using System.Web.Security;
using SportsStore.WebUI.Infrastructure.Abstract;

namespace SportsStore.WebUI.Infrastructure.Concrete {

    public class FormsAuthProvider : IAuthProvider {

        public bool Authenticate(string username, string password)

                bool result = FormsAuthentication.Authenticate(u
password);
            if (result) {
                FormsAuthentication.SetAuthCookie(username, false)
            }
            return result;
        }
    }
}
```

Note You will see a warning from Visual Studio that the `FormsAuthentication.Authenticate` been deprecated. This is part of Microsoft's ongoing efforts to rationalize user security, which is a thorny area for a application framework. For this chapter, the deprecated method will suffice and allow me to perform authentication u static details I added to the `Web.config` file.

The implementation of the `Authenticate` method calls the static `FormsAuthentication` methods to keep out of the controller. The final step is to register the `FormsAuthProvider` in the `AddBindings` the `NinjectDependencyResolver` class, as shown in Listing 12-6.

Listing 12-6. Registering the Authentication Provider in the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Web.Mvc;
using Moq;
using Ninject;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Concrete;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Infrastructure.Concrete;
```

eBooks-IT.org

```
namespace SportsStore.WebUI.Infrastructure {

    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            kernel.Bind<IProductRepository>().To<EFProductReposito

            EmailSettings emailSettings = new EmailSettings {
                WriteAsFile = bool.Parse(ConfigurationManager
                    .AppSettings["Email.WriteAsFile"] ?? "false")
            };

            kernel.Bind<IOrderProcessor>().To<EmailOrderProcessor>
                .WithConstructorArgument("settings", emailSettings

            kernel.Bind<IAuthProvider>().To<FormsAuthProvider>();
        }
    }
}
```

Creating the Account Controller

The next task is to create the `Account` controller and the `Login` action method referred to in the `Web.conf` fact, I will create two versions of the `Login` method. The first will render a view that contains a login prompt, and handle the `POST` request when users submit their credentials.

To get started, I created a view model class that I will pass between the controller and the view. Add a new class `LoginViewModel.cs` to the `Models` folder of the `SportsStore.WebUI` project and edit the cont matches Listing 12-7.

Listing 12-7. The Contents of the LoginViewModel.cs File

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.WebUI.Models {

    public class LoginViewModel {
        [Required]
        public string UserName { get; set; }

        [Required]
```

```
    }
}
```

This class contains properties for the username and password, and uses data annotation attributes to specify that v are required. Given that there are only two properties, you might be tempted to do without a view model and rely on `ViewBag` to pass data to the view. However, it is good practice to define view models so that the data passed fror controller to the view and from the model binder to the action method is typed consistently.

Next, I create the `Account` controller that will handle authentication. Create a new class file called `AccountController.cs` in the `Controllers` folder and edit the file contents to match Listing 12-8.

Listing 12-8. The the Contents of the AccountController.cs File

```
using System.Web.Mvc;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class AccountController : Controller {
        IAuthProvider authProvider;

        public AccountController(IAuthProvider auth) {
            authProvider = auth;
        }

        public ViewResult Login() {
            return View();
        }

        [HttpPost]
         public ActionResult Login(LoginViewModel model, string re
{

            if (ModelState.IsValid) {
                                if  (authProvider.Authenticate(model.U
model.Password)) {
                            return Redirect(returnUrl ?? Url.Action(
"Admin"));
                } else {
                        ModelState.AddModelError("", "Incorrect use
password");
                    return View();
                }
            } else {
                return View();
            }
        }
    }
}
```

Creating the View

To create the view that will ask the users for their credentials, create the `Views/Account` folder in the `SportsStore.WebUI` folder. Right-click on the new folder, select `Add` ➤ `MVC 5 View Page (Ra`

eBooks-IT.org

the new

Listing 12-9.

Listing 12-9. The Contents of the Login.cshtml File

```
@model SportsStore.WebUI.Models.LoginViewModel

@{
    ViewBag.Title = "Admin: Log In";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<div class="panel">
    <div class="panel-heading">
        <h3> Log In</h3>
    </div>
    <div class="panel-body">
         <p class="lead">Please log in to access the administrati
</p>
        @using (Html.BeginForm()) {
            @Html.ValidationSummary()
            <div class="form-group">
                <label>User Name:</label>
                  @Html.TextBoxFor(m => m.UserName, new { @class
control" })
            </div>
            <div class="form-group">
                <label>Password:</label>
                  @Html.PasswordFor(m => m.Password, new { @class
control" })
            </div>
            <input type="submit" value="Log in"  class="btn btn-pr
        }
    </div>
</div>
```

This view uses the `_AdminLayout.cshtml` layout and Bootstrap classes to style the content. There are n techniques in this view, other than the use of the `Html.PasswordFor` helper method, which generates an `in` whose `type` attribute is set to `password`. I describe the complete set of HTML helper methods in Chapter 21. how the view appears by starting the app and navigating to the `/Admin/Index` URL, as shown in Figure 12-2.

eBooks-IT.org

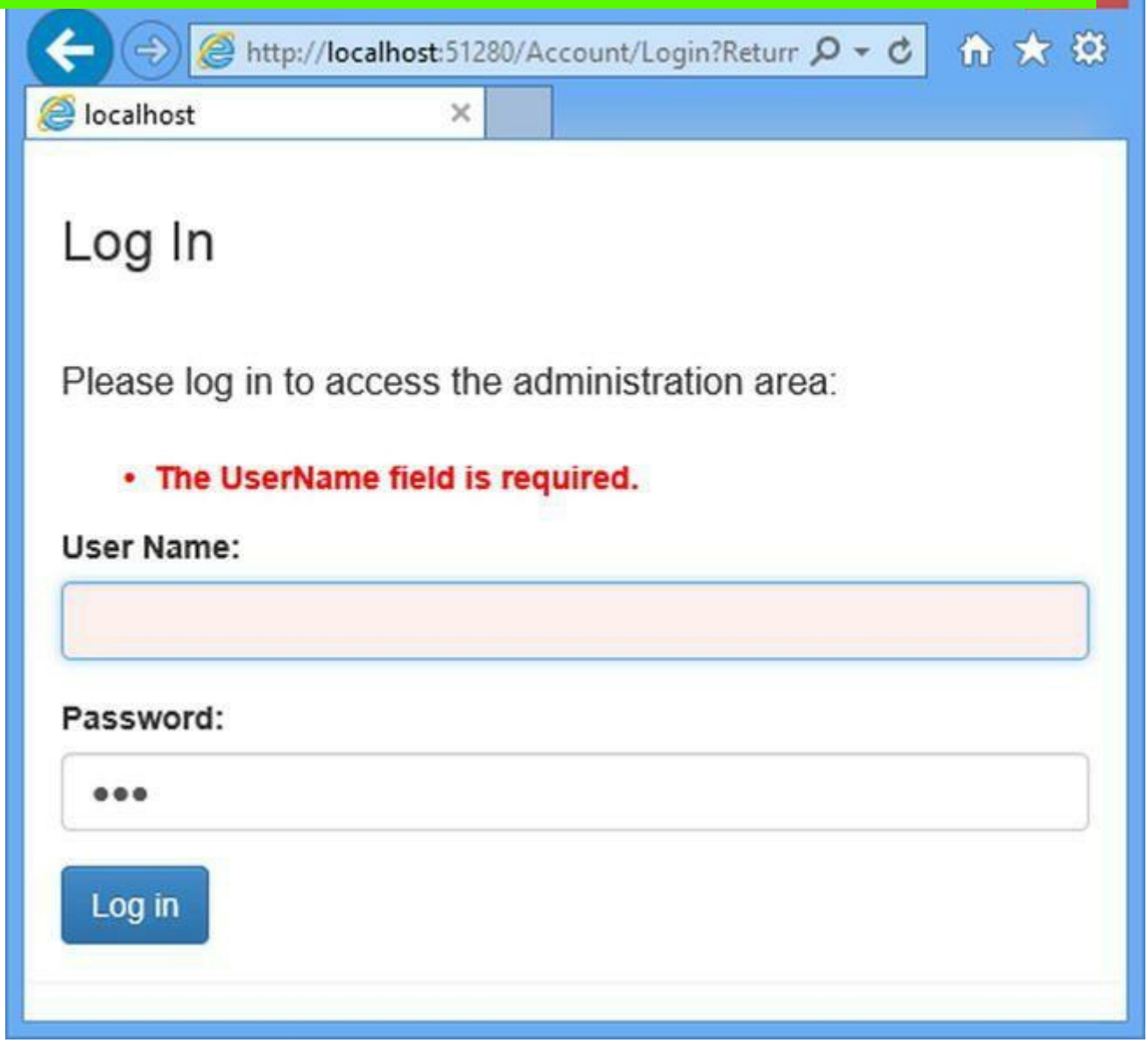


Figure 12-2. The Login view

The `Required` attributes that I applied to the properties of the view model are enforced using client-side valida (Remember that the required JavaScript libraries are included in the `_AdminLayout.cshtml` layout created previous chapter.) Users can submit the form only after they have provided both a username and password, and the is performed at the server when I call the `FormsAuthentication.Authenticate` method.

Caution In general, using client-side data validation is a good idea. It offloads some of the work from your serve users immediate feedback about the data they are providing. However, you should not be tempted to perform authent client, as this would typically involve sending valid credentials to the client so they can be used to check the usernam password that the user has entered, or at least trusting the client's report of whether they have successfully authentic Authentication must always be done at the server.

When I receive bad credentials, I add an error to the `ModelState` and re-render the view. This causes a mess displayed in the validation summary area, which I created by calling the `Html.ValidationSummary` helpe the view. This takes care of protecting the SportsStore administration functions. Users will be allowed to access thes only after they have supplied valid credentials and received a cookie, which will be attached to subsequent requests.

UNIT TEST: AUTHENTICATION

Testing the `Account` controller requires me to check two behaviors: a user should be authenticated when va are supplied, and a user should *not* be authenticated when invalid credentials are supplied. I perform these tests

eBooks-IT.org

sult of

Login method. I created the following tests in a new unit test file called AdminSecurityTests.cs

```
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.WebUI.Controllers;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Models;

namespace SportsStore.UnitTests {
  [TestClass]
  public class AdminSecurityTests {

    [TestMethod]
    public void Can_Login_With_Valid_Credentials() {

      // Arrange - create a mock authentication provider
      Mock<IAuthProvider> mock = new Mock<IAuthProvider>();
      mock.Setup(m => m.Authenticate("admin", "secret")).Returns

      // Arrange - create the view model
      LoginViewModel model = new LoginViewModel {
        UserName = "admin",
        Password = "secret"
      };

      // Arrange - create the controller
      AccountController target = new AccountController(mock.Objec

      // Act - authenticate using valid credentials
      ActionResult result = target.Login(model, "/MyURL");

      // Assert
      Assert.IsInstanceOfType(result, typeof(RedirectResult));
      Assert.AreEqual("/MyURL", ((RedirectResult)result).Url);
    }

    [TestMethod]
    public void Cannot_Login_With_Invalid_Credentials() {

      // Arrange - create a mock authentication provider
      Mock<IAuthProvider> mock = new Mock<IAuthProvider>();
      mock.Setup(m => m.Authenticate("badUser",
"badPass")).Returns(false);

      // Arrange - create the view model
      LoginViewModel model = new LoginViewModel {
        UserName = "badUser",
        Password = "badPass"
      };

      // Arrange - create the controller
      AccountController target = new AccountController(mock.Objec

      // Act - authenticate using valid credentials
```

```
            // Assert
            Assert.IsInstanceOfType(result, typeof(ViewResult));
            Assert.IsFalse(((ViewResult)result).ViewData.ModelState.IsV
        }
    }
}
```

Image Uploads

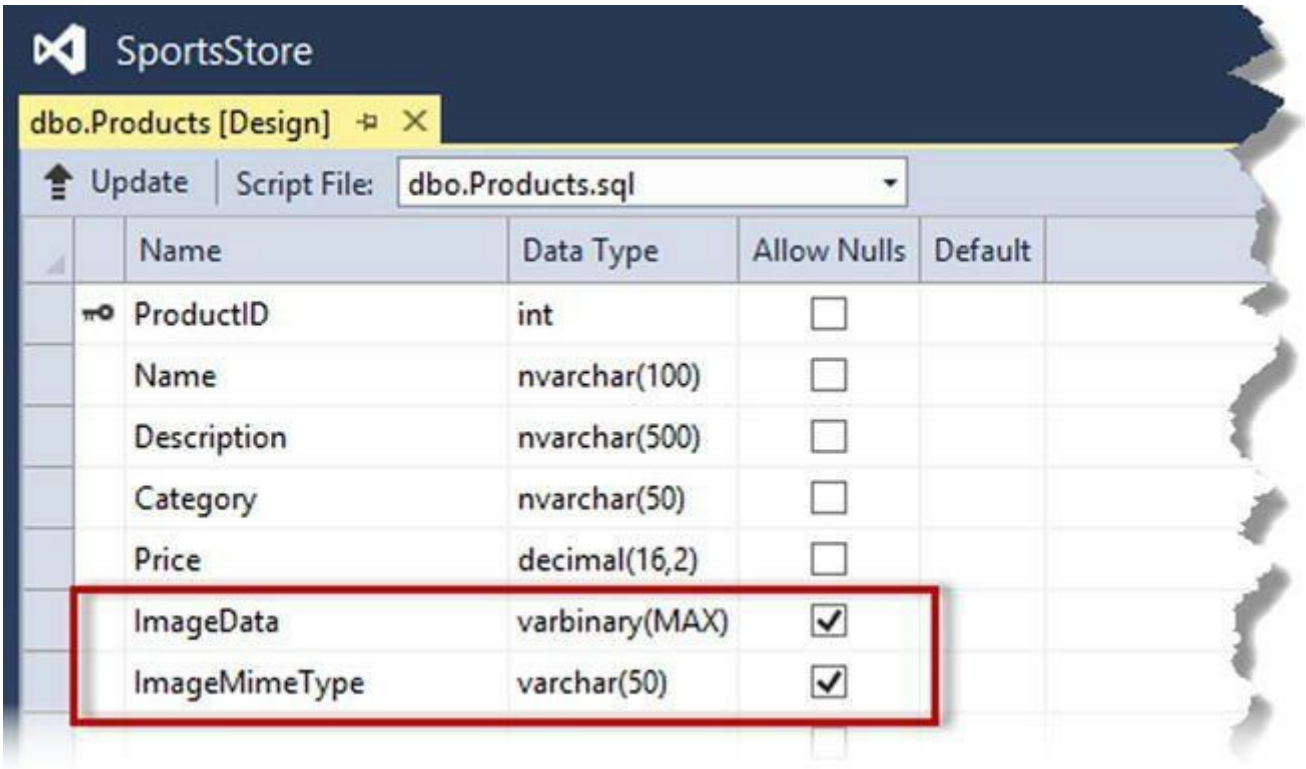
I am going to complete the SportsStore user experience with something a little more sophisticated: I will add the abili administrator to upload product images and store them in the database so that they are displayed in the product catalo something that is especially interesting or useful in its own right, but it does allow me to demonstrate some important Framework features.

Extending the Database

Open the Visual Studio `Server Explorer` window and navigate to the `Products` table in the database cr Chapter 7. The name of the data connection may have changed to be `EFDbContext`, which is the name assigne connection in the `Web.config` file. Visual Studio is a little bit inconsistent about when it renames the connection might also see the original name that was shown when the connection was created. Right-click on the `Products` select `New Query` from the pop-up menu and enter the following SQL into the text area:

```
ALTER TABLE [dbo].[Products]
    ADD [ImageData]     VARBINARY (MAX) NULL,
        [ImageMimeType] VARCHAR (50)    NULL;
```

Click the `Execute` button (which is marked with an arrow) in the top-left cover of the window and Visual Stu the database, adding two new columns to the table. To test the update, right-click on the `Products` table in the S `Explorer` window and select `Open Table Definition` from the menu. You will see that there are n called `ImageData` and `ImageMimeType`, as shown in Figure 12-3.



eBooks-IT.org

Tip If the columns are not visible, close the design window, right-click on the data connection in the `Server Explorer` window and select `Refresh` from the menu. The new columns should now be visible when you se `Open Table Definition` menu item again.

Enhancing the Domain Model

I need to add two new fields to the `Products` class in the `SportsStore.Domain` project that correspon database columns, as shown in Listing 12-10.

Listing 12-10. Adding Properties in the Product.cs File

```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace SportsStore.Domain.Entities {

    public class Product {

        [HiddenInput(DisplayValue = false)]
        public int ProductID { get; set; }

        [Required(ErrorMessage = "Please enter a product name")]
        public string Name { get; set; }
        [DataType(DataType.MultilineText)]

        [Required(ErrorMessage = "Please enter a description")]
        public string Description { get; set; }

        [Required]
           [Range(0.01, double.MaxValue, ErrorMessage = "Please
positive price")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Please specify a category")]
        public string Category { get; set; }

        public byte[] ImageData { get; set; }
        public string ImageMimeType { get; set; }
    }
}
```

Caution Make sure that the names of the properties that you add to the `Product` class exactly match the nan to the new columns in the database.

Creating the Upload User Interface Elements

The next step is to add support for handling file uploads. This involves creating a UI that the administrator can use to image. Modify the `Views/Admin/Edit.cshtml` view so that it matches Listing 12-11.

Listing 12-11. Adding Support for Images in the Edit.cshtml File

```
@model SportsStore.Domain.Entities.Product

@{
```

```
    Layout = "/Views/Shared/_AdminLayout.cshtml";
}

<div class="panel">
    <div class="panel-heading">
        <h3>Edit @Model.Name</h3>
    </div>

@using (Html.BeginForm("Edit", "Admin",
        FormMethod.Post, new { enctype = "multipart/form-data" }))
    <div class="panel-body">
        @Html.HiddenFor(m => m.ProductID)
        @foreach (var property in ViewData.ModelMetadata.Propertie
            switch (property.PropertyName) {
                case "ProductID":
                case "ImageData":
                case "ImageMimeType":
                    // do nothing
                    break;
                default:
                    <div class="form-group">
                                                <label>@(property.Display
property.PropertyName)</label>
                        @if (property.PropertyName == "Description
                            @Html.TextArea(property.PropertyName,
                                new { @class = "form-control", row
                        } else {
                            @Html.TextBox(property.PropertyName, n
                                new { @class = "form-control" })
                        }
                        @Html.ValidationMessage(property.PropertyN
                    </div>
                    break;
            }
        }

        <div class="form-group">
            <div style="position:relative;">
                <label>Image</label>
                <a class='btn' href='javascript:;'>
                    Choose File...
                    <input type="file" name="Image" size="40"
                        style="position:absolute;z-index:2;top:0;
                            left:0;filter: alpha(opacity=0); opaci
                                                                bac
color:transparent;color:transparent;"
                                                    onchange='$("#uplo
info").html($(this).val());'>
                </a>
                    <span class='label label-info' id="upload-fil
</span>
            </div>
            @if (Model.ImageData == null) {
```

eBooks-IT.org

```
                } else {
                    <img class="img-thumbnail" width="150" height="150
                            src="@Url.Action("GetImage", "Product",
                            new { Model.ProductID })" />
                }
            </div>
        </div>

        <div class="panel-footer">
            <input type="submit" value="Save" class="btn btn-primary"
            @Html.ActionLink("Cancel and return to List", "Index", nul
            @class = "btn btn-default"
        })
        </div>
}
</div>
```

You may already be aware that Web browsers will upload files properly only when the HTML `form` element defi
`enctype` value of `multipart/form-data`. In other words, for a successful upload, the `form` element
like this:

```
...
<form action="/Admin/Edit" enctype="multipart/form-data" method="p
...
```

Without the `enctype` attribute, the browser will transmit only the name of the file and not its content, which is
To ensure that the `enctype` attribute appears, I must use an overload of the `Html.BeginForm` helper meth
me specify HTML attributes, like this:

```
...
@using (Html.BeginForm("Edit", "Admin",
    FormMethod.Post, new { enctype = "multipart/form-data" })) {
...
```

There are two other changes in the view. The first is that I have replaced the Razor `if` expression I used when g
`input` elements with a `switch` statement. The effect is the same, but it allows me to specify the model propert
skip more concisely, and I don't want to display the image-related properties directly to the user.

Instead, I have made the remaining change, which is to add an `input` element whose `type` is file to allow file
with an `img` element to display the image associated with a product, if there is one in the database.

The horrific mess of inline CSS and JavaScript addresses a shortcoming in the Bootstrap library: it does not prope
`input` elements. There are a number of extensions that add the missing functionality, but I have chosen the magic
shown in the listing because it is self-contained and is reliable. It doesn't change the way that the MVC Framework
way in which the elements in the `Edit.cshtml` file are styled.

Saving Images to the Database

I need to enhance the `POST` version of the `Edit` action method in the `Admin` controller so that I take the image
been uploaded and save it in the database. Listing 12-12 shows the changes that are required.

Listing 12-12. Handling Image Data in the AdminController.cs File

```
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
```

```
namespace SportsStore.WebUI.Controllers {

    [Authorize]
    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo) {
            repository = repo;
        }

        public ViewResult Index() {
            return View(repository.Products);
        }

        public ViewResult Edit(int productId) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            return View(product);
        }

        [HttpPost]
        public ActionResult Edit(Product product, HttpPostedFileBa
= null) {
            if (ModelState.IsValid) {
                if (image != null) {
                    product.ImageMimeType = image.ContentType;
                    product.ImageData = new byte[image.ContentLeng
                              image.InputStream.Read(product.ImageD
image.ContentLength);
                }
                repository.SaveProduct(product);
                 TempData["message"] = string.Format("{0} has been
product.Name);
                return RedirectToAction("Index");
            } else {
                // there is something wrong with the data values
                return View(product);
            }
        }

        public ViewResult Create() {
            return View("Edit", new Product());
        }

        [HttpPost]
        public ActionResult Delete(int productId) {
            Product deletedProduct = repository.DeleteProduct(prod
            if (deletedProduct != null) {
                TempData["message"] = string.Format("{0} was delet
                    deletedProduct.Name);
            }
            return RedirectToAction("Index");
        }
```

eBooks-IT.org

}

I have added a new parameter to the `Edit` method, which the MVC Framework uses to pass the uploaded file da action method. I check to see if the parameter value is `null`; if it is not, I copy the data and the MIME type from t to the `Product` object so that it is saved to the database. I must also update the `EFProductRepository` `SportsStore.Domain` project to ensure that the values assigned to the `ImageData` and `ImageMime` properties are stored in the database. Listing 12-13 shows the required changes to the `SaveProduct` method.

Listing 12-13. Ensuring That the Image Values Are Stored in the Database in the EFProductRepository.cs File

```
...
public void SaveProduct(Product product) {

    if (product.ProductID == 0) {
        context.Products.Add(product);
    } else {
        Product dbEntry = context.Products.Find(product.ProductID)
        if (dbEntry != null) {
            dbEntry.Name = product.Name;
            dbEntry.Description = product.Description;
            dbEntry.Price = product.Price;
            dbEntry.Category = product.Category;
            dbEntry.ImageData = product.ImageData;
            dbEntry.ImageMimeType = product.ImageMimeType;
        }
    }
    context.SaveChanges();
}
...
```

Implementing the GetImage Action Method

In Listing 12-11, I added an `img` element whose content was obtained through a `GetImage` action method on th `Product` controller. I am going to implement this action method so that I can display images contained in the data 12-14 shows the definition of the action method.

Listing 12-14. The GetImage Action Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productReposit
```

eBooks-IT.org

```
        }

        public ViewResult List(string category, int page = 1) {
            ProductsListViewModel model = new ProductsListViewMode
                Products = repository.Products
                            .Where(p => category == null || p.Cate
category)
                    .OrderBy(p => p.ProductID)
                    .Skip((page - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = category == null ?
                        repository.Products.Count() :
                             repository.Products.Where(e => e.Cat
category).Count()
                },
                CurrentCategory = category
            };
            return View(model);
        }

        public FileContentResult GetImage(int productId) {
            Product prod = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            if (prod != null) {
                return File(prod.ImageData, prod.ImageMimeType);
            } else {
                return null;
            }
        }
    }
}
```

This method tries to find a product that matches the ID specified by the parameter. The `FileContentResu` used as the result from an action method when you want to return a file to the client browser, and instances are crea `File` method of the base controller class. I discuss the different types of results you can return from action metho 17.

UNIT TEST: RETRIEVING IMAGES

I want to make sure that the `GetImage` method returns the correct MIME type from the repository and mak data is returned when I request a product ID that doesn't exist. Here are the test methods I created, which I def unit test file called `ImageTests.cs`:

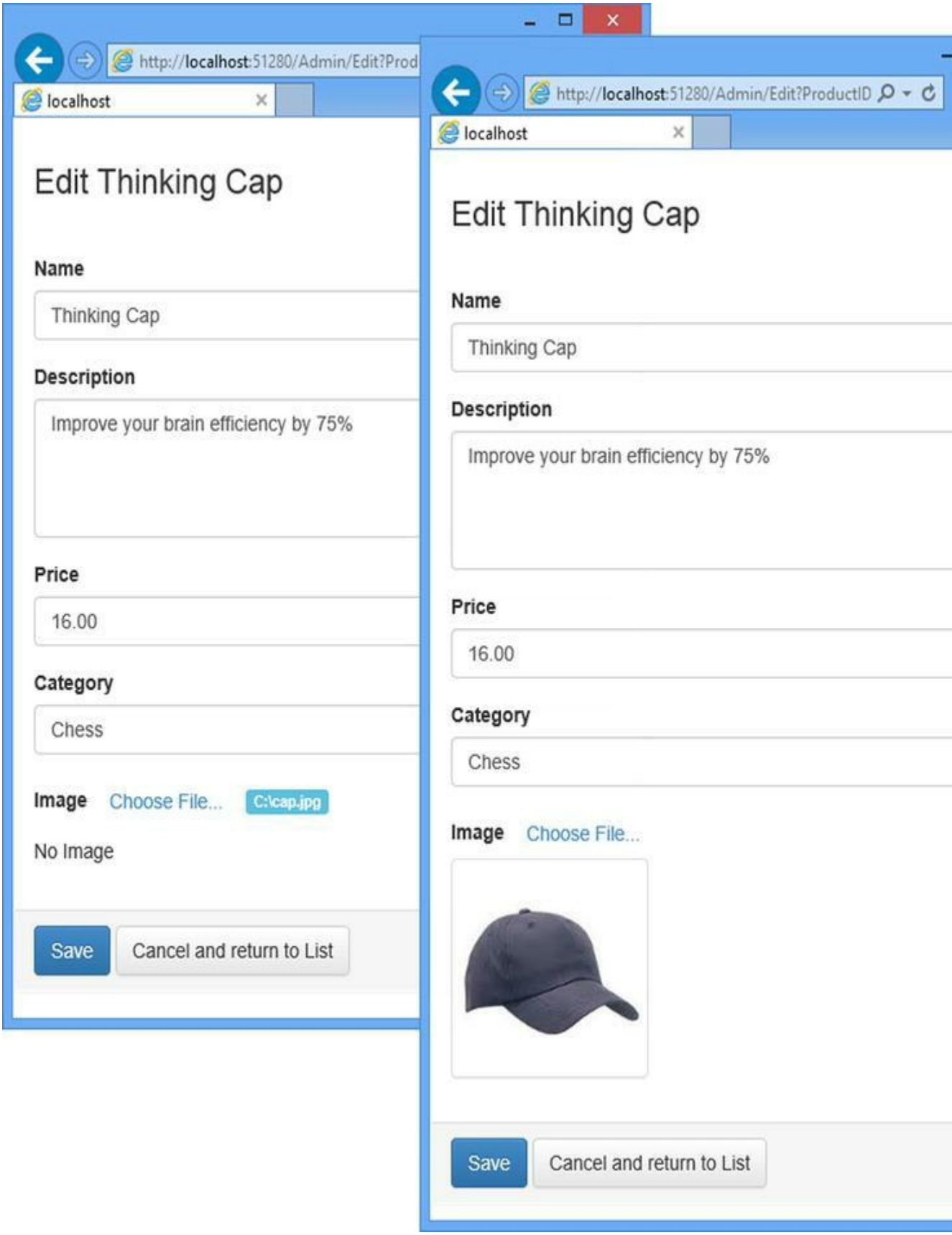
```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using System.Linq;
using System.Web.Mvc;

namespace SportsStore.UnitTests {
```

```
  public class ImageTests {

    [TestMethod]
    public void Can_Retrieve_Image_Data() {

      // Arrange - create a Product with image data
      Product prod = new Product {
        ProductID = 2,
        Name = "Test",
        ImageData = new byte[] { },
        ImageMimeType = "image/png"
      };

      // Arrange - create the mock repository
      Mock<IProductRepository> mock = new Mock<IProductRepository
      mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        prod,
        new Product {ProductID = 3, Name = "P3"}
      }.AsQueryable());

      // Arrange - create the controller
      ProductController target = new ProductController(mock.Objec

      // Act - call the GetImage action method
      ActionResult result = target.GetImage(2);

      // Assert
      Assert.IsNotNull(result);
      Assert.IsInstanceOfType(result, typeof(FileResult));
      Assert.AreEqual(prod.ImageMimeType,
((FileResult)result).ContentType);
    }

    [TestMethod]
    public void Cannot_Retrieve_Image_Data_For_Invalid_ID() {

      // Arrange - create the mock repository
      Mock<IProductRepository> mock = new Mock<IProductRepository
      mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"}
      }.AsQueryable());

      // Arrange - create the controller
      ProductController target = new ProductController(mock.Objec

      // Act - call the GetImage action method
      ActionResult result = target.GetImage(100);

      // Assert
      Assert.IsNull(result);
    }
  }
}
```

eBooks-IT.org

content type matches the type in the mock data. The `FileResult` class does not let me access the binary d and the file, so I must be satisfied with a less than perfect test. When I request an invalid product ID, I check to ens result is `null`.

The administrator can now upload images for products. You can try this yourself by starting the application, navig `/Admin/Index` URL and editing one of the products. Figure 12-4 shows an example.



eBooks-IT.org

Displaying Product Images

All that remains is to display the images alongside the product description in the product catalog. Edit the `Views/Shared/ProductSummary.cshtml` view to reflect the changes shown in bold in Listing 12-

Listing 12-15. Displaying Images in the ProductSummary.cshtml File

```
@model SportsStore.Domain.Entities.Product

<div class="well">

    @if (Model.ImageData != null) {
        <div class="pull-left" style="margin-right: 10px">
            <img class="img-thumbnail" width="75" height="75"
               src="@Url.Action("GetImage", "Product",
                new { Model.ProductID })" />
        </div>
    }

    <h3>
        <strong>@Model.Name</strong>
                                <span    class="pull-right    label
primary">@Model.Price.ToString("c")</span>
    </h3>
    @using (Html.BeginForm("AddToCart", "Cart")) {
        <div class="pull-right">
            @Html.HiddenFor(x => x.ProductID)
            @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
               <input type="submit" class="btn btn-success" value=
cart" />
        </div>
    }
    <span class="lead"> @Model.Description</span>
</div>
```

With these changes in place, the customers will see images displayed as part of the product description when they catalog, as shown in Figure 12-5.

eBooks-IT.org

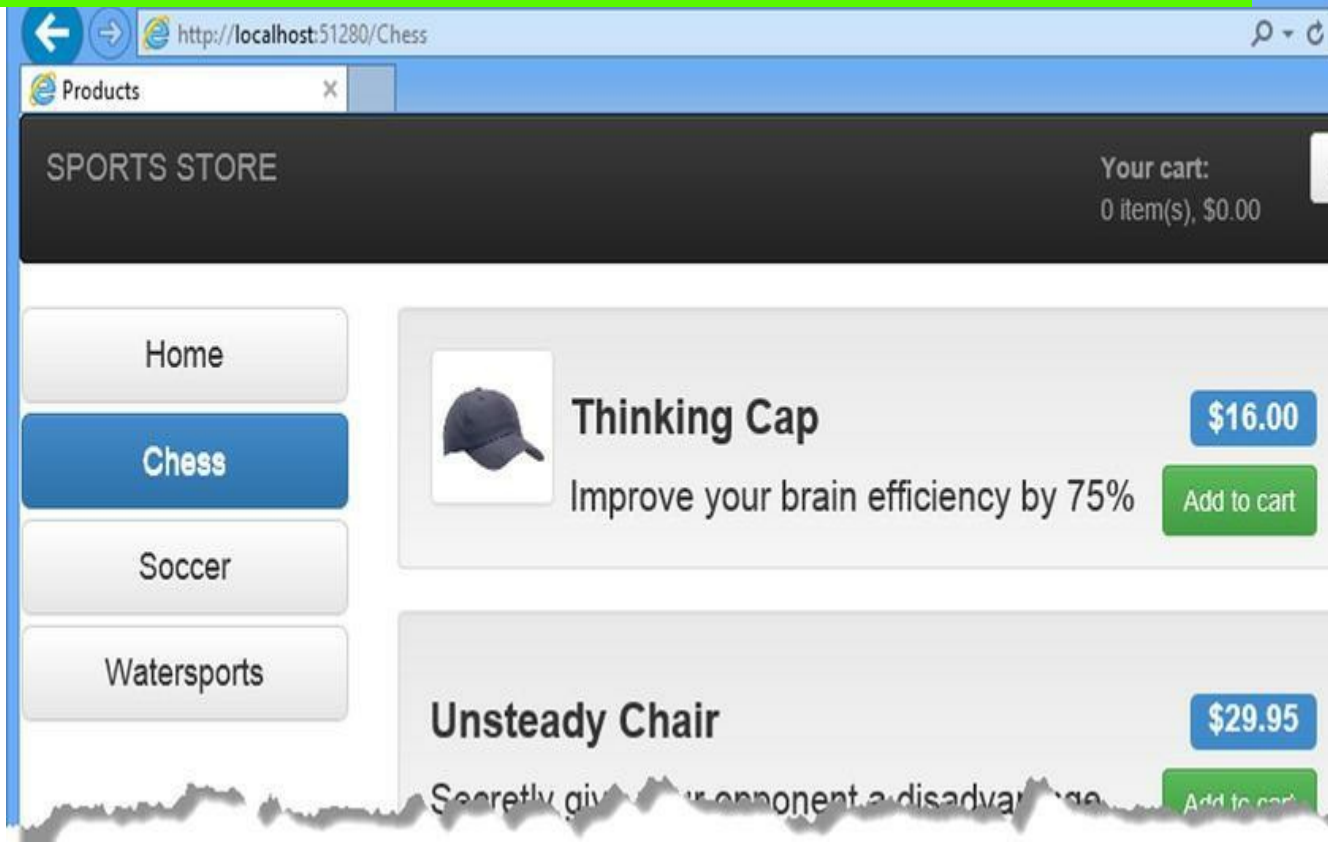


Figure 12-5. Displaying product images

Summary

In this and previous chapters, I have demonstrated how the ASP.NET MVC Framework can be used to create a real commerce application. This extended example has introduced many of the framework's key features: controllers, act routing, views, model binding, metadata, validation, layouts, authentication, and more. You have also seen how some technologies related to MVC can be used. These included the Entity Framework, Ninject, Moq, and the Visual Studio unit testing. The result is an application that has a clean, component-oriented architecture that separates out the vario and a code base that will be easy to extend and maintain. In the next chapter, I show you how to deploy the SportsS application into production.

eBooks-IT.org

Deployment

The final, and critical, step in application development is deployment: making your application available for the world chapter, I'll show you how to prepare and deploy the SportsStore application.

There are lots of different ways to deploy MVC Framework applications and a wide range of deployment targets. deploy to a Windows Server machine running Internet Information Services (IIS) which you run and manage locally deploy to a remote hosting service that manages servers for you; or, increasingly, you can deploy to a cloud infrastru platform that provisions and scales your application to seamlessly meet demand.

I thought long and hard about how to create a useful example deployment in this chapter. I ruled out showing you deploy directly to IIS because the server configuration process is long and complicated, and most MVC Framework that are targeting local servers rely on an IT operations group to perform configuration and deployment tasks. I also demonstrating deployment to a managed hosting company because each has its own custom deployment processes a company sets the standard for hosting.

So, somewhat by default, I settled on demonstrating a deployment to Windows Azure, which is Microsoft's cloud which has some nice support for MVC applications. I am not suggesting that Azure is suitable for all deployments, b way it works and using it in this chapter allows me to demonstrate the deployment process rather than getting bogge and Windows configuring issues. There is a free 90-day trial available on Azure as I write this (and MSDN subscript Azure), which means that you should be able to follow the example in this chapter, even if you don't intend to use A your application.

Caution I recommend that you practice deployment using a test application and server before attempting to depl application into a production environment. Like every other aspect of the software development life cycle, the deploy benefits from testing. I have a stock of horror stories of project teams who have destroyed operational applications t hasty and poorly tested deployment procedures. It is not that the ASP.NET deployment features are especially dange are not—but rather, any interaction that involves a running application with real user data deserves careful thought ar

Deploying a web application used to be a tedious and error-prone process, but Microsoft has put a lot of effort int the deployment tools in Visual Studio. So even if you need to deploy to a different kind of infrastructure, you will fin Studio is able to do a lot of the heavy lifting for you.

Preparing Windows Azure

You have to create an account before you can use Azure, which you can do by going to www.windowsazure the time of writing, Microsoft is offering free trial accounts, and MSDN packages include Azure services. Once you your account, you can manage your Azure services by going to http://manage.windowsazure.com your credentials. When you start, you will see the summary view shown in Figure 13-1.

eBooks-IT.org

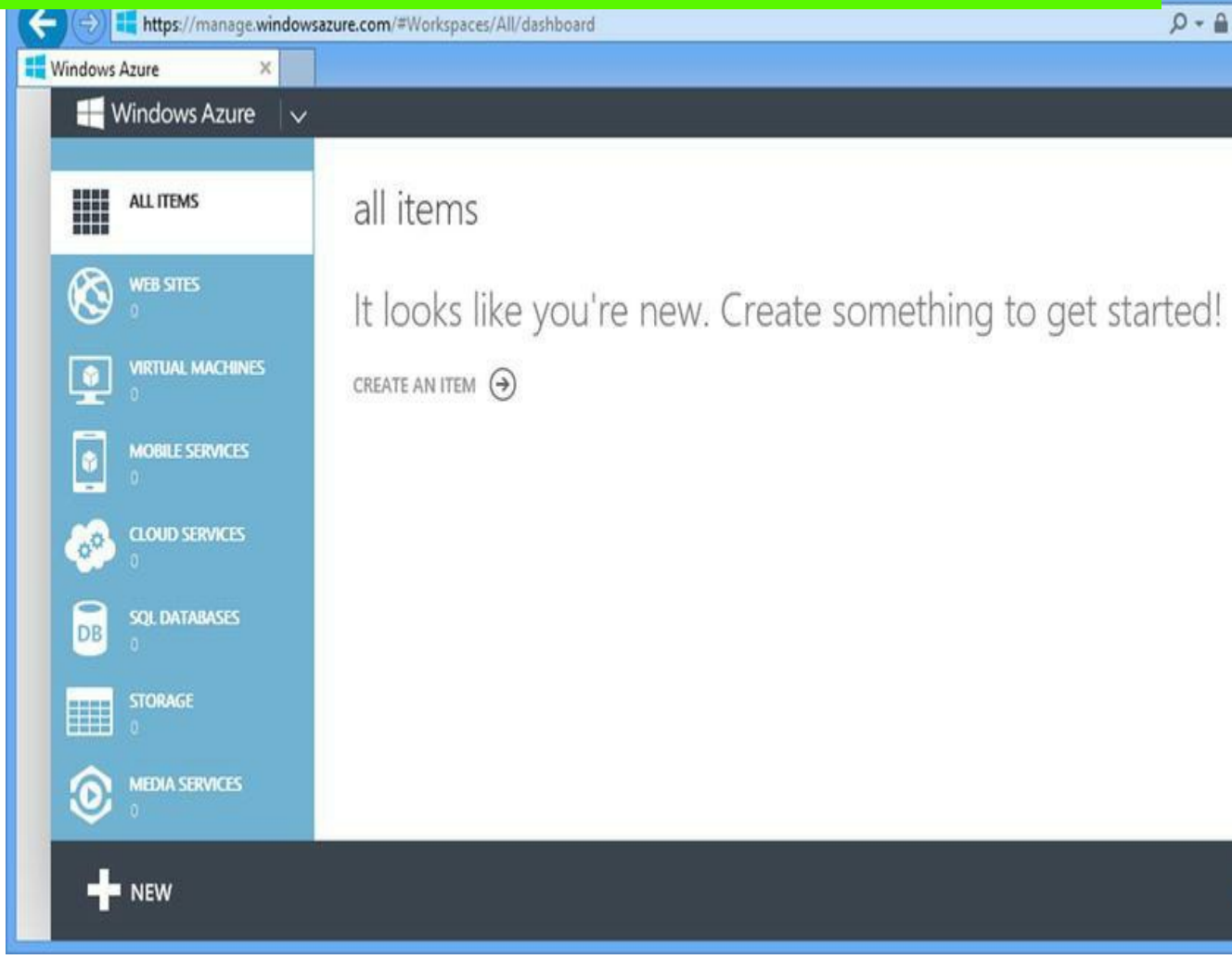


Figure 13-1. The Azure portal

Creating the Web Site and Database

I start by creating a new web site and database service, which are two of the cloud services offered by Azure. Click sign in the bottom-left corner of the portal window and select `Compute` ➤ `Web Site` ➤ `Custom Crea` see the form illustrated in Figure 13-2.

eBooks-IT.org

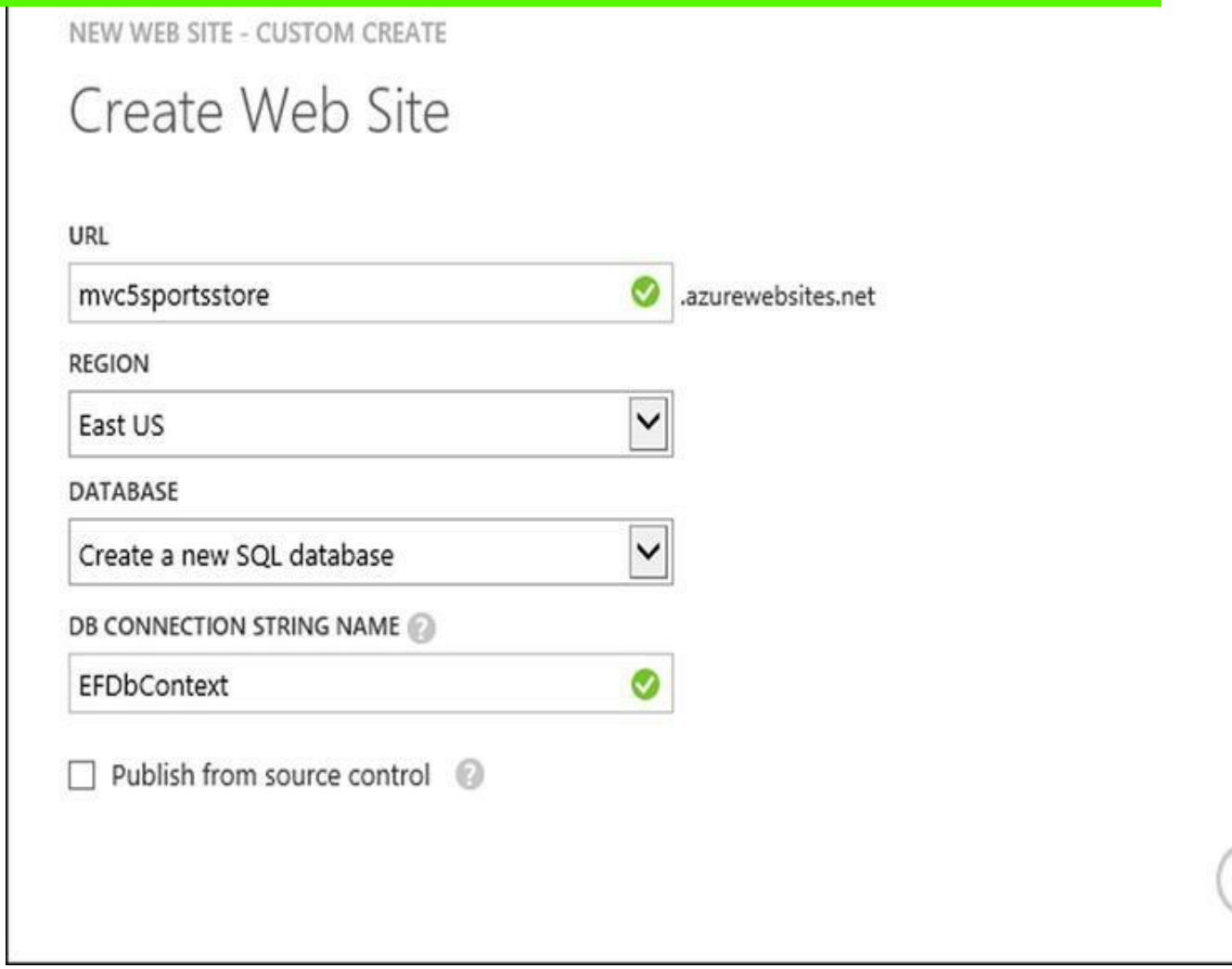


Figure 13-2. Creating a new web site and database

I need to select a URL for my application. For the free Azure services, I am restricted to names in the `azurewebsites.net` domain. I have chosen the name `mvc5sportsstore`, but you will have to choo name since each Azure web site requires a unique name.

Select the region that you want your application deployed to and ensure that the `Create a new SQL da` option is selected for the `Database` field. (Azure can use MySql, which the SportsStore application is not set up want the option that gives me a SQL Server database.)

Set the DB `Connection String Name` field to `EFDbContext`. This is the name the `SportsSt` application uses to get a database connection from the `Web.config` file, and by using this name in the Azure ser that the application code works in deployment without modification.

When you have filled out the form, click the arrow button to proceed to the form shown in Figure 13-3.

eBooks-IT.org

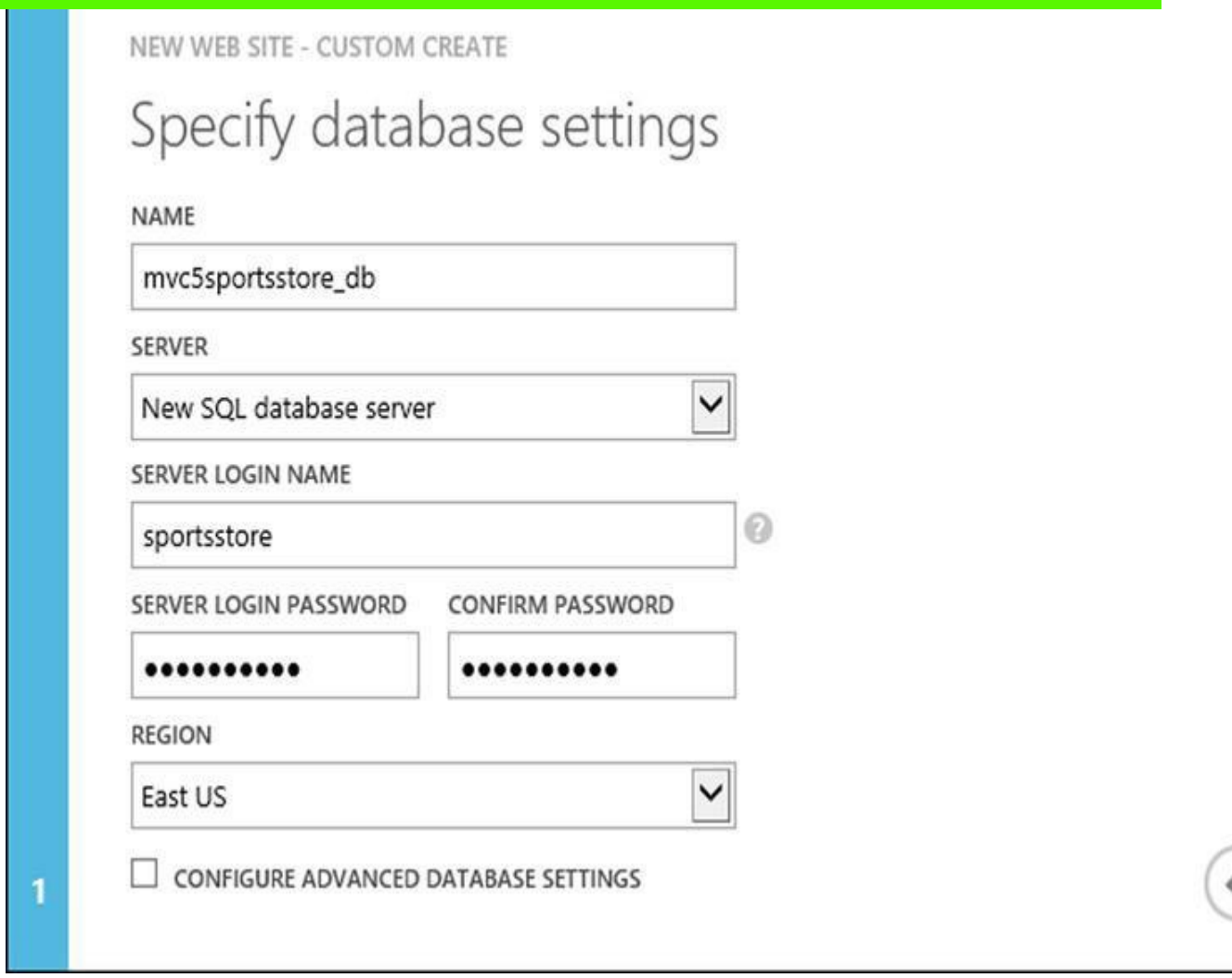


Figure 13-3. Configuring the database

Set a name for the database. I have used `mvc5sportsstore_db` so that it is obvious which application th relates to. Select the `New SQL Data Server` option for the `Server` field and enter a login name and pas specified a name of `sportsstore` and followed the guidance provided by the form to select a password contai case letters and numbers. Make a note of the username and password you use because you will need them in the nex Click the check mark button to complete the setup process. Azure will create new web site and database services, w a few minutes. You will be returned to the overview when setup is complete and you will see that the `Web Site` `Databases` categories each report one item, as shown in Figure 13-4.

eBooks-IT.org

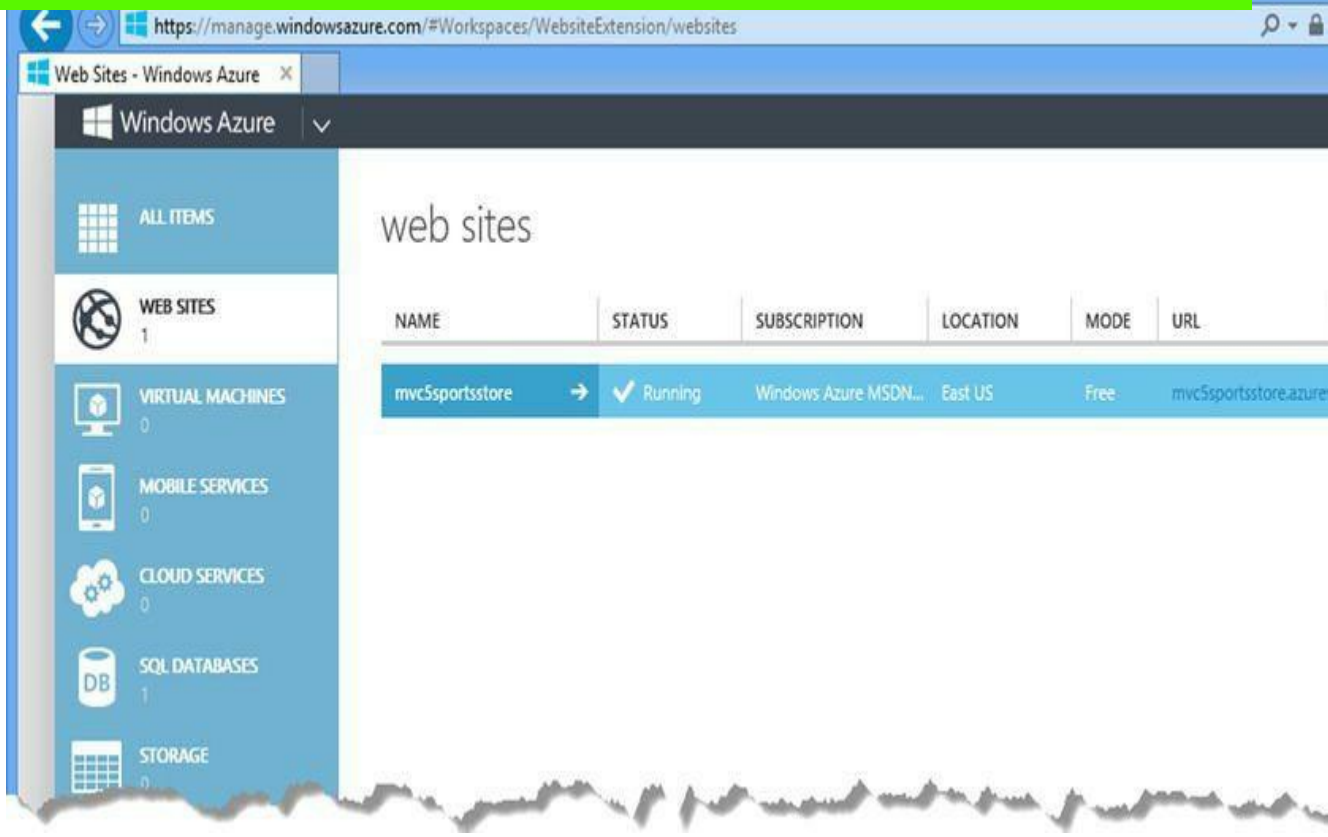


Figure 13-4. The effect of creating a web site with a database

Preparing the Database for Remote Administration

The next step is to configure the Azure database so that it contains the same schema and data that I used in Chapter
`SQL Databases` link in the Azure summary page and then click the entry that appears in the `SQL Databa`
(If you are following my example, the database will be called `mvc5sportsstore_db`.)

The portal will show you details of the database and various options for configuring and managing it. Click the `Se`
`Windows Azure firewall rules for this address` link, which you will find in the `Des`
`Database` section of the page. You will see a message that tells you that your current IP address (which is to say
address of your workstation) is not in the firewall rules. Click the `Yes` button as shown in Figure 13-5.

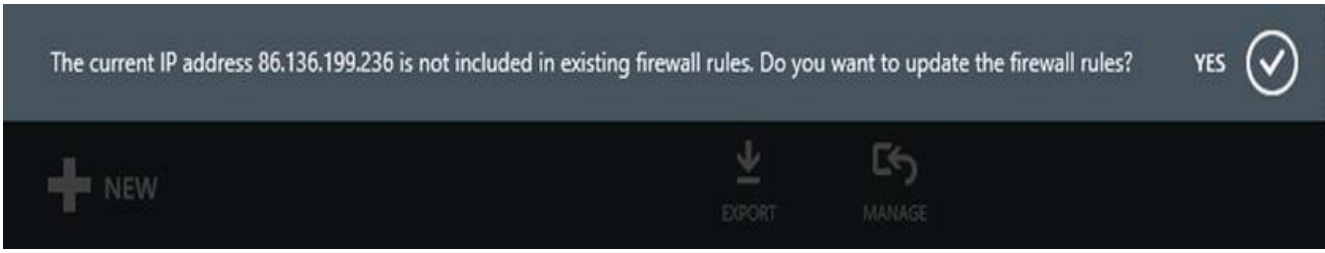


Figure 13-5. Adding the workstation IP address to the Azure firewall rules

Caution Visual Studio has support for deploying the database along with the application. I recommend against us
feature since it is trivially easy to wipe out your real application data with a careless menu selection. Always update y
separately and test thoroughly before doing so.

Creating the Schema

The next step is to create the schema for the database. Click the `Design your SQL database` link in th
`to your Database` section. Enter the database name (`mvc5sportsstore_db`), username (`sport`
and the password that you defined when creating the database, and click the Log On button, as shown in Figure 13-

eBooks-IT.org

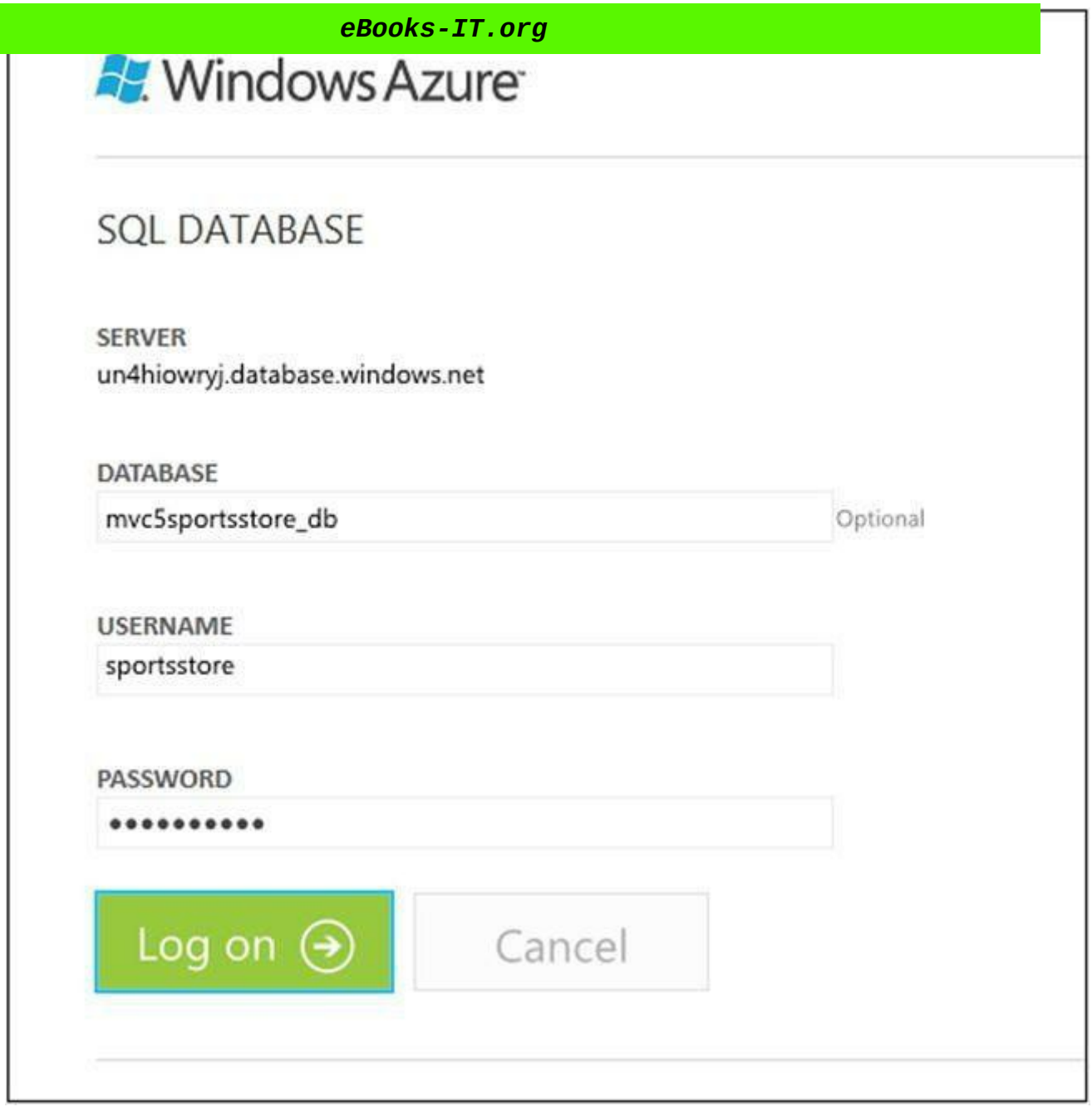


Figure 13-6. Connecting to the database

Tip Managing the database requires Silverlight, which you may need to install in your browser before you can co

At the top of the window you will see a `New Query` button. Click the button and a text area will enter into wh type SQL commands. This is where I am going to provide the SQL commands that will create the database table I ne

Getting the Schema Command

I can get the SQL statements I need from Visual Studio. Open the `Server Explorer` window and expand th contains until you reach the entry for the `Products` table in the development SportsStore application. Right-click `Products` table and select `Open Table Definition`. The editor for the table schema will be opened. tab, you will see the SQL shown in Listing 13-1.

Listing 13-1. The Statement to Create the Products Table

```
CREATE TABLE [dbo].[Products] (
```

eBooks-IT.org

```
    [Name]            NVARCHAR (100)  NOT NULL,
    [Description]     NVARCHAR (500)  NOT NULL,
    [Category]        NVARCHAR (50)   NOT NULL,
    [Price]           DECIMAL (16, 2) NOT NULL,
    [ImageData]       VARBINARY (MAX) NULL,
    [ImageMimeType]   VARCHAR (50)    NULL,
    PRIMARY KEY CLUSTERED ([ProductID] ASC)
);
```

Copy the SQL from Visual Studio, paste it into the text area in the browser, and click the `Run` button at the top o window. After a second, you will see the message `Command(s) completed successfully`, which the Azure database contains a `Product` database using the same schema as I defined in the `SportsStore` ap

Adding the Table Data

Now that I have created the table, I can populate it with the product data that I used in Chapter 7. Return to the `Pro` entry in the `Database Explorer` window, right-click, and select `Show Table Data` from the pop-u will find a `Script` button at the top of the window that is opened, as shown in Figure 13-7.

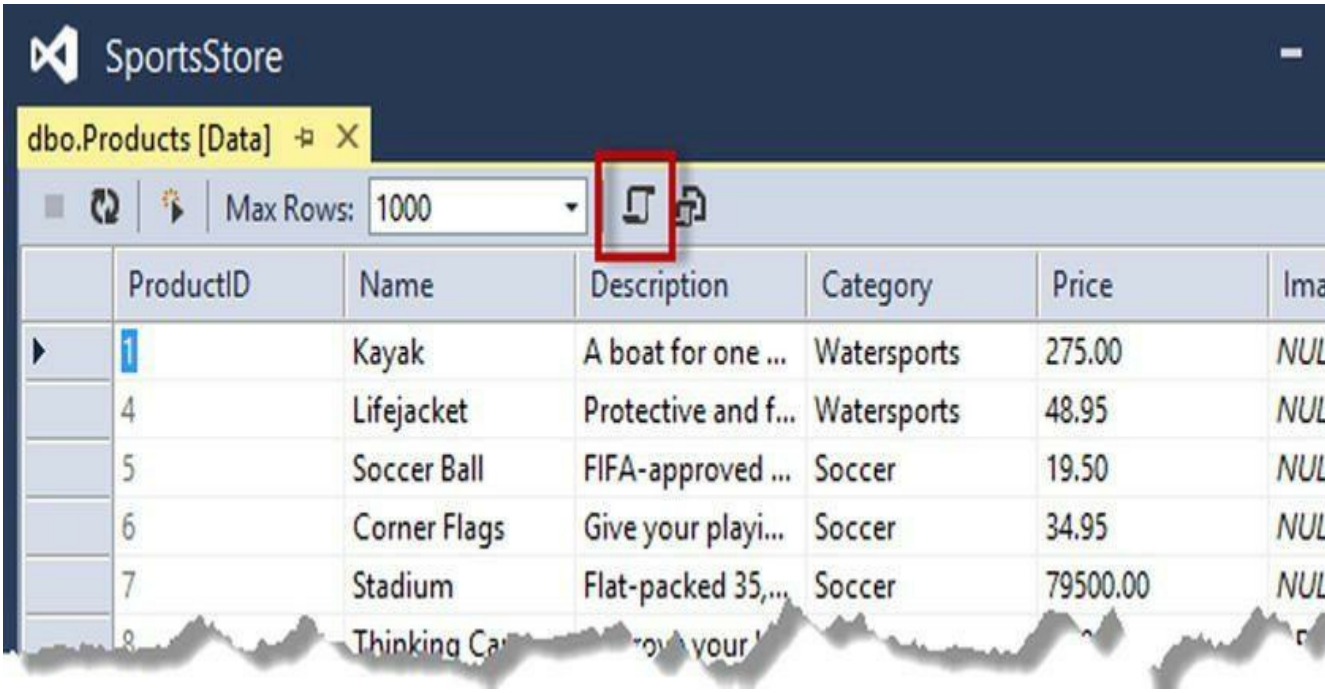


Figure 13-7. The script button in the table data display

A new window will open containing another SQL statement, which I have shown in Listing 13-2.

Listing 13-2. The SQL Statement to Add Data to the Products Table

```
SET IDENTITY_INSERT [dbo].[Products] ON
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
[Category], [Price], [ImageMimeType]) VALUES (1, N'Kayak', N'A b
one person', N'Watersports', CAST(275.00 AS Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
[Category], [Price], [ImageMimeType]) VALUES (4, N'Life
N'Protective and fashionable', N'Watersports', CAST(48.95 AS Dec
2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
```

eBooks-IT.org

```
approved size and weight', N'Soccer', CAST(19.50 AS Decimal(16, 2)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
[Category], [Price], [ImageMimeType]) VALUES (6, N'Corner Flags'
your playing field a professional touch', N'Soccer', CAST(3
Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
[Category], [Price], [ImageMimeType]) VALUES (7, N'Stadium', N'Fla
35,000-seat stadium', N'Soccer', CAST(79500.00 AS Decimal(16, 2)),
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
[Category], [Price], [ImageMimeType]) VALUES (8, N'Thinking
N'Improve your brain efficiency by 75%', N'Chess', CAST(1
Decimal(16, 2)), N'image/jpeg')
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
[Category], [Price], [ImageMimeType]) VALUES (9, N'Unsteady
N'Secretly give your opponent a disadvantage', N'Chess', CAST(2
Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
[Category], [Price], [ImageMimeType]) VALUES (10, N'Human Chess
N'A fun game for the family', N'Chess', CAST(75.00 AS Decimal(1
NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Descr
[Category], [Price], [ImageMimeType]) VALUES (11, N'Bling-Bling
N'Gold-plated, diamond-studded King', N'Chess', CAST(1200
Decimal(16, 2)), NULL)
SET IDENTITY_INSERT [dbo].[Products] OFF
```

Clear the text area in the Azure browser window and paste the SQL shown in the listing in its place. Click the Run
script will be executed and add the data to the table.

Deploying the Application

Now that the setup is complete, deploying the application is simple. Return to the main Azure portal and click the We
button. Click the `mvc5sportsstore` web site to open the dashboard page and click the `Download the
profile` link in the `Publish your app` section. Save this file in a prominent location.

For my Azure service, the file is called `mvc5sportsstore.azurewebsites.net.PublishSe
and I saved it to the desktop. This file contains the details that Visual Studio needs to publish your app to the Azure in

Return to Visual Studio and right-click the `SportsStore.WebUI` project in the Solution Explorer and selec
from the pop-up menu. You will see the Publish Web dialog window, as illustrated by Figure 13-8.

eBooks-IT.org

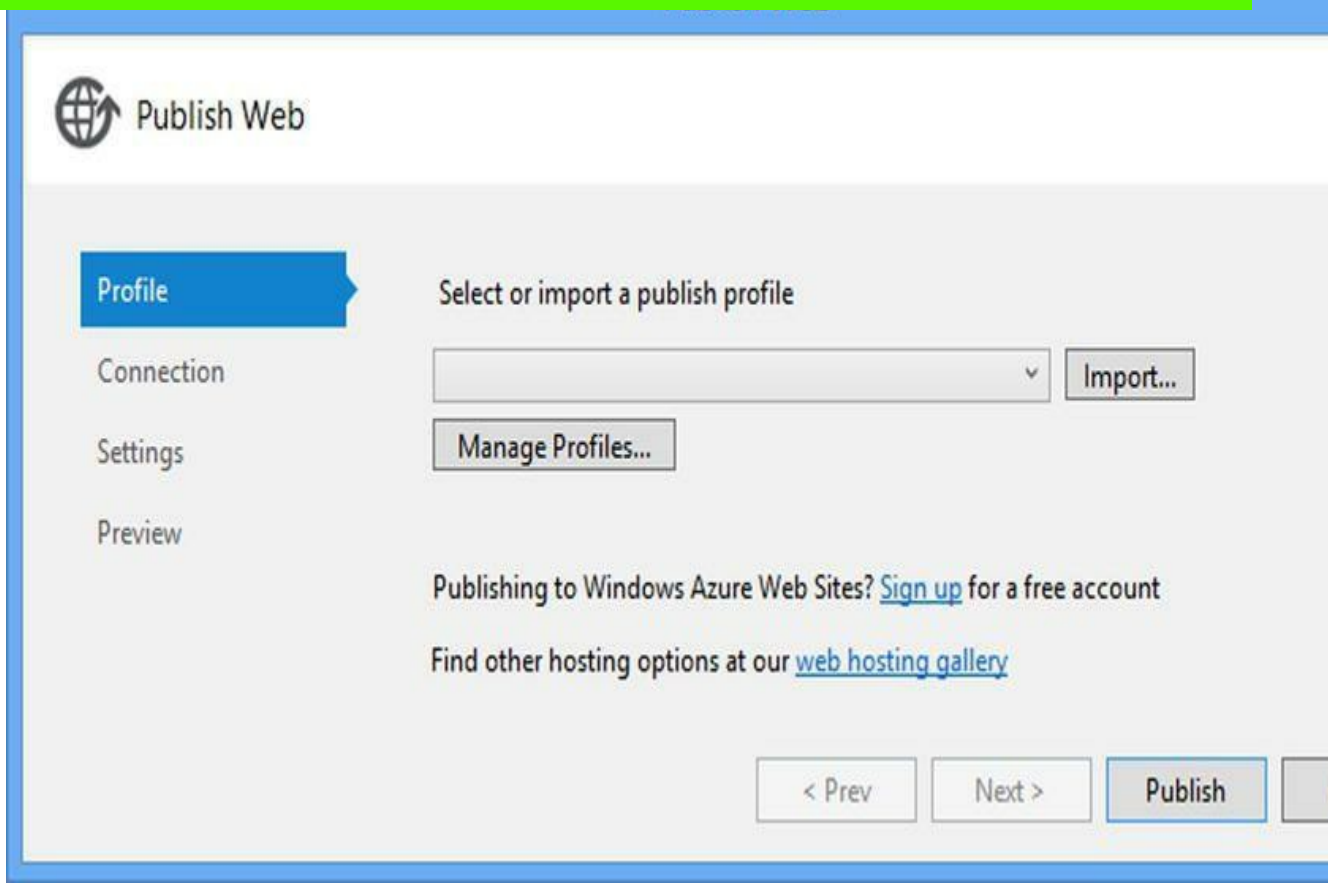


Figure 13-8. The Publish Web dialog

Click the `Import` button and locate the file that you downloaded from the Azure portal. Visual Studio will proce display the details of your Azure service configuration, as shown in Figure 13-9. Your details will reflect the name yo your web site.

eBooks-IT.org

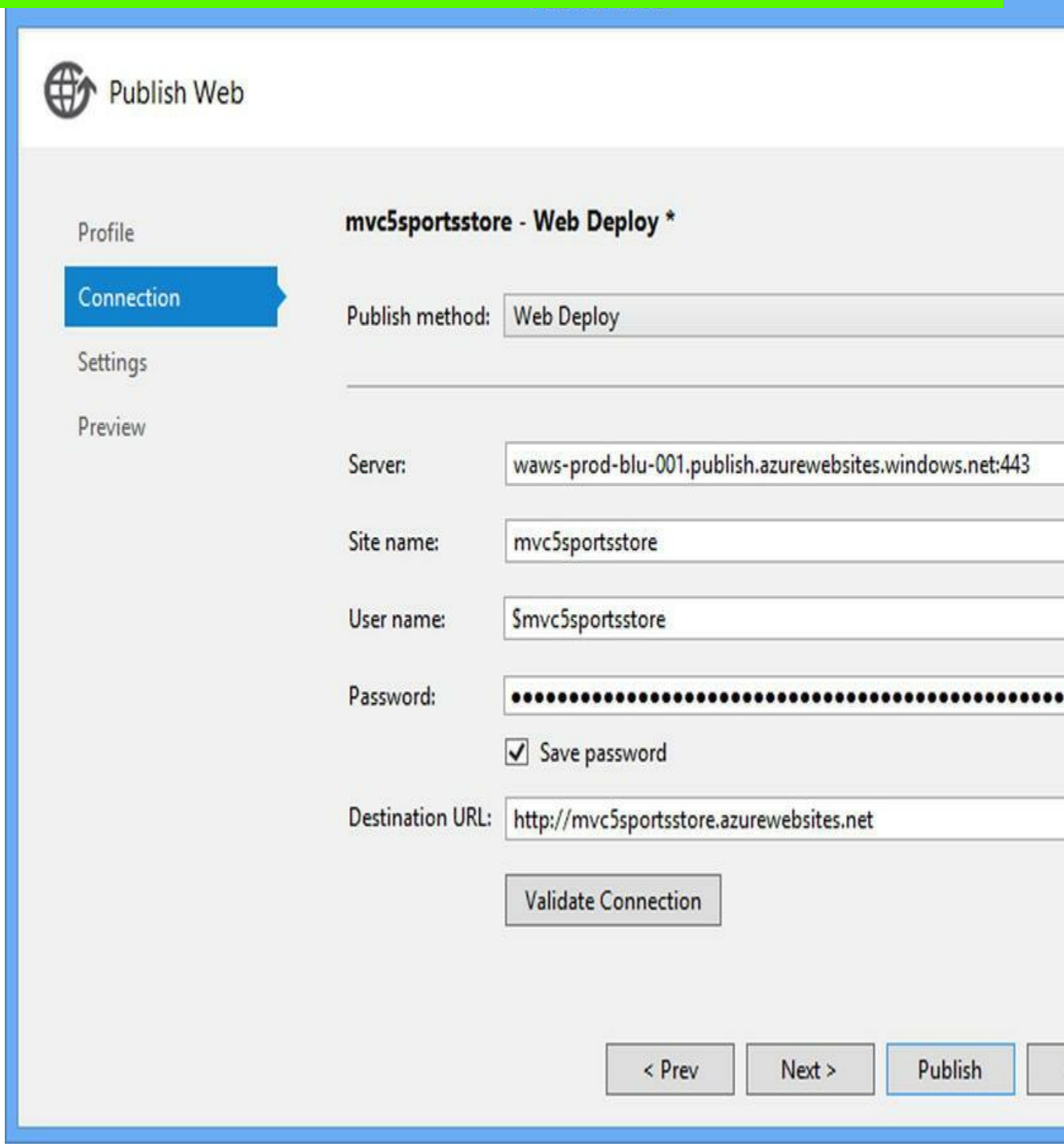


Figure 13-9. Details of the Azure service that the application will be deployed to

There is no need to change any of the values that are displayed. Click the `Next` button to move to the next stage deployment process, which you can see in Figure 13-10.

eBooks-IT.org

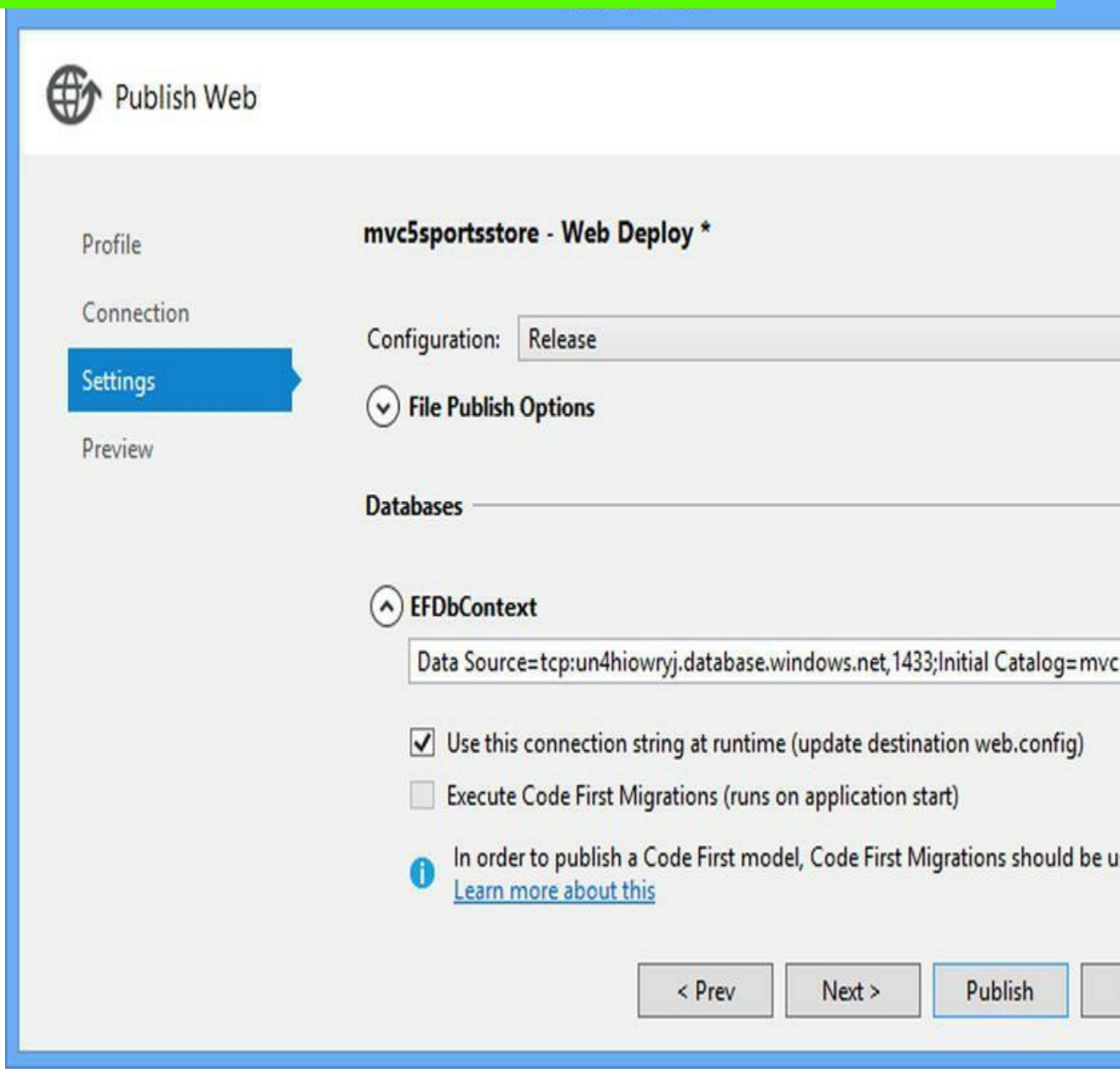


Figure 13-10. Settings for the deployed application

You can choose the configuration that will be used in deployment. This will usually be `Release`, but you can s if you intend to test your application on the Azure infrastructure and want the debug settings for the compiler and yo bundles.

The other part of this process is configuring database connections. Visual Studio gives you the opportunity to crea between the database connections defined in your project and the databases that are associated with your Azure web `Web.config` file contains only one set of details, and since I only created one Azure database, there is only one from the drop-down list. If you have multiple databases in your application, you should take care to ensure that the r database is selected.

Click the `Next` button to preview the effect of your deployment, as shown in Figure 13-11. When you click the `Preview` button, Visual Studio walks through the deployment process, but does not actually send the files to the s are upgrading an application that is already deployed, this can be a useful check to ensure that you are only replacing you expect.

eBooks-IT.org

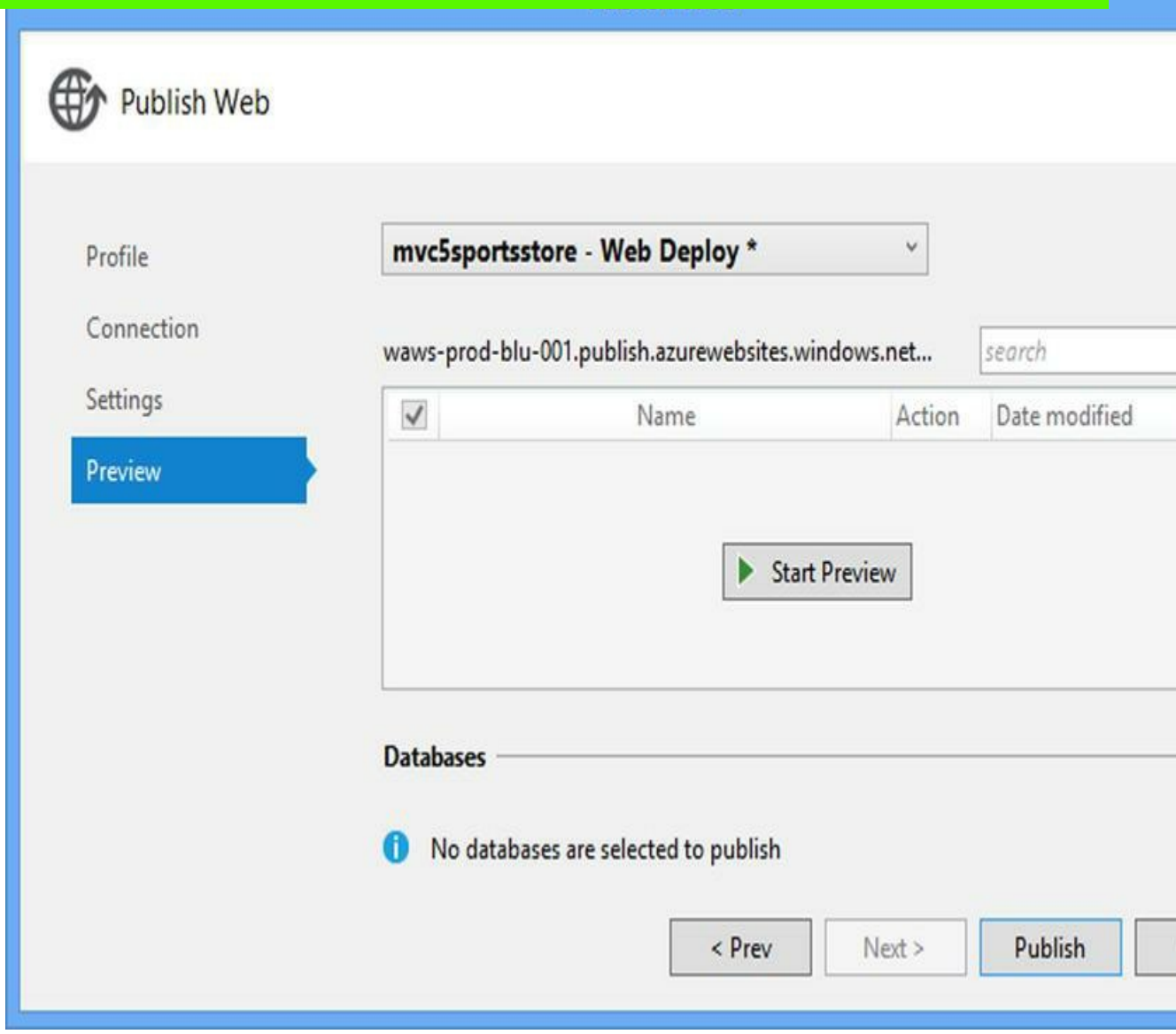


Figure 13-11. The Preview section of the Publish Web dialog

This is the first time that I have deployed this application, so all the files in the project will appear in the preview w shown in Figure 13-12. Notice that each file has a check box next to it. You can prevent individual files from being d although you should be careful when doing this. I am pretty conservative in this regard and I would rather deploy file not need rather than forget to deploy one that I do.

eBooks-IT.org

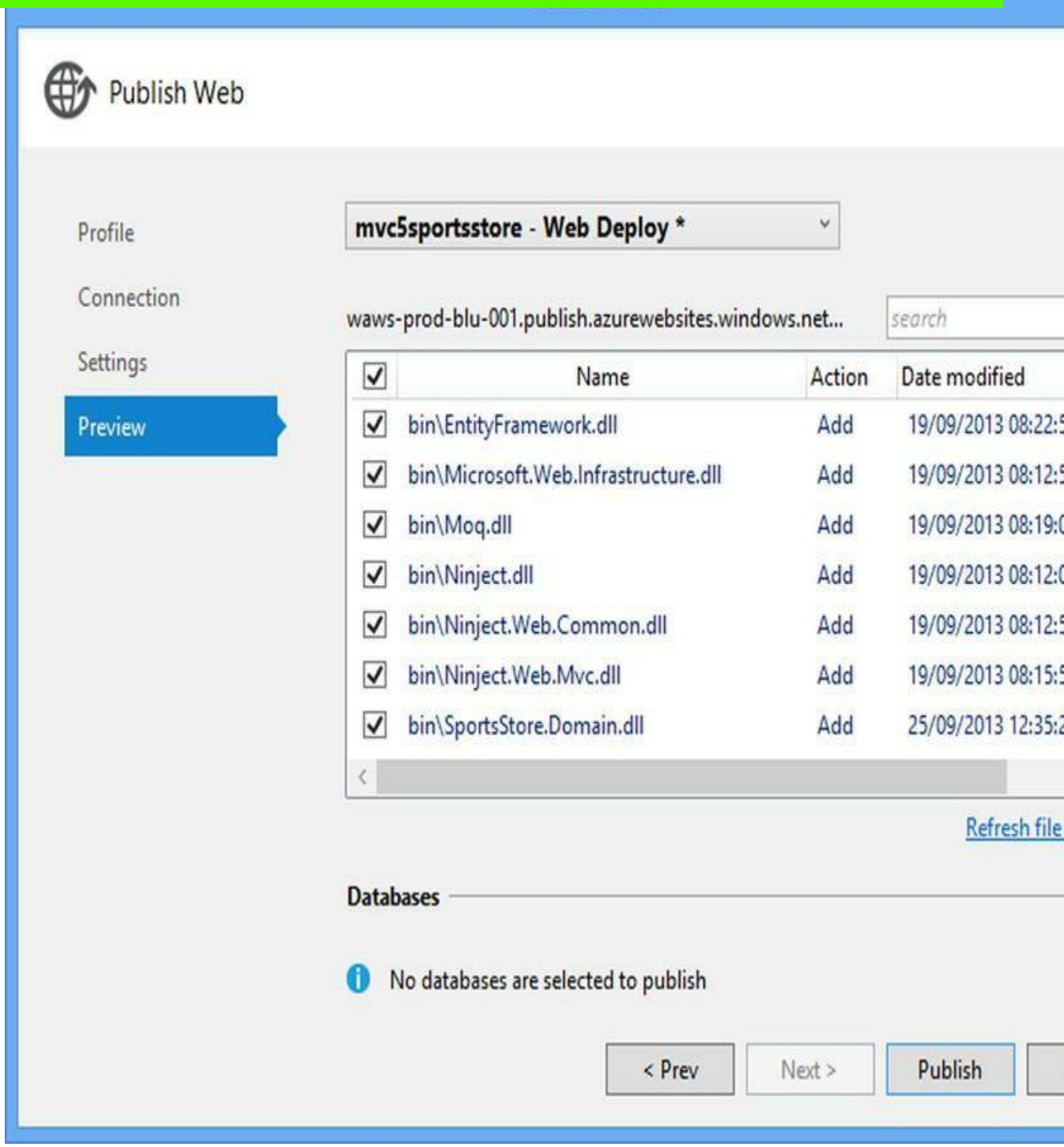


Figure 13-12. Previewing the deployment changes

Click the `Publish` button to deploy your application to the Azure platform. The `Publish Web` dialog win and you will be able to see details of the deployment progress in the Visual Studio `Output` window, as shown in F

eBooks-IT.org

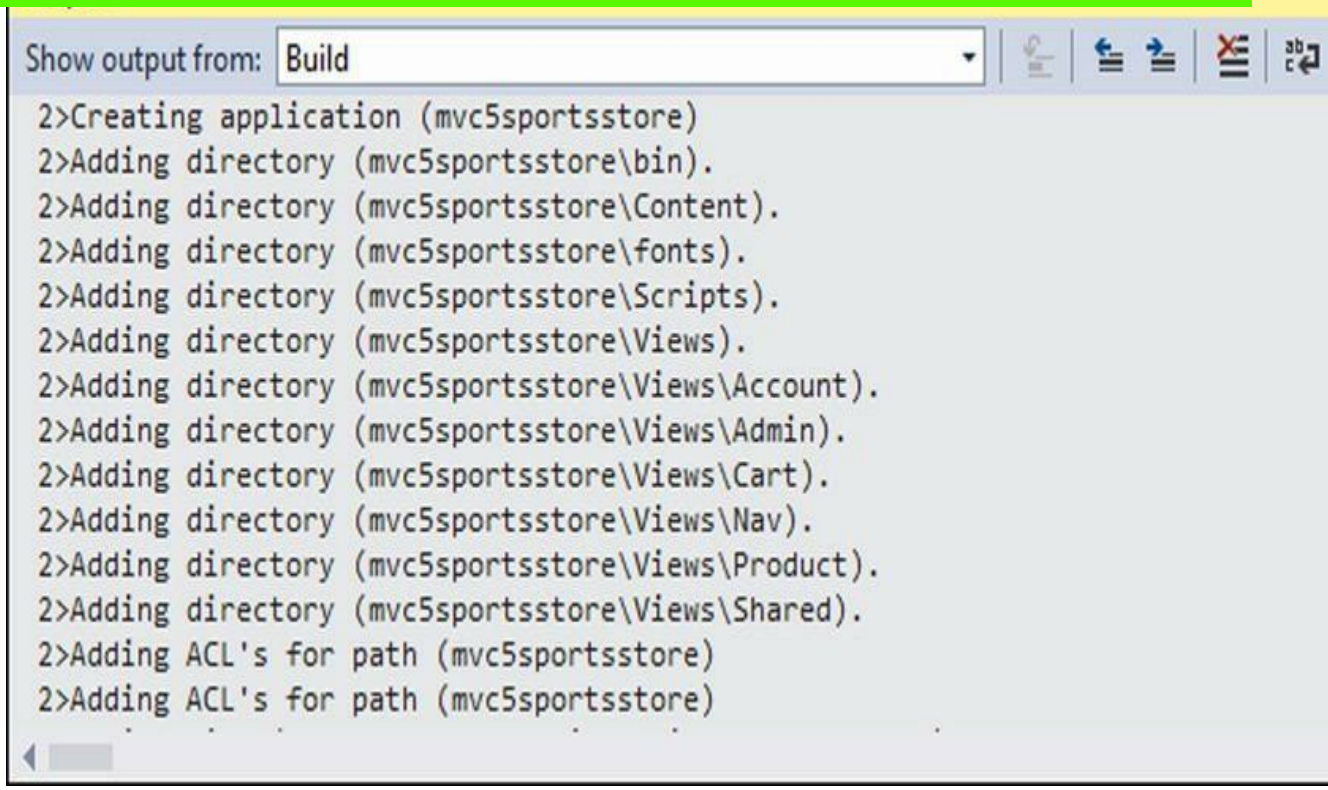


Figure 13-13. Deploying an application to the Azure platform

It can take a few minutes to deploy an application, but then the process is complete. Visual Studio will open a brov that navigates to the URL of your Azure web site. For me, this URL is `http://mvc5sportsstore.azurewebsites.net`, as shown in Figure 13-14.

eBooks-IT.org

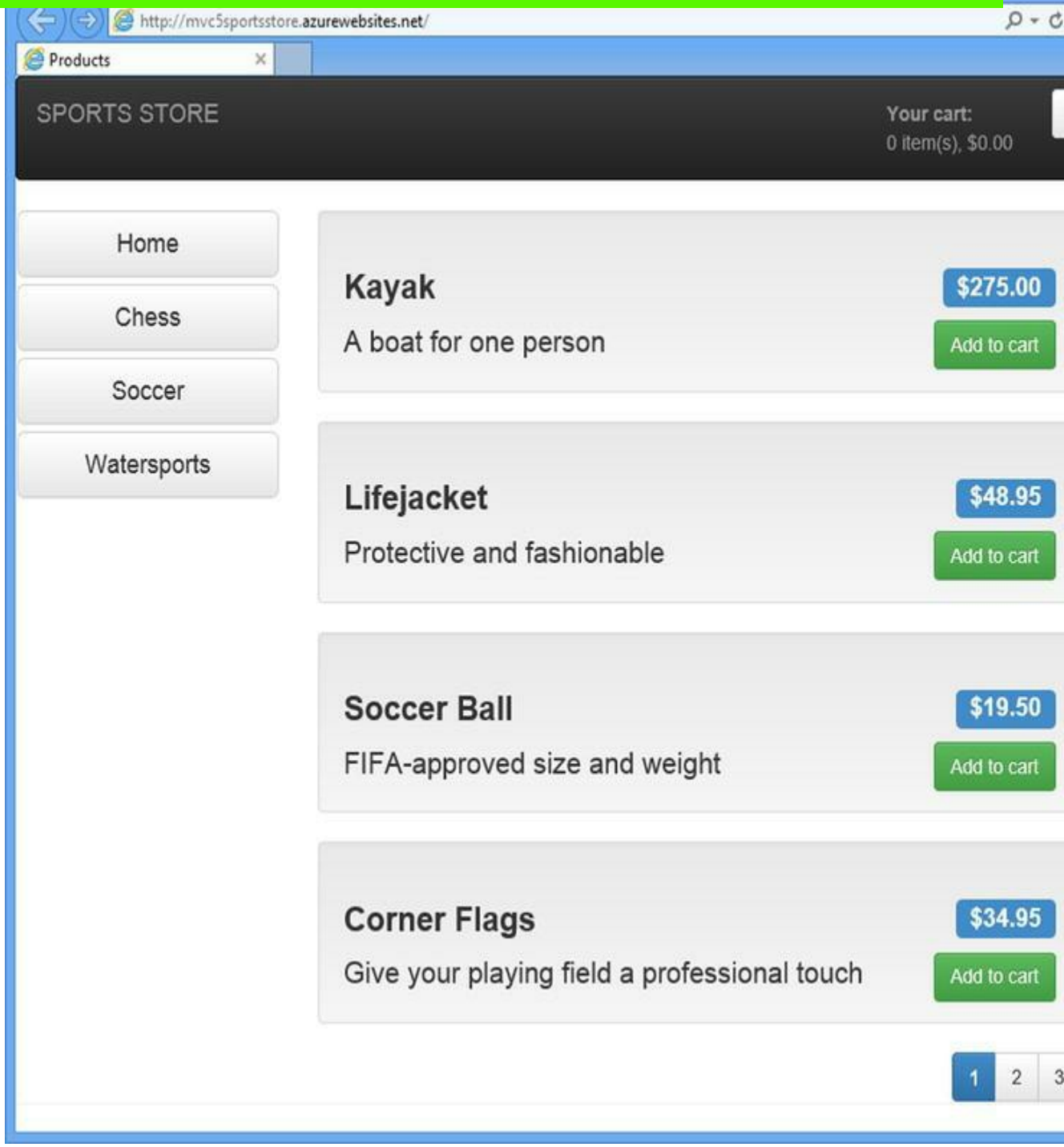


Figure 13-14. The SportsStore application running on the Windows Azure platform

Summary

In this chapter, I have shown you how to deploy an MVC Framework to the Windows Azure platform. There are ma ways to deploy applications and many different platforms that you can target, but the process I have shown you in th representative of what you can expect, even if you don't use Azure.

And that's the end of the SportsStore application and this part of the book. In Part 2, I begin the process of diggin detail and showing you how the features I used to create the application work in depth.

eBooks-IT.org

Overview of MVC Projects

I am going to provide some additional context before diving into MVC Framework features. This chapter gives an ov structure and nature of an ASP.NET MVC application, including the default project structure and naming convention which are optional and some of which are hard-coded into the way that the MVC Framework operates.

Working with Visual Studio MVC Projects

When you create a new ASP.NET project, Visual Studio gives you a number of choices as to the initial content you w project. The idea is to ease the learning process for new developers and apply some time-saving best practices for co features and tasks. This support continues with templates used to create controllers and views that are created with t for listing data objects, editing model properties and so on.

With Visual Studio 2013 and MVC 5, Microsoft has updated the templates and *scaffolding*, as it is known, to blur boundaries between different kinds of ASP.NET project and to provide a wider range of project templates and code configurations.

Part 1 of this book will have left you in no doubt that I am not a fan of this kind of approach to cookie cutter proje The intent is good, but the execution is always underwhelming. One of the characteristics I like most about ASP.NET MVC Framework is just how much flexibility I have in tailoring the platform to suit my development style, and the pr classes and views that Visual Studio creates and populates make me feel constrained to work in someone else's style the content and configuration too generic and too bland to be useful. In Chapter 10, I mentioned that one of the dang responsive design to target mobile devices is a kind of averaging that ends up compromising the experience for all de something similar has happened to the Visual Studio templates. Microsoft can't possibly know what kind of applicati to create and so they cover all the bases, but in such a drab and generalized way that I end up just ripping out the def anyway.

My advice (given to anyone who makes the mistake of asking) is to start with an empty project and add the folder packages that you need. Not only will you learn more about the way that the MVC Framework works, but you will h control over what your application contains.

But my preferences should not color your development experience. You may find the templates and scaffolding m than I do, especially if you are new to ASP.NET development and you have not yet developed a distinctive personal c style that suits you. You may also find the project templates a useful resource and a source of ideas, although you sh cautious about adding any functionality to an application before you completely understand how it works.

Creating the Project

When you first create a new MVC Framework project, you have two basic starting points to choose from: the `Emp` and the `MVC` template. The names are a little misleading, because you can add the basic folders and assemblies requi MVC Framework to any project template by checking the `MVC` option in the `Add Folders and Core References` section of the `New ASP.NET Project` dialog window, as shown in Figure 14-1. For the this option is checked for you.

eBooks-IT.org

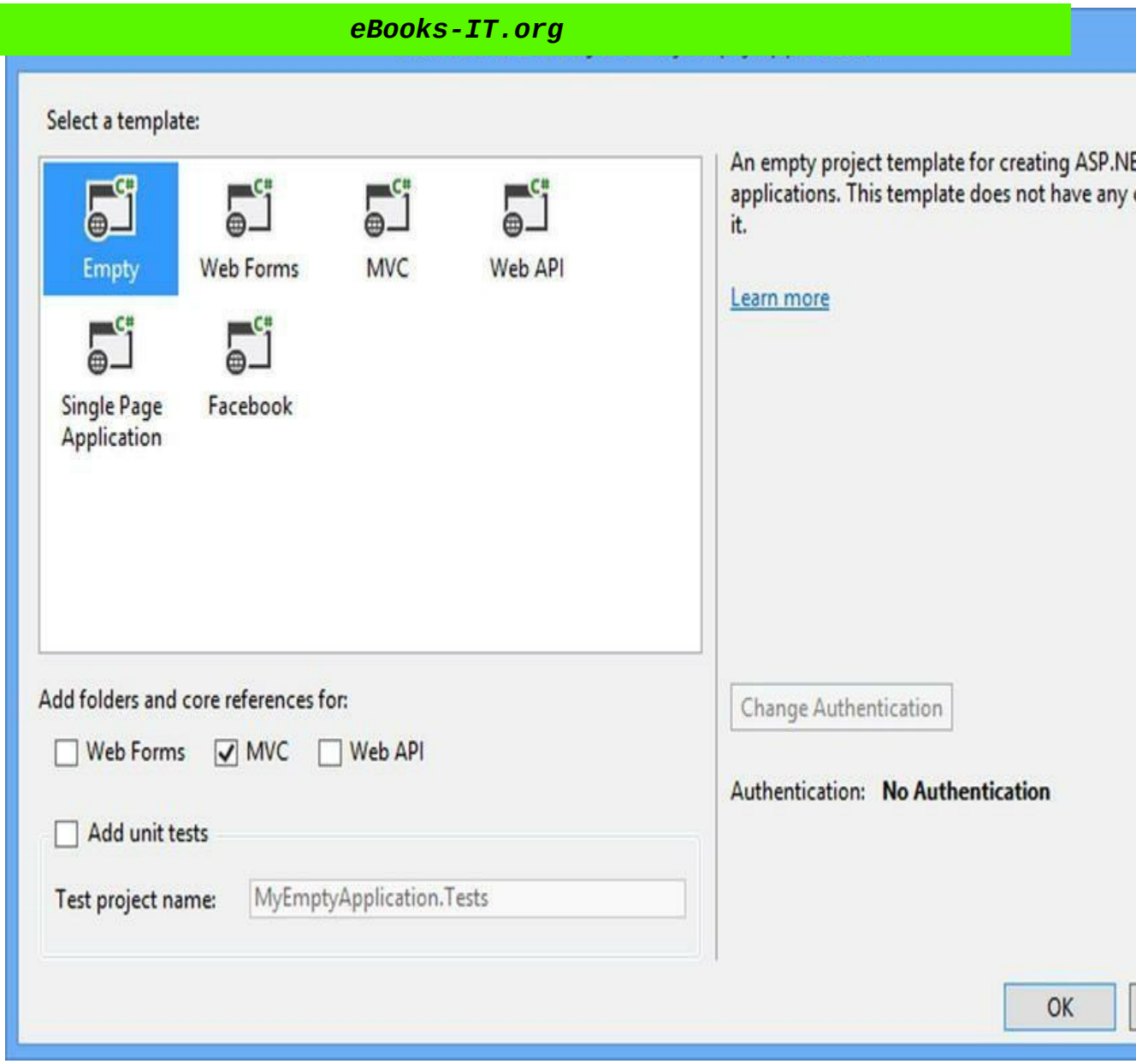


Figure 14-1. Selecting the project type, folders and assemblies for a new project

The real difference is the additional content that the `MVC` project template adds to new projects, which provides a starting point that includes some default controllers and views, a security configuration, some popular JavaScript anc packages (such as jQuery and Bootstrap) and a layout that uses Bootstrap to provide a theme for the application cont `Empty` project option just contains the basic references required for an MVC framework and the barebones folder There is a fair amount of content added by the `MVC` template and you can see the differences in Figure 14-2, which contents of two newly created projects. The one on the left was created with the `Empty` template with the MVC fo references option checked. The others show the content of a project that was created with the `MVC` template, and to show the files on the page, I had to focus the Solution Explorer on different folders because a single listing was too l printed page.

eBooks-IT.org

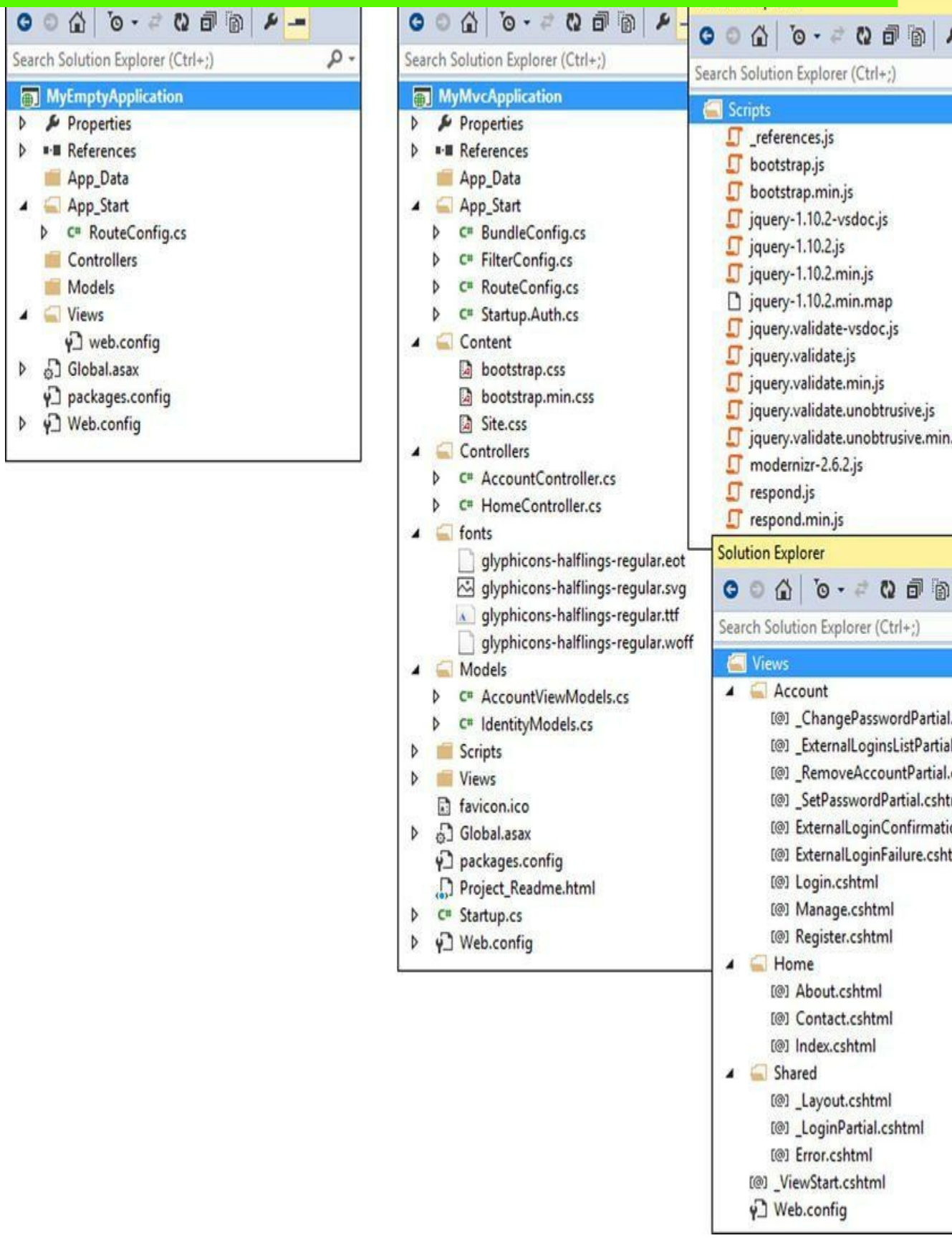


Figure 14-2. The default content added to a project by the Empty and MVC templates

The extra files that the MVC project adds look worse than they really are. Some are related to authentication and o
JavaScript and CSS files for which there are regular and minified versions. (I describe how these can be used in Cha

Tip Visual Studio assembles a project created with the MVC template using NuGet packages, which means that y

eBooks-IT.org

Studio Tools ➤ Library Package Manager menu. It also means that you can add the same pa project, including one created with the `Empty` template (and this is what I was doing for the examples in Part 1 of

Whichever template you choose, you will notice that the resulting projects have similar folder structures. Some of an MVC project have special roles, which are hard-coded into ASP.NET or the MVC Framework. Others are subject conventions. I have described each of the core files and folders in Table 14-1, some of which are not present in proj default but which I introduce in later chapters.

Table 14-1. *Summary of MVC Project Items*

Folder or File	Description	Notes
/App_Data	This folder is where you put private data, such as XML files or databases if you are using SQL Server Express, SQLite, or other file-based repositories.	IIS will not serve the contents of this folder.
/App_Start	This folder contains some core configuration settings for your project, including the definition of routes and filters and content bundles.	I describe routes in Chapters 15 and 16, filte and content bundles in Chapter 26.
/Areas	Areas are a way of partitioning a large application into smaller pieces.	I describe areas in Chapter 15.
/bin	The compiled assembly for your MVC application is placed here, along with any referenced assemblies that are not in the GAC.	You won't see the **bin** directory in the **Solu** window unless you click the **Show All Files** bu are binary files generated on compilation, you s normally store them in source control.
/Content	This is where you put static content such as CSS files and images.	This is a convention but not required. You can content anywhere that suits you.
/Controllers	This is where you put your controller classes.	This is a convention. You can put your control anywhere you like, because they are all compile assembly.
/Models	This is where you put your view model and domain model classes, although all but the simplest applications benefit from defining the domain model in a dedicated project, as I demonstrated for SportsStore.	This is a convention. You can define your mod anywhere in the project or in a separate projec
/Scripts	This directory is intended to hold the JavaScript libraries for your application.	This is a convention. You can put script files in as they are just another type of static content. 26 for more information about managing script
/Views	This directory holds views and partial views, usually grouped together in folders named after the controller with which they are associated.	The **/Views/web.config** file prevents IIS fr content of these directories. Views must be rend action method.
/Views/Shared	This directory holds layouts and views which are not specific to a single controller.	
/Views/Web.config	This is *not* the configuration file for your application. It contains the configuration required to make views work with ASP.NET and prevents views from being served by IIS and the namespaces imported into views by default.	
/Global.asax	This is the global ASP.NET application class. Its code-behind class (**Global.asax.cs**) is the place to register routing configuration, as well as set up any code to run on application initialization or shutdown, or when unhandled exceptions occur.	The **Global.asax** file has the same role in a application as it does in a Web Forms applicati
/Web.config	This is the configuration file for your application.	The **Web.config** file has the same role in a application as it does in a Web Forms applicati

Understanding MVC Conventions

There are two kinds of conventions in an MVC project. The first kind is just suggestions as to how you might like to your project. For example, it is conventional to put your JavaScript files in the `Scripts` folder. This is where oth developers would expect to find them, and where NuGet packages will install them. But you are free to rename the S folder, or remove it entirely and put your scripts somewhere else. That would not prevent the MVC Framework from your application as long as the `script` elements in your views refer to the location you settle on.

eBooks-IT.org

points that made Ruby on Rails so popular. Convention over configuration means that you don't need to explicitly co
associations between controllers and their views, for example. You just follow a certain naming convention for your
everything just works. There is less flexibility in changing your project structure when dealing with this kind of conv
following sections explain the conventions that are used in place of configuration.

Tip All of the conventions can be changed if you are using a custom view engine (which I cover in Chapter 20),
a step to be taken lightly and, for the most part, these are the conventions you will be dealing with in MVC projects.

Following Conventions for Controller Classes

Controller classes must have names that end with `Controller`, such as `ProductController`, `AdminController`, and `HomeController`. When referencing a controller from elsewhere in the projec
when using an HTML helper method, you specify the first part of the name (such as `Product`), and the MVC Fra
automatically appends `Controller` to the name and starts looking for the controller class.

Tip You can change this behavior by creating your own implementation of the `IControllerFactory` in
which I describe in Chapter 19.

Following Conventions for Views

Views and partial views go into the folder `/Views/` *Controllername*. For example, a view associated with the `ProductController` class would go in the `/Views/Product` folder.

Tip Notice that I omit the `Controller` part of the class from the `Views` folder: `/Views/Product`, `/Views/ProductController`. This may seem counterintuitive at first, but it quickly becomes second na

The MVC Framework expects that the default view for an action method should be named after that method. For
default view associated with an action method called `List` should be called `List.cshtml`. Thus, for the `Lis`
method in the `ProductController` class, the default view is expected to be `/Views/Product/List.cshtml`. The default view is used when you return the result of calling the `Vi`
an action method, like this:

```
...
return View();
...
```

You can specify a different view by name, like this:

```
...
return View("MyOtherView");
...
```

Notice that I do not include the file name extension or the path to the view. When looking for a view, the MVC Fra
looks in the folder named after the controller and then in the `/Views/Shared` folder. This means that I can put
will be used by more than one controller in the `/Views/Shared` folder and the framework will find them.

Following Conventions for Layouts

The naming convention for layouts is to prefix the file with an underscore (_) character, and layout files are placed i
`/Views/Shared` folder. This layout is applied to all views by default through the `/Views/_ViewStart`
file. If you do not want the default layout applied to views, you can change the settings in `_ViewStart.csht`
the file entirely) to specify another layout in the view, like this:

```
@{
    Layout = "~/Views/Shared/_MyLayout.cshtml";
```

Or you can disable any layout for a given view, like this:

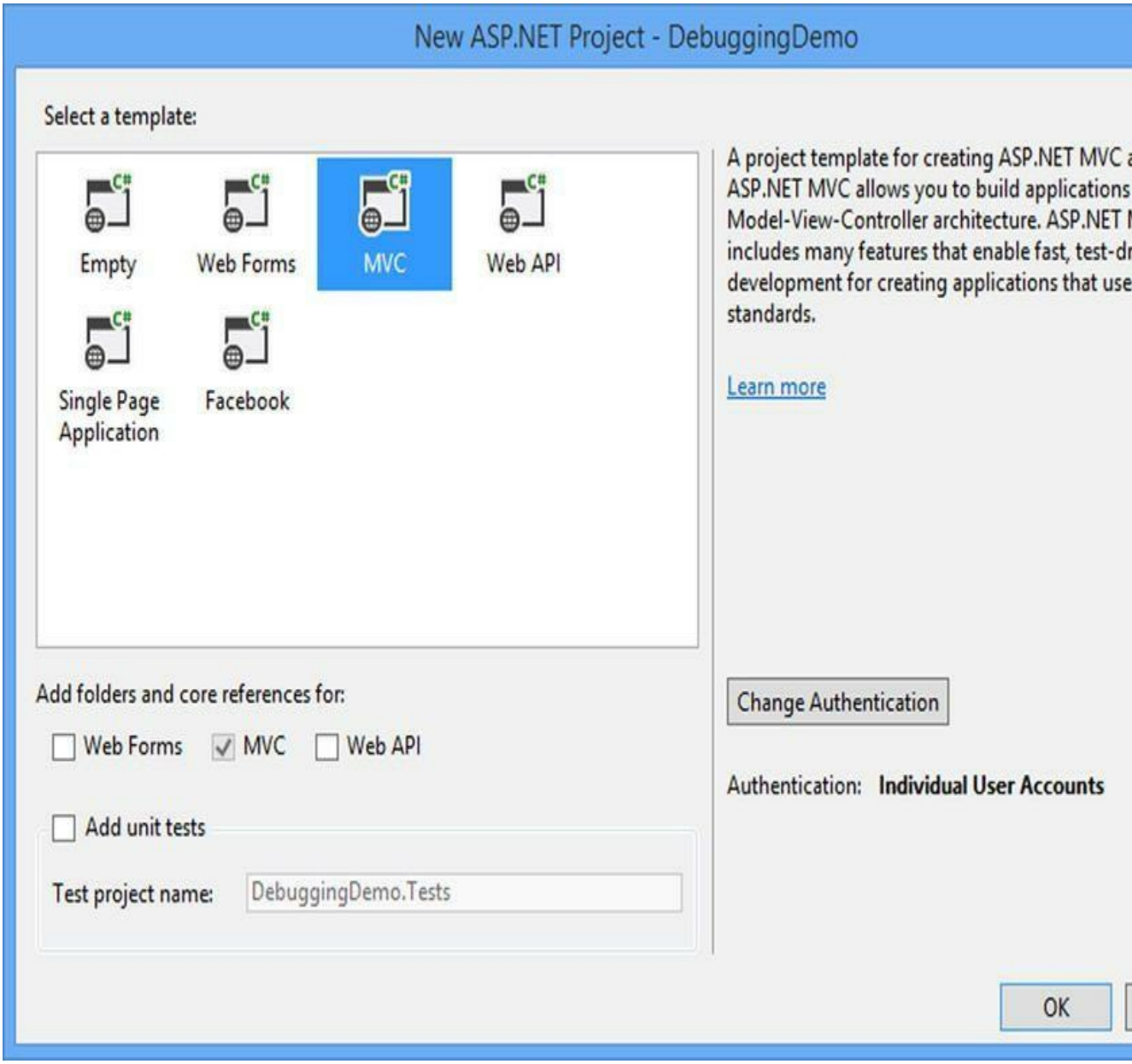
```
@{
    Layout = null;
}
```

Debugging MVC Applications

You debug an ASP.NET MVC application in exactly the same way as you debug an ASP.NET Web Forms applicatio
Studio debugger is a powerful and flexible tool, with many features and uses. I can only scratch the surface in this b
the sections that follow I show you how to set up the debugger and perform different debugging activities on your M

Preparing the Example Project

To demonstrate using the debugger, I have created a new MVC project using the `MVC` project template, just so you
the default content and configuration is set up and the effect of the default theme that is applied to views. I called the
`DebuggingDemo`, as shown in Figure 14-3. I have chosen the `Individual User Accounts` authe
option, which sets up a basic user security system.



eBooks-IT.org

Click the OK button and Visual Studio will create the project and add the default packages, files, and folders that th template includes. You can see how the files and settings added to the project are applied by starting the application, a Figure 14-4.

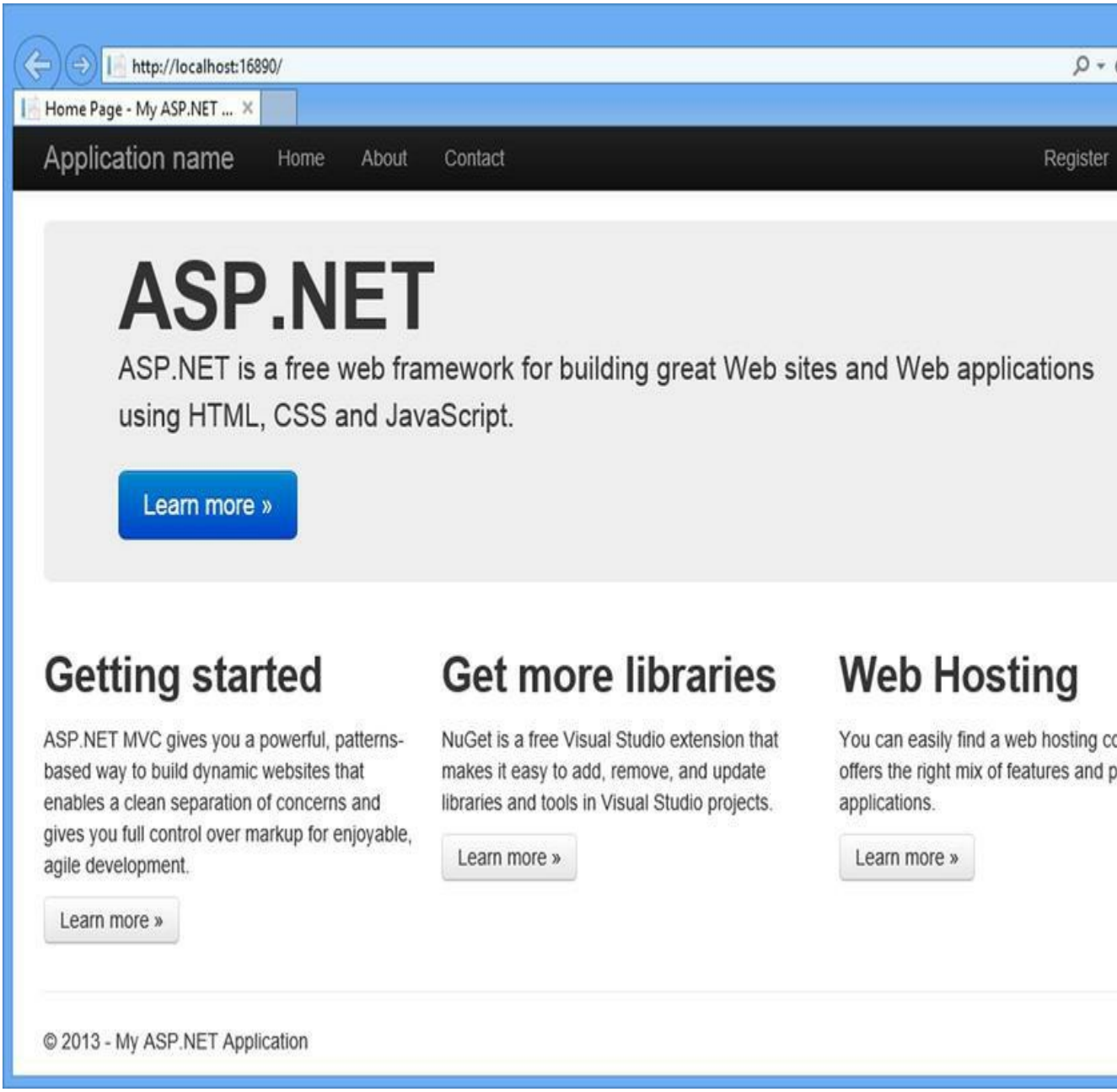


Figure 14-4. The effect of the additions made by the MVC project template

There are some placeholder elements for the name of the application and branding, and some pointers to the MVC NuGet, and some hosting options. The navigation bar at the top of the screen is the same kind I used for the SportsS application and the layout has some responsive features. Change the width of the window to see the effect.

Creating the Controller

Visual Studio creates a Home controller as part of the initial project content, but I am going to replace the code that added with that shown in Listing 14-1.

Listing 14-1. The Contents of the HomeController.cs File

```
namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
            int secondVal = 5;
            int result = firstVal / secondVal;

            ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
    }
}
```

Creating the View

Visual Studio also created the `Views/Home/Index.cshtml` view file as part of the project setup. I don't
default content and have to replace it with the markup shown in Listing 14-2.

Listing 14-2. The Contents of the Index.cshtml File

```
@model int

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Site.css" rel="stylesheet" type="text/cs
    <title>Index</title>
</head>
<body>
    <h2 class="message">@ViewData["Message"]</h2>
    <p>
        The calculation result value is: @Model
    </p>
</body>
</html>
```

The last preparatory step I need to take is to add a style to the `/Content/Site.css` file and change one o
ones, as shown in Listing 14-3. The `Site.css` file is created by Visual Studio as part of the `MVC` project templa
default location for application CSS styles. (I added a `link` element to the view in Listing 14-2 that imports this file
`Index.cshtml` view.)

Listing 14-3. Adding a Style to the /Content/Site.css File

```
body { padding-top: 5px; padding-bottom: 5px; }
.field-validation-error { color: #b94a48; }
```

eBooks-IT.org

```
input.input-validation-error { border: 1px solid #b94a48; }
input[type="checkbox"].input-validation-error { border: 0 none; }
.validation-summary-errors { color: #b94a48; }
.validation-summary-valid { display: none; }
.no-color { background-color: white; border-style:none; }
.message { font-size: 20pt; text-decoration: underline;}
```

Launching the Visual Studio Debugger

Visual Studio prepares new projects for debugging automatically, but it is useful to understand how to change the co
The important setting is in the `Web.config` file in the root project folder and can be found in the `system.we`
as shown in Listing 14-4.

Listing 14-4. The Debug Attribute in the Web.config File

```
...
<system.web>
   <compilation debug="true" targetFramework="4.5.1" />
   <httpRuntime targetFramework="4.5.1" />
</system.web>
...
```

A lot of compilation in an MVC Framework project is done when the application is running in IIS, and so you need
that the `debug` attribute on the compilation attribute is set to `true` during the development process. This ensures
debugger is able to operate on the classes files produced through on-demand compilation

Caution Do not deploy your application to a production server without setting the `debug` value to `false`. If
using Visual Studio to deploy your application (as I did in Chapter 13), then the setting will be changed automatically
select the Release configuration for the project.

In addition to the `Web.config` file, I want to ensure that Visual Studio includes debugging information in the
it creates. This isn't critical, but it can cause problems if the different debug settings are not in sync. Ensure that the
configuration is selected in the Visual Studio toolbar, as shown in Figure 14-5.

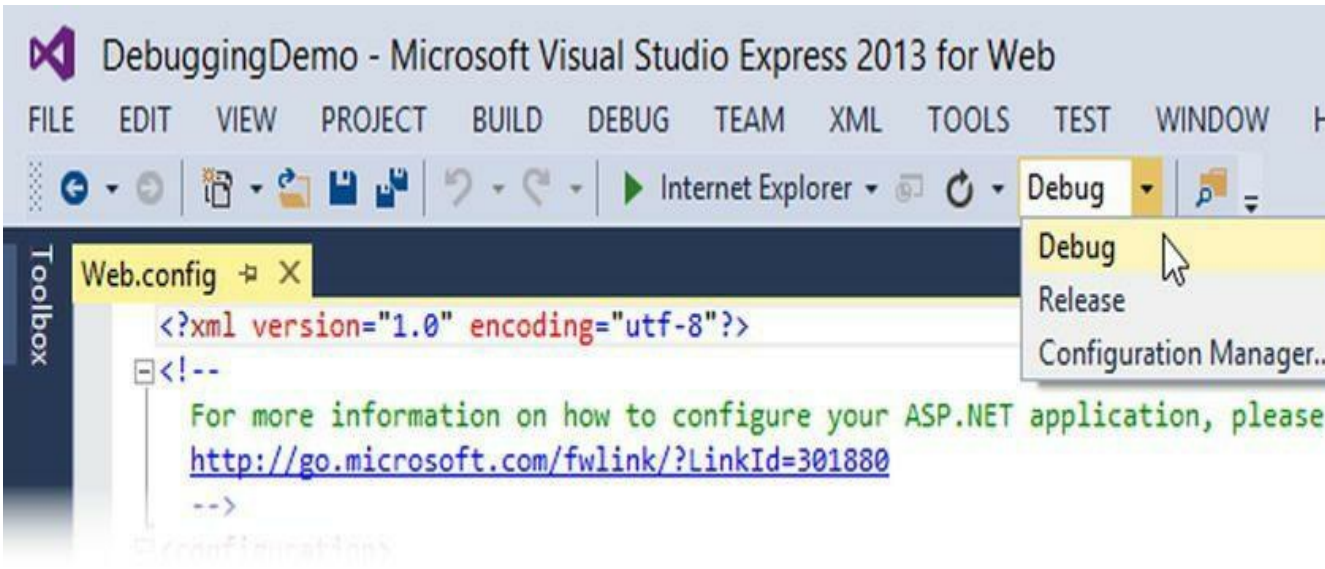


Figure 14-5. Selecting the Debug configuration

To debug an MVC Framework application, select `Start Debugging` from the Visual Studio `Debug` men
the green arrow in the Visual Studio toolbar (which you can see in Figure 14-5, next to the name of the browser that
to display the app—Internet Explorer in this case).

eBooks-IT.org

display the dialog shown in Figure 14-6. Select the option which allows Visual Studio to edit the `Web.Config` fil
the `OK` button and the debugger will start.

Figure 14-6. The dialog that Visual Studio displays when the Web.config File disables debugging

At this point, your application will be displayed in a new browser window, as shown in Figure 14-7.

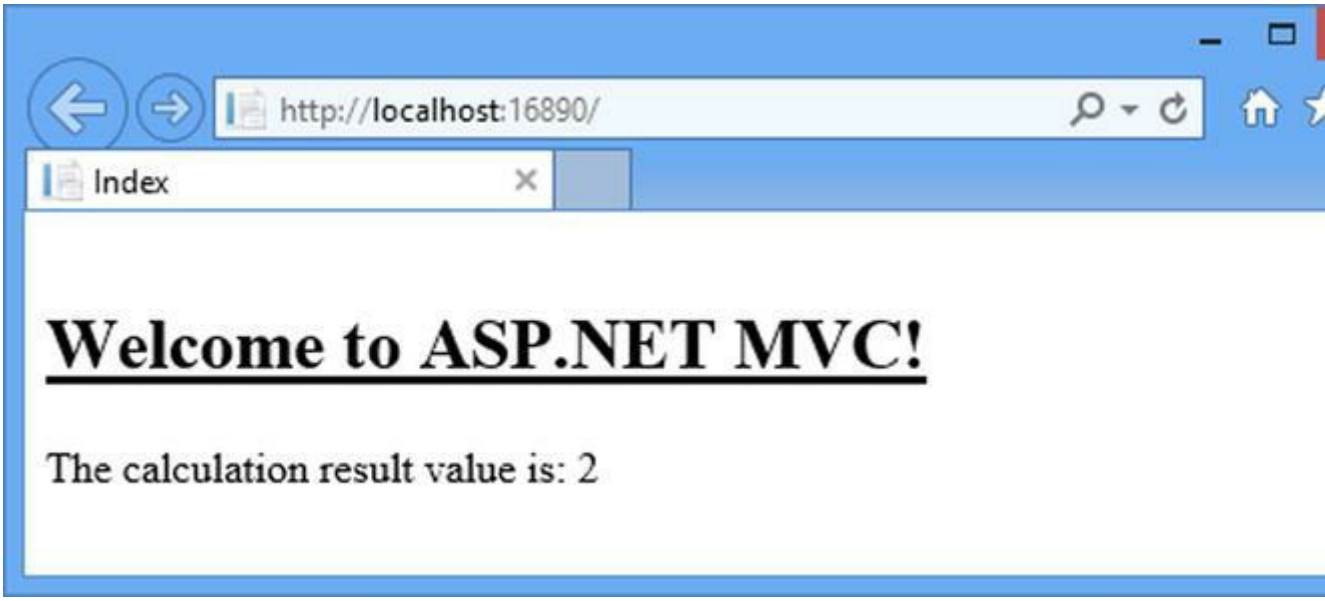


Figure 14-7. Running the debugger

The debugger will be attached to your application, but you will not notice any difference until the debugger breaks
what this means in the next section.) To stop the debugger, select `Stop Debugging` from the Visual Studio D
or close the browser window.

Causing the Visual Studio Debugger to Break

An application that is running with the debugger attached will behave normally until a *break* occurs, at which point th

eBooks-IT.org

the application. Breaks occur for two main reasons: when a breakpoint is reached and when an unhandled exception control will see examples of both in the following sections.

Using Breakpoints

A *breakpoint* is an instruction that tells the debugger to halt execution of the application and hand control to the prog can inspect the state of the application and see what is happening and, optionally, resume execution again.

To create a breakpoint, right-click a code statement and select `Breakpoint` ➤ `Insert Breakpoint` pop-up menu. As a demonstration, apply a breakpoint to the first statement in the `Index` action method of the `Ho` and you will see a red dot appear in the margin of the text editor, as shown in Figure 14-8.

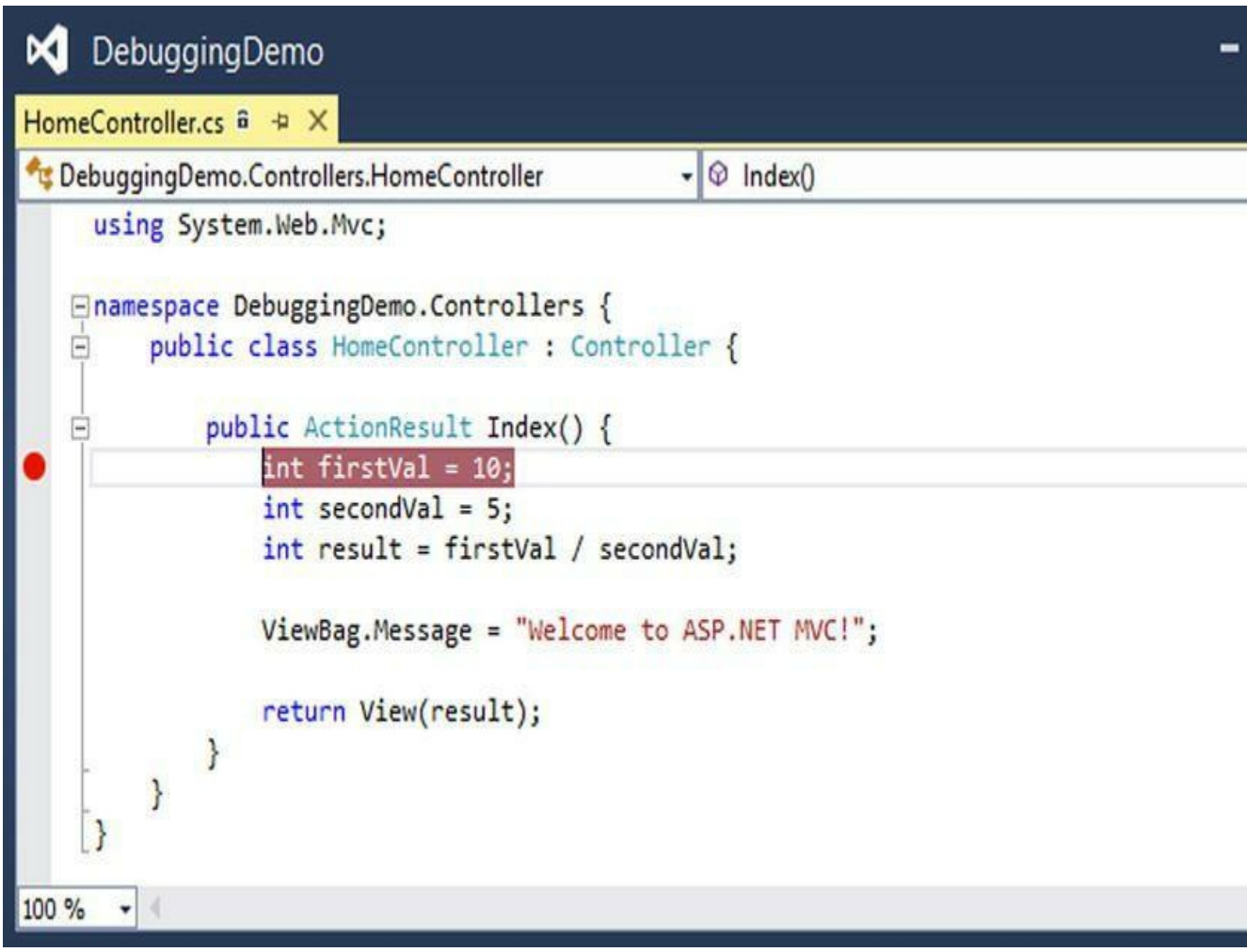


Figure 14-8. Applying a breakpoint to the first statement in the Index action method

To see the effect of the breakpoint, start the debugger by selecting `Start Debugging` from the Visual Stu menu. The application will run until the statement to which the breakpoint has been applied is reached, at which poin will break, halting execution of the application and transferring control. Visual Studio highlights the point at which th has been stopped with yellow highlights, as shown in Figure 14-9.

eBooks-IT.org

HomeController.cs

DebuggingDemo.Controllers.HomeController | Index()

```
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
            int secondVal = 5;
            int result = firstVal / secondVal;

            ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
    }
}
```

100 %

Figure 14-9. Hitting a breakpoint

Note A breakpoint is triggered only when the statement it is associated with is executed. My example breakpoint as soon as the application was started because it is inside the action method that is called when a request for the defa received. If you place a breakpoint inside another action method, you must use the browser to request a URL associa method. This can mean working with the application in the way a user would or navigating directly to the URL in the window.

Once you have control of the application's execution, you can move to the next statement, follow execution into o and generally explore the state of your application. You can do this using the toolbar buttons or using the items in the `Debug` menu. In addition to giving you control of the execution of the app, Visual Studio provides you with a lot of information about the state of your app. In fact, there is so much information that I only have room to show you the

Viewing Data Values in the Code Editor

The most common use for is to track down bugs in your code. Before you can fix a bug, you have to figure out wha and one of the most useful features that Visual Studio provides is the ability to view and monitor the values of variabl code editor.

As an example, start the app using the debugger and wait until the breakpoint I added in the previous section is rea the debugger breaks, move the mouse pointer over the statement that defines the `result` variable. You will see a s which shows you the current value, as illustrated by Figure 14-10. It can be hard to make out the pop-up, so I have magnified version in the figure.

eBooks-IT.org

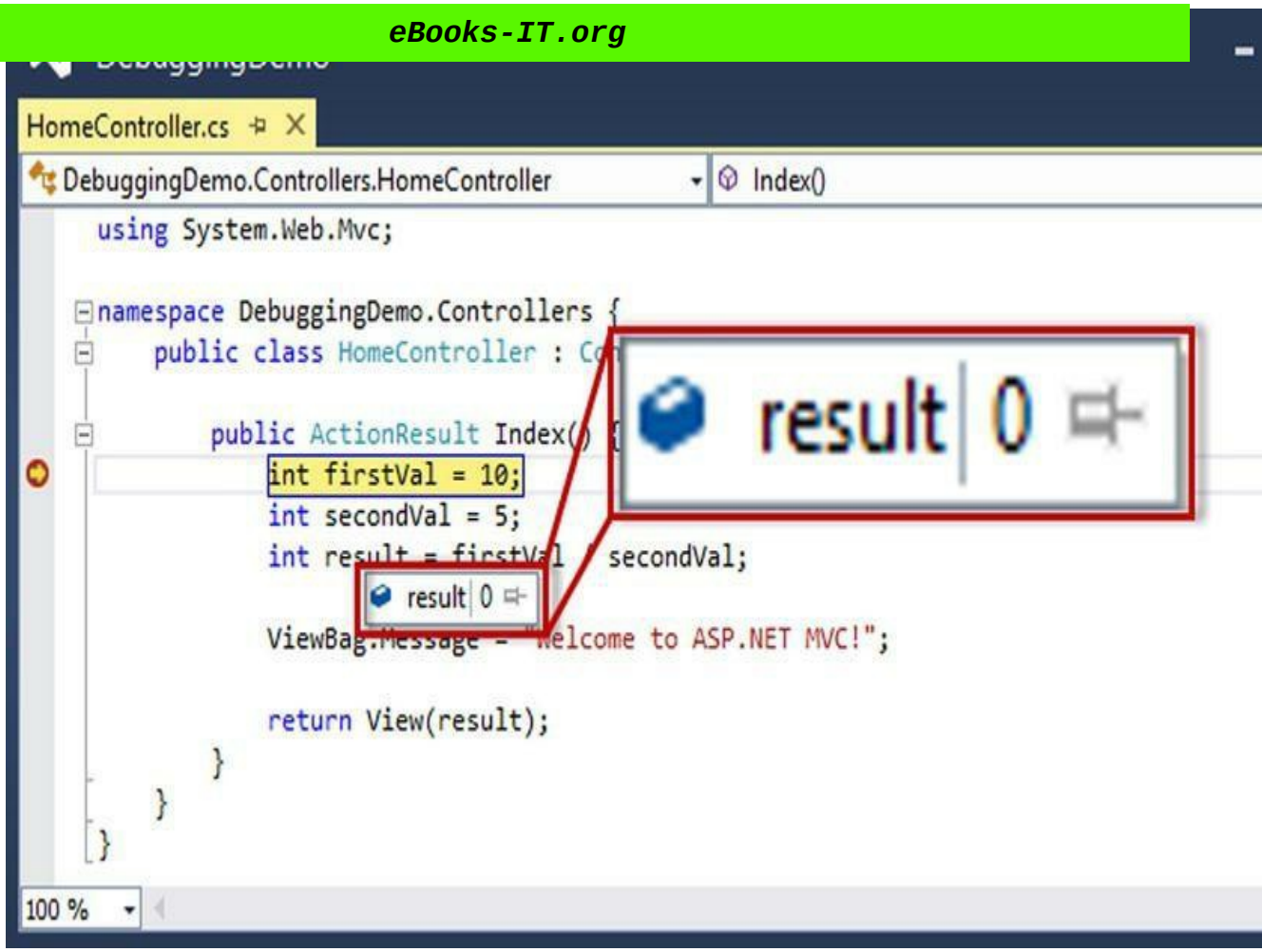


Figure 14-10. Displaying the value of a variable in the Visual Studio code editor

The execution of the statements in the `Index` action method has not reached the point where a value has been a `result` variable, so Visual Studio shows the default value, which is `0` for the `int` type. Select the `Step Ove` in the Visual Studio `Debug` menu (or press `F10`) to advance the point of execution to the statement which defines `ViewBag.Message` property and hold your mouse over the `result` variable again. The debugger executed that assigns a value to the `result` variable, and you can see the effect in Figure 14-11.

eBooks-IT.org

```
HomeController.cs
DebuggingDemo.Controllers.HomeController        Index()
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
            int secondVal = 5;
            int result = firstVal / secon
                result 2
            ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
    }
}
100 %
```

Figure 14-11. The effect of assigning a value of a variable

I use this feature when I start the process of tracking down a bug because it gives an immediate insight into what inside the app. I find it especially useful for spotting `null` values, which indicate that a variable has not been assign cause of many bugs early in the development process, in my experience).

You will notice that there is a pushpin icon to the right of the value in the pop-up. If you click on this, the pop-up permanent and will indicate when the value of the variable changes. This allows you to monitor one or more variables when they change and what their new values are.

Viewing Application State in the Debugger

Visual Studio provides a number of different windows that you can use to get information about your app while the been halted following a breakpoint. A complete list of the windows available is shown on the `Debug` ➤ `Window` two of the most useful are the `Locals` and `Call Stack` windows. The `Locals` window automatically dis value of all of the variables in the current scope, as Figure 14-12 illustrates. This gives you an all-in-one view of the which are likely to be of interest.

Locals

Name	Value	Type
this	{DebuggingDemo.Controllers.HomeController}	DebuggingDemo.Controllers.HomeContro
firstVal	10	int
secondVal	5	int
result	2	int

eBooks-IT.org

Variables whose values were changed by the previously executed statement are shown in red. In the figure, the `re` variable is red because I just executed the statement that assigns a value.

Tip The set of variables shown in the `Locals` window changes as you navigate through the application, but if keep an eye on a variable globally, then right-click on one of the items shown in the `Locals` window and select th `Watch` option. The items in the `Watch` window don't change as you execute statements in the app, providing yo point of reference.

The `Call Stack` window shows you the sequence of calls that have led to the current statement being execu be helpful if you are trying to figure out odd behavior because you can unwind the call stack and explore the circums led to the breakpoint being triggered. (I have not shown you the `Call Stack` window in a figure because the ex doesn't have enough call depth to provide a useful insight. But I recommend you explore this and the other Visual Stu to get more of an idea of what information the debugger is able to provide.)

Tip You can add breakpoints to views. This can be helpful for inspecting the values of view model properties, fo You add a breakpoint to a view just as I did in the code file: right-click the Razor statement that you are interested in `Breakpoint` ➤ `Insert Breakpoint`.

Breaking on Exceptions

Unhandled exceptions are a fact of development. One of the reasons that I unit and integration test my projects is to likelihood that such an exception will occur in production. As an aid to finding and fixing unhandled exceptions, the V debugger will break automatically when it encounters one.

Note Only *unhandled* exceptions cause the debugger to break. An exception becomes *handled* if you catch and h `try...catch` block. Handled exceptions can be a useful programming tool. They are used to represent the scen method was unable to complete its task and needs to notify its caller. Unhandled exceptions are bad, because they rep *unforeseen* condition and because they generally drop the user into an error page.

To demonstrate breaking on an exception, I have made a small change to the `Index` action method in the `Home` as shown in Listing 14-5.

Listing 14-5. Adding a Statement That Will Cause an Exception in the HomeController.cs File

```
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
            int secondVal = 0;
            int result = firstVal / secondVal;

            ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
    }
}
```

I changed the value of the `secondVal` variable to be `0`, which will cause an exception in the statement that div `firstVal` by `secondVal`.

Note I also removed the breakpoint from the Index action method by right-clicking on the breakpoint icon in the

When you start the debugger, the application will run until the exception is thrown, at which point the exception he will appear, as shown in Figure 14-13.

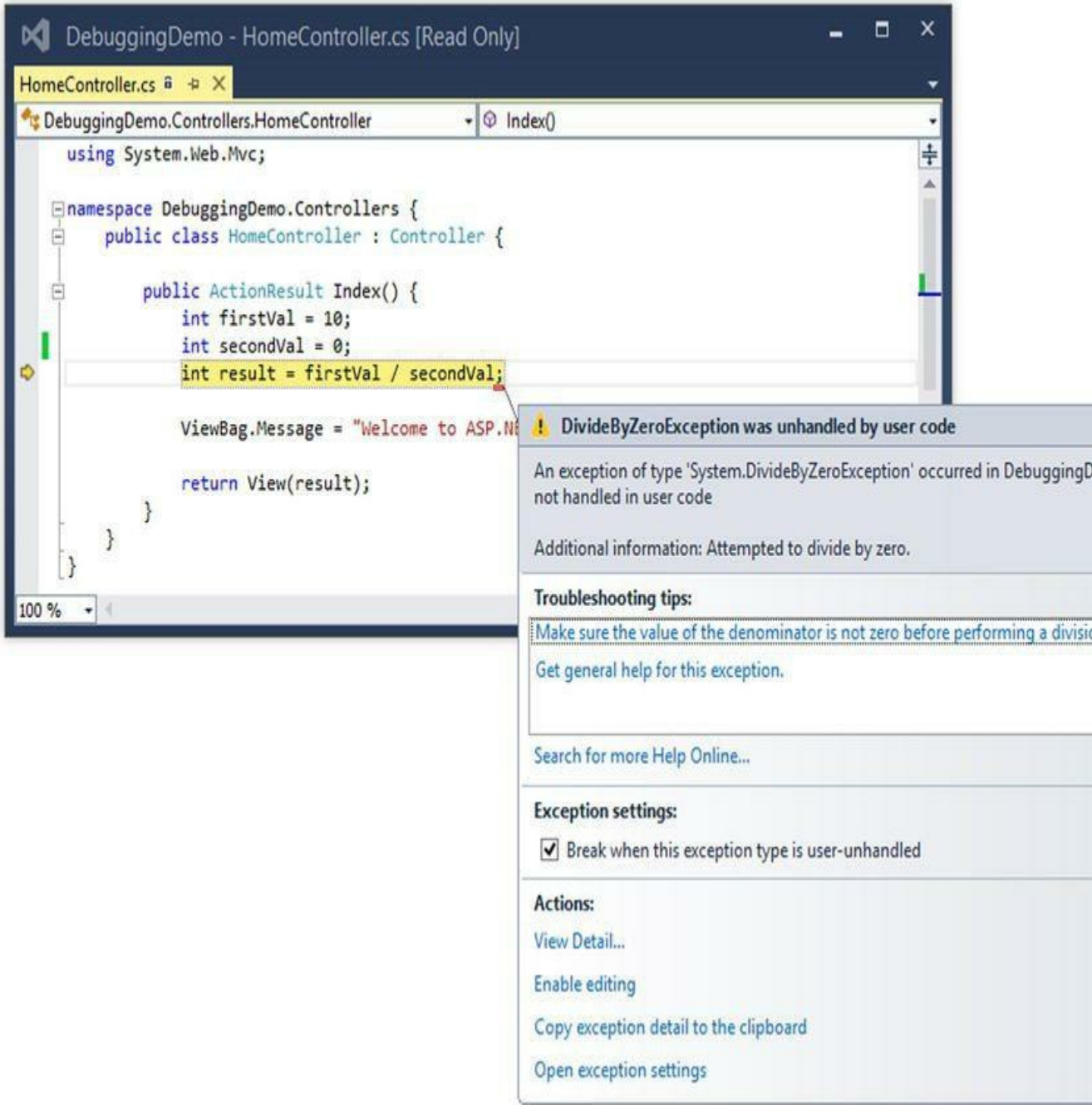


Figure 14-13. The exception helper

The exception helper gives you details of the exception. When the debugger breaks on an exception, you can inspe application state and control execution, just as when a breakpoint is hit.

Using Edit and Continue

An interesting Visual Studio debugging feature is called *Edit and Continue*. When the debugger breaks, you can edit y then continue debugging. Visual Studio recompiles your application and re-creates the state of your application at the the debugger break.

Enabling Edit and Continue

- In the `Edit and Continue` section of the `Debugging` options (select `Options` from the `Tools` menu), make sure that `Enable Edit and Continue` is checked, as shown in Figure 14

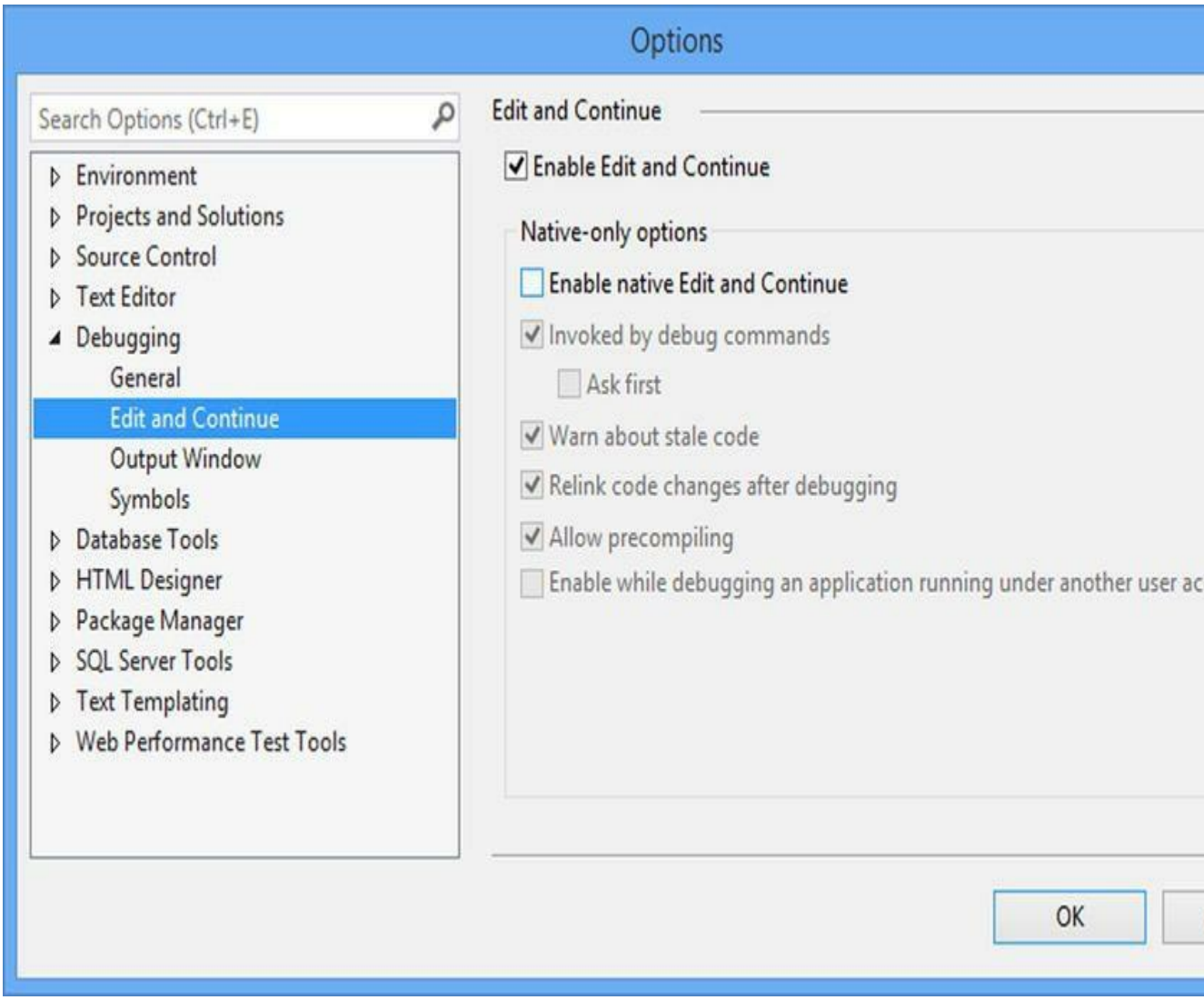


Figure 14-14. Enabling Edit and Continue in the Options dialog box

- In the project properties (select `DebuggingDemo Properties` from the Visual Studio `Project` the `Web` section and ensure that `Enable Edit and Continue` is checked, as shown in Figure 1

eBooks-IT.org

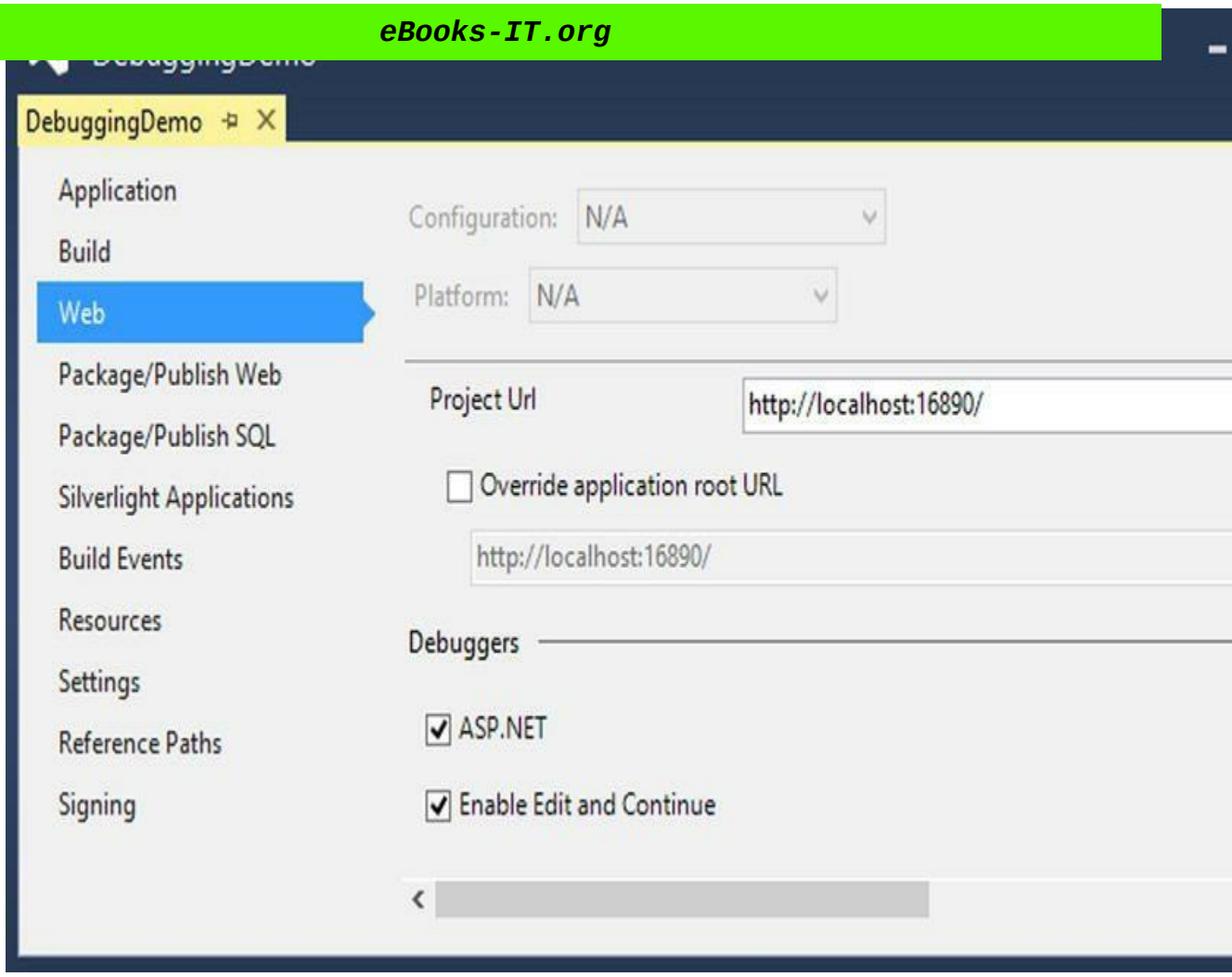


Figure 14-15. Enabling Edit and Continue in the project properties

Modifying the Project

The Edit and Continue feature is somewhat picky. There are some conditions under which it cannot work. One such
present in the `Index` action method of the `HomeController` class: the use of dynamic objects. To work ar
have commented out the line that uses the view bag in the `HomeController.cs` class, as shown in Listing 1

Listing 14-6. Removing the ViewBag Call from the Index Method in the HomeController.cs File

```
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
            int secondVal = 0;
            int result = firstVal / secondVal;

            // This statement has been commented out
            //ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
```

eBooks-IT.org

}

I need to make a corresponding change in the `Index.cshtml` view, as shown in Listing 14-7.

Listing 14-7. Removing the ViewBag Call from the Index.cshtml File

```
@model int

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Site.css" rel="stylesheet" type="text/cs
    <title>Index</title>
</head>
<body>
    <!-- This element has been commented out -->
    <!-- <h2 class="message">@ViewData["Message"]</h2> -->
    <p>
        The calculation result value is: @Model
    </p>
</body>
</html>
```

Editing and Continuing

I am now ready for a demonstration of the Edit and Continue feature. Begin by selecting `Start Debugging` f Visual Studio `Debug` menu. The application will be started with the debugger attached and run until it reaches the li perform the calculation in the `Index` method. The value of the second parameter is zero, which causes an exceptio thrown. At this point, the debugger halts execution, and the exception helper pops up (just like the one shown in Figu

Click the `Enable editing` link in the exception helper window. In the code editor, change the expression t the value for the `result` variable, as follows:

```
...
int result = firstVal / 2 ;
...
```

I have removed the reference to the `secondVal` variable and replaced it with a numeric literal value of `2`. Now `Continue` from the Visual Studio `Debug` menu to resume execution of the application. The new value is used t value for the `result` variable, producing the output shown in Figure 14-16.

eBooks-IT.org

Figure 14-16. The effect of correcting a bug using the Edit and Continue feature

Take a moment to reflect on what happened here. I started the application with a bug in it: an attempt to divide a v
The debugger detected the exception and stopped executing the program. I edited the code and then told the debugge
the execution. At this point, Visual Studio recompiled the application, restored its state and continued execution as no
new value. The browser received the rendered result, which reflected the new data value. Without Edit and Continue
have needed to stop the application, make a change, compile the application, and restart the debugger. I would then u
browser to repeat the steps that I took up to the moment of the debugger break. It is avoiding this last step that can b
important. Complex bugs may require many steps through the application to re-create, and the ability to test potential
needing to repeat those steps over and over can save the programmer's time and sanity.

Using Browser Link

Visual Studio 2013 includes a feature called *browser link* that allows you to view the application in multiple browsers
simultaneously and have them all reload when you make a change. This feature is most useful once an application has
and you are doing fit and finish work on the HTML and CSS that your views generate. (I'll explain why shortly.)

To use browser link, click on the small down arrow next to the selected browser on the Visual Studio toolbar and
`Browse With` from the menu, as shown in Figure 14-17.

eBooks-IT.org

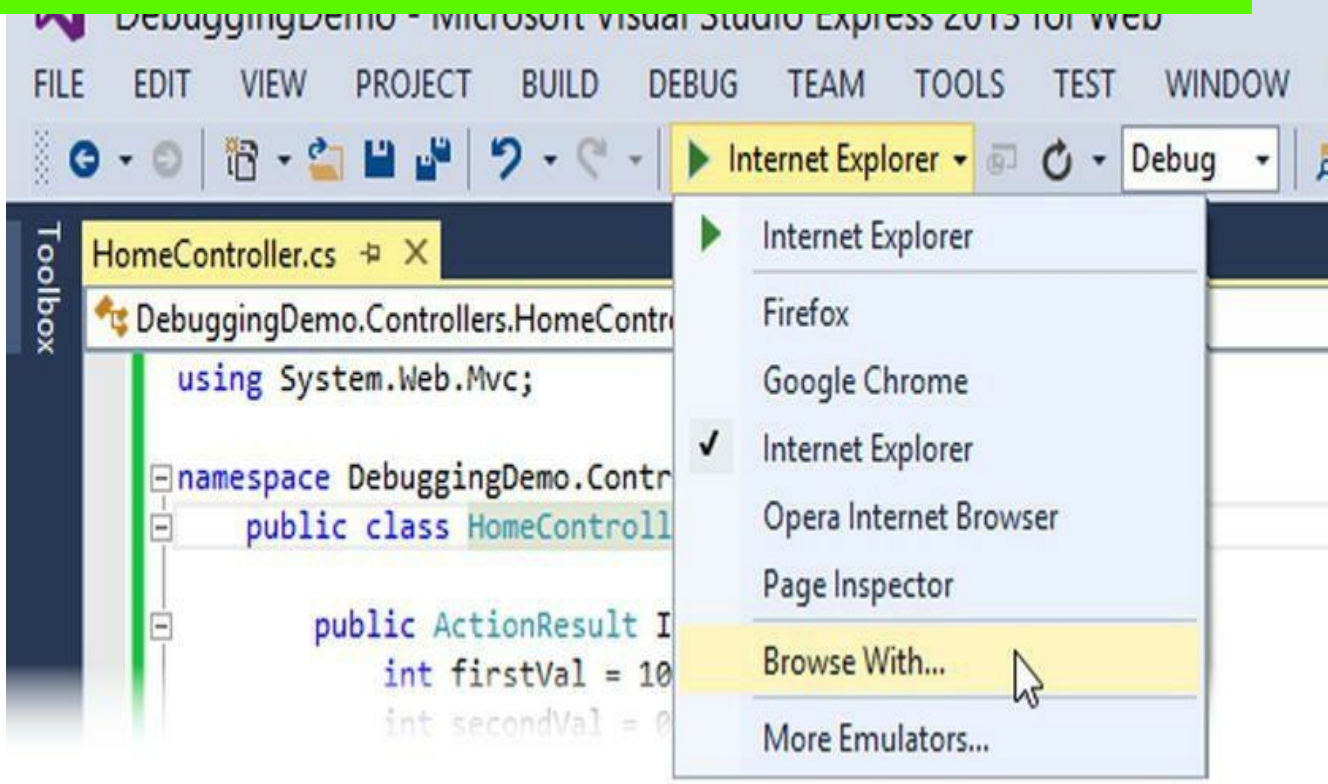


Figure 14-17. Preparing to select the browsers used for the browser link feature

The `Browse With` dialog window will appear. Hold the control key and select the browsers that you want to
Figure 14-18, you can see that I have chosen Internet Explorer and Chrome. You can also use this dialog to add new
(although Visual Studio is pretty good at detecting most of the mainstream ones).

eBooks-IT.org

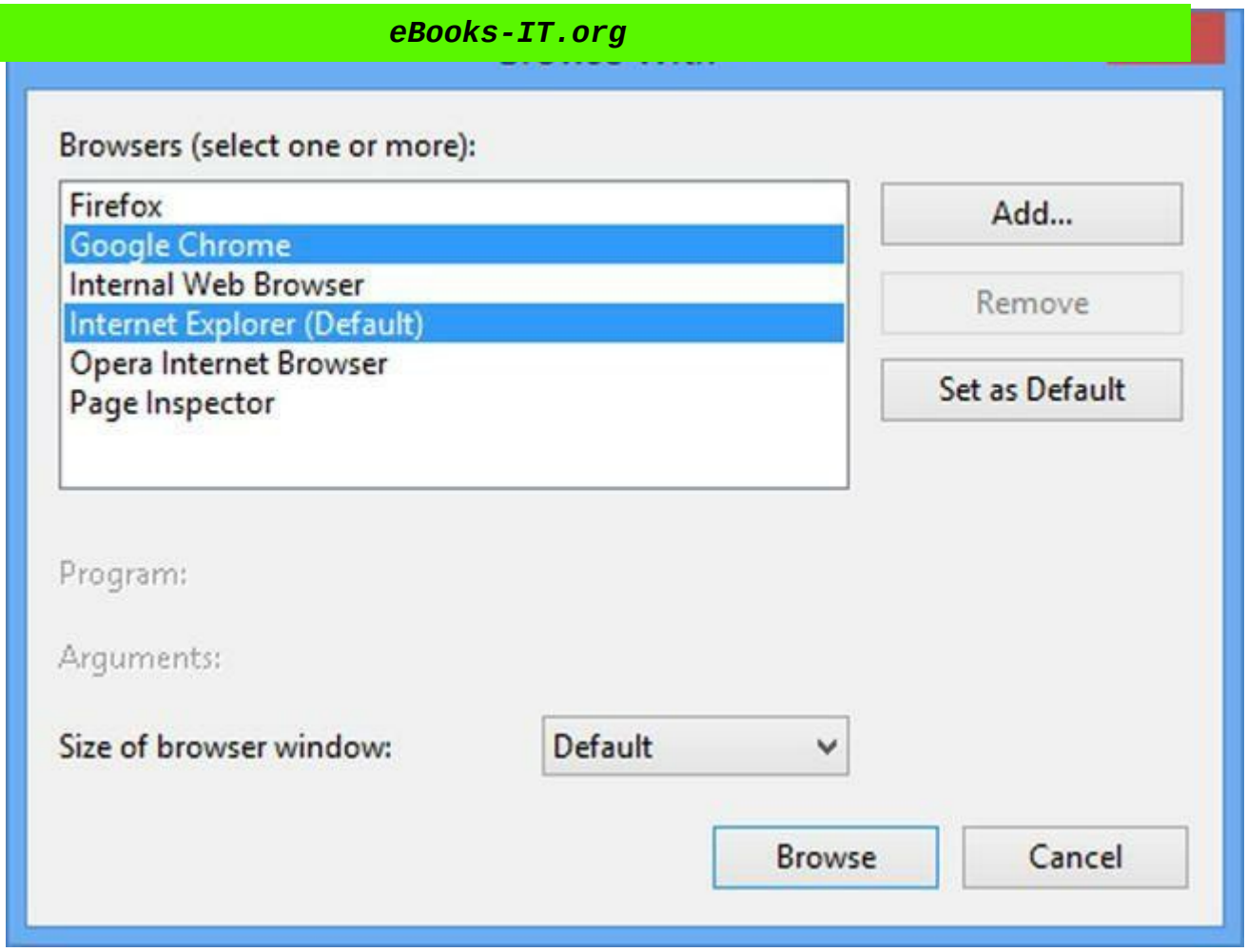


Figure 14-18. Selecting multiple browsers

Click the `Browse` button and Visual Studio will open the browsers you have selected and have each of them loa URL. You can edit the code and views in the application and then update all of the browser windows by selecting Re Browsers from the Visual Studio toolbar, as shown in Figure 14-19. The application will be compiled automatically s see changes.

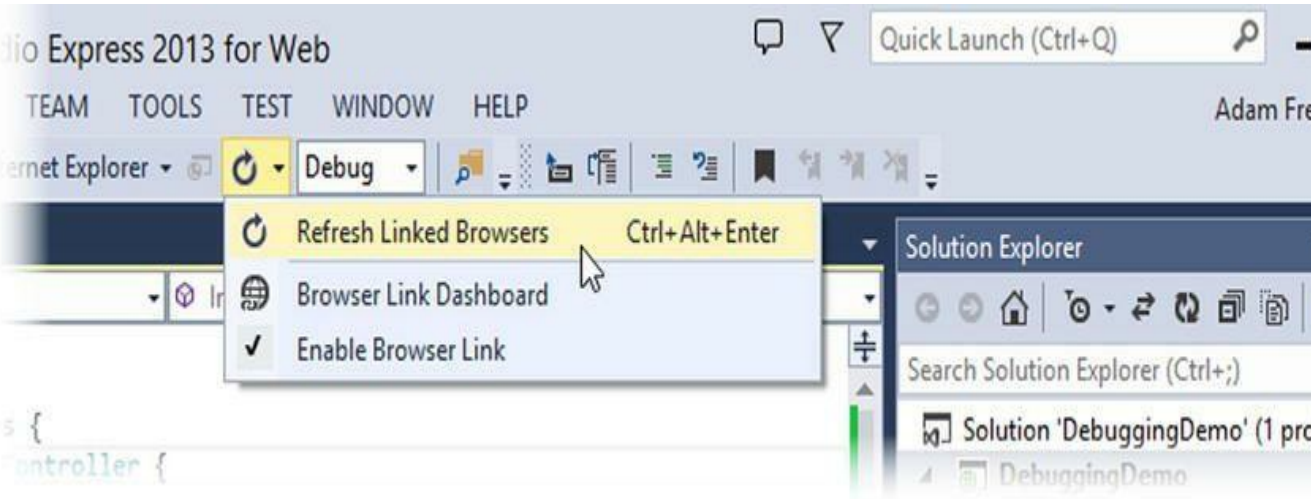


Figure 14-19. Refreshing linked browsers

This feature works by including some JavaScript in the HTML sent to the browser and it can be a nice way of dev iteratively. The reason that I recommend it only for working on views is that they are less likely to cause IIS to send messages to the browser, which is what happens when there is an error in the code. The JavaScript code isn't added responses, which means that the link between Visual Studio and the browsers is lost. You have to start over using th `With` menu. The browser link feature is a good idea, but the use of JavaScript is a problem. I use a similar tool call

eBooks-IT.org

browser plugins that are not affected by HTTP error messages. The value of Visual Studio browser link will be limite because Microsoft takes a similar approach.

Summary

In this chapter, I have shown you the structure of a Visual Studio MVC project and how the various parts fit togethe touched on one of the most important characteristics of the MVC Framework: convention. These are topics that I w again and again in the chapters that follow, as I dig deeper into how the MVC Framework operates.

eBooks-IT.org

URL Routing

Before the introduction of the MVC Framework, ASP.NET assumed that there was a direct relationship between req
and the files on the server hard disk. The job of the server was to receive the request from the browser and deliver t
from the corresponding file.

This approach works just fine for Web Forms, where each ASPX page is both a file and a self-contained response
It *doesn't* make sense for an MVC application, where requests are processed by action methods in controller classes
no one-to-one correlation to the files on the disk.

To handle MVC URLs, the ASP.NET platform uses the *routing system*. In this chapter, I will show you how to us
system to create powerful and flexible URL handling for your projects. As you will see, the routing system lets you c
pattern of URLs you desire and express them in a clear and concise manner. The routing system has two functions:

- Examine an *incoming URL* and figure out for which controller and action the request is intended.
- Generate *outgoing URLs*. These are the URLs that appear in the HTML rendered from views so that a speci
be invoked when the user clicks the link (at which point, it has become an incoming URL again).

In this chapter, I will focus on defining routes and using them to process incoming URLs so that the user can reac
controllers and actions. There are two ways to create routes in an MVC Framework application: *convention-based ro*
attribute routing. You will be familiar with convention-based routing if you have used earlier versions of the MVC Fr
attribute routing is new to MVC 5. I explain both approaches in this chapter.

Then, in the next chapter, I will show you how to use those same routes to generate the outgoing URLs you will n
in your views, as well as show you how to customize the routing system and use a related feature called *areas*. Table
provides the summary for this chapter.

***Table 15-1**. Chapter Summary*

Problem	Solution	L
Map between URLs and action methods	Define a route	1
Allow URL segments to be omitted	Define default values for segment variables	9
Match URL segments that don't have corresponding routing variables	Use static segments	1
Pass URL segments to action methods	Define custom segment variables	1
Allow URL segments for which there is no default value to be omitted	Define optional segments	1
Define routes that match any number of URL segments	Use a catchall segment	2
Avoid controller name confusion	Specify priority namespaces in a route	2
Limit the URLs that a route can match	Apply a route constraint	2
Enable attribute routing	Call the `MapMvcAttributeRoutes` method	3
Define a route within a controller	Apply the `Route` attribute to the action methods	3
Constrain an attribute route	Apply a constraint to the segment variable in the route pattern	3
Define a common prefix for all of the attribute routes in a controller	Apply the `RoutePrefix` attribute to the controller class	4

Preparing the Example Project

eBooks-IT.org

Empty template, and I called the project UrlsAndRoutes. I added a test project to the Visual Studio solution UrlsAndRoutes.Tests by checking the Add Unit Tests option, as shown in Figure 15-1.

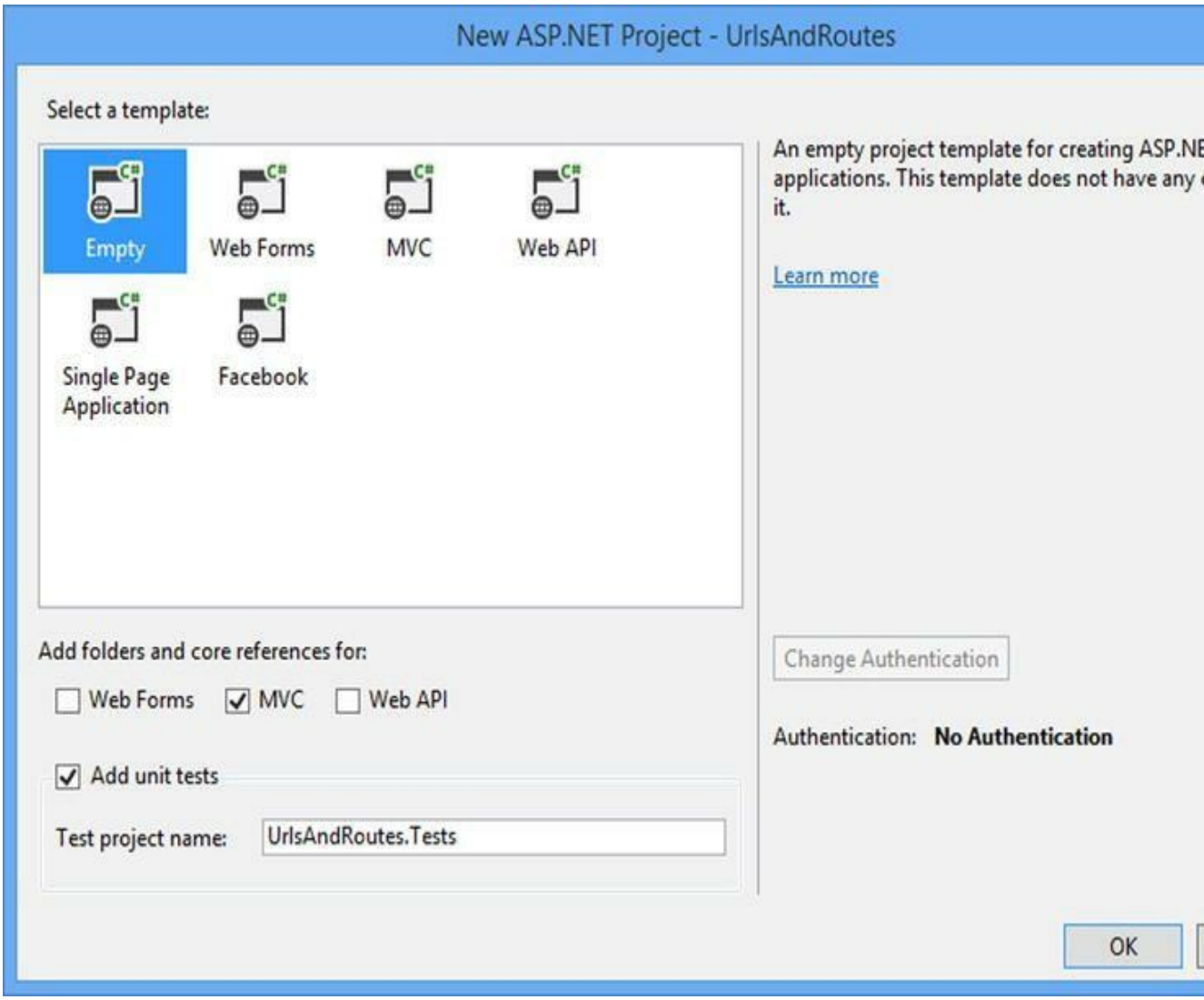


Figure 15-1. Creating an Empty MVC application project with unit tests

I showed you how to create the unit tests manually for the SportsStore chapter, but this produces the same result the references between projects automatically. You will still need to add Moq, however, and so enter the following c the NuGet console:

```
Install-Package Moq -version 4.1.1309.1617 -pro
UrlsAndRoutes.Tests
```

Creating the Example Controllers

To demonstrate the routing feature, I am going to add some simple controllers to the example application. I only care way in which URLs are interpreted in order to call action methods, so the view models I use are string values in the which report the controller and action method name. First, create a Home controller and set its contents to match th 15-1.

Listing 15-1. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
```

```
        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

Create a `Customer` controller and set its contents to match Listing 15-2.

Listing 15-2. The Contents of the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

Create an `Admin` controller and edit its contents to match the code shown in Listing 15-3.

Listing 15-3. The Contents of the AdminController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class AdminController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Admin";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

Creating the View

I specified the `ActionName` view in all of the action methods in these controllers, which allows me to define one use it throughout the example application. Create a folder called `Shared` in the `Views` folder and add a new view `ActionName.cshtml` to it, setting the contents of the view to match Listing 15-4.

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
</body>
</html>
```

Setting the Start URL and Testing the Application

As I explained in Part 1 of this book, Visual Studio will try to figure out the URL you want the browser to request ba you are editing when you start the debugger. This is a good idea that quickly becomes annoying and is a feature that disable. Select `UrlsAndRoutes Properties` from the Visual Studio `Project` menu, switch to the `W` check the `Specific Page` option in the `Start Action` section. You don't have to provide a value—jus option is enough. If you start the example app, you will see the response shown in Figure 15-2.

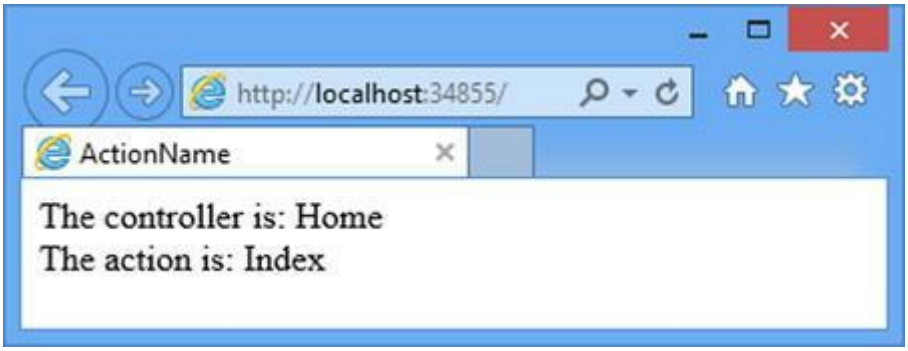


Figure 15-2. Running the example app

Introducing URL Patterns

The routing system works its magic using a set of *routes*. These routes collectively comprise the URL *schema* or *sch* application, which is the set of URLs that your application will recognize and respond to.

I do not need to manually type out all of the individual URLs I am willing to support in my applications. Instead, ea contains a *URL pattern*, which is compared to incoming URLs. If a URL matches the pattern, then it is used by the r to process that URL. Let's start with a URL for the example application:

```
http://mysite.com/Admin/Index
```

URLs can be broken down into *segments*. These are the parts of the URL, excluding the hostname and query string separated by the / character. In the example URL, there are two segments, as shown in Figure 15-3.

eBooks-IT.org

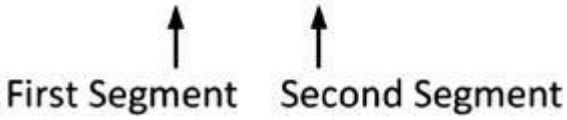


Figure 15-3. The segments in an example URL

The first segment contains the word `Admin`, and the second segment contains the word `Index`. To the humar obvious that the first segment relates to the controller and the second segment relates to the action. But, of course, I express this relationship in a way that the routing system can understand. Here is a URL pattern that does this:

```
{controller}/{action}
```

When processing an incoming request, the job of the routing system is to match the URL that has been requested and extract values from the URL for the *segment variables* defined in the pattern. The segment variables are expresse braces (the `{` and `}` characters). The example pattern has two segment variables with the names `controller` a `action`, and so the value of the `controller` segment variable will be `Admin` and the value of the `actio` variable will be `Index`.

I say match to *a* pattern, because an MVC application will usually have several routes and the routing system will incoming URL to the URL pattern of each route until it finds a match.

Note The routing system does not have any special knowledge of controllers and actions. It just extracts values segment variables. It is later in the request handling process, when the request reaches the MVC Framework proper, is assigned to the `controller` and `action` variables. This is why the routing system can be used with Web the Web API. (I introduce the Web API in Chapter 27 and I describe the ASP.NET request handling process in detail *ASP.NET MVC 5 Platform* book.)

By default, a URL pattern will match any URL that has the correct number of segments. For example, the pattern `{controller}/{action}` will match any URL that has two segments, as illustrated by Table 15-2.

***Table 15-2**. Matching URLs*

Request URL	Segment Variables
http://mysite.com/**Admin/Index**	`controller` = `Admin` `action` = `Index`
http://mysite.com/**Index/Admin**	`controller` = `Index` `action` = `Admin`
http://mysite.com/**Apples/Oranges**	`controller` = `Apples` `action` = `Oranges`
http://mysite.com/**Admin**	No match—too few segments
http://mysite.com/**Admin/Index/Soccer**	No match—too many segments

Table 15-2 highlights two key behaviors of URL patterns:

- URL patterns are *conservative*, and will match only URLs that have the same number of segments as the pat see this in the fourth and fifth examples in the table.
- URL patterns are *liberal*. If a URL *does* have the correct number of segments, the pattern will extract the segment variable, whatever it might be.

These are the default behaviors, which are the keys to understanding how URL patterns function. I show you how the defaults later in this chapter.

As already mentioned, the routing system does not know anything about an MVC application, and so URL patterns even when there is no controller or action that corresponds to the values extracted from a URL. You can see this der the second example in Table 15-2. I transposed the `Admin` and `Index` segments in the URL, and so the values ex the URL have also been transposed, even though there is no `Index` controller in the example project.

eBooks-IT.org

Once you have a URL pattern in mind, you can use it to define a route. Routes are defined in the RouteConfig which is in the App_Start project folder. You can see the initial content that Visual Studio defines for this file in

Listing 15-5. The Default Contents of the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Ind
                    id = UrlParameter.Optional }
            );
        }
    }
}
```

The static RegisterRoutes method that is defined in the RouteConfig.cs file is called from the Global.asax.cs file, which sets up some of the core MVC features when the application is started. You can default contents of the Global.asax.cs file in Listing 15-6, and I have highlighted the call to the RouteConfig.RegisterRoutes method, which is made from the Application_Start metho

Listing 15-6. The Default Contents of the Global.asax.cs File

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

namespace UrlsAndRoutes {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

The Application_Start method is called by the underlying ASP.NET platform when the MVC applicati

eBooks-IT.org

value of the static RouteTable.Routes property, which is an instance of the RouteCollection cla to this describe shortly.

Tip The other call made in the Application_Start method sets up a related feature called *areas*, which the next chapter.

Listing 15-7 shows how to create a route using the example URL pattern from the previous section in the RegisterRoutes method of the RouteConfig.cs file. (I have removed the other statements in the me focus on the example.)

Listing 15-7. Registering a Route in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
                Route myRoute = new Route("{controller}/{action
MvcRouteHandler());
            routes.Add("MyRoute", myRoute);
        }
    }
}
```

I created a new Route using a URL pattern as a constructor parameter, which I express as a string. I also pass MvcRouteHandler to the constructor. Different ASP.NET technologies provide different classes to tailor the behavior, and this is the class used for ASP.NET MVC applications. Once I have created the route, I add it to the RouteCollection object using the Add method, passing in the name I want the route to be known by and t itself.

Tip Naming your routes is optional, and there is a philosophical argument that doing so sacrifices some of the cle of concerns that otherwise comes from routing. I am relaxed about naming, but I explain why this can be a problem "Generating a URL from a Specific Route" section in Chapter 16.

A more convenient way of registering routes is to use the MapRoute method defined by the RouteCollec class. Listing 15-8 shows how I can use this method to register a route, which has the same effect as the previous e has a cleaner syntax.

Listing 15-8. Registering a Route Using the MapRoute Method in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
```

eBooks-IT.org

```
            }
        }
    }
```

This approach is more compact, mainly because I do not need to create an instance of the `MvcRouteHandl` done for me, behind the scenes). The `MapRoute` method is solely for use with MVC applications. ASP.NET Web applications can use the `MapPageRoute` method, also defined in the `RouteCollection` class.

Using the Simple Route

You can see the effect of the changes I made to the routing by starting the example application. You will see an error browser tries to navigate to the root URL for the application, but if you navigate to a route that matches the `{controller}/{action}` pattern, you will see a result like the one shown in Figure 15-4, which illustrate navigating to `/Admin/Index`.

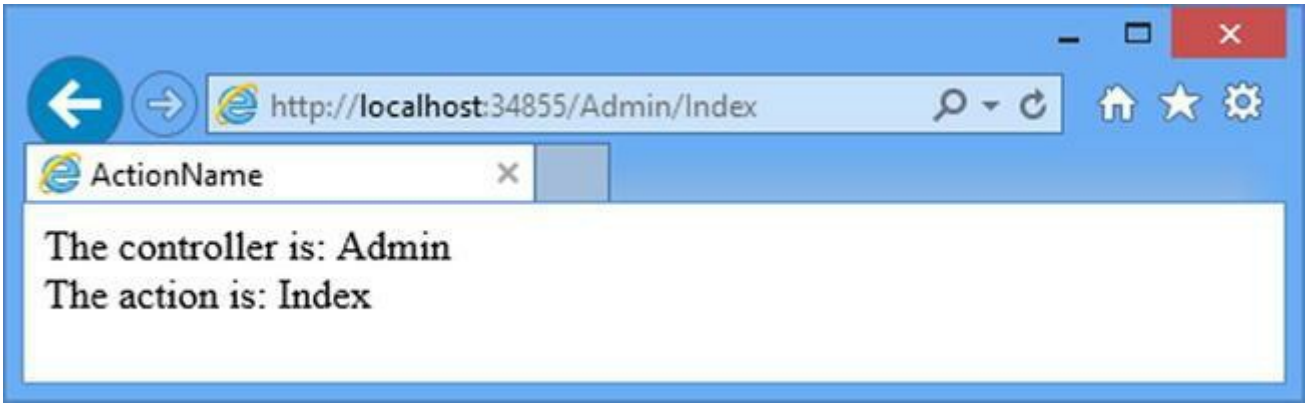


Figure 15-4. Navigating using a simple route

My simple route in Listing 15-8 does not tell the MVC Framework how to respond to requests for the root URL ar supports a single, specific, URL pattern. I have temporarily taken a step back from the functionality that Visual Studi `RouteConfig.cs` file when it creates the project, but I will show you how to build more complex patterns an throughout the rest of this chapter.

UNIT TEST: TESTING INCOMING URLS

I recommend that you unit test your routes to make sure they process incoming URLs as expected, even if you unit test the rest of your application. URL schemas can get pretty complex in large applications, and it is easy to something that has unexpected results.

In previous chapters, I avoided creating common helper methods to be shared among tests in order to keep each description self-contained. For this chapter, I am taking a different approach. Testing the routing schema for an most readily done when you can batch several tests in a single method, and this becomes much easier with som methods.

To test routes, I need to mock three classes from the MVC Framework: `HttpRequestBase`, `HttpContextBase`, and `HttpResponseBase`. (This last class is required for testing outgoing U cover in the next chapter.) Together, these classes recreate enough of the MVC infrastructure to support the ro I added a new Unit Tests file called `RouteTests.cs` to the `UrlsAndRoutes.Tests` unit test pro first addition is the helper method that creates the mock `HttpContextBase` objects, as follows:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System;
using System.Reflection;
using System.Web;
using System.Web.Routing;
```

eBooks-IT.org

```
namespace UrlsAndRoutes.Tests {
  [TestClass]
  public class RouteTests {

    private HttpContextBase CreateHttpContext(string targetUrl =
                    string httpMethod = "GET") {

      // create the mock request
      Mock<HttpRequestBase> mockRequest = new Mock<HttpRequestBas
      mockRequest.Setup(m => m.AppRelativeCurrentExecutionFilePat
        .Returns(targetUrl);
      mockRequest.Setup(m => m.HttpMethod).Returns(httpMethod);

      // create the mock response
      Mock<HttpResponseBase> mockResponse = new Mock<HttpRespons
();
      mockResponse.Setup(m => m.ApplyAppPathModifier(
        It.IsAny<string>())).Returns<string>(s => s);

      // create the mock context, using the request and response
      Mock<HttpContextBase> mockContext = new Mock<HttpContextBa
      mockContext.Setup(m => m.Request).Returns(mockRequest.Objec
      mockContext.Setup(m => m.Response).Returns(mockResponse.Obj
      // return the mocked context
      return mockContext.Object;
    }
  }
}
```

The setup here is simpler than it looks. I expose the URL I want to test through the `AppRelativeCurrentExecutionFilePath` property of the `HttpRequestBase` class, the `HttpRequestBase` through the `Request` property of the mock `HttpContextBase` class. helper method lets me test a route:

```
...
private void TestRouteMatch(string url, string controller, stri
action,
  object routeProperties = null, string httpMethod = "GET") {

  // Arrange
  RouteCollection routes = new RouteCollection();
  RouteConfig.RegisterRoutes(routes);

  // Act - process the route
  RouteData result
    = routes.GetRouteData(CreateHttpContext(url, httpMethod));

  // Assert
  Assert.IsNotNull(result);
  Assert.IsTrue(TestIncomingRouteResult(result, controller,
    action, routeProperties));
}
...
```

The parameters of this method let me specify the URL to test, the expected values for the `controller` and

eBooks-IT.org

you how to create such variables later in the chapter and in the next chapter. I also defined a parameter for the I define method, which I will explain in the "Constraining Routes" section.

The `TestRouteMatch` method relies on another method, `TestIncomingRouteResult`, to co result obtained from the routing system with the segment variable values I expect. This method uses .NET refle can use an anonymous type to express any additional segment variables. Do not worry if this method doesn't m this is just to make testing more convenient; it is not a requirement for understanding MVC. Here is the `TestIncomingRouteResult` method:

```
...
private bool TestIncomingRouteResult(RouteData routeResult,
  string controller, string action, object propertySet = null) {

  Func<object, object, bool> valCompare = (v1, v2) => {
    return StringComparer.InvariantCultureIgnoreCase
      .Compare(v1, v2) == 0;
  };

  bool result = valCompare(routeResult.Values["controller"],
controller)
    && valCompare(routeResult.Values["action"], action);

  if (propertySet != null) {
    PropertyInfo[] propInfo = propertySet.GetType().GetPropertie
    foreach (PropertyInfo pi in propInfo) {
      if (!(routeResult.Values.ContainsKey(pi.Name)
          && valCompare(routeResult.Values[pi.Name],
          pi.GetValue(propertySet, null)))) {

        result = false;
        break;
      }
    }
  }
  return result;
}
...
```

I also need a method to check that a URL does not work. As you will see, this can be an important part of defin schema.

```
...
private void TestRouteFail(string url) {

  // Arrange
  RouteCollection routes = new RouteCollection();
  RouteConfig.RegisterRoutes(routes);

  // Act - process the route
  RouteData result = routes.GetRouteData(CreateHttpContext(url))

  // Assert
  Assert.IsTrue(result == null || result.Route == null);
}
...
```

eBooks-IT.org

vs an e
assertion fails. Because C# exceptions are propagated up the call stack, I can create simple test methods that tes
URLs and get the test behavior I require. Here is a test method that tests the route I defined in Listing 15-8:

```
...
[TestMethod]
public void TestIncomingRoutes() {

  // check for the URL that is hoped for
  TestRouteMatch("~/Admin/Index", "Admin", "Index");

  // check that the values are being obtained from the segments
  TestRouteMatch("~/One/Two", "One", "Two");

  // ensure that too many or too few segments fails to match
  TestRouteFail("~/Admin/Index/Segment");
  TestRouteFail("~/Admin");
}
...
```

This test uses the `TestRouteMatch` method to check the URL I am expecting and also checks a URL in
format to make sure that the `controller` and `action` values are being obtained properly using the URL
use the `TestRouteFail` method to make sure that the application won't accept URLs that have a differen
segments. When testing, I must prefix the URL with the tilde (~) character, because this is how the ASP.NET F
presents the URL to the routing system.

Notice that I didn't need to define the routes in the test methods. This is because I am loading them directly usi
`RegisterRoutes` method in the `RouteConfig` class.

Defining Default Values

The reason that I got an error when I requested the default URL for the application is that it didn't match the route I
The default URL is expressed as `~/` to the routing system and there are no segments in this string that can be match
`controller` and `action` variables defined by the simple route pattern.

I explained earlier that URL patterns are conservative, in that they will match only URLs with the specified number
I also said that this was the default behavior and one way to change this behavior is to use *default values*. A default v
applied when the URL doesn't contain a segment that can be matched to the value. Listing 15-9 provides an example
that contains a default value.

Listing 15-9. Providing a Default Value in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { action = "Index" });
        }
    }
```

Default values are supplied as properties in an anonymous type. In Listing 15-9, I provided a default value of `Ind` `action` variable. This route will match all two-segment URLs, as it did previously. For example, if the URL `http://mydomain.com/Home/Index` is requested, the route will extract `Home` as the value for the `controller` and `Index` as the value for the `action`.

Now that I have provided a default value for the `action` segment, the route will *also* match single-segment UR When processing a single-segment URL, the routing system will extract the `controller` value from the sole UI and use the default value for the `action` variable. In this way, I can request the URL `http://mydomain.` and invoke the `Index` action method on the `Home` controller.

I can go further and define URLs that do not contain any segment variables at all, relying on just the default values the action and controller. And as an example, Listing 15-10 shows how I have mapped the root URL for the applicati providing default values for both segments.

Listing 15-10. Providing Action and Controller Default Values in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}
```

By providing default values for both the `controller` and `action` variables, I have created a route that wil that have zero, one, or two segments, as shown in Table 15-3.

Table 15-3. *Matching URLs*

Number of Segments	Example	Maps To
0	mydomain.com	controller = Homeaction = Index
1	mydomain.com/**Customer**	controller = Customeraction = Index
2	mydomain.com/**Customer/List**	controller = Customeraction = List
3	mydomain.com/**Customer/List/All**	No match—too many segments

The fewer segments I receive in the incoming URL, the more I rely on the default values, up until the pointI receiv no segments and only default values are used. You can see the effect of the default values by starting the example ap time, when the browser requests the root URL for the application, the default values for the `controller` and `a` segment variables will be used, which will lead the MVC Framework to invoke the `Index` action method on the `H` controller, as shown in Figure 15-5.

eBooks-IT.org

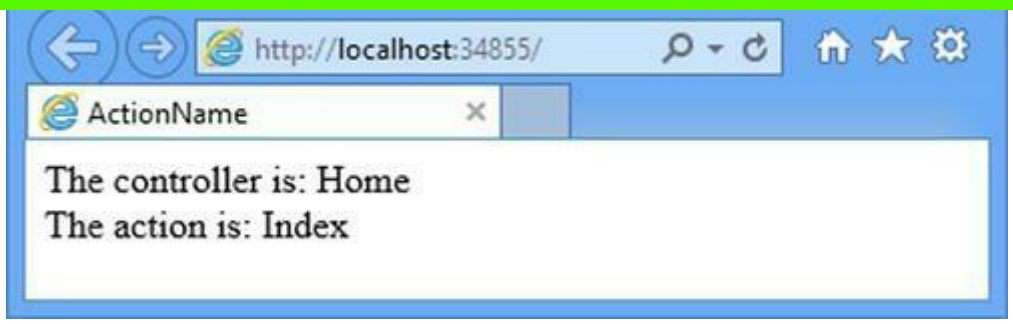


Figure 15-5. Using default values to broaden the scope of a route

UNIT TESTING: DEFAULT VALUES

I do not need to take any special actions to use the helper methods to test routes that define default values. Here
revisions I made to the `TestIncomingRoutes` test method in the `RouteTests.cs` file for the ro
Listing 15-10:

```
...
[TestMethod]
public void TestIncomingRoutes() {
  TestRouteMatch("~/", "Home", "Index");
  TestRouteMatch("~/Customer", "Customer", "Index");
  TestRouteMatch("~/Customer/List", "Customer", "List");
  TestRouteFail("~/Customer/List/All");
}
...
```

The only point of note is that I must specify the default URL as `~/`, as this is how ASP.NET presents the URL
system. If I specify the empty string (`""`) that I used to define the route or `/`, the routing system will throw ar
and the test will fail.

Using Static URL Segments

Not all of the segments in a URL pattern need to be variables. You can also create patterns that have *static segments*.
I want to match a URL like this to support URLs that are prefixed with `Public`:

http://mydomain.com/Public/Home/Index

I can do so by using a pattern like the one shown in Listing 15-11.

Listing 15-11. A URL Pattern with Static Segments in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

            routes.MapRoute("MyRoute", "{controller}/{action}",
```

```
            routes.MapRoute("", "Public/{controller}/{action}",
               new { controller = "Home", action = "Index" });
        }
    }
}
```

This new pattern will match only URLs that contain three segments, the first of which *must* be `Public`. The ot segments can contain any value, and will be used for the `controller` and `action` variables. If the last two s omitted, then the default values will be used.

I can also create URL patterns that have segments containing both static and variable elements, such as the one sh 15-12.

Listing 15-12. A URL Pattern with a Mixed Segment in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

            routes.MapRoute("", "X{controller}/{action}");

            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });

            routes.MapRoute("", "Public/{controller}/{action}",
               new { controller = "Home", action = "Index" });
        }
    }
}
```

The pattern in this route matches any two-segment URL where the first segment starts with the letter `X`. The valu `controller` is taken from the first segment, excluding the `X`. The `action` value is taken from the second se can see the effect of this route if you start the application and navigate to `/XHome/Index`, the result of which i Figure 15-6.

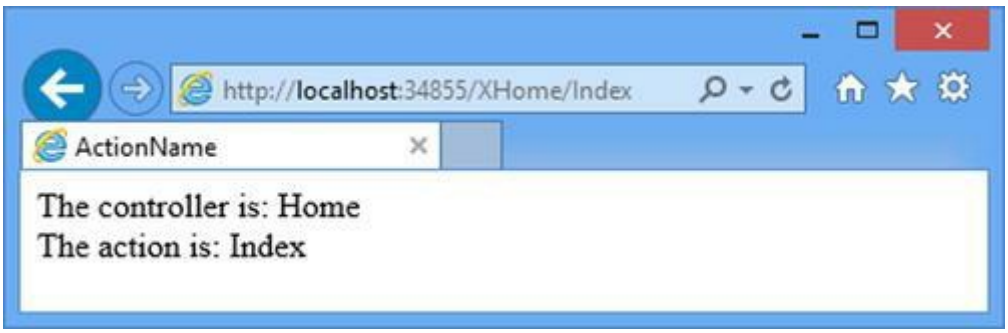


Figure 15-6. Mixing static and variable elements in a single segment

ROUTE ORDERING

eBooks-IT.org

s meth

because routes are applied in the order in which they appear in the RouteCollection object. The MapR method adds a route to the end of the collection, which means that routes are generally applied in the order in w defined. I say "generally" because there are methods that insert routes in specific locations. I tend not to use the because having routes applied in the order in which they are defined makes understanding the routing for an app simpler.

The route system tries to match an incoming URL against the URL pattern of the route that was defined first, an the next route only if there is no match. The routes are tried in sequence until a match is found or the set of rou exhausted. The result of this is that the most specific routes must be defined first. The route I added in Listing 1 specific than the route that follows. Suppose that I reversed the order of the routes, like this:

```
...
routes.MapRoute("MyRoute", "{controller}/{action}",
  new { controller = "Home", action = "Index" });
routes.MapRoute("", "X{controller}/{action}");
...
```

Then the first route, which matches *any* URL with zero, one, or two segments, will be the one that is used. The specific route, which is now second in the list, will never be reached. The new route excludes the leading X of a this won't be done by the older route. Therefore, a URL such as this:

http://mydomain.com/XHome/Index

will be targeted to a controller called XHome, which does not exist, and so will lead to a 404—Not Found sent to the user.

I can combine static URL segments and default values to create an alias for a specific URL. This can be useful if y published your URL schema publicly and it has formed a contract with your user. If you refactor an application in th you need to preserve the previous URL format so that any URL favorites, macros or scripts the user has created con work. Imagine that I used to have a controller called Shop, which has now been replaced by the Home controller. shows how I can create a route to preserve the old URL schema.

Listing 15-13. Mixing Static URL Segments and Default Values in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

            routes.MapRoute("ShopSchema", "Shop/{action}",
                new { controller = "Home" });

            routes.MapRoute("", "X{controller}/{action}");

            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });

            routes.MapRoute("", "Public/{controller}/{action}",
               new { controller = "Home", action = "Index" });
        }
```

eBooks-IT.org

}

The route I added matches any two-segment URL where the first segment is Shop. The action value is take second URL segment. The URL pattern doesn't contain a variable segment for controller, so the default valu supplied is used. This means that a request for an action on the Shop controller is translated to a request for the H controller. You can see the effect of this route by starting the app and navigating to the /Shop/Index URL. As shows, the new route causes the MVC Framework to target the Index action method in the Home controller.

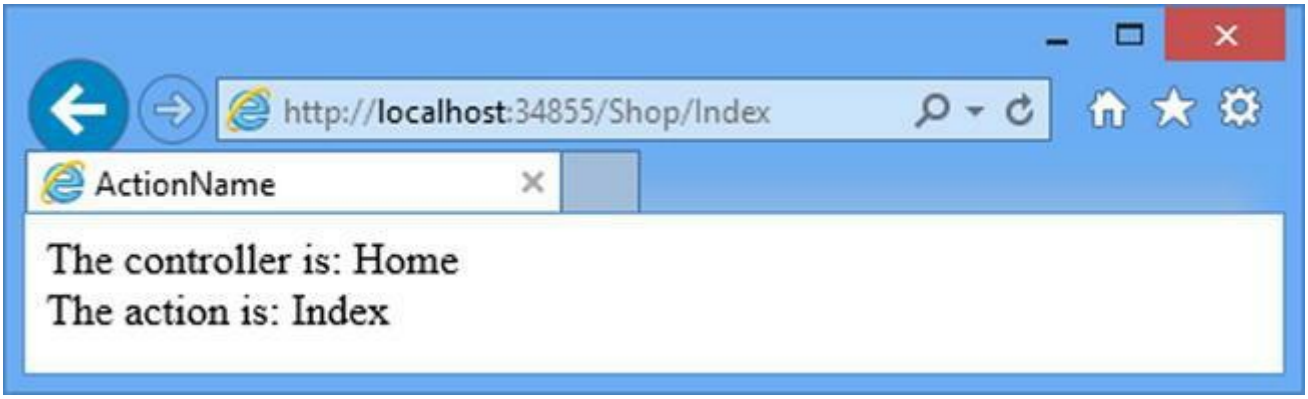


Figure 15-7. Creating an alias to preserve URL schemas

I can go one step further and create aliases for action methods that have been refactored away as well and are no in the controller. To do this, I create a static URL and provide the controller and action values as default in Listing 15-14.

Listing 15-14. Aliasing a Controller and an Action in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

            routes.MapRoute("ShopSchema2", "Shop/OldAction",
                new { controller = "Home", action = "Index" });

            routes.MapRoute("ShopSchema", "Shop/{action}",
                new { controller = "Home" });

            routes.MapRoute("", "X{controller}/{action}");

            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });

            routes.MapRoute("", "Public/{controller}/{action}",
               new { controller = "Home", action = "Index" });
        }
    }
}
```

eBooks-IT.org

cific th

that follow. If a request for `Shop/OldAction` were processed by the next defined route, for example, I would different result from the one I want. The request would be dealt with using a `404—Not Found` error, rather tha translated in order to preserve a contract with my clients.

UNIT TEST: TESTING STATIC SEGMENTS

Once again, I can use my helper methods to routes whose URL patterns contain static segments. Here is the add to the `TestIncomingRoutes` unit test method to test the route added in Listing 15-14:

```
...
[TestMethod]
public void TestIncomingRoutes() {
  TestRouteMatch("~/", "Home", "Index");
  TestRouteMatch("~/Customer", "Customer", "Index");
  TestRouteMatch("~/Customer/List", "Customer", "List");
  TestRouteFail("~/Customer/List/All");
  TestRouteMatch("~/Shop/Index", "Home", "Index");
}
...
```

Defining Custom Segment Variables

The `controller` and `action` segment variables have special meaning to the MVC Framework and, obviousl correspond to the controller and action method that will be used to service the request. But these are only the built-in variables. I can also define my own variables, as shown in Listing 15-15. (I have removed the existing routes from th section so I can start over.)

Listing 15-15. Defining Additional Variables in a URL Pattern in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
            routes.MapRoute("MyRoute", "{controller}/{action}/{id}
                new { controller = "Home", action = "Index",
                    id = "DefaultId" });
        }
    }
}
```

The route's URL pattern defines the standard `controller` and `action` variables, as well as a custom vari `id`. This route will match any zero-to-three-segment URL. The contents of the third segment will be assigned to the and if there is no third segment, the default value will be used.

Caution Some names are reserved and not available for custom segment variable names. These are `control` `action`, and `area`. The meaning of the first two is obvious, and I will explain *areas* in the next chapter.

I can access any of the segment variables in an action method by using the `RouteData.Values` property.

eBooks-IT.org

hown i

16.

Listing 15-16. Accessing a Custom Segment Variable in an Action Method in the HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {
`

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = RouteData.Values["id"];
            return View();
        }
    }
}
```

This method obtains the value of the custom variable in the route URL pattern and passes it to the view using the V
To create the view for the action method, create the `Views/Home` folder, right-click on it, select `Add` ➤ `MVC Page (Razor)` from the pop-up menu and set the name to `CustomVariable.cshtml`. Click the `OK` create the view and edit the contents to match Listing 15-17.

Listing 15-17. The Contents of the CustomVariable.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Custom Variable</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>The custom variable is: @ViewBag.CustomVariable</div>
</body>
</html>
```

To see the effect of the custom segment variable, start the application and navigate to the URL `/Home/CustomVariable/Hello`. The `CustomVariable` action method in the `Home` controller i the value of the custom segment variable is retrieved from the `ViewBag` and passed to the view. You can see the r Figure 15-8.

eBooks-IT.org

Figure 15-8. Displaying the value of a custom segment variable

I have provided a default value for the `id` segment variable in the route, which means that you will see the results Figure 15-9 if you navigate to `/Home/CustomVariable`.

Figure 15-9. The default value for a custom segment variable

UNIT TEST: TESTING CUSTOM SEGMENT VARIABLES

I included support for testing custom segment variables in the test helper methods. The `TestRouteMatch` an optional parameter that accepts an anonymous type containing the names of the properties I want to test for I expect. Here are the changes I made to the `TestIncomingRoutes` test method to test the route defin 15-15:

```
...
[TestMethod]
public void TestIncomingRoutes() {
   TestRouteMatch("~/", "Home", "Index", new { id = "DefaultId" });
   TestRouteMatch("~/Customer", "Customer", "index", new { id = "DefaultId" });
   TestRouteMatch("~/Customer/List", "Customer", "List",
      new { id = "DefaultId" });
   TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
   TestRouteFail("~/Customer/List/All/Delete");
}
...
```

Using Custom Variables as Action Method Parameters

Using the `RouteData.Values` property is only one way to access custom route variables. The other way is elegant. If I define parameters to the action method with names that match the URL pattern variables, the MVC Fram pass the values obtained from the URL as parameters to the action method. For example, the custom variable I defin

eBooks-IT.org

so tha

matching parameter, as shown in Listing 15-18.

Listing 15-18. Adding an Action Method Parameter in the HomeController.cs File

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

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable(string id) {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id;
            return View();
        }
    }
}
```

When the routing system matches a URL against the route defined in Listing 15-18, the value of the third segment assigned to the custom variable `id`. The MVC Framework compares the list of segment variables with the list of act parameters, and if the names match, passes the values from the URL to the method.

I have defined the `id` parameter as a `string`, but the MVC Framework will try to convert the URL value to w parameter type I define. If I declared the `id` parameter as an `int` or a `DateTime`, then I would receive the valu URL parsed to an instance of that type. This is an elegant and useful feature that removes the need for me to handle t myself.

Note The MVC Framework uses the *model binding* feature to convert the values contained in the URL to .NET t handle much more complex situations than shown in this example. I cover model binding in Chapter 24.

Defining Optional URL Segments

An *optional* URL segment is one that the user does not need to specify, but for which no default value is specified. I shows an example, and you can see that I specify that a segment variable is optional by setting the default value to `UrlParameter.Optional`.

Listing 15-19. Specifying an Optional URL Segment in the RouteConfig.cs File

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

eBooks-IT.org

```
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional });
        }
    }
}
```

This route will match URLs whether or not the `id` segment has been supplied. Table 15-4 shows how this works URLs.

Table 15-4. Matching URLs with an Optional Segment Variable

Segments	Example URL	Maps To
0	mydomain.com	controller = Homeaction = Index
1	mydomain.com/**Customer**	controller = Customeraction = Index
2	mydomain.com/**Customer/List**	controller = Customeraction = List
3	mydomain.com/**Customer/List/All**	controller = Customeraction = Listid = All
4	mydomain.com/**Customer/List/All/Delete**	No match—too many segments

As you can see from the table, the `id` variable is added to the set of variables only when there is a corresponding the incoming URL. This feature is useful if you need to know whether the user supplied a value for a segment variab value has been supplied for an optional segment variable, the value of the corresponding parameter will be `null`. I the controller to respond when no value is provided for the `id` segment variable in Listing 15-20.

Listing 15-20. Checking for an Optional Segment Variable in the HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable(string id) {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id ?? "<no value>";
            return View();
        }

    }
}
```

You can see the result of starting the application and navigating to the `/Home/CustomVariable` controll (which doesn't define a value for the `id` segment variable) in Figure 15-10.

eBooks-IT.org

Figure 15-10. Detecting when a URL doesn't contain a value for an optional segment variable

Using Optional URL Segments to Enforce Separation of Concerns

Some developers who are focused on the separation of concerns in the MVC pattern do not like putting the default va segment variables into the routes for an application. If this is an issue, you can use C# optional parameters along with segment variable in the route to define the default values for action method parameters. As an example, Listing 15-21 `CustomVariable` action method to define a default value for the `id` parameter that will be used if the URL do a value.

Listing 15-21. Defining a Default Value for an Action Method Parameter in the HomeController.cs File

```
...
public ActionResult CustomVariable(string id = "DefaultId") {
    ViewBag.Controller = "Home";
    ViewBag.Action = "CustomVariable";
    ViewBag.CustomVariable = id;
    return View();
}
...
```

There will always be a value for the `id` parameter (either one from the URL or the default), so I am able to remov which deals with the `null` value. This action method combined with the route I defined in Listing 15-21 is the func equivalent to the route shown in Listing 15-22:

Listing 15-22. An Equivalent Route

```
...
routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = "DefaultId"
...
```

The difference is that the default value for the `id` segment variable is defined in the controller code and not in the definition.

UNIT TESTING: OPTIONAL URL SEGMENTS

The issue to be aware of when testing optional URL segments is that the segment variable will not be added to t `RouteData.Values` collection unless a value was found in the URL. This means that you should not inc variable in the anonymous type unless you are testing a URL that contains the optional segment. Here are the cha `TestIncomingRoutes` unit test method for the route defined in Listing 15-22.

```
[TestMethod]
public void TestIncomingRoutes() {
  TestRouteMatch("~/", "Home", "Index");
  TestRouteMatch("~/Customer", "Customer", "index");
  TestRouteMatch("~/Customer/List", "Customer", "List");
  TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
  TestRouteFail("~/Customer/List/All/Delete");
}
...
```

Defining Variable-Length Routes

Another way of changing the default conservatism of URL patterns is to accept a variable number of URL segments. you to route URLs of arbitrary lengths in a single route. You define support for variable segments by designating one segment variables as a *catchall*, done by prefixing it with an asterisk (*), as shown in Listing 15-23.

Listing 15-23. Designating a Catchall Variable in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

                    routes.MapRoute("MyRoute",  "{controller}/{actio
{*catchall}",
                 new { controller = "Home", action = "Index",
                     id = UrlParameter.Optional });
        }
    }
}
```

I have extended the route from the previous example to add a catchall segment variable, which I imaginatively calle `catchall`. This route will now match *any* URL, irrespective of the number of segments it contains or the value o those segments. The first three segments are used to set values for the `controller`, `action`, and `id` variab respectively. If the URL contains additional segments, they are all assigned to the `catchall` variable, as shown i

Table 15-5. *Matching URLs with a Catchall Segment Variable*

Segments	Example URL	Maps To
0	/	controller = Homeaction = Index
1	/Customer	controller = Customeraction = Index
2	/Customer/List	controller = Customeraction = List
3	/Customer/List/All	controller = Customeraction = Listid = All
4	/Customer/List/All/Delete	controller = Customeraction = Listid = Allcatchall = D
5	/Customer/List/All/Delete/Perm	controller = Customeraction = Listid = Allcatchall = D

There is no upper limit to the number of segments that the URL pattern in this route will match. Notice that the seg

eBooks-IT.org

break out the individual segments.

sing the

UNIT TEST: TESTING CATCHALL SEGMENT VARIABLES

I can treat a catchall variable just like a custom variable. The only difference is that I must expect multiple segm concatenated in a single value, such as *segment / segment / segment*. Notice that I will not receive the leading character. Here are the changes to the `TestIncomingRoutes` method that demonstrate testing for a ca segment, using the route defined in Listing 15-23 and the URLs shown in Table 15-5:

```
...
[TestMethod]
public void TestIncomingRoutes() {
  TestRouteMatch("~/", "Home", "Index");
  TestRouteMatch("~/Customer", "Customer", "Index");
  TestRouteMatch("~/Customer/List", "Customer", "List");
  TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
  TestRouteMatch("~/Customer/List/All/Delete", "Customer", "List",
    new { id = "All", catchall = "Delete" });
  TestRouteMatch("~/Customer/List/All/Delete/Perm", "Customer", "List",
    new { id = "All", catchall = "Delete/Perm" });
}
...
```

Prioritizing Controllers by Namespaces

When an incoming URL matches a route, the MVC Framework takes the value of the `controller` variable and appropriate name. For example, when the value of the `controller` variable is `Home`, then the MVC Framewo controller called `HomeController`. This is an *unqualified* class name, which means that the MVC Framewor know what to do if there are two or more classes called `HomeController` in different .

To demonstrate the problem, create a new folder in the root of the example project called `AdditionalCont` and add a new `Home` controller, setting the contents to match those in Listing 15-24.

Listing 15-24. The Contents of the AdditionalControllers/HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.AdditionalControllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Additional Controllers - Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

When you start the app, you will see the error shown in Figure 15-11.

eBooks-IT.org

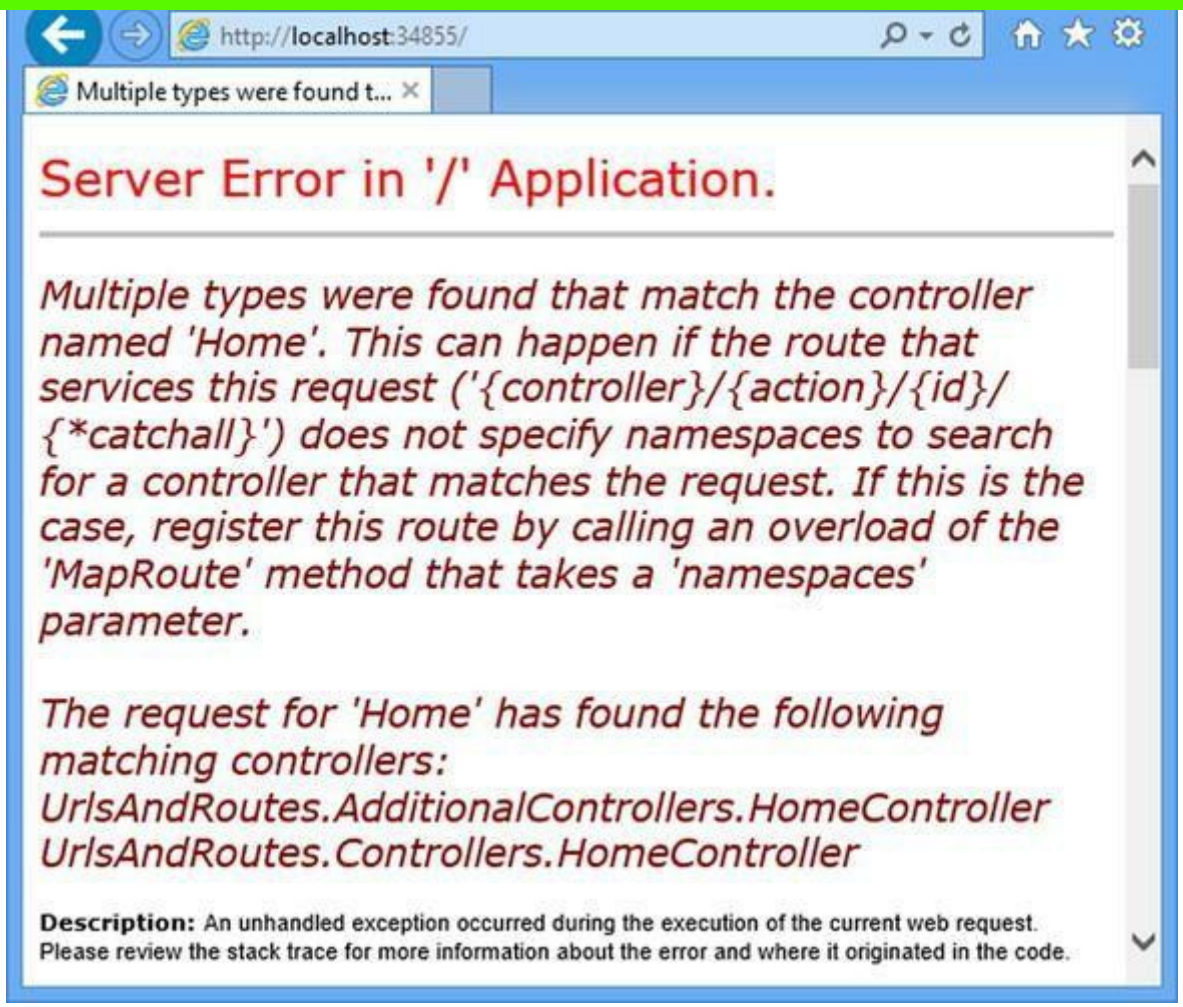


Figure 15-11. The error displayed when there are two controllers with the same name

The MVC Framework searched for a class called `HomeController` and found two: one in the original `RoutesAndUrls.Controllers` namespace and one in the new `RoutesAndUrls.AdditionalControllers` namespace. If you read the text of the error shown in you can see that the MVC Framework helpfully reports which classes it has found.

This problem arises more often than you might expect, especially if you are working on a large MVC project that u of controllers from other development teams or third-party suppliers. It is natural to name a controller relating to use `AccountController`, for example, and it is only a matter of time before you encounter a naming clash.

To address this problem, I can tell the MVC Framework to give preference to certain namespaces when attemptin the name of a controller class, as demonstrated in Listing 15-25.

Listing 15-25. Specifying Namespace Resolution Order in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

                                                    routes.MapRoute("MyRout
```

eBooks-IT.org

```
                    new { controller = "Home", action = "Index",
                        id = UrlParameter.Optional
                    } , new[] { "URLsAndRoutes.AdditionalControllers"
            }
        }
}
```

I express the namespaces as an array of strings and in the listing I have told the MVC Framework to look in the `URLsAndRoutes.AdditionalControllers` namespace before looking anywhere else.

If a suitable controller cannot be found in that namespace, then the MVC Framework will default to its regular beh in all of the available namespaces. If you start the app once you have made this addition to the route, you will see the in Figure 15-12, which shows that the request for the root URL, which is translated in to a request for the `Index` in the `Home` controller, has been sent to the controller in the `AdditionalControllers` namespace.

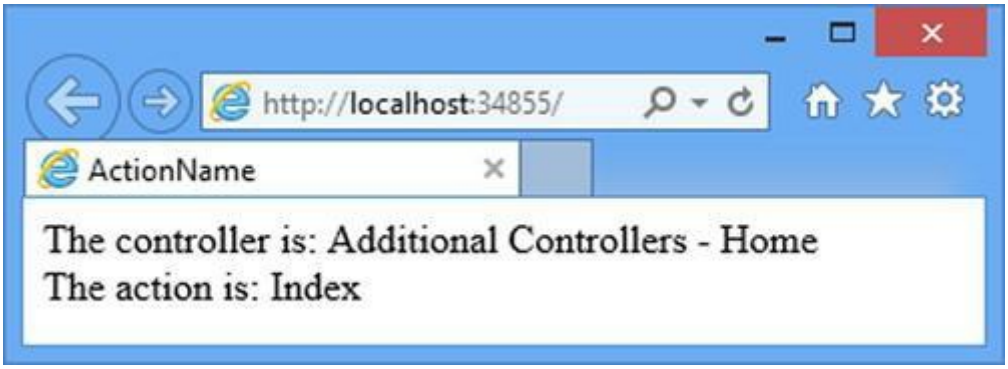


Figure 15-12. Giving priority to controllers in a specified namespaces

The namespaces added to a route are given equal priority. The MVC Framework does not check the first namespa moving on to the second and so forth. For example, suppose that I added both of the project namespaces to the rout

```
...
routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}
            new { controller = "Home", action = "Index",
UrlParameter.Optional },
                        new[] { "URLsAndRoutes.AdditionalContr
"UrlsAndRoutes.Controllers" });
...
```

I would see the same error as shown in Figure 15-11, because the MVC Framework is trying to resolve the contro name in *all* of the namespaces added to the route. If I want to give preference to a single controller in one namespac other controllers resolved in another namespace, I need to create multiple routes, as shown in Listing 15-26.

Listing 15-26. Using Multiple Routes to Control Namespace Resolution in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
```

routes.MapRoute("AddContolle

eBooks-IT.org

```
                    new { controller = "Home", action = "Index",
                        id = UrlParameter.Optional },
                    new[] { "URLsAndRoutes.AdditionalControllers" });

                    routes.MapRoute("MyRout
{controller}/{action}/{id}/{*catchall}",
                    new { controller = "Home", action = "Index",
                        id = UrlParameter.Optional },
                    new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

The first route applies when the user explicitly requests a URL whose first segment is `Home` and will target the H controller in the `AdditionalControllers` folder. All other requests, including those where no first segme specified, will be handled by controllers in the `Controllers` folder.

I can tell the MVC Framework to look *only* in the namespaces that I specify. If a matching controller cannot be fo framework will not search elsewhere. Listing 15-27 shows how this feature is used.

Listing 15-27. Disabling Fallback Namespaces in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

            Route myRoute = routes.MapRoute("AddContollerRoute",
                    "Home/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "URLsAndRoutes.AdditionalControllers" });

            myRoute.DataTokens["UseNamespaceFallback"] = false;
        }
    }
}
```

The `MapRoute` method returns a `Route` object. I have been ignoring this in previous examples, because I dic make any adjustments to the routes that were created. To disable searching for controllers in other namespaces, I tak `Route` object and set the `UseNamespaceFallback` key in the `DataTokens` collection property to `f`

This setting will be passed along to the component responsible for finding controllers, which is known as the *cont* and which I discuss in detail in Chapter 19. The effect of this addition is that requests that cannot be satisfied by the controller in the `AdditionalControllers` folder will fail.

Constraining Routes

At the start of the chapter, I described how URL patterns are conservative in how they match segments and liberal in

eBooks-IT.org

the deg
conservatism: making a route match more or fewer segments using default values, optional variables, and so on.

It is now time to look at how to control the liberalism in matching the content of URL segments: how to restrict th that a route will match against. Once I have control over both of these aspects of the behavior of a route, I can creat schemas that are expressed with laser-like precision.

Constraining a Route Using a Regular Expression

The first technique is constraining a route using regular expressions. Listing 15-28 contains an example.

Listing 15-28. Using a Regular Expression to Constrain a Route in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

                                                routes.MapRoute("MyRout
{controller}/{action}/{id}/{*catchall}",
                      new { controller = "Home", action = "Index
UrlParameter.Optional },
                new { controller = "^H.*" },
                new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

Constraints are defined by passing them as a parameter to the `MapRoute` method. Like default values, constrai expressed as an anonymous type, where the properties of the type correspond to the names of the segment variables constrain. In this example, I have used a constraint with a regular expression that matches URLs only where the valu `controller` variable begins with the letter `H`.

Note Default values are applied before constraints are checked. So, for example, if I request the URL `/`, the defa `controller`, which is `Home`, is applied. The constraints are then checked, and since the `controller` va with `H`, the default URL will match the route.

Constraining a Route to a Set of Specific Values

Regular expressions can constrain a route so that only specific values for a URL segment will cause a match. I do th bar (|) character, as shown in Listing 15-29.

Listing 15-29. Constraining a Route to a Specific Set of Segment Variable Values in the RouteConfig.cs File

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

eBooks-IT.org

```
namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

                                                    routes.MapRoute("MyRout
{controller}/{action}/{id}/{*catchall}",
                        new { controller = "Home", action = "Index
UrlParameter.Optional },
                    new { controller = "^H.*", action = "^Index$|^About
                    new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

This constraint will allow the route to match only URLs where the value of the action segment is `Index` or `Abo`
Constraints are applied together, so the restrictions imposed on the value of the `action` variable are combined wit
imposed on the `controller` variable. This means that the route in Listing 15-29 will match URLs only when th
`controller` variable begins with the letter `H` and the `action` variable is `Index` or `About`. So now you c
mean about creating precise routes.

Constraining a Route Using HTTP Methods

Routes can be constrained so that they match a URL only when it is requested using a specific HTTP method, as der
Listing 15-30.

Listing 15-30. Constraining a Route Based on an HTTP Method in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)

                                                    routes.MapRoute("MyRout
{controller}/{action}/{id}/{*catchall}",
                        new { controller = "Home", action = "Index
UrlParameter.Optional },
                    new {
                        controller = "^H.*", action = "Index|About",
                        httpMethod = new HttpMethodConstraint("GET")
                    },
                    new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

The format for specifying an HTTP method constraint is slightly odd. It does not matter what name is given to the
long as it is assigned to an instance of the `HttpMethodConstraint` class. In the listing, I called the constra
`httpMethod` to help distinguish it from the value-based constraints I defined previously.

eBooks-IT.org

using a
as HttpGet and HttpPost. The route constraints are processed much earlier in the request pipeline, and they
name of the controller and action required to process a request. The action method attributes are used to determine w
action method will be used to service a request by the controller. I provide details of how to handle different kinds of
methods (including the more unusual ones such as PUT and DELETE) in Chapter 16.

I pass the names of the HTTP methods I want to support as string parameters to the constructor of the HttpMethodConstraint class. In the listing, I limited the route to GET requests, but I could have easily a
for other methods, like this:

```
...
httpMethod = new HttpMethodConstraint("GET", "POST") },
...
```

UNIT TESTING: ROUTE CONSTRAINTS

When testing constrained routes, it is important to test for both the URLs that will match and the URLs you are
exclude, which you can do by using the helper methods introduced at the start of the chapter. Here are the chan
TestIncomingRoutes test method that I used to test the route defined in Listing 15-30:

```
...
[TestMethod]
public void TestIncomingRoutes() {
  TestRouteMatch("~/", "Home", "Index");
  TestRouteMatch("~/Home", "Home", "Index");
  TestRouteMatch("~/Home/Index", "Home", "Index");

  TestRouteMatch("~/Home/About", "Home", "About");
  TestRouteMatch("~/Home/About/MyId", "Home", "About", new { id = "MyId" });
  TestRouteMatch("~/Home/About/MyId/More/Segments", "Home", "About",
    new {
      id = "MyId",
      catchall = "More/Segments"
});

  TestRouteFail("~/Home/OtherAction");
  TestRouteFail("~/Account/Index");
  TestRouteFail("~/Account/About");
}
...
```

Using Type and Value Constraints

The MVC Framework contains a number of built-in constraints that can be used to restrict the URLs that a route ma
on the type and value of segment variables. In Listing 15-31, you can see how I have applied one of these constraints
routing configuration of the example application.

Listing 15-31. Using a Built-in Type/Value Constraint in the RouteConfig.cs File

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

eBooks-IT.org

```
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)

                                                    routes.MapRoute("MyRout
{controller}/{action}/{id}/{*catchall}",
                        new { controller = "Home", action = "Index
UrlParameter.Optional },
                new {
                    controller = "^H.*", action = "Index|About",
                    httpMethod = new HttpMethodConstraint("GET"),
                    id = new RangeRouteConstraint(10, 20)
                },
                new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

In the constraint classes, which are in the `System.Web.Mvc.Routing.Constraints` namespace if segment variables are values for different C# types and can perform basic checks. In the listing, I have used the `RangeRouteConstraint` class, which checks that the value provided for a segment variable is a valid `int` falls between two bounds – , in this case `10` and `20`. Table 15-6 describes the complete set of constraint classes. N accept arguments and so I have shown the class names as they would be used to configure routes. Ignore the *Attrib* column for the moment. I'll refer back to it when I introduce the attribute routing feature later in this chapter.

Table 15-6*. The route constraint classes*

Name	Description	Attribute Constra
`AlphaRouteConstraint()`	Matches alphabet characters, irrespective of case (A–Z, a–z)	`alpha`
`BoolRouteConstraint()`	Matches a value that can be parsed into a `bool`	`bool`
`DateTimeRouteConstraint()`	Matches a value that can be parsed into a `DateTime`	`datetime`
`DecimalRouteConstraint()`	Matches a value that can be parsed into a `decimal`	`decimal`
`DoubleRouteConstraint()`	Matches a value that can be parsed into a `double`	`double`
`FloatRouteConstraint()`	Matches a value that can be parsed into a `float`	`float`
`IntRouteConstraint()`	Matches a value that can be parsed into an `int`	`int`
`LengthRouteConstraint(len)` `LengthRouteConstraint(min, max)`	Matches a value with the specified number of characters or that is between `min` and `max` characters in length.	`length(len)` `max)`
`LongRouteConstraint()`	Matches a value that can be parsed into a `long`	`long`
`MaxRouteConstraint(val)`	Matches an `int` value if the value is less than `val`	`max(val)`
`MaxLengthRouteConstraint(len)`	Matches a string with no more than `len` characters	`maxlength(l`
`MinRouteConstraint(val)`	Matches an `int` value if the value is more than `val`	`min(val)`

RangeRouteConstraint(min, max)	Matches an int value if the value is between min and max	range(min,

You can combine different constraints for a single segment variable by using the CompoundRouteConstr which accepts an array of constraints as its constructor argument. In Listing 15-32, you can see how I have used th apply both the AlphaRouteConstraint and the MinLengthRouteConstraint to the id segr to ensure that the route will only match string values that contain solely alphabetic characters and have at least six ch

Listing 15-32. Combining Route Constraints in the RouteConfig.cs File

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

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)

                                                routes.MapRoute("MyRout
{controller}/{action}/{id}/{*catchall}",
                      new { controller = "Home", action = "Index
UrlParameter.Optional },
                new {
                    controller = "^H.*", action = "Index|About",
                    httpMethod = new HttpMethodConstraint("GET"),
                    id = new CompoundRouteConstraint(new IRouteCons
{
                        new AlphaRouteConstraint(),
                        new MinLengthRouteConstraint(6)
                     })
                },
                new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

Defining a Custom Constraint

If the standard constraints are not sufficient for your needs, you can define your own custom constraints by implem IRouteConstraint interface. To demonstrate this feature, I added an Infrastructure folder to the project and created a new class file called UserAgentConstraint.cs, the contents of which are shown i 33.

Listing 15-33. The Contents of the UserAgentConstraint.cs File

```
using System.Web;
using System.Web.Routing;
```

```
    public class UserAgentConstraint : IRouteConstraint {

        private string requiredUserAgent;

        public UserAgentConstraint(string agentParam) {
            requiredUserAgent = agentParam;
        }

        public bool Match(HttpContextBase httpContext, Route route
parameterName,
                                RouteValueDictionary values, RouteD
routeDirection) {

            return httpContext.Request.UserAgent != null &&
                httpContext.Request.UserAgent.Contains(requiredUse
        }
    }
}
```

The `IRouteConstraint` interface defines the `Match` method, which an implementation can use to indic routing system if its constraint has been satisfied. The parameters for the `Match` method provide access to the req client, the route that is being evaluated, the parameter name of the constraint, the segment variables extracted from th details of whether the request is to check an incoming or outgoing URL. For the example, I check the value of the U property of the client request to see if it contains a value that was passed to the constructor. Listing 15-34 shows the constraint used in a route.

Listing 15-34. Applying a Custom Constraint in a Route in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.Mvc.Routing.Constraints;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)

            routes.MapRoute("ChromeRoute", "{*catchall}",
                new { controller = "Home", action = "Index" },
                new { customConstraint = new UserAgentConstraint("
},
                new[] { "UrlsAndRoutes.AdditionalControllers" });

                                        routes.MapRoute("MyRout
{controller}/{action}/{id}/{*catchall}",
                    new { controller = "Home", action = "Index
UrlParameter.Optional },
                new {
                    controller = "^H.*", action = "Index|About",
```

eBooks-IT.org

```
                                                                    ),
                        id = new CompoundRouteConstraint(new IRouteCons
{
                                new AlphaRouteConstraint(),
                                new MinLengthRouteConstraint(6)
                            })
                    },
                    new[] { "URLsAndRoutes.Controllers" });
            }
        }
}
```

In the listing, I have constrained the first route so that it will match only requests made from browsers whose use contains `Chrome`. If the route matches, then the request will be sent to the `Index` action method in the `Home` c defined in the `AdditionalControllers` folder, irrespective of the structure and content of the URL that l requested. The URL pattern consists of just a catchall segment variable, which means that the values for the `contr` `action` segment variables will always be taken from the defaults and not the URL itself.

The second route will match all other requests and target controllers in the `Controllers` folder, subject to th value constraints I applied in the previous section. The effect of these routes is that one kind of browser always ends same place in the application. You can see this in Figure 15-13, which shows the effect of navigating to the app usin Chrome.

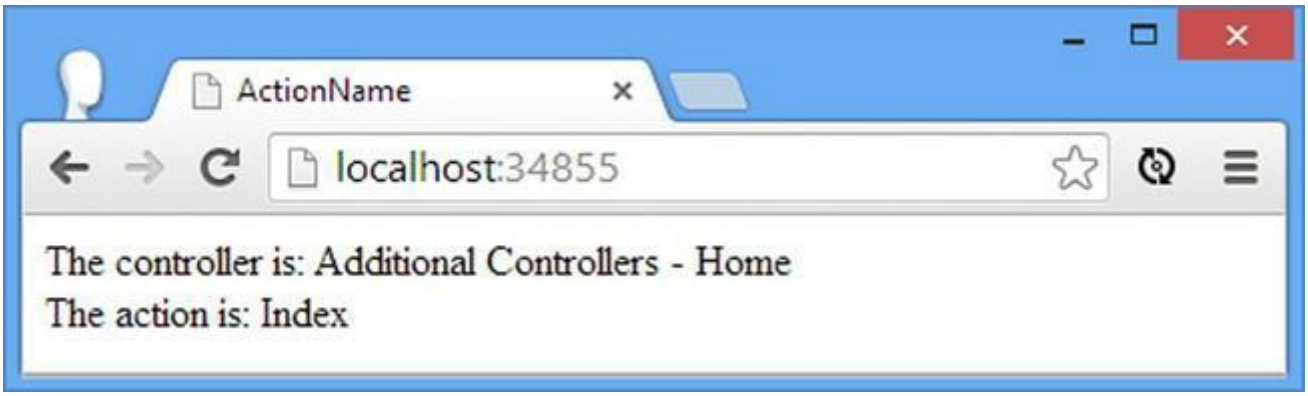


Figure 15-13. Navigating to the app using the Google Chrome browser

Figure 15-14 shows the result of navigating to the example application using Internet Explorer. (Notice that I have segment that contains six or more alpha characters to make the second route match the URL because of the constrai in the last section.)

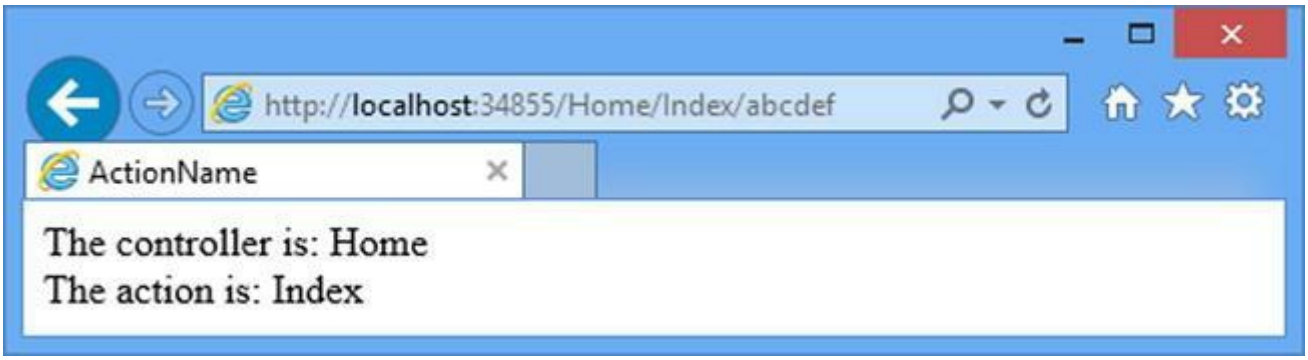


Figure 15-14. Navigating to the app using Internet Explorer

Note To be clear, because this is the kind of thing I get angry letters about, I am not suggesting that you restrict application so that it supports only one kind of browser. I used user-agent strings solely to demonstrate custom route and believe in equal opportunities for all browsers. I really hate Web sites that try to force their preference for brows

eBooks-IT.org

All of the examples so far in this chapter have been defined using a technique known as *convention-based routing*. M support for a new technique known as *attribute routing*, in which routes are defined by C# attributes that are applied the controller classes. In the sections that follow, I'll show you how to create and configure routes using attributes, mixed freely with the standard convention-based routes.

CONVENTION VERSUS ATTRIBUTE ROUTING

Attribute routing is one of the major additions to MVC 5, but I must admit that I am not a fan. One of the main MVC pattern is, as I described in Chapter 3, to separate out different parts of the application to make them easie test and maintain. I prefer convention-based routing because the controllers have no knowledge or dependency configuration of the application. By contrast, I find attribute routing muddies the waters and blurs the lines betw important components of an application.

That said, attribute routing is supported as part of the MVC Framework as of MVC 5 and you should read the f sections and make up your own mind. The fact that I don't like a feature shouldn't deter you from giving it due for use in your projects.

The good news is that both approaches to creating routes use the same underlying infrastructure and that means in the sections that follow, that you can use both approaches in a single project with no ill effects.

Enabling and Applying Attribute Routing

Attribute routing is disabled by default and is enabled by the `MapMvcAttributeRoutes` extension method, called on the `RouteCollection` object passed as the argument to the static `RegisterRoutes` method this method call to the `RouteConfig.cs` file in Listing 15-35, as well as simplifying the routes in the applicatio can focus on using attributes.

Listing 15-35. Enabling Attribute Routing in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)

            routes.MapMvcAttributeRoutes();

            routes.MapRoute("Default", "{controller}/{action}/{id}
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}
```

Calling the `MapMvcAttributeRoutes` method causes the routing system to inspect the controller classe application and look for attributes that configure routes. The most important attribute is called `Route` and you can

Listing 15-36. Applying the Route Attribute in the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

This is the basic use of the `Route` attribute, which is to define a static route for an action method. The `Route` properties, as described in Table 15-7.

Table 15-7. The Parameters Supported by the Route Attribute

Name	Description
`Name`	Assigns a name to the route, used for generating outgoing URLs from a specific route
`Template`	Defines the pattern that will be used to match URLs that target the action method

If you supply just one value when applying the `Route` attribute—as I did in the listing—then the value is assume pattern that will be used to match routes. Patterns for the `Route` attribute follow the same structure as for convent routing, although there are some differences when it comes to constraining route matching (which I describe in the *Route Constraints* section, later in the chapter). In this example, I used the `Route` attribute to specify that the `Inc` the `Customer` controller can be accessed through the URL `/Test`. You can see the result in Figure 15-15. I sh to use the `Name` property in Chapter 16.

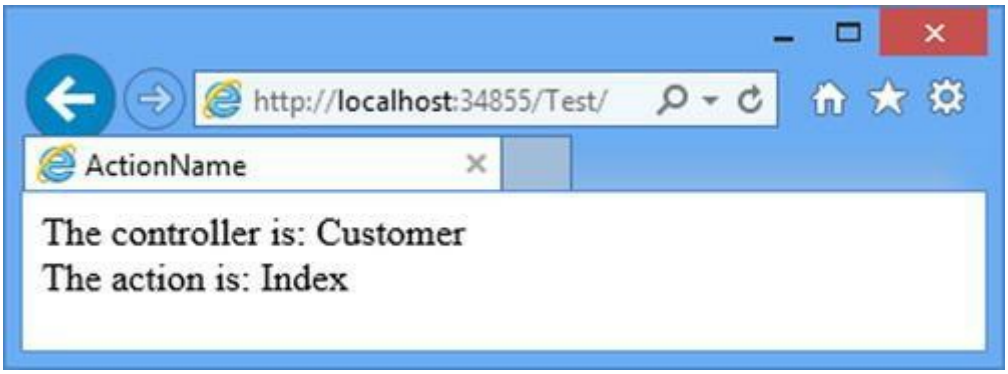


Figure 15-15. The effect of applying the Route attribute to create a static route

When an action method is decorated with the `Route` attribute, it can no longer be accessed through the convent routes defined in the `RouteConfig.cs` file. For my example, this means that the `Index` action of `Custor` controller can no longer be reached through the `/Customer/Index` URL.

eBooks-IT.org

disabled. Take care to call the MapMvcAttributeRoutes method in the RouteConfig.cs file or yo
unreachable action methods.

The Route attribute only affects the methods that it is applied to, which means that although the Index action
the Customer controller is reachable via the /Test URL, the List action must still be targeted using the
/Customer/List URL.

Tip You can apply the Route attribute to the same action method multiple times and each instance will create a

Creating Routes with Segment Variables

The attribute routing feature supports all of the same features as convention-based routing, albeit expressed through a
This includes creating routes that contain segment variables and you can see an example of such a route in Listing 15

Listing 15-37. Creating an Attribute Route with a Segment Variable in the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        [Route("Users/Add/{user}/{id}")]
        public string Create(string user, int id) {
            return string.Format("User: {0}, ID: {1}", user, id);
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

I have added an action method called Create that takes string and int arguments. For simplicity, I retur
result from the method so that I don't have to create a view. The route I defined with the Route attribute mixes a
(Users/Add) with user and id segment variables that correspond to the method arguments. The MVC Fram
the model binding feature, which I describe in Chapter 25, to convert the segment variable values to the correct types
invoke the Create method. Figure 15-16 shows the effect of navigating to the URL /Users/Add/Adam/1

eBooks-IT.org

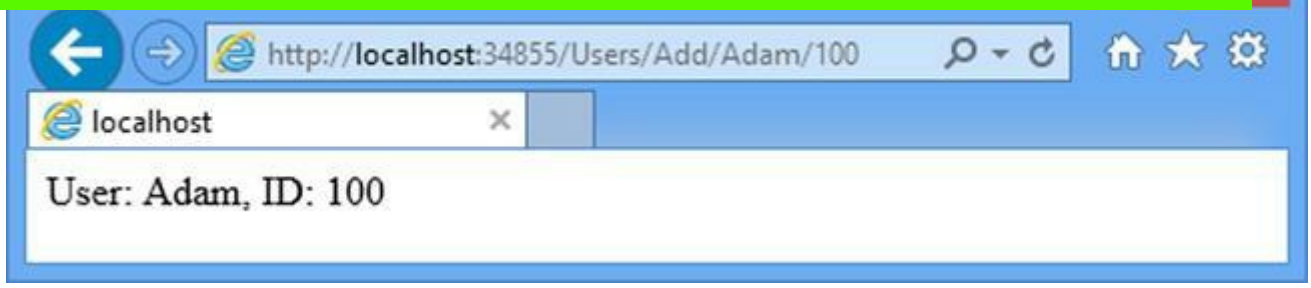


Figure 15-16. Navigating to a URL with segment variables

Notice that each instance of the `Route` attribute operates independently, and that means that I am able to create different routes to target each of the action methods in the controller, as described in Table 15-8.

***Table 15-8**. The Actions in the Customer Controller and the Routes that Target Them*

Action	URL
`Index`	`/Test`
`Create`	`/Users/Add/Adam/100` (or any values for the last two segments)
`List`	`/Customer/List` (through the route defined in the `RouteConfig.cs` file)

Applying Route Constraints

Routes defined using attributes can be constrained just like those defined in the `RouteConfig.cs` file, althoug technique is more direct. To demonstrate how this works, I have added an additional action method to the `Custom` controller, as shown in Listing 15-38.

***Listing 15-38**. Adding an Action Method and Route to the CustomerController.cs File*

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        [Route("Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        [Route("Users/Add/{user}/{id:int}")]
        public string Create(string user, int id) {
              return string.Format("Create Method - User: {0}, I
user, id);
        }

        [Route("Users/Add/{user}/{password}")]
        public string ChangePass(string user, string password) {
              return string.Format("ChangePass Method - User: {0
{1}",
                user, password);
        }
```

eBooks-IT.org

```
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

The new action method, called `ChangePass`, takes two string arguments. But I have used the `Route` attribu associate the action with the same URL pattern as for the `Create` action method: a static prefix of `/Users/Ac` by two segment variables. To differentiate between the actions, I applied a constraint to the `Route` attribute for the method, as follows:

```
...
[Route("Users/Add/{user}/{id :int }")]
...
```

I followed the name of the segment variable—`id`—with a colon and then `int`. This tells the routing system that `Create` action method should only be targeted by requests where the value provided for the `id` segment is a valid The `int` constraint corresponds to the `IntRouteConstraint` constraint class and Table 15-6 includes the constraint names you can use to access the built-in type and value constraints.

You can see the effect of my constraints by starting the application and requesting the `/Users/Add/Adam/` `/Users/Add/Adam/Secret` URLs. The final segment of the first URL is a valid `int` and is directed to th method. The final segment of the second URL isn't an `int` value and so is directed to the `ChangePass` method Figure 15-17.

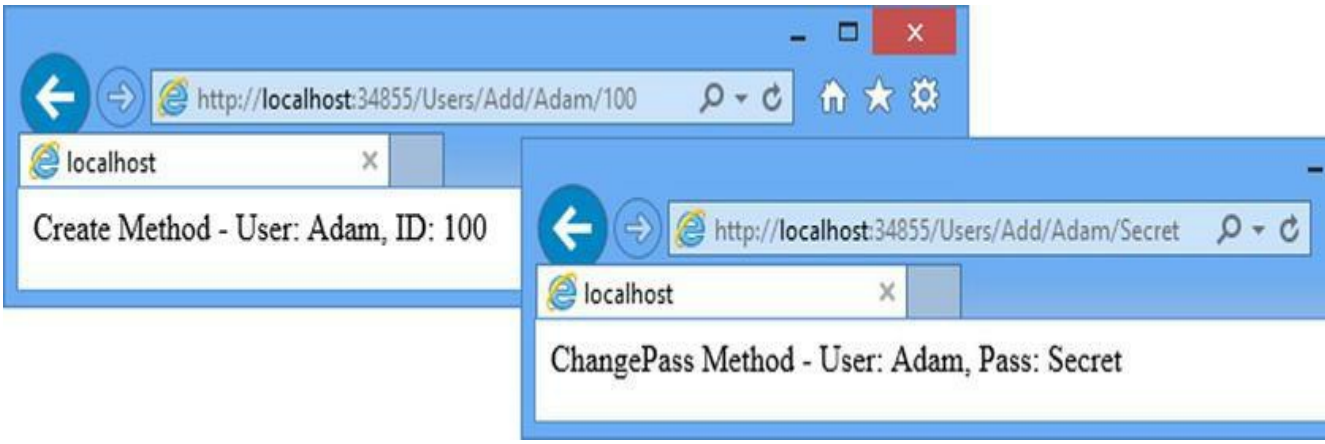


Figure 15-17. The effect of applying a constraint through the Route attribute

Combining Constraints

You can apply multiple constraints to a segment variable to further restrict the range of values that the route will mat 15-39, you can see how I have combined the `alpha` and `length` constraints on the route for the `ChangePa`

Listing 15-39. Applying Multiple Constraints to a Route in the CustomerController.cs File

```
...
[Route("Users/Add/{user}/{password :alpha:length(6) }")]
public string ChangePass(string user, string password) {
    return string.Format("ChangePass Method - User: {0}, Pass: {1}
        user, password);
}
...
```

eBooks-IT.org

by the
constraint and, if required, a value in parentheses. The route created by the attribute in this example will only match a
strings that have exactly six characters.

Caution Be careful when applying constraints. The routes defined by the `Route` attribute work in just the sam
those defined in the `RouteConfig.cs` file and a `404—Not Found` result will be sent to the browser for
can't be matched to an action method. Always define a fallback route that will match irrespective of the values that th
contains.

Using a Route Prefix

You can use the `RoutePrefix` attribute to define a common prefix that will be applied to all of the routes define
controller, which can be useful when you have multiple action methods that should be targeted using the same URL
see how I have applied the `RoutePrefix` attribute to the `CustomerController` in Listing 15-40.

Listing 15-40. Setting a Common Route Prefix in the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    [RoutePrefix("Users")]
    public class CustomerController : Controller {

        [Route("~/Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        [Route("Add/{user}/{id:int}")]
        public string Create(string user, int id) {
              return string.Format("Create Method - User: {0}, I
user, id);
        }

        [Route("Add/{user}/{password}")]
        public string ChangePass(string user, string password) {
              return string.Format("ChangePass Method - User: {0
{1}",
                user, password);
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

I used the `RoutePrefix` attribute to specify that the routes for the action method should be prefixed with `Us`
the prefix defined, I am able to update the `Route` attribute for the `Create` and `ChangePass` action methods
the prefix. The MVC Framework will combine the prefix with the URL pattern automatically when the routes are cre

Notice that I have also changed the URL pattern for the `Route` attribute applied to the `Index` action method, a

eBooks-IT.org

```
[Route("~/Test")]
...
```

Prefixing the URL with `~/` tells the MVC Framework that I don't want the `RoutePrefix` attribute applied to action method, which means that it will still be accessible through the URL `/Test`.

Summary

In this chapter, I took an in-depth look at the routing system. You have seen how routes are defined by convention o attributes. You have seen how incoming URLs are matched and handled, how to customize routes by changing the w match URL segments and by using default values and optional segments. I also showed you how to constrain routes range of requests that they will match, using both built-in constraints and using custom constraint classes.

In the next chapter, I show you how to generate outgoing URLs from routes in your views and how to use the MV Framework *areas* feature, which relies on the routing system and which can be used to manage large and complex M Framework applications.

eBooks-IT.org

Advanced Routing Features

In the previous chapter, I showed you how to use the routing system to handle incoming URLs, but this is only part
You also need to be able use your URL schema to generate *outgoing URLs* you can embed in your views, so that use
links and submit forms back to your application in a way that will target the correct controller and action. In this cha
show you different techniques for generating outgoing URLs, show you how to customize the routing system by rep
standard MVC routing implementation classes and use the MVC Framework *areas* feature, which allows you to breal
complex MVC application into manageable chucks. I finish this chapter with some best-practice advice about URL s
MVC Framework applications. Table 16-1 provides the summary for this chapter.

Table 16-1. *Chapter Summary*

Problem	Solution
Generate an `a` element with an outgoing URL	Use the `Html.ActionLink` helper method
Provide values for segment variables	Pass an anonymous object to the `ActionLink` helper whose proper correspond to the segment variable names.
Define attributes for the `a` element	Pass an anonymous object to the `ActionLink` helper whose proper correspond to the attribute names
Generate an outgoing URL without the `a` element	Use the `Url.Action` helper method
Generate a URL from a specific route	Specify the route name when calling the helper
Create a custom URL matching and generation policy	Derive from the `RouteBase` class
Create a custom mapping between URLs and action methods	Implement the `IRouteHandler` interface
Break an application into sections	Create areas or apply the `RouteArea` attribute
Resolve controller name ambiguity in areas	Give priority to a controller namespace
Prevent IIS and ASP.NET processing requests for static files before they are passed to the routing system	Use the `RouteExistingFiles` property
Prevent the routing system from processing a request	Use the `IgnoreRoute` method

Preparing the Example Project

I am going to continue to use the `UrlsAndRoutes` project from the previous chapter, but I need to make a cou
changes. First, I deleted the `AdditionalControllers` folder and `HomeController.cs` file that i
perform the deletion, right-click on the `AdditionalControllers` folder and select `Delete` from the p

Simplifying the Routes

I need to simplify the routes in the application. Edit your `App_Start/RouteConfig.cs` file so that it mat
content shown in Listing 16-1.

Listing 16-1. Simplifying the Example Routes in the RouteConfig.cs File

```
using System.Web.Mvc;
```

```
namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)

            routes.MapMvcAttributeRoutes();

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}
                new {
                    controller = "Home", action = "Index",
                    id = UrlParameter.Optional
                });
        }
    }
}
```

Adding the Optimization Package

Later in the chapter I describe the *areas* feature, which requires a new package be installed into the project. Enter the command into the NuGet console:

```
Install-Package Microsoft.AspNet.Web.Optimization -version 1.1.0
    -projectname UrlsAndRoutes
```

This package contains functionality for optimizing the JavaScript and CSS files in the project, which I describe in won't be using this feature directly in this chapter, but the areas feature has a dependency on it.

Updating the Unit Test Project

I need to make two changes to the unit test project. The first is to delete the `TestIncomingRoutes` method won't be using since this chapter is about generating outgoing routes. To avoid failed tests, simply remove the metho `RouteTests.cs` file.

The second change is to add a reference to the `System.Web.Mvc` namespace, which I do by installing the M package into the unit test project. Enter the following command into the NuGet console:

```
Install-Package Microsoft.Aspnet.Mvc -version 5.0.0 -pro
UrlsAndRoutes.Tests
```

I need to add the MVC 5 package so that I can use some helper methods to generate outgoing URLs. I didn't need the last chapter because the support for dealing with incoming URLs is in the `System.Web` and `System.Web.Routing` namespaces.

Generating Outgoing URLs in Views

In almost every MVC Framework application, you will want to allow the user to navigate from one view to another, usually rely on including a link in the first view that targets the action method that generates the second view.

It is tempting to just add a static `a` element whose `href` attribute targets the action method, like this:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

With the standard routing configuration, the HTML element creates a link that will target the `CustomVariab` method on the `Home` controller. Manually defined URLs like this one are quick and simple to create. They are also e dangerous and you will break all of the URLs you have hard-coded when you change the URL schema for your appli then must trawl through all of the views in your application and update all of the references to your controllers and a

eBooks-IT.org

ystem t
outgoing URLs, which ensures that the URLs scheme is used to produce the URLs dynamically and in a way that is g
reflect the URL schema of the application.

Using the Routing System to Generate an Outgoing URL

The simplest way to generate an outgoing URL in a view is to call the `Html.ActionLink` helper method with
illustrated by Listing 16-2, which shows an addition I made to the `/Views/Shared/ActionName.csht`

Listing 16-2. Using the Html.ActionLink Helper Method in the ActionName.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL", "CustomVariabl
    </div>
</body>
</html>
```

The parameters to the `ActionLink` method are the text for the link and the name of the action method that th
target. You can see the result of this addition by starting the app and allowing the browser to navigate to the root UR
illustrated by Figure 16-1.

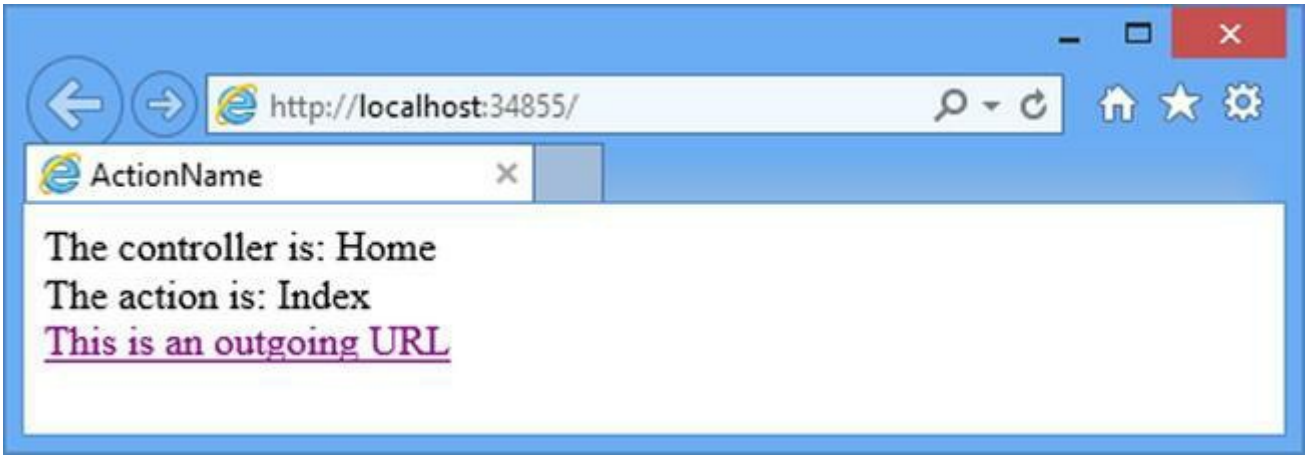


Figure 16-1. Adding an outgoing URL to a view

The HTML that the `ActionLink` method generates is based on the current routing configuration. For exampl
schema defined in Listing 16-1 (and assuming that the view is being rendered by a request to the `Home` controller) g
HTML:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

eBooks-IT.org

is that it automatically responds to changes in the routing configuration. As a demonstration, I have changed the rout
added a new route to the `RouteConfig.cs` file, as shown in Listing 16-3.

Listing 16-3. Adding a Route to the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.MapMvcAttributeRoutes();

    routes.MapRoute("NewRoute", "App/Do{action}",
        new { controller = "Home" });

    routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index",
            id = UrlParameter.Optional });
}
...
```

The new route changes the URL schema for requests that target the `Home` controller. If you start the app, you w
this change is reflected in the HTML that is generated by the `ActionLink` HTML helper method, as follows:

```
<a href="/App/DoCustomVariable">This is an outgoing URL</a>
```

You can see how generating links in this way addresses the issue of maintenance. I am able to change the routing
have the outgoing links in the views reflect the change automatically. And, of course an outgoing URL becomes a reg
when you click on the link, and so the routing system is used again to target the action method correctly, as shown i

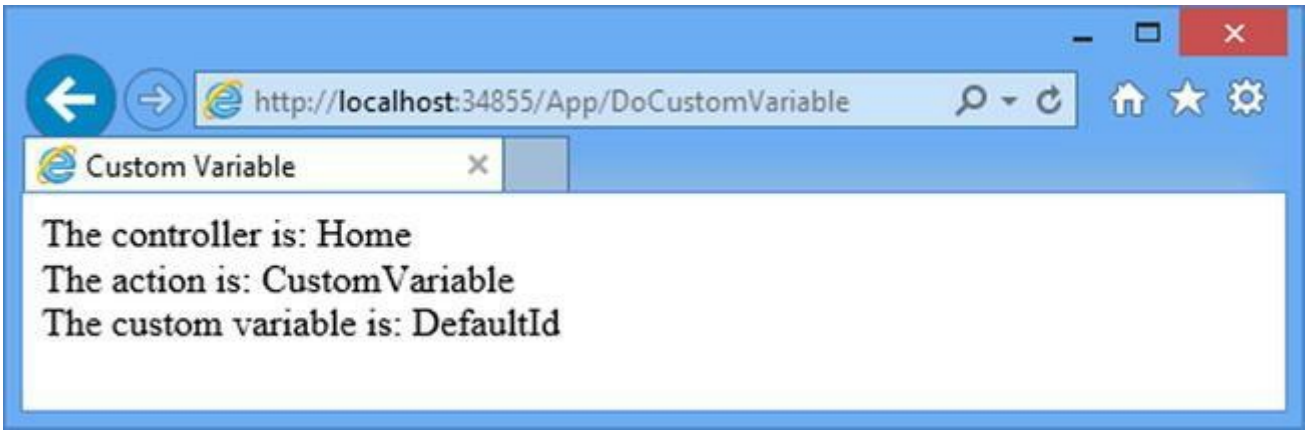


Figure 16-2. The effect of clicking on a link is to make an outgoing URL into an incoming request

UNDERSTANDING OUTBOUND URL ROUTE MATCHING

You have seen how changing the routes that define your URL schema changes the way that outgoing URLs are
Applications will usually define several routes, and it is important to understand just how routes are selected for
generation. The routing system processes the routes in the order that they were added to the `RouteCollec`
object passed to the `RegisterRoutes` method. Each route is inspected to see if it is a match, which requ
conditions to be met:

- A value must be available for every segment variable defined in the URL pattern. To find values for each segm
the routing system looks first at the values you have provided (using the properties of an anonymous ty
variable values for the current request, and finally at the default values defined in the route. (I return to the s
of these values later in this chapter.)

eBooks-IT.org

These are variables for which default values have been provided, but which do not occur in the URL defined pattern. in this route definition, `myVar` is a default-only variable:

```
routes.MapRoute("MyRoute", "{controller}/{action}",
                new { myVar = "true" });
```

For this route to be a match, I must take care to not supply a value for `myVar` or to make sure that the value matches the default value.

- The values for all of the segment variables must satisfy the route constraints. See the "Constraining Routes" previous chapter for examples of different kinds of constraints.

To be clear: the routing system doesn't try to find the route that provides the *best* matching route. It finds only match, at which point it uses the route to generate the URL; any subsequent routes are ignored. For this reason, define your most specific routes first. It is important to test your outbound URL generation. If you try to genera which no matching route can be found, you will create a link that contains an empty `href` attribute, like this:

```
<a href="">About this application</a>
```

The link will render in the view properly, but won't function as intended when the user clicks it. If you are gene URL (which I show you how to do later in the chapter), then the result will be `null`, which renders as the em views. You can exert some control over route matching by using named routes. See the "Generating a URL fro Route" section later in this chapter for details.

The first `Route` object meeting these criteria will produce a non-`null` URL, and that will terminate the URL process. The chosen parameter values will be substituted for each segment parameter, with any trailing sequenc values omitted. If you have supplied explicit parameters that do not correspond to segment parameters or defaul then the method will append them as a set of query string name/value pairs.

Targeting Other Controllers

The default version of the `ActionLink` method assumes that you want to target an action method in the same c has caused the view to be rendered. To create an outgoing URL that targets a *different* controller, you can use an ov allows you to specify the controller name, as illustrated by Listing 16-4.

Listing 16-4. Targeting a Different Controller in the ActionName.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL", "CustomVariabl
    </div>
    <div>
```

```
    Admin")
    </div>
</body>
</html>
```

Caution The routing system has no more knowledge of the application when generating outgoing URLs than wh incoming requests. This means that the values you supply for action methods and controllers are not validated, and y care not to specify nonexistent targets.

When you render the view, you will see the following HTML generated:

```
<a href="/Admin">This targets another controller</a>
```

My call for a URL that targets the `Index` action method on the `Admin` controller has been expressed as `/Adm` `ActionLink` method. The routing system is pretty clever and it knows that the route defined in the application w `Index` action method by default, allowing it to omit unneeded segments.

The routing system includes routes that have been defined using the `Route` attribute when determining how to ta action method. In Listing 16-5, you can see how I have changed the controller name in the `ActionLink` call so the Index action in the `Customer` controller.

Listing 16-5. Targeting an Action Decorated with the Route Attribute in the ActionName.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL", "CustomVariabl
    </div>
    <div>
            @Html.ActionLink("This targets another controller",
"Customer")
    </div>
</body>
</html>
```

The link that is generated is as follows:

```
<a href="/Test">This targets another controller</a>
```

This corresponds to the `Route` attribute I applied to the `Index` action method in the `Customer` controller

```
...
[Route("~/Test")]
public ActionResult Index() {
    ViewBag.Controller = "Customer";
```

eBooks-IT.org

```
    return View( ActionName );
}
...
```

Passing Extra Values

You can pass values for segment variables using an anonymous type, with properties representing the segments. List provides an example, which I added to the `ActionName.cshtml` view file.

Listing 16-6. Supplying Values for Segment Variables in the ActionName.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL",
            "CustomVariable", new { id = "Hello" })
    </div>
</body>
</html>
```

I have supplied a value for a segment variable called `id`. If the application uses the route shown in Listing 16-3, th following HTML will be rendered in the view:

```
<a href="/App/DoCustomVariable?id=Hello">This is an outgoing URL</
```

Notice that the value I supplied has been added as part of the query string to fit into the URL pattern described by This is because there is no segment variable that corresponds to `id` in that route. In Listing 16-7, I have edited the r `RouteConfig.cs` file to leave only a route that *does* have an `id` segment.

Listing 16-7. Editing the Routes in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {

   routes.MapMvcAttributeRoutes();

    routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
        new {
            controller = "Home", action = "Index",
            id = UrlParameter.Optional
        });
}
```

Start the application again and you will see that the call to the `ActionLink` helper method in the `ActionName.cshtml` view produces the following HTML element:

```
<a href="/Home/CustomVariable/Hello">This is an outgoing URL</a>
```

This time, the value I assigned to the `id` property is included as a URL segment, in keeping with the active route i application configuration.

UNDERSTANDING SEGMENT VARIABLE REUSE

When I described the way that routes are matched for outbound URLs, I explained that when trying to find valu the segment variables in a route's URL pattern, the routing system will look at the values from the current reque behavior that confuses many programmers and can lead to a lengthy debugging session.

Imagine the application has a single route, as follows:

```
...
routes.MapRoute("MyRoute", "{controller}/{action}/{color}/{page
...
```

Now imagine that a user is currently at the URL `/Catalog/List/Purple/123`, and I render a link a

```
...
@Html.ActionLink("Click me", "List", "Catalog", new {page=789},
...
```

You might expect that the routing system would be unable to match the route, because I have not supplied a val `color` segment variable, and there is no default value defined. You would, however, be wrong. The routing s match against the route I defined. It will generate the following HTML:

```
<a href="/Catalog/List/Purple/789">Click me</a>
```

The routing system is keen to make a match against a route, to the extent that it will reuse segment variable valu incoming URL. In this case, I end up with the value `Purple` for the color variable, because of the URL from imaginary user started.

This is *not* a behavior of last resort. The routing system will apply this technique as part of its regular assessme even if there is a subsequent route that would match without requiring values from the current request to be reu routing system will reuse values only for segment variables that occur earlier in the URL pattern rather than any that are supplied to the `Html.ActionLink` method. Suppose I tried to create a link like this:

```
...
@Html.ActionLink("Click me", "List", "Catalog", new {color="Aqua
null)
...
```

I have supplied a value for `color`, but not for `page`. But `color` appears before `page` in the URL pattern routing system *won't* reuse the values from the incoming URL, and the route will not match.

The best way to deal with this behavior is to prevent it from happening. I strongly recommend that you do not r behavior, and that you supply values for all of the segment variables in a URL pattern. Relying on this behavior v make your code harder to read, but you end up making assumptions about the order in which your users make which is something that will ultimately bite you as your application enters maintenance.

Specifying HTML Attributes

I have focused on the URL that the `ActionLink` helper method generates, but remember that the method genera complete HTML anchor (`a`) element. I can set attributes for this element by providing an anonymous type whose pr

eBooks-IT.org

iew to
attribute and assign a class to the HTML a element.

Listing 16-8. Generating an Anchor Element with Attributes in the ActionName.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL",
            "Index", "Home", null, new {
                id = "myAnchorID",
                @class = "myCSSClass"
            })
    </div>
</body>
</html>
```

I created a new anonymous type that has `id` and `class` properties, and passed it as a parameter to the `Actio` method. I passed `null` for the additional segment variable values, indicating that I do not have any values to supply

Tip Notice that I prepended the `class` property with a `@` character. This is a C# language feature that lets rese keywords be used as the names for class members. This is the technique that I used to assign elements to Bootstrap SportsStore application in Part 1 of this book.

When this call to `ActionLink` is rendered, I get the following HTML:

```
<a class="myCSSClass" href="/" id="myAnchorID">This is an outgoing
```

Generating Fully Qualified URLs in Links

All of the links that I have generated so far have contained relative URLs, but I can also use the `ActionLink` he generate fully qualified URLs, as shown in Listing 16-9.

Listing 16-9. Generating a Fully Qualified URL in the ActionNane.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
```

eBooks-IT.org

```
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL", "Index", "Home
            "https", "myserver.mydomain.com", " myFragmentName",
            new { id = "MyId" },
            new { id = "myAnchorID", @class = "myCSSClass" })
    </div>
</body>
</html>
```

This is the ActionLink overload with the most parameters, and accepts values for the protocol (https, in the name of the target server (myserver.mydomain.com), and the URL fragment (myFragmentNam all of the other options you saw previously. When rendered in a view, the ActionLink helper generates the follo

```
<a class="myCSSClass"
     href="https://myserver.mydomain.com/Home/Index/MyId#myFragmen
     id="myAnchorID">This is an outgoing URL</a>
```

I recommend using relative URLs wherever possible. Fully qualified URLs create dependencies on the way that yo infrastructure is presented to your users. I have seen large applications that relied on absolute URLs broken by uncoo changes to the network infrastructure or domain name policy, which are often outside the control of the programme

Generating URLs (and Not Links)

The Html.ActionLink helper method generates complete HTML <a> elements, which is usually what is re creating views, but there will be times when you want just a URL, without the surrounding HTML. In these circums Url.Action method can be used to generate just the URL and not the surrounding HTML. Listing 16-10 shows I made to the ActionName.cshtml file to create a URL with the Url.Action helper.

Listing 16-10. Generating a URL Without the Surrounding HTML in the ActionName.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        This is a URL:
        @Url.Action("Index", "Home", new { id = "MyId" })
    </div>
</body>
```

The `Url.Action` method works in the same way as the `Html.ActionLink` method, except that it ger the URL. The overloaded versions of the method and the parameters they accept are the same for both methods, and all of the things with `Url.Action` that I demonstrated with `Html.ActionLink` in the previous sections. how the URL in Listing 16-10 is rendered in the view in Figure 16-3.

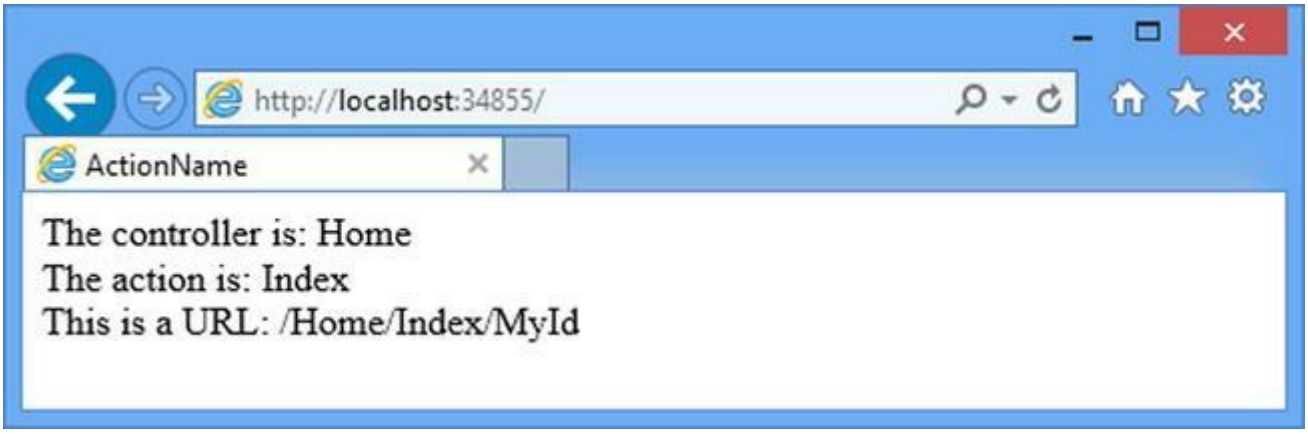


Figure 16-3. Rendering a URL (as opposed to a link) in a view

Generating Outgoing URLs in Action Methods

Mostly, you will need to generate outgoing URLs in views, but there are times when you may need to do something s an action method. This can be achieved using the same helper method used in the view, as illustrated by Listing 16-1 shows a new action method I added to the `Home` controller.

Listing 16-11. Generating an Outgoing URL in the HomeController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable(string id = "DefaultId"
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id;
            return View();
        }

        public ViewResult MyActionMethod() {
            string myActionUrl = Url.Action("Index", new { id = "M
            string myRouteUrl = Url.RouteUrl(new { controller = "H
                action = "Index" });
            //... do something with URLs...
            return View();
        }
    }
```

For the routing in the example app, the `myActionUrl` variable would be set to `/Home/Index/MyID` a
`myRouteUrl` variable would be set to `/`, which is consistent with the results that calling these helpers in a view
produce.

A more common requirement is to redirect the client browser to another URL, which can be achieved by returning
calling the `RedirectToAction` method, as shown in Listing 16-12.

Listing 16-12. Redirecting to Another Action in the HomeController.cs File

```
...
public RedirectToRouteResult MyActionMethod() {
    return RedirectToAction("Index");
}
...
```

The result of the `RedirectToAction` method is a `RedirectToRouteResult`, which instructs t
Framework to issue a redirect instruction to a URL that will invoke the specified action. There are the usual overload
the `RedirectToAction` method that specify the controller and values for the segment variables in the genera

If you want to send a redirect using a URL generated from just object properties, you can use the `RedirectT`
method, as shown in Listing 16-13. This method also returns a `RedirectToRouteResult` object and has
same effect as calling the `RedirectToAction` method.

Listing 16-13. Redirecting to a URL in the HomeController.cs File

```
...
public RedirectToRouteResult MyActionMethod() {
    return RedirectToRoute(new { controller = "Home", action = "In
= "MyID" });
}
...
```

Generating a URL from a Specific Route

In the previous examples, I left the routing system to select the route which will be used to generate a URL or a link.
section, I will show you how to control this process and select specific routes. In Listing 16-14, I have changed the
information in the `RouteConfig.cs` file to better demonstrate this feature.

Listing 16-14. Changing the Routing Configuration in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapMvcAttributeRoutes();
    routes.MapRoute("MyRoute", "{controller}/{action}");
      routes.MapRoute("MyOtherRoute", "App/{action}", new { contr
"Home" });
}
...
```

There are two routes in this configuration and I have specified names for both of these routes: `MyRoute` and
`MyOtherRoute`. There are two reasons for naming your routes:

- As a reminder of the purpose of the route
- So that you can select a specific route to be used to generate an outgoing URL

eBooks-IT.org

ActionLink method like this:

```
...
@Html.ActionLink("Click me", "Index", "Customer")
...
```

The outgoing link would always be generated using `MyRoute`, as follows:

```
<a href="/Customer/Index">Click me</a>
```

You can override the default route matching behavior by using the `Html.RouteLink` method, which lets yo which route you want to use, as follows:

```
...
@Html.RouteLink("Click me", "MyOtherRoute","Index", "Customer")
...
```

The result is that the link generated by the helper looks like this:

```
<a Length="8" href="/App/Index?Length=5">Click me</a>
```

In this case, the controller I specified, `Customer`, is overridden by the route and the link targets the `Home` co instead.

You can also give names to the routes you define with the `Route` attribute. In Listing 16-15, you can see how I name to such a route in the `Customer` controller.

Listing 16-15. Naming a Route in the CustomerController.cs File

```
...
[Route("Add/{user}/{id:int}", Name="AddRoute")]
public string Create(string user, int id) {
    return string.Format("Create Method - User: {0}, ID: {1}", use
}
...
```

The addition in this example sets a value for the `Name` property that I described in Chapter 15. In this example, I name `AddRoute` to the route that the attribute creates, which allows me to generate routes by name.

THE CASE AGAINST NAMED ROUTES

The problem with relying on route names to generate outgoing URLs is that doing so breaks through the separat concerns that is so central to the MVC design pattern. When generating a link or a URL in a view or action meth focus on the action and controller that the user will be directed to, not the format of the URL that will be used. I knowledge of the different routes into the views or controllers, I am creating dependencies that I would prefer t tend to avoid naming my routes (by specifying `null` for the route name parameter) and prefer to use code co remind myself of what each route is intended to do.

Customizing the Routing System

You have seen how flexible and configurable the routing system is, but if it does not meet your requirements, you ca the behavior. In this section, I will show you the two ways to do this.

Creating a Custom RouteBase Implementation

If you do not like the way that standard `Route` objects match URLs, or want to implement something unusual, you an alternative class from `RouteBase`. This gives you control over how URLs are matched, how parameters are

- GetRouteData (HttpContextBase httpContext): This is the mechanism by which *matching* works. The framework calls this method on each RouteTable.Routes entry in turn, unt returns a non-null value.
- GetVirtualPath (RequestContext requestContext, RouteValueDi values): This is the mechanism by which *outbound URL generation* works. The framework calls this m RouteTable.Routes entry in turn, until one of them returns a non-null value.

To demonstrate this kind of customization, I am going to create a RouteBase class that will handle legacy UR Imagine that I have migrated an existing application to the MVC Framework, but some users have bookmarked the p URLs or hard-coded them into scripts. I still want to support those old URLs. I could handle this using the regular ro but this problem provides a nice example for this section.

To begin, I need to create a controller that will receive the legacy requests. I have called the controller LegacyController, and its contents are shown in Listing 16-16.

Listing 16-16. The Contents of the LegacyController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {

    public class LegacyController : Controller {

        public ActionResult GetLegacyURL(string legacyURL) {
            return View((object)legacyURL);
        }
    }
}
```

In this simple controller, the GetLegacyURL action method takes the parameter and passes it as a view mode If I were implementing this controller in a real project, I would use this method to retrieve the files that were request is, I am simply going to display the URL in a view.

Tip Notice that I cast the parameter to the View method in Listing 16-16 to object. One of the overloaded the View method takes a string specifying the name of the view to render and, without the cast, this would be that the C# compiler thinks I want. To avoid this, I cast to object so that I unambiguously call the overload that model and uses the default view. I could also have solved this by using the overload that takes both the view name ar model, but I prefer not to make explicit associations between action methods and views if I can help it.

Create a view called GetLegacyURL.cshtml within the Views/Legacy folder and set the content c to match Listing 16-17.

Listing 16-17. The Contents of the GetLegacyURL.cshtml File

```
@model string

@{
    ViewBag.Title = "GetLegacyURL";
    Layout = null;
}

<h2>GetLegacyURL</h2>

The URL requested was: @Model
```

eBooks-IT.org

and I just display the model value. I have now reached the point where I can create a custom derivation of the Rout action class.

Routing Incoming URLs

I added a class file called `LegacyRoute.cs` to the `Infrastructure` folder (which is where I like to pu classes that do not really belong anywhere else). The contents of this file are shown in Listing 16-18.

Listing 16-18. The Contents of the LegacyRoute.cs File

```
using System;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes.Infrastructure {

    public class LegacyRoute : RouteBase {
        private string[] urls;

        public LegacyRoute(params string[] targetUrls) {
            urls = targetUrls;
        }

                public  override  RouteData  GetRouteData(HttpCon
httpContext) {
            RouteData result = null;

            string requestedURL =
                httpContext.Request.AppRelativeCurrentExecutionFil
                                        if    (urls.Contains(reque
StringComparer.OrdinalIgnoreCase)) {
                result = new RouteData(this, new MvcRouteHandler()
                result.Values.Add("controller", "Legacy");
                result.Values.Add("action", "GetLegacyURL");
                result.Values.Add("legacyURL", requestedURL);
            }
            return result;
        }

            public  override  VirtualPathData  GetVirtualPath(Reques
requestContext,
            RouteValueDictionary values) {

            return null;
        }
    }
}
```

The constructor of this class takes a string array that represents the individual URLs that this routing class will sup specify these when I register the route later. Of note in this class is the `GetRouteData` method, which is what system calls to see if the `LegacyRoute` class can match an incoming URL.

If the class cannot match the request, then I just return `null`, and the routing system will move on to the next ro

eBooks-IT.org

for the controller and action variables, and anything else I want to pass along to the action method.

When I create the RouteData object, I need to pass in the handler that I want to deal with the values that gen the standard MvcRouteHandler class, which is what assigns meaning to the controller and actio

```
...
result = new RouteData(this, new MvcRouteHandler());
...
```

For the vast majority of MVC applications, this is the class that you will require, as it connects the routing system controller/action model of an MVC application. But you can implement a replacement for MvcRouteHandler, demonstrate in the Creating a Custom Route Handler section later in the chapter.

In this custom RouteBase implementation, I am willing to match any request for the URLs that were passed t constructor. When I get such a URL, I add hard-coded values for the controller and action method to the RouteV object. I also pass along the requested URL as the legacyURL property. Notice that the name of this property m name of the parameter of the action method in the Legacy controller, ensuring that the value I generate will be pas action method via the parameter.

The last step is to register a new route that uses the custom RouteBase class. You can see how to do this in which shows the addition to the RouteConfig.cs file.

Listing 16-19. Registering the Custom RouteBase Implementation in the RouteConfig.cs File

```
using System.Web.Mvc;
using System.Web.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)
            routes.MapMvcAttributeRoutes();

            routes.Add(new LegacyRoute(
                    "~/articles/Windows_3.1_Overview.html",
                    "~/old/.NET_1.0_Class_Library"));

            routes.MapRoute("MyRoute", "{controller}/{action}");
                   routes.MapRoute("MyOtherRoute",  "App/{action}"
controller = "Home" });
        }

    }
}
```

I create a new instance of the LegacyRoute class and pass in the URLs I want it to route. I then add the obj RouteCollection using the Add method. Now when I start the application and request one of the legacy U defined, the request is routed by the LegacyRoute class and directed toward the Legacy controller, as show 16-4.

eBooks-IT.org

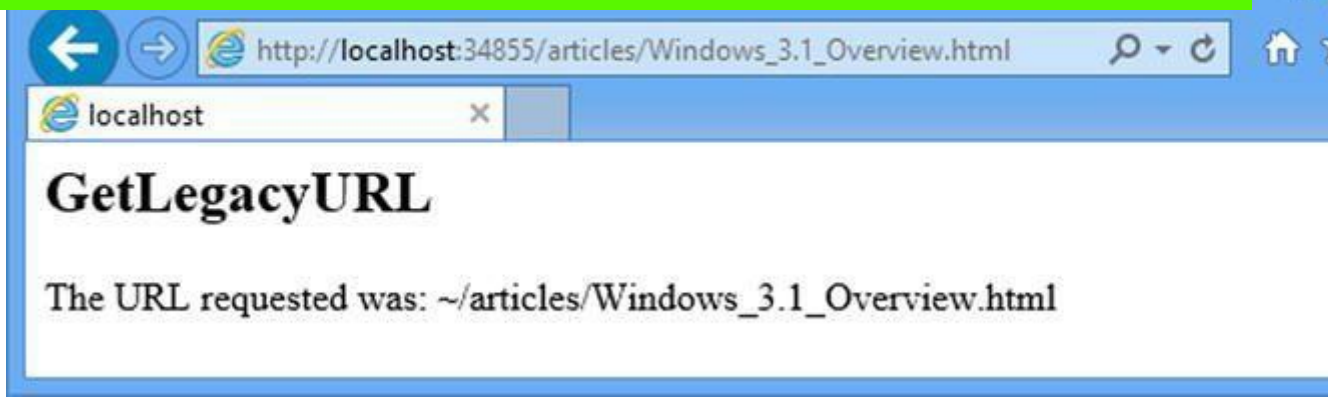


Figure 16-4. Routing requests using a custom RouteBase implementation

Generating Outgoing URLs

To support outgoing URL generation, I need to implement the `GetVirtualPath` method in the `LegacyRo` the class is unable to generate a specific URL, I let the routing system know by returning `null`. Otherwise, I retur of the `VirtualPathData` class. Listing 16-20 shows the implementation of this method.

Listing 16-20. Implementing the GetVirtualPath Method in the LegacyRoute.cs File

```
...
public     override     VirtualPathData     GetVirtualPath(Reques
requestContext,
        RouteValueDictionary values) {

    VirtualPathData result = null;

    if (values.ContainsKey("legacyURL") &&
                                urls.Contains((string)values["lega
StringComparer.OrdinalIgnoreCase)) {
        result = new VirtualPathData(this,
            new UrlHelper(requestContext)
                .Content((string)values["legacyURL"]).Substring(1)
    }
    return result;
}
...
```

I have been passing segment variables and other details around in earlier chapters using anonymous types. But beh scenes, the routing system has been converting these into `RouteValueDictionary` objects so they can be `RouteBase` implementations. Listing 16-21 shows an addition to the `ActionName.cshtml` view file that outgoing URL using the custom routing class.

Listing 16-21. Generating an Outgoing URL via a Custom Route in the ActionName.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>
```

```
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        This is a URL:
        @Html.ActionLink("Click me", "GetLegacyURL",
            new { legacyURL = "~/articles/Windows_3.1_Overview.htm
    </div>
</body>
</html>
```

When this view is rendered the `ActionLink` helper generates the following HTML if you request a URL such /`Home/Index`, just as you would expect:

```
<a href="/articles/Windows_3.1_Overview.html">Click me</a>
```

The anonymous type created with the `legacyURL` property is converted into a `RouteValueDictiona` that contains a key of the same name. In this example, I decide I am able to deal with a request for an outbound URL key named `legacyURL` and if its value is one of the URLs passed to the constructor. I could be more specific an `controller` and `action` values, but for a simple example, this is sufficient.

If I get a match, I create a new instance of `VirtualPathData`, passing in a reference to the current object outbound URL. I have used the `Content` method of the `UrlHelper` class to convert the application-relative U that can be passed to browsers. The routing system prepends an additional / to the URL, so I must take care to rem leading character from the generated URL.

Creating a Custom Route Handler

I have relied on the `MvcRouteHandler` in my routes because it connects the routing system to the MVC Fran focus of this book. Even so, the routing system lets me define my own route handler by implementing the `IRoute` interface. Listing 16-22 shows the content of the `CustomRouteHandler.cs` class file that I added to the `Infrastructure` folder in the example project.

Listing 16-22. Implementing the IRouteHandler Interface in the CustomRouteHandler.cs File

```
using System.Web;
using System.Web.Routing;

namespace UrlsAndRoutes.Infrastructure {

    public class CustomRouteHandler : IRouteHandler {
         public IHttpHandler GetHttpHandler(RequestContext request
{
            return new CustomHttpHandler();
        }
    }

    public class CustomHttpHandler : IHttpHandler {
        public bool IsReusable {
            get { return false; }
        }
```

eBooks-IT.org

```
            context.Response.Write("Hello");
        }
    }
}
```

The purpose of the `IRouteHandler` interface is to provide a means to generate implementations of the `IHttpHandler` interface, which is responsible for processing requests. In the MVC implementation of these in controllers are found, action methods are invoked, views are rendered, and the results are written to the response. M implementation is a little simpler: it just writes the word `Hello` to the client (not an HTML document containing th just the text).

Note The `IHttpHandler` interface is defined by the ASP.NET platform and is part of the standard request system, which I describe in my *Pro ASP.NET MVC 5 Platform* book, published by Apress in 2014. You don't need the way that ASP.NET handles requests to write an MVC Framework application, but there are facilities to customiz the process that can be useful for advanced or complex applications.

I register the custom route handler in the `RouteConfig.cs` file when I define a route, as shown in Listing 1

Listing 16-23. Using a Custom Routing Handler in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapMvcAttributeRoutes();

    routes.Add(new Route("SayHello", new CustomRouteHandler()));

    routes.Add(new LegacyRoute(
            "~/articles/Windows_3.1_Overview.html",
            "~/old/.NET_1.0_Class_Library"));

    routes.MapRoute("MyRoute", "{controller}/{action}");
      routes.MapRoute("MyOtherRoute", "App/{action}", new { contr
"Home" });
}
...
```

When I request the URL `/SayHello`, the custom route handler is used to process the request. Figure 16-5 sho result.

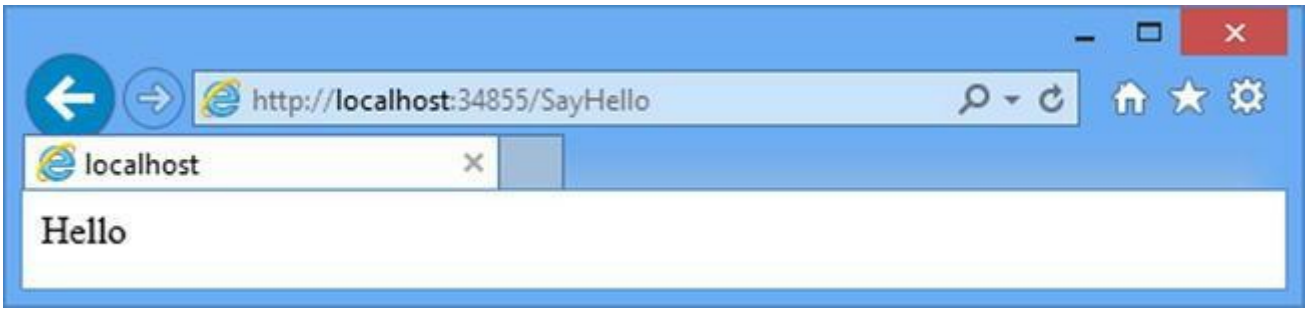


Figure 16-5. Using a custom request handler

Implementing custom route handling means taking on responsibility for functions that are usually handled for you, controller and action resolution. But it does give you incredible freedom: you can co-opt some parts of the MVC Fra ignore others, or even implement an entirely new architectural pattern.

Working with Areas

eBooks-IT.org

nal seg
application, such as administration, billing, customer support, and so on. This is useful in a large project, where havi
of folders for all of the controllers, views, and models, and it can become difficult to manage.

Each MVC area has its own folder structure, allowing you to keep everything separate. This makes it more obviou
project elements relate to each functional area of the application, helping multiple developers to work on the project w
colliding with one another. Areas are supported largely through the routing system, which is why I have chosen to c
feature alongside URLs and routes. In this section, I will show you how to set up and use areas in your MVC project

Creating an Area

To add an area to the example MVC application, right-click the `UrlsAndRoutes` project item in the `Soluti`
`Explorer` window and select `Add` ➤ `Area` from the pop-up menu. Visual Studio will prompt you for the nan
as shown in Figure 16-6. In this case, I have specified an area called `Admin`. This is a pretty common area to crea
many Web applications need to separate the customer-facing and administration functions. Click the `Add` button to
area.

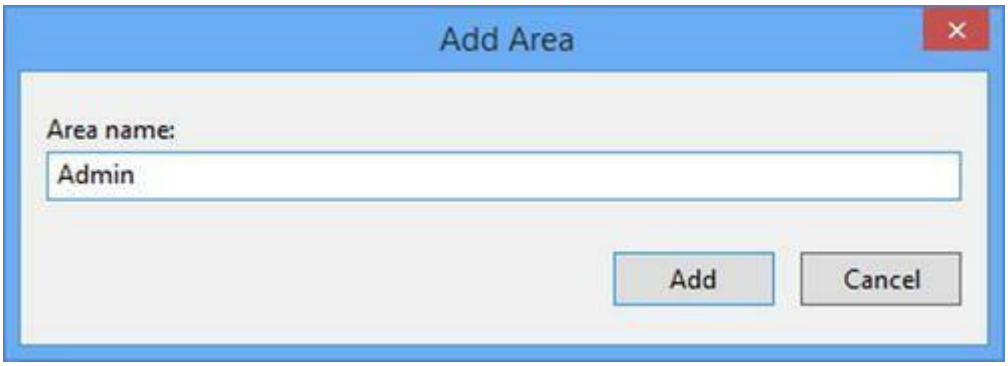


Figure 16-6. Adding an area to an MVC application

After you click `Add`, Visual Studio will add an `Areas` folder to the project. This contains a folder called `Admi`
represents the area that I just created. If I were to create additional areas, other folders would be created here.

Inside the `Areas/Admin` folder, you will see a mini-MVC project. There are folders called `Controllers`
and `Views`. The first two are empty, but the `Views` folder contains a `Shared` folder (and a `web.config`
configures the view engine, but I am not interested in the view engine until Chapter 20).

The `Areas` folder also contains a file called `AdminAreaRegistration.cs`, which defines the
`AdminAreaRegistration` class, as shown in Listing 16-24.

Listing 16-24. The Contents of the AdminAreaRegistration.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Areas.Admin {
    public class AdminAreaRegistration : AreaRegistration {

        public override string AreaName {
            get {
                return "Admin";
            }
        }

        public override void RegisterArea(AreaRegistrationContext
{
            context.MapRoute(
                "Admin_default",
                "Admin/{controller}/{action}/{id}",
```

eBooks-IT.org

```
            );
        }
    }
}
```

The interesting part of this class is the `RegisterArea` method. As you can see from the listing, this method route with the URL pattern `Admin/{controller}/{action}/{id}`. I can define additional routes in which will be unique to this area.

Caution If you assign names to your routes, you must ensure that they are unique across the entire application a the area for which they are intended.

I do not need to take any action to make sure that this registration method is called. Visual Studio added a statemen `Global.asax` file that takes care of setting up areas when it created the project, which you can see in Listing 1

Listing 16-25. Area Registration Setup in the Global.asax File

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

namespace UrlsAndRoutes {
    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

The call to the static `AreaRegistration.RegisterAllAreas` method causes the MVC Framewo through all of the classes in the application, find those that are derived from the `AreaRegistration` class an `RegisterArea` method on each of them.

Caution Do not change the order of the statements related to routing in the `Application_Start` metho `RegisterRoutes` before `AreaRegistration.RegisterAllAreas` is called, then your routes defined before the area routes. Given that routes are evaluated in order, this will mean that requests for area controlle be matched against the wrong routes.

The `AreaRegistrationContext` class that is passed to each area's `RegisterArea` method exp `MapRoute` methods that the area can use to register routes in the same way as your main application does in the `RegisterRoutes` method of `Global.asax`.

Note The `MapRoute` methods in the `AreaRegistrationContext` class automatically limit the rou register to the namespace that contains the controllers for the area. This means that when you create a controller in a must leave it in its default namespace; otherwise, the routing system will not be able to find it.

Populating an Area

You can create controllers, views, and models in an area just as you have seen in previous examples. To create a con click the `Controllers` folder within the `Admin` area and select `Add` ➤ `Controller` from the pop-up m `MVC 5 Controller - Empty` from the list of options, click the `Add` button, set the controller name and `Add` button to create the new controller class.

eBooks-IT.org

min a

see the contents of the Areas/Admin/Controllers/HomeController.cs file in Listing 16-26.

Listing 16-26. The Contents of the Areas/Admin/Controllers/HomeController.cs File

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

namespace UrlsAndRoutes.Areas.Admin.Controllers {
    public class HomeController : Controller {

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

I am not going to change the controller code. Just rendering the default view associated with the `Index` action r enough for this sample. Create the `Areas/Admin/Views/Home` folder, right-click it in the Solution Explore `Add` ➤ `MVC 5 View Page (Razor)` from the menu. Set the name to `Index.cshtml`, click `OK` t file and edit the content to match those shown in Listing 16-27.

Listing 16-27. The Contents of the Areas/Admin/Views/Home/Index.cshtml File

```
@{
    ViewBag.Title = "Index";
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        <h2>Admin Area Index</h2>
    </div>
</body>
</html>
```

The point of all of this is to show that working inside an area is just the same as working in the main part of an M you start the application and navigate to `/Admin/Home/Index`, you will see the view that was added to the A as shown in Figure 16-7.

eBooks-IT.org

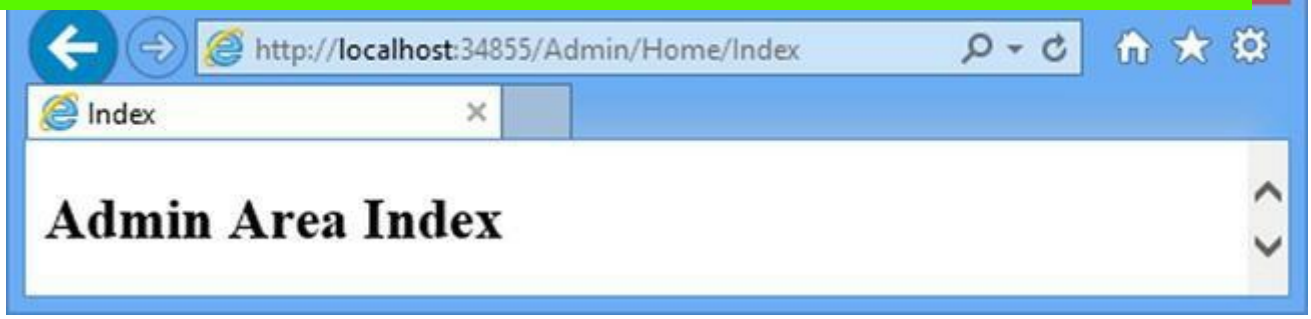


Figure 16-7. Rendering an area view

Resolving the Ambiguous Controller Issue

Okay, so I lied slightly: areas are not quite as self-contained as they might be. If you navigate to the `/Home/Inde` will see the error shown in Figure 16-8.

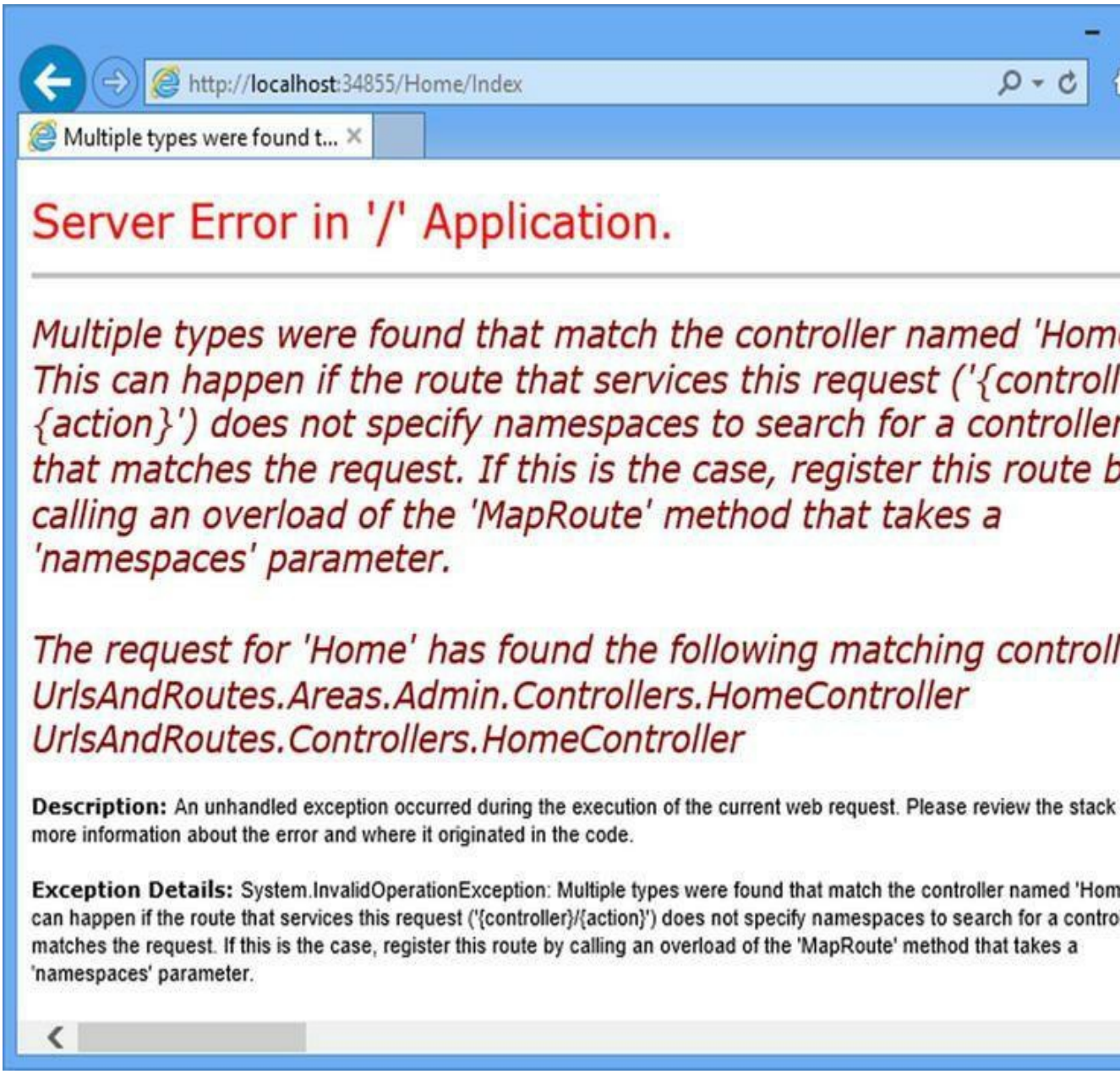


Figure 16-8. The ambiguous controller error

When an area is registered, any routes that it defines are limited to the namespace associated with the area. This is

eBooks-IT.org

UrlsAndRoutes.Areas.Admin.Controllers namespace.

However, routes defined in the `RegisterRoutes` method of `RouteConfig.cs` are not similarly restr
Listing 16-28, as a reminder, I have listed the current routing configuration of the example app.

Listing 16-28. The Routing Configuration for the Example MVC App in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.MapMvcAttributeRoutes();

    routes.Add(new Route("SayHello", new CustomRouteHandler()));

    routes.Add(new LegacyRoute(
        "~/articles/Windows_3.1_Overview.html",
        "~/old/.NET_1.0_Class_Library"));

    routes.MapRoute("MyRoute", "{controller}/{action}");
      routes.MapRoute("MyOtherRoute", "App/{action}", new { contr
"Home" });
}
...
```

The route named `MyRoute` translates the incoming URL from the browser to the `Index` action on the `Home`
At that point, I get an error, because there are no namespace restrictions in place for this route and the MVC Framew
two `HomeController` classes. To resolve this, I need to prioritize the main controller namespace in all of the
might lead to a conflict, as shown in Listing 16-29.

Listing 16-29. Resolving the Area Namespace Conflict in the RouteConfig.cs File

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.MapMvcAttributeRoutes();

    routes.Add(new Route("SayHello", new CustomRouteHandler()));

    routes.Add(new LegacyRoute(
            "~/articles/Windows_3.1_Overview.html",
            "~/old/.NET_1.0_Class_Library"));

    routes.MapRoute("MyRoute", "{controller}/{action}", null,
        new[] {"UrlsAndRoutes.Controllers"});
      routes.MapRoute("MyOtherRoute", "App/{action}", new { contr
"Home" },
        new[] { "UrlsAndRoutes.Controllers" });
}
...
```

This change ensures that the controllers in the main project are given priority in resolving requests. Of course, if y
give preference to the controllers in an area, you can do that instead.

Creating Areas with Attributes

eBooks-IT.org

ave ass

action methods in the Customer controller to a new area called Services in Listing 16-30.

Listing 16-30. Creating an Area Using an Attribute in the CustomerController.cs File

```
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    [RouteArea("Services")]
    [RoutePrefix("Users")]
    public class CustomerController : Controller {

        [Route("~/Test")]
        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        [Route("Add/{user}/{id:int}", Name="AddRoute")]
        public string Create(string user, int id) {
              return string.Format("Create Method - User: {0}, I
user, id);
        }

        [Route("Add/{user}/{password}")]
        public string ChangePass(string user, string password) {
              return string.Format("ChangePass Method - User: {0
{1}",
                user, password);
        }

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

The RouteArea attribute moves all of the routes defined by the Route attribute into the specified area. The attribute combined with the RoutePrefix attribute means that to reach the Create action method, for exam create a URL like this:

http://localhost:34855/Services/Users/Add/Adam/100

The RouteArea attribute doesn't affect routes that are defined by the Route attribute but that start with ~/ for example, that I continue to reach the Index action method with this URL:

http://localhost:34855/Test/

The RouteArea has no effect on action methods to which the Route attribute has not been defined, which the routing for the List action method is determined by the RouteConfig.cs file and not attribute-based ro

Generating Links to Actions in Areas

eBooks-IT.org

urrent r

to. The MVC Framework detects that a request relates to a particular area and ensures that outbound URL generation match only among routes defined for that area. For example, adding a call to the `Html.ActionLink` helper fr the `Admin` area like this:

```
...
@Html.ActionLink("Click me", "About")
...
```

will generate the following HTML:

```
<a href="/Admin/Home/About">Click me</a>
```

To create a link to an action in a *different* area, or no area at all, you must create a variable called `area` and use i the name of the area you want, like this:

```
...
@Html.ActionLink("Click me to go to another area", "Index", new
"Support" })
...
```

It is for this reason that `area` is reserved from use as a segment variable name. The HTML generated by this cal (assuming that you created an area called `Support` that has a suitable route defined):

```
<a href="/Support/Home">Click me to go to another area</a>
```

If you want to link to an action on one of the top-level controllers (a controller in the `/Controllers` folder), should specify the area as an empty string, like this:

```
...
@Html.ActionLink("Click me to go to another area", "Index", new
"" })
...
```

Routing Requests for Disk Files

Not all of the requests for an MVC application are for controllers and actions. Most applications need a way to serve as images, static HTML files, JavaScript libraries, and so on. As a demonstration, I created a `Content` folder and called `StaticContent.html` to it using the `HTML Page` item template. Listing 16-31 shows the content HTML file.

Listing 16-31. The Contents of the StaticContent.html File

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>Static HTML Content</title></head>
    <body>
        This is the static html file (~/Content/StaticContent.html
    </body>
</html>
```

The routing system provides integrated support for serving such content. If you start the application and request t `/Content/StaticContent.html`, you will see the contents of this simple HTML file displayed in the b shown in Figure 16-9.

eBooks-IT.org

Figure 16-9. Requesting the static content file

By default, the routing system checks to see if a URL matches a disk file *before* evaluating the application's routes why I didn't have to add a route to get the result shown in Figure 16-9.

If there is a match between the requested URL and a disk on the file, then the disk file is served and the routes def application are never used. This behavior can be reversed so that the routes are evaluated before disk files are checke the `RouteExistingFiles` property of the `RouteCollection` to `true`, as shown in Listing 16-32.

Listing 16-32. Enabling Route Evaluation Before File-Checking in the RouteConfig.cs File

```
using System.Web.Mvc;
using System.Web.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)

            routes.RouteExistingFiles = true;

            routes.MapMvcAttributeRoutes();

            routes.Add(new Route("SayHello", new CustomRouteHandle

            routes.Add(new LegacyRoute(
                    "~/articles/Windows_3.1_Overview.html",
                    "~/old/.NET_1.0_Class_Library"));

            routes.MapRoute("MyRoute", "{controller}/{action}", nu
                new[] { "UrlsAndRoutes.Controllers" });
                   routes.MapRoute("MyOtherRoute",  "App/{action}"
controller = "Home" },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}
```

The convention is to place this statement close to the top of the `RegisterRoutes` method, although it will t even if you set it after you have defined your routes.

Configuring the Application Server

Visual Studio uses IIS Express as the application server for MVC application projects. Not only do I have to set the `RouteExistingFiles` property to `true` in the `RegisterRoutes` method, I also have to tell IIS Ex

eBooks-IT.org

First of all, start IIS Express. The easiest way to do this is to start the MVC application from Visual Studio, which the IIS Express icon to appear on the task bar. Right-click on the icon and select `Show All Application` pop-up menu. Click on `UrlsAndRoutes` in the `Site Name` column to display the IIS configuration inform shown in Figure 16-10.

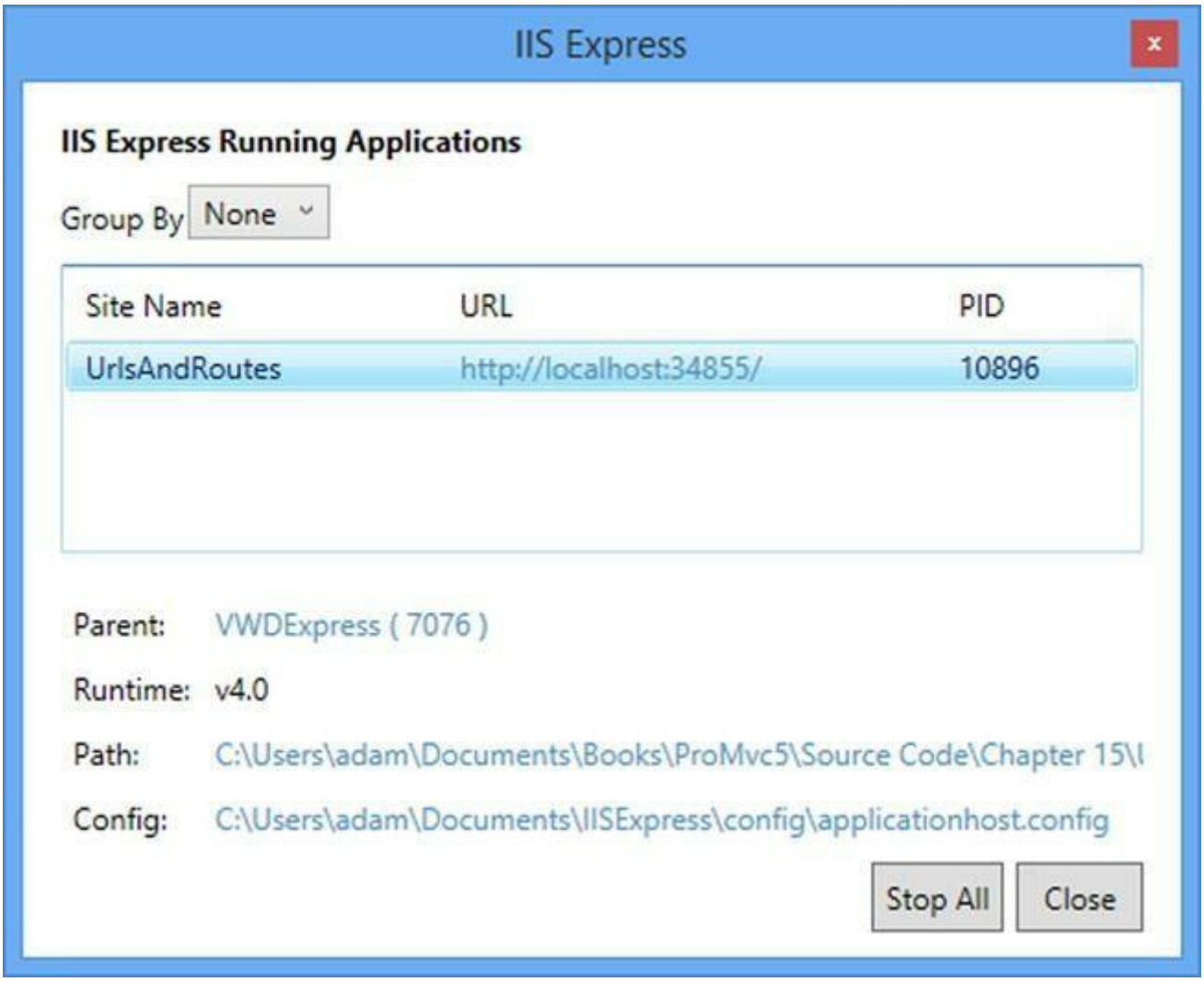


Figure 16-10. The IIS Express configuration information

Click on the `Config` link at the bottom of the window to open the IIS Express configuration file in Visual Studi `Control+F` and search for `UrlRoutingModule-4.0`. There will be an entry found in the `modules` configuration file and you will need to set the `preCondition` attribute to the empty string, like this:

```
...
<add                                                name="UrlRoutingMod
type="System.Web.Routing.UrlRoutingModule"
    preCondition="" />
...
```

Now restart the application in Visual Studio to let the modified settings take effect and navigate to the `/Content/StaticContent.html` URL. Rather than see the contents of the file, you will see the error shown in Figure 16-11. This error occurs because the request for the HTML file has been passed to the MVC routin the route that matches the URL directs the request to the `Content` controller, which does not exist.

eBooks-IT.org

Figure 16-11. Requesting a static content URL which is handled by the routing system

Defining Routes for Disk Files

Once the property has been set to `true`, I can define routes that match URLs that correspond to disk files, such as shown in Listing 16-33.

Listing 16-33. A Route Whose URL Pattern Corresponds to a Disk File in the RouteConfig.cs File

```
using System.Web.Mvc;
using System.Web.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)

            routes.RouteExistingFiles = true;

            routes.MapMvcAttributeRoutes();

            routes.MapRoute("DiskFile", "Content/StaticContent.htm
                new {
                    controller = "Customer",
                    action = "List",
                });
```

eBooks-IT.org

```
            routes.Add(new Route("SayHello", new CustomRouteHandle

            routes.Add(new LegacyRoute(
                    "~/articles/Windows_3.1_Overview.html",
                    "~/old/.NET_1.0_Class_Library"));

            routes.MapRoute("MyRoute", "{controller}/{action}", nu
                new[] { "UrlsAndRoutes.Controllers" });
                    routes.MapRoute("MyOtherRoute",  "App/{action}"
controller = "Home" },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}
```

This route maps requests for the URL `Content/StaticContent.html` to the `List` action of the C controller. You can see the URL mapping at work in Figure 16-12, which I created by starting the app and navigating `/Content/StaticContent.html` URL again.

Figure 16-12. Intercepting a request for a disk file using a route

Tip Your browser may cache the old response, especially if you are using the browser link feature I described in If this happens, reload the web page and you should see the response shown in the figure.

Routing requests intended for disk files requires careful thought, not least because URL patterns will match these k as eagerly as any other. For example, as you saw in the previous section, a request for `/Content/StaticContent.html` will be matched by a URL pattern such as `{controller}/{a` Unless you are careful, you can end up with some exceptionally strange results and reduced performance. So, enabli is a last resort.

Bypassing the Routing System

Setting the `RouteExistingFiles` property, which I demonstrated in the previous section, makes the routin more inclusive. Requests that would normally bypass the routing system are now evaluated against the routes the app defines.

The counterpart to this feature is the ability to make the routing system *less* inclusive and prevent URLs from being against routes. This is done using the `IgnoreRoute` method of the `RouteCollection` class, as shown 34.

Listing 16-34. Using the IgnoreRoute Method in the RouteConfig.cs File

```
using System.Web.Mvc;
using System.Web.Routing;
using UrlsAndRoutes.Infrastructure;
```

eBooks-IT.org

```
namespace UrlsAndRoutes {
    public class RouteConfig {

        public static void RegisterRoutes(RouteCollection routes)

            routes.RouteExistingFiles = true;

            routes.MapMvcAttributeRoutes();

            routes.IgnoreRoute("Content/{filename}.html");

            routes.Add(new Route("SayHello", new CustomRouteHandle

            routes.Add(new LegacyRoute(
                    "~/articles/Windows_3.1_Overview.html",
                    "~/old/.NET_1.0_Class_Library"));

            routes.MapRoute("MyRoute", "{controller}/{action}", nu
                new[] { "UrlsAndRoutes.Controllers" });
                   routes.MapRoute("MyOtherRoute",  "App/{action}"
controller = "Home" },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}
```

You can use segment variables like `{filename}` to match a range of URLs. In this case, the URL pattern will two-segment URL where the first segment is `Content` and the second content has the `.html` extension.

The `IgnoreRoute` method creates an entry in the `RouteCollection` where the route handler is an in `StopRoutingHandler` class, rather than `MvcRouteHandler`. The routing system is hard-coded to r handler. If the URL pattern passed to the `IgnoreRoute` method matches, then no subsequent routes will be eva when a regular route is matched. It follows, therefore, that the location of the call to the `IgnoreRoute` method If you start the app and navigate to the `/Content/StaticContent.html` URL again, you will see the c HTML file because the `StopRoutingHandler` object is processed before any other route which might have URL.

URL Schema Best Practices

After all of this, you may be left wondering where to start in designing your own URL schema. You could just accep schema that Visual Studio generates for you, but there are benefits in giving your schema some thought. In recent ye design of an application's URLs have been taken increasingly seriously and a few important design principles have en follow these design patterns, you will improve the usability, compatibility, and search-engine rankings of your applica

Make Your URLs Clean and Human-Friendly

Users notice the URLs in your applications. If you do not agree, then just think back to the last time you tried to send Amazon URL. Here is the URL for an earlier edition of this book:

http://www.amazon.com/Pro-ASP-NET-MVC-Professional-Apress/dp/1430242361/ref=la_B001IU0SNK_1_5?ie=UTF8&qid=1349978167&

It is bad enough sending someone such a URL by e-mail, but try reading this over the phone. When I needed to do I ended up quoting the ISBN number and asking the caller to look it up for himself. It would be nice if I could access with a URL like this:

That is the kind of URL that I *could* read out over the phone and that doesn't look like I dropped something on the while composing an e-mail message.

Note To be clear, I have only the highest respect for Amazon, who sells more of my books than everyone else c know for a fact that each and every member of the Amazon team is a strikingly intelligent and beautiful person. Not would be so petty as to stop selling my books over something so minor as criticism of their URL format. I love Ama Amazon. I just wish they would fix their URLs.

Here are some simple guidelines to make friendly URLs:

- Design URLs to describe their content, not the implementation details of your appl `/Articles/AnnualReport` rather `/Website_v2/CachedContentServer/FromCache/AnnualReport`.
- Prefer content titles over ID numbers. Use `/Articles/AnnualReport` rather than `/Article` you must use an ID number (to distinguish items with identical titles, or to avoid the extra database query nee item by its title), then use both (`/Articles/2392/AnnualReport`). It takes longer to type, but sense to a human and improves search-engine rankings. Your application can just ignore the title and dis matching that ID.
- Do *not* use file name extensions for HTML pages (for example, `.aspx` or `.mvc`), but *do* use them for types (such as `.jpg`, `.pdf`, and `.zip`). Web browsers do not care about file name extensions if you type appropriately, but humans still expect PDF files to end with `.pdf`.
- Create a sense of hierarchy (for example, `/Products/Menswear/Shirts/Red`), so your vis the parent category's URL.
- Be case-insensitive. (Someone might want to type in the URL from a printed page.) The ASP.NET routing sy insensitive by default.
- Avoid symbols, codes, and character sequences. If you want a word separator, use a dash (as in `/m article`). Underscores are unfriendly, and URL-encoded spaces are bizarre (`/my+great+ar` disgusting (`/my%20great%20article`).
- Do not change URLs. Broken links equal lost business. When you do change URLs, continue to support schema for as long as possible via permanent (301) redirections.
- Be consistent. Adopt one URL format across your entire application.

URLs should be short, easy to type, hackable (human-editable), and persistent, and they should visualize site struct Nielsen, usability guru, expands on this topic at http://www.useit.com/alertbox/990321.ht Berners-Lee, inventor of the Web, offers similar advice. (See http://www.w3.org/Provider/Styl

GET and POST: Pick the Right One

The rule of thumb is that `GET` requests should be used for all read-only information retrieval, while `POST` requests used for any operation that changes the application state. In standards-compliance terms, `GET` requests are for *safe* (having no side effects besides information retrieval), and `POST` requests are for *unsafe* interactions (making a deci changing something). These conventions are set by the World Wide Web Consortium (W3C), at http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

`GET` requests are *addressable*: all the information is contained in the URL, so it's possible to bookmark and link to addresses.

Do not use `GET` requests for operations that change state. Many Web developers learned this the hard way in 200 Google Web Accelerator was released to the public. This application pre-fetched all the content linked from each pag legal within the HTTP because `GET` requests should be safe. Unfortunately, many Web developers had ignored the H conventions and placed simple links to "delete item" or "add to shopping cart" in their applications. Chaos ensued.

One company believed their content management system was the target of repeated hostile attacks, because all the kept getting deleted. They later discovered that a search-engine crawler had hit upon the URL of an administrative pa

Summary

In this chapter, I have shown you the advanced features of the MVC Framework routing system, showing you how outgoing links and URLs, and how to customize the routing system. Along the way, I introduced the concept of area my views on how to create a useful and meaningful URL schema. In the next chapter, I turn to controllers and actio the heart of the MVC Framework. I explain how these work in detail and show you how to use them to get the best application.

eBooks-IT.org

Controllers and Actions

Every request that comes to your application is handled by a controller. The controller is free to handle the request ar fit, as long as it doesn't stray into the areas of responsibility that belong to the model and view. This means that cont contain or store data, nor do they generate user interfaces.

In the ASP.NET MVC Framework, controllers are .NET classes that contain the logic required to handle a request 3, I explained that the role of the controller is to encapsulate your application logic. This means that controllers are re processing incoming requests, performing operations on the domain model, and selecting views to render to the user chapter, I show you how controllers are implemented and the different ways that you can use controllers to receive a output. Table 17-1 provides the summary for this chapter.

Table 17-1. *Chapter Summary*

Problem	Solution
Create a controller	Implement the `IController` interface or derive from the `Controll` class
Get information about a request	Use the context objects and properties or define action method parameters
Generate a response from controller that directly implements the `IController` interface	Use the `HttpResponse` context object
Generate a response from a controller derived from the `Controller` class	Use an action result
Tell the MVC Framework to render a view	Use a `ViewResult`
Pass data from the controller to the view	Use a view model object or the view bag
Redirect the browser to a new URL	Use the `Redirect` or `RedirectPermanent` methods
Redirect the browser to a URL generated by a route	Use the `RedirectToRoute` or `RedirectToRoutePermanent` methods
Redirect the browser to another action method	Use the `RedirectToAction` method
Send an HTTP result code to the browser	Return an `HttpStatusCodeResult` object or use one of the convenience methods such as `HttpNotFound`.

Preparing the Example Project

To prepare for this chapter, I created a new project called `ControllersAndActions` using the `Empty` t checking the option for the MVC folders and references to create a unit test project called `ControllersAndActions.Tests`. The unit tests that I create in this chapter don't need mock impleme so I don't need to install the Moq package, but I do need to install the MVC package so that my tests have access to controller classes. Enter the following command into the Visual Studio NuGet Package Manager Console:

```
Install-Package Microsoft.Aspnet.Mvc -version 5.0.0 -projectname
     ControllersAndActions.Tests
```

Setting the Start URL

Once you have created the project, select `ControllersAndActions Properties` from the Visual S

eBooks-IT.org

don't have to provide a value, just check the option.

Introducing the Controller

You have seen the use of controllers in almost all of the chapters so far. Now it is time to take a step back and look b scenes.

Creating a Controller with IController

In the MVC Framework, controller classes must implement the `IController` interface from the `System.W` namespace, which I have shown in Listing 17-1.

Listing 17-1. The System.Web.Mvc.IController Interface

```
public interface IController {
    void Execute(RequestContext requestContext);
}
```

Tip I got the definition of this interface by downloading the MVC Framework source code, which is endlessly us figuring out how things work behind the curtain. You can download the source code from `http://aspnet.codeplex.com`.

This is a simple interface. The sole method, `Execute`, is invoked when a request is targeted at the controller cl Framework knows which controller class has been targeted in a request by reading the value of the `controller` generated by the routing data, or through your custom routing classes as described in the Chapters 15 and 16.

You can create controller classes by implementing `IController`, but it is a low-level interface and you must work to get anything useful done. That said, the `IController` interface makes for a useful demonstration of h operate and, to that end, I created a new class file called `BasicController.cs` in the `Controllers` f content shown in Listing 17-2.

Listing 17-2. The Contents of the BasicController.cs File

```
using System.Web.Mvc;
using System.Web.Routing;

namespace ControllersAndActions.Controllers {

    public class BasicController : IController {

        public void Execute(RequestContext requestContext) {

                                                    string    control
(string)requestContext.RouteData.Values["controller"];
                                                        string    acti
(string)requestContext.RouteData.Values["action"];

                  requestContext.HttpContext.Response.Write(
                 string.Format("Controller: {0}, Action: {1}", con
action));
        }
    }
}
```

The `Execute` method of the `IController` interface is passed to a

eBooks-IT.org

that matched it (and led to this controller being invoked to process that request). The `RequestContext` class
properties, which I have described in Table 17-2.

***Table 17-2**. The Properties Defined by the RequestContext Class*

Name	Description
`HttpContext`	Returns an `HttpContextBase` object that describes the current request
`RouteData`	Returns a `RouteData` object that describes the route that matched the request

The `HttpContextBase` object provides access to a set of objects that describe the current request, known
context objects, and which I'll come back to later in the chapter. The `RouteData` object describes the route. I ha
the important `RouteData` properties in Table 17-3.

***Table 17-3**. The Properties Defined by the RouteData Class*

Name	Description
`Route`	Returns the `RouteBase` implementation that matched the route
`RouteHandler`	Returns the `IRouteHandler` that handled the route
`Values`	Returns a collection of segment values, indexed by name

CLASS NAMES THAT END WITH BASE

The MVC Framework relies on the ASP.NET platform to process requests, which makes a lot of sense because
feature-rich and integrates well into the IIS application server. One problem is that the classes that the ASP.NET
uses to provide information about requests are not well-suited to unit testing, a key benefit of using the MVC Fr
Microsoft needed to introduce testability while maintaining compatibility with existing ASP.NET Web Forms app
so introduced the *Base* classes, so-called because they have the same names as core ASP.NET platform classes
the word `Base`. So, for example, the ASP.NET platform provides context information about the current reque
key application services through an `HttpContext` object. The `Base` class counterpart is `HttpConte`
an instance of which is passed to the `Execute` method defined by the `IController` interface (and you
`Base` classes in the examples that follow). The original and the `Base` classes define the same properties and
the Base classes are always abstract, which means that they can easily be used for unit testing.

Sometime you will receive an instance of one of the original ASP.NET classes, such as `HttpContext` but
an MVC-friendly `Base` class, such as `HttpContextBase`. You can do this using one of the `Wrappe`
which have the same name as the original classes plus the word `Wrapper`, such as `HttpContextWra`
wrapper classes are derived from the `Base` classes and have constructors that accept an instance of the origin
this:

```
...
HttpContext myContext = getOriginalObjectFromSomewhere();
HttpContextBase myBase = new HttpContextWrapper(myContext);
...
```

There are original, `Base` and `Wrapper` classes throughout the `System.Web` namespace so that ASP.N
support the MVC Framework and older Web Forms applications seamlessly.

I showed you how to use the `RouteBase` and `IRouteHandler` types to customize the routing system i
In this example, I use the `Values` property to get the values of the `controller` and `action` segment vari
write them to the response.

Note Part of the problem when creating custom controllers is that you don't have access to features like views.
that you have to work at a lower level and is the reason that I write my content directly to the client. The
`HttpContextBase.Response` property returns an `HttpResponseBase` object that allows you to
and add to the response that will be sent to the client. This is another touch-point between the ASP.NET platform an
Framework and one that I describe in depth in the *Pro ASP.NET MVC 5 Framework Platform* book, which will be p

eBooks-IT.org

If you run the application and navigate to /Basic/Index, you can see the output generated by the custom c shown in Figure 17-1.

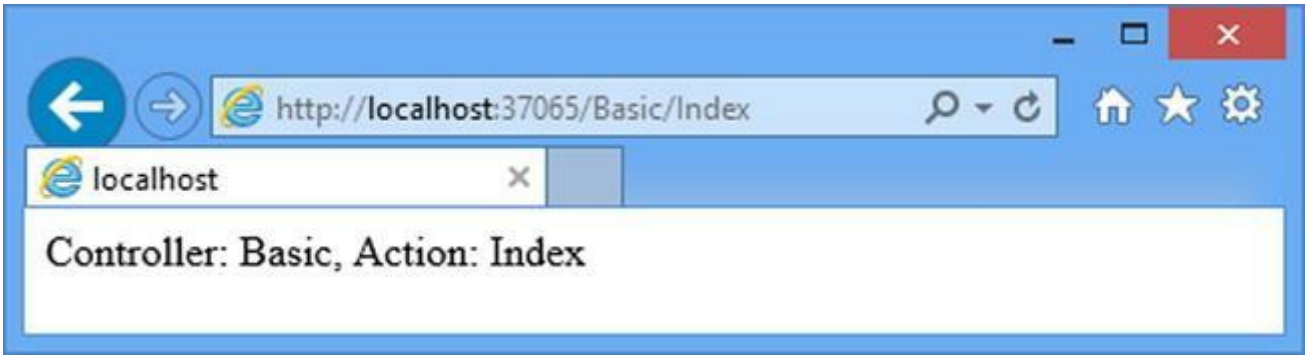


Figure 17-1. A result generated from the BasicController class

Implementing the IController interface allows you to create a class that the MVC Framework recognizes a and sends requests to, without any limitation on how the request is processed and responded to. This is a nice examp shows you how extensible the MVC Framework is, even for key building blocks like controllers, but it would be hard complex application this way.

Creating a Controller by Deriving from the Controller Class

As the previous example suggested, the MVC Framework is endlessly customizable and extensible. You can impleme IController interface to create any kind of request handling and result generation you require. Don't like actio Don't care for rendered views? Then you can just take matters in your own hands and write a better, faster, and mo way of handling requests. Or you can build on the features that the Microsoft MVC Framework team has provided, achieved by deriving your controllers from the System.Web.Mvc.Controller class.

System.Web.Mvc.Controller is the class that provides the request handling support that most MVC will be familiar with. It is the class I have been using in all of the examples in previous chapters. The Controlle provides three key features:

- *Action methods*: A controller's behavior is partitioned into multiple methods (instead of having jus Execute() method). Each action method is exposed on a different URL and is invoked with parame from the incoming request.
- *Action results*: You can return an object describing the result of an action (for example, rendering a view, or re different URL or action method), which is then carried out on your behalf. The separation between *specifyi executing them* simplifies unit testing.
- *Filters*: You can encapsulate reusable behaviors (for example, authentication, as you saw in Chapter 12) as fi tag each behavior onto one or more controllers or action methods by putting an attribute in your source code.

Unless you have a specific requirement in mind, the best way to create controllers is to derive from the Contro and, as you might hope, this is what Visual Studio does when it creates a new controller in response to the Add ➤ menu item. Listing 17-3 shows a simple controller created this way, called DerivedController, generated MVC 5 Controller - Empty option, with some simple changesto set a ViewBag property and selec

Listing 17-3. The ontents of the DerivedController.cs File

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

eBooks-IT.org

```
    public class DerivedController : Controller {

        public ActionResult Index() {
                ViewBag.Message = "Hello from the DerivedControll
method";
            return View("MyView");
        }
    }
}
```

The `Controller` class is also the link to the Razor view system. In the listing, I return the result of the `Vie` passing in the name of the view I want rendered to the client as a parameter. To create this view, create the `Views/Derived` folder, right-click on it and select `Add` ➤ `MVC 5 View Page (Razor)` from the the name to `MyView.cshtml` and click the `OK` button to create the view file. Set the content of the view file to Listing 17-4.

Listing 17-4. The Contents of the MyView.cshtml File

```
@{
    ViewBag.Title = "MyView";
}

<h2>MyView</h2>

Message: @ViewBag.Message
```

If you start the application and navigate to `/Derived/Index`, the action method will be invoked and the vie will be rendered, as shown in Figure 17-2.

Figure 17-2. A result generated from the DerivedController class

The job of a derivation of the `Controller` class is to implement action methods, obtain whatever input is nee process a request, and generate a suitable response. The myriad of ways to do this are covered in the rest of this cha

Receiving Request Data

Controllers frequently need to access data from the incoming request, such as query string values, form values, and parsed from the URL by the routing system. There are three main ways to access that data:

- Extract it from a set of *context objects*.

eBooks-IT.org

- Explicitly invoke the framework's *model binding* feature.

Here, I look at the approaches for getting input for your action methods, focusing on using context objects and ac parameters. In Chapter 24, I cover model binding in depth.

Getting Data from Context Objects

When you create a controller by deriving from the `Controller` base class, you get access to a set of convenier to access information about the request. These properties include `Request`, `Response`, `RouteData`, `HttpContext`, and `Server`. Each provides information about a different aspect of the request. I refer to thes *convenience properties*, because they each retrieve different types of data from the request's `ControllerCon` instance (which can be accessed through the `Controller.ControllerContext` property). I have des of the most commonly used context objects and properties in Table 17-4.

***Table 17-4**. Commonly Used Context Objects and Properties*

Property	Type	Description
`Request.QueryString`	`NameValueCollection`	`GET` variables sent with this request
`Request.Form`	`NameValueCollection`	`POST` variables sent with this request
`Request.Cookies`	`HttpCookieCollection`	Cookies sent by the browser with this request
`Request.HttpMethod`	`string`	The HTTP method (verb, such as `GET` or `POST`) used for t
`Request.Headers`	`NameValueCollection`	The full set of HTTP headers sent with this request
`Request.Url`	`Uri`	The URL requested
`Request.UserHostAddress`	`string`	The IP address of the user making this request
`RouteData.Route`	`RouteBase`	The chosen `RouteTable.Routes` entry for this request
`RouteData.Values`	`RouteValueDictionary`	Active route parameters (either extracted from the URL or def
`HttpContext.Application`	`HttpApplicationStateBase`	Application state store
`HttpContext.Cache`	`Cache`	Application cache store
`HttpContext.Items`	`IDictionary`	State store for the current request
`HttpContext.Session`	`HttpSessionStateBase`	State store for the visitor's session
`User`	`IPrincipal`	Authentication information about the logged-in user
`TempData`	`TempDataDictionary`	Temporary data items stored for the current user

The individual properties that I refer to here—`Request`, `HttpContext`, and so on—provide *context objec* going to go into them in detail in this book (because they are part of the ASP.NET platform), but they provide access useful information and features and are worth exploring. An action method can use any of these context objects to g about the request, as Listing 17-5 demonstrates in the form of a hypothetical action method.

***Listing 17-5**.* An Action Method Using Context Objects to Get Information About a Request

```
...
public ActionResult RenameProduct() {
    // Access various properties from context objects
    string userName = User.Identity.Name;
    string serverName = Server.MachineName;
    string clientIP = Request.UserHostAddress;
    DateTime dateStamp = HttpContext.Timestamp;
      AuditRequest(userName, serverName, clientIP, dateStamp, "
product");
```

eBooks-IT.org

```
    string oldProductName = Request.Form["OldName"];
    string newProductName = Request.Form["NewName"];
    bool result = AttemptProductRename(oldProductName, newProductN

    ViewData["RenameResult"] = result;
    return View("ProductRenamed");
}
...
```

You can explore the vast range of available request context information using IntelliSense (in an action method, typ and browse the pop-up), and the Microsoft Developer Network (look up `System.Web.Mvc.Controller` classes, or `System.Web.Mvc.ControllerContext`).

Using Action Method Parameters

As you've seen in previous chapters, action methods can take parameters. This is a neater way to receive incoming d extracting it manually from context objects, and it makes your action methods easier to read. For example, suppose I action method that uses context objects like this:

```
...
public ActionResult ShowWeatherForecast() {
    string city = (string)RouteData.Values["city"];
    DateTime forDate = DateTime.Parse(Request.Form["forDate"]);
    // ... implement weather forecast here ...
    return View();
}
...
```

I can rewrite it to use parameters, like this:

```
...
public ActionResult ShowWeatherForecast(string city, DateTime forD
    // ... implement weather forecast here ...
    return View();
}
...
```

Not only is this easier to read, but it also helps with unit testing. I can test the action method without needing to m convenience properties of the controller class.

Tip It is worth noting that action methods aren't allowed to have `out` or `ref` parameters. It wouldn't make an they did and the MVC Framework will simply throw an exception if it sees such a parameter.

The MVC Framework will provide values for action method parameters by checking the context objects and prope automatically, including `Request.QueryString`, `Request.Form`, and `RouteData.Values`. T the parameters are treated case-insensitively, so that an action method parameter called `city` can be populated by a `Request.Form["City"]`, for example.

Understanding How Parameters Objects Are Instantiated

The base `Controller` class obtains values for action method parameters using MVC Framework components c *providers* and *model binders*. Value providers represent the set of data items available to your controller. There are bu providers that fetch items from `Request.Form`, `Request.QueryString`, `Request.Files`, and `RouteData.Values`. The values are then passed to model binders that try to map them to the types that your methods require as parameters.

eBooks-IT.org

types. You saw an example of this in Chapter 17 when form posts from administrators were presented to an action m
single `Product` object, even though the individual values were dispersed among the elements of the HTML form.
providers and model binders in depth in Chapter 24.

Understanding Optional and Compulsory Parameters

If the MVC Framework cannot find a value for a reference type parameter (such as a `string` or `object`), the
method will still be called, but using a `null` value for that parameter. If a value cannot be found for a value type pa
as `int` or `double`), then an exception will be thrown, and the action method will *not* be called. Here is another w
about it:

- Value-type parameters are compulsory. To make them optional, either specify a default value (see the nex
change the parameter type to a nullable type (such as `int?` or `DateTime?`), so the MVC Framew
`null` if no value is available.
- Reference-type parameters are optional. To make them compulsory (to ensure that a non-`null` value is
some code to the top of the action method to reject `null` values. For example, if the value equals nul
`ArgumentNullException`.

Specifying Default Parameter Values

If you want to process requests that do not contain values for action method parameters, but you would rather not c
`null` values in your code or have exceptions thrown, you can use the C# optional parameter feature instead. Listin
provides a demonstration.

Listing 17-6. Using the C# Optional Parameter Feature in an Action Method

```
...
public ActionResult Search(string query = "all" , int page = 1) {
    // ...process request...
    return View();
}
...
```

You mark parameters as optional by assigning values when you define them. In the listing, I have provided default
`query` and `page` parameters. The MVC Framework will try to obtain values from the request for these paramete
there are no values available, the defaults I have specified will be used instead.

For the `string` parameter, `query`, this means that I do not need to check for `null` values. If the request I
processing didn't specify a query, then the action method will be called with the string `all`. For the `int` paramete
need to worry about requests resulting in errors when there is no `page` value: the method will be called with the de
`1`. Optional parameters can be used for *literal types*, which are types that you can define without using the `new` key
including `string`, `int`, and `double`.

Caution If a request *does* contain a value for a parameter but it cannot be converted to the correct type (for exa
user gives a nonnumeric string for an `int` parameter), then the framework will pass the default value for that paran
example, 0 for an `int` parameter), and will register the attempted value as a validation error in a special context obje
`ModelState`. Unless you check for validation errors in `ModelState`, you can get into odd situations where
entered bad data into a form, but the request is processed as though the user had not entered any data or had entered
value. See Chapter 25 for details of validation and `ModelState`, which can be used to avoid such problems.

Producing Output

eBooks-IT.org

e bare-
controller by implementing the IController interface directly, I needed to take responsibility for every aspect
a request, including generating the response to the client. If I want to send an HTML response, for example, then I n
and assemble the HTML data and send it to the client using the `Response.Write` method. Similarly, if I want
the user's browser to another URL, I need to call the `Response.Redirect` method and pass the URL I am i
directly. Both of these approaches are shown in Listing 17-7, which shows enhancements to the `BasicContro`

Listing 17-7. Generating Results in the BasicController.cs File

```
using System.Web.Mvc;
using System.Web.Routing;

namespace ControllersAndActions.Controllers {

    public class BasicController : IController {

        public void Execute(RequestContext requestContext) {

                                                    string    control
(string)requestContext.RouteData.Values["controller"];
                                                        string    acti
(string)requestContext.RouteData.Values["action"];

            if (action.ToLower() == "redirect") {
                requestContext.HttpContext.Response.Redirect("/Der
            } else {
                requestContext.HttpContext.Response.Write(
                    string.Format("Controller: {0}, Action: {1}",
                    controller, action));
            }
        }
    }
}
```

You can use the same approach when you have derived your controller from the `Controller` class. The `HttpResponseBase` class that is returned when you read the `requestContext.HttpContext.Response` property in your `Execute` method is available throu `Controller.Response` property, as shown in Listing 17-8, which shows enhancements to the `DerivedController` class.

Listing 17-8. Using the Response Property to Generate Output in the DerivedController.cs File

```
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class DerivedController : Controller {

        public ActionResult Index() {
                ViewBag.Message = "Hello from the DerivedControll
method";
            return View("MyView");
        }
```

eBooks-IT.org

```
            if (Server.MachineName == "TINY") {
                Response.Redirect("/Basic/Index");
            } else {
                                Response.Write("Controller:  Derived,
ProduceOutput");
            }
        }
    }
}
```

The `ProduceOutput` method uses the value of the `Server.MachineName` property to decide what send to the client. (TINY is the name of one of my development machines.) This approach works, but it has a few p

- The controller classes must contain details of HTML or URL structure, which makes the classes harde maintain.
- It is hard to unit test a controller that generates its response directly to the output. You need to implementations of the `Response` object, and then be able to process the output you receive from the con to determine what the output represents. This can mean parsing HTML for keywords, for example, which i and painful process.
- Handling the fine detail of every response this way is tedious and error-prone. Some programmers will lik control that building a raw controller gives, but normal people get frustrated pretty quickly.

Fortunately, the MVC Framework has a nice feature that addresses all of these issues, called *action results*. The fo sections introduce the action result concept and show you the different ways that it can be used to generate respons controllers.

Understanding Action Results

The MVC Framework uses action results to separate *stating intentions* from *executing intentions*. The concept is sim have mastered it, but it takes a while to get your head around the approach at first because there is a little bit of indire on.

Instead of working directly with the `Response` object, action methods return an object derived from the `ActionResult` class that describes what the response from controller will be, such as rendering a view or red another URL or action method. But—and this is where the indirection comes in—you don't generate the response di Instead, you create an `ActionResult` object that the MVC Framework processes to produce the result for you action method has been invoked.

Note The system of action results is an example of the *command pattern*. This pattern describes scenarios where and pass around objects that describe operations to be performed. See http://en.wikipedia.org/wiki/Command_pattern for more details.

When the MVC Framework receives an `ActionResult` object from an action method, it calls the `Execut` method defined by that object. The action result implementation then deals with the `Response` object for you, ge output that corresponds to your intention. To demonstrate how this works, I created an `Infrastructure` fol a new class file called `CustomRedirectResult.cs` to it, which I then used to define the custom `Acti` implementation shown in Listing 17-9.

Listing 17-9. The Contents of the CustomRedirectResult.cs File

```
using System.Web.Mvc;

namespace ControllersAndActions.Infrastructure {
    public class CustomRedirectResult : ActionResult {
```

```
        public override void ExecuteResult(ControllerContext conte
                    string fullUrl = UrlHelper.GenerateContent
context.HttpContext);
            context.HttpContext.Response.Redirect(fullUrl);
        }
    }
}
```

I based this class on the way that the `System.Web.Mvc.RedirectResult` class works. One of the the MVC Framework being open source is that you can see how things work behind the scenes. The `CustomRedirectResult` class is a lot simpler than the MVC equivalent, but is enough for my purposes in

When I create an instance of the `RedirectResult` class, I pass in the URL I want to redirect the user to. T `ExecuteResult` method, which will be executed by the MVC Framework when the action method has finishe `Response` object for the query through the `ControllerContext` object that the framework provides, ar `Redirect` method, which is exactly what I was doing in the bare-bones `IController` implementation in L You can see how I have used the `CustomRedirectResult` class in the `Derived` controller in Listing 1

Listing 17-10. Using the CustomRedirectResult Class in the DerivedController.cs File

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

namespace ControllersAndActions.Controllers {
    public class DerivedController : Controller {

        public ActionResult Index() {
                ViewBag.Message = "Hello from the DerivedControll
method";
            return View("MyView");
        }

        public ActionResult ProduceOutput() {
            if (Server.MachineName == "TINY") {
                return new CustomRedirectResult { Url = "/Basic/In
            } else {
                            Response.Write("Controller: Derived,
ProduceOutput");
                return null;
            }
        }
    }
}
```

Notice that I have had to change the result of the action method to return an `ActionResult`. I return `null` want the MVC Framework to do anything after the action method has been executed, which is what I have done wh return a `CustomRedirectResult` instance.

UNIT TESTING CONTROLLERS AND ACTIONS

Many parts of the MVC Framework are designed to facilitate unit testing, and this is especially true for actions a

- You can test actions and controllers outside a web server. The context objects are accessed through their (such as `HttpRequestBase`), which are easy to mock.
- You do not need to parse any HTML to test the result of an action method. You can inspect the `Acti` object that is returned to ensure that you received the expected result.
- You do not need to simulate client requests. The MVC Framework model binding system allows you to methods that receive input as method parameters. To test an action method, you simply call the action metho provide the parameter values that interest you.

I will show you how to create unit tests for the different kinds of action results throughout this chapter.

Do not forget that unit testing isn't the complete story. Complex behaviors in an application arise when action m called in sequence. Unit testing is best combined with other testing approaches.

Now that you have seen how a custom redirection action result works, I can switch to the equivalent one provided Framework, which has more features and has been thoroughly tested by Microsoft. Listing 17-11 shows the change `Derived` controller.

Listing 17-11. Using the Built-in RedirectResult Object in the DerivedController.cs File

```
...
public ActionResult ProduceOutput() {
    return new RedirectResult("/Basic/Index");
}
...
```

I have removed the conditional statement from the action method, which means that if you start the application an the `/Derived/ProduceOutput` method, your browser will be redirected to the `/Basic/Index` URL

To make action method code simpler, the `Controller` class includes convenience methods for generating di of `ActionResult` objects. So, as an example, I can achieve the effect in Listing 17-11 by returning the result o `Redirect` method, as shown in Listing 17-12.

Listing 17-12. Using a Controller Convenience Method in the DerivedController.cs File

```
...
public ActionResult ProduceOutput() {
    return Redirect("/Basic/Index");
}
...
```

There is nothing in the action result system that is especially complex, but it helps you create with simpler, cleaner consistent code, which is easier to read and easier to unit test.

In the case of a redirection, for example, you can simply check that the action method returns an instance of `RedirectResult` and that the `Url` property contains the target you expect.

The MVC Framework contains a number of built-in action result types, which are shown in Table 17-5. All of the derived from `ActionResult`, and many of them have convenient helper methods in the `Controller` class following sections, I will show you how to use the most important of these result types.

Table 17-5. *Built-in ActionResult Types*

Type	Description	Helper Methods
`ViewResult`	Renders the specified or default view template	`View`
`PartialViewResult`	Renders the specified or default partial view template	`PartialView`

eBooks-IT.org

RedirectToRouteResult	entry, generating a URL according to your routing configuration	RedirectToRoute RedirectToRoutePe
RedirectResult	Issues an HTTP 301 or 302 redirection to a specific URL	Redirect Redire
ContentResult	Returns raw textual data to the browser, optionally setting a content-type header	Content
FileResult	Transmits binary data (such as a file from disk or a byte array in memory) directly to the browser	File
JsonResult	Serializes a .NET object in JSON format and sends it as the response. This kind of response is more typically generated using the Web API feature, which I describe in Chapter 27, but you can see this action type used in Chapter 23.	Json
JavaScriptResult	Sends a snippet of JavaScript source code that should be executed by the browser	JavaScript
HttpUnauthorizedResult	Sets the response HTTP status code to 401 (meaning "not authorized"), which causes the active authentication mechanism (forms authentication or Windows authentication) to ask the visitor to log in	None
HttpNotFoundResult	Returns a HTTP 404—Not found error	HttpNotFound
HttpStatusCodeResult	Returns a specified HTTP code	None
EmptyResult	Does nothing	None

Returning HTML by Rendering a View

The most common kind of response from an action method is to generate HTML and send it to the browser. To dem to render views, I added a controller called Example to the project. You can see the contents of the ExampleController.cs class file in Listing 17-13.

Listing 17-13. The Contents of the ExampleController.cs File

```
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            return View("Homepage");
        }
    }
}
```

When using the action result system, you specify the view that you want the MVC Framework to render using an the ViewResult class. The simplest way to do this is to call the controller's View method, passing the name o an argument. In the listing, I called the View method with an argument of Homepage, which specifies that I wa HomePage.cshtml view to be used.

Note Notice that that the return type for the action method in the listing is ViewResult. The method would work just as well if I had specified the more general ActionResult type. In fact, some MVC programmers wi result of every action method as ActionResult, even when they know it will always return a more specific ty

When the MVC Framework calls the ExecuteResult method of the ViewResult object, a search will view that you have specified. If you are using areas in your project, then the framework will look in the following loc

- /Areas/ <*AreaName*> /Views /<*ControllerName*> / <*ViewName*> .aspx

- /Areas/ <*AreaName*> /Views/Shared/ <*ViewName*> .aspx
- /Areas/ <*AreaName*> /Views/Shared/ <*ViewName*> .ascx
- /Areas/ <*AreaName*> /Views/ <*ControllerName*> / <*ViewName*> .cshtml
- /Areas/ <*AreaName*> /Views/ <*ControllerName*> / <*ViewName*> .vbhtml
- /Areas/ <*AreaName*> /Views/Shared/<ViewName>.cshtml
- /Areas/ <*AreaName*> /Views/Shared/ <*ViewName*> .vbhtml

You can see from the list that the framework looks for views that have been created for the legacy ASPX view en .aspx and .ascx file extensions), even though the MVC Framework uses Razor. This is to preserve compatibil versions of the MVC Framework that used the rendering features from ASP.NET Web Forms.

The framework also looks for C# and Visual Basic .NET Razor templates. (The .cshtml files are the C# ones .vbhtml files are Visual Basic. The Razor syntax is the same in these files, but the code fragments are, as the nar in different languages.) The MVC Framework checks to see if each of these files exists in turn. As soon as it locates uses that view to render the result of the action method.

If you are not using areas, or you are using areas but none of the files in the preceding list have been found, then t continues its search, using the following locations:

- /Views/ <*ControllerName*> / <*ViewName*> .aspx
- /Views/<ControllerName>/<ViewName>.ascx
- /Views/Shared/ <*ViewName*> .aspx
- /Views/Shared/ <*ViewName*> .ascx
- /Views/ <*ControllerName*> / <*ViewName*> .cshtml
- /Views/ <*ControllerName*> / <*ViewName*> .vbhtml
- /Views/Shared/ <*ViewName*> .cshtml
- /Views/Shared/ <*ViewName*> .vbhtml

Once again, as soon as the MVC Framework tests a location and finds a file, then the search stops, and the view t found is used to render the response to the client.

I am not using areas in the example application, so the first place that the framework will look will be /Views/Example/Index.aspx. Notice that the Controller part of the class name is omitted, so t ViewResult in ExampleController leads to a search for a directory called Example.

UNIT TEST: RENDERING A VIEW

To test the view that an action method renders, you can inspect the ViewResult object that it returns. This the same thing (after all, you are not following the process through to check the final HTML that is generated) b enough, as long as you have reasonable confidence that the MVC Framework view system works properly. I ad unit test file called ActionTests.cs to the test project to hold the unit tests for this chapter.

The first situation I want to test is when an action method selects a specific view, like this:

```
...
public ViewResult Index() {
  return View("Homepage");
}
...
```

You can determine which view has been selected by reading the ViewName property of the ViewResult

```
using System.Web.Mvc;
using ControllersAndActions.Controllers;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ControllersAndActions.Tests {
  [TestClass]
  public class ActionTests {

    [TestMethod]
    public void ControllerTest() {

      // Arrange - create the controller
      ExampleController target = new ExampleController();

// Act - call the action method
      ViewResult result = target.Index();

      // Assert - check the result
      Assert.AreEqual("Homepage", result.ViewName);
    }
  }
}
```

A slight variation arises when you are testing an action method that selects the default view, like this:

```
...
public ViewResult Index() {
  return View();
}
...
```

In such situations, you need to accept the empty string (`""`) for the view name, like this:

```
...
Assert.AreEqual("", result.ViewName);
...
```

The empty string is how the `ViewResult` object signals to the Razor view engine that the default view asso the action method has been selected.

The sequence of directories that the MVC Framework searches for a view is another example of convention over You do not need to register your view files with the framework. You just put them in one of a set of known location framework will find them. I can take the convention a step further by omitting the name of the view I want rendered the `View` method, as shown in Listing 17-14.

Listing 17-14. Creating a ViewResult Without Specifying a View in the ExampleController.cs File

```
using System.Web.Mvc;
using System;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
```

eBooks-IT.org

```
        }
    }
}
```

The MVC Framework assumes that I want to render a view that has the same name as the action method. This m
call to the `View` method in Listing 17-14 starts a search for a view called `Index`.

Note The MVC Framework actually gets the name of the action method from the `RouteData.Values["action"]` value, which I explained as part of the routing system in Chapters 15 a
action method name and the routing value will be the same if you are using the built-in routing classes, but this may r
case if you have implemented custom routing classes which do not follow the MVC Framework conventions.

There are a number of overridden versions of the `View` method. They correspond to setting different properties `ViewResult` object that is created. For example, you can override the layout used by a view by explicitly namin
alternative, like this:

```
...
public ViewResult Index() {
    return View("Index", "_AlternateLayoutPage");
}
...
```

SPECIFYING A VIEW BY ITS PATH

The naming convention approach is convenient and simple, but it does limit the views you can render. If you wa
specific view, you can do so by providing an explicit path and bypass the search phase. Here is an example:

```
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

  public class ExampleController : Controller {

    public ViewResult Index() {

      return View("~/Views/Other/Index.cshtml");
    }
  }
}
```

When you specify a view like this, the path must begin with `/` or `~/` and include the file name extension (such
`.cshtml` for Razor views containing C# code).

If you find yourself using this feature, I suggest that you take a moment and ask yourself what you are trying to
you are attempting to render a view that belongs to another controller, then you might be better off redirecting t
action method in that controller (see the "Redirecting to an Action Method" section later in this chapter for an ex
you are trying to work around the naming scheme because it doesn't suit the way you have organized your proj
Chapter 20, which explains how to implement a custom search sequence.

Passing Data from an Action Method to a View

The MVC Framework provides a number of different ways to pass data from an action method to a view, which I d
following sections. I touch on the topic of views, which I cover in depth in Chapter 20. In this chapter, I discuss on
view functionality to demonstrate the controller features of interest.

Providing a View Model Object

Listing 17-15. Specifying a View Model Object in the ExampleController.cs File

```
using System;
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            DateTime date = DateTime.Now;
            return View(date);
        }
    }
}
```

I passed a `DateTime` object as the view model and I can access the object in the view using the Razor `Mode` To demonstrate the `Model` keyword, I added a view called `Index.cshtml` in the `Views/Example` fol contents shown in Listing 17-16.

Listing 17-16. Accessing a View Model in the Index.cshtml File

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

The day is: @(((DateTime)Model).DayOfWeek)
```

This is an *untyped* or *weakly typed* view. The view does not know anything about the view model object, and treat instance of `object`. To get the value of the `DayOfWeek` property, I need to cast the object to an instance of D This works, but produces messy views. I can tidy this up by creating *strongly typed views*, in which the view includ the type of the view model object, as demonstrated in Listing 17-17.

Listing 17-17. Adding Strong Typing to the Index.cshtml File

```
@model DateTime
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

The day is: @Model.DayOfWeek
```

I specified the view model type using the Razor `model` keyword. Notice that I use a lowercase `m` when specifyi type and an uppercase `M` when reading the value. Not only does this help tidy up the view, but Visual Studio support for strongly typed views, as shown in Figure 17-3.

eBooks-IT.org

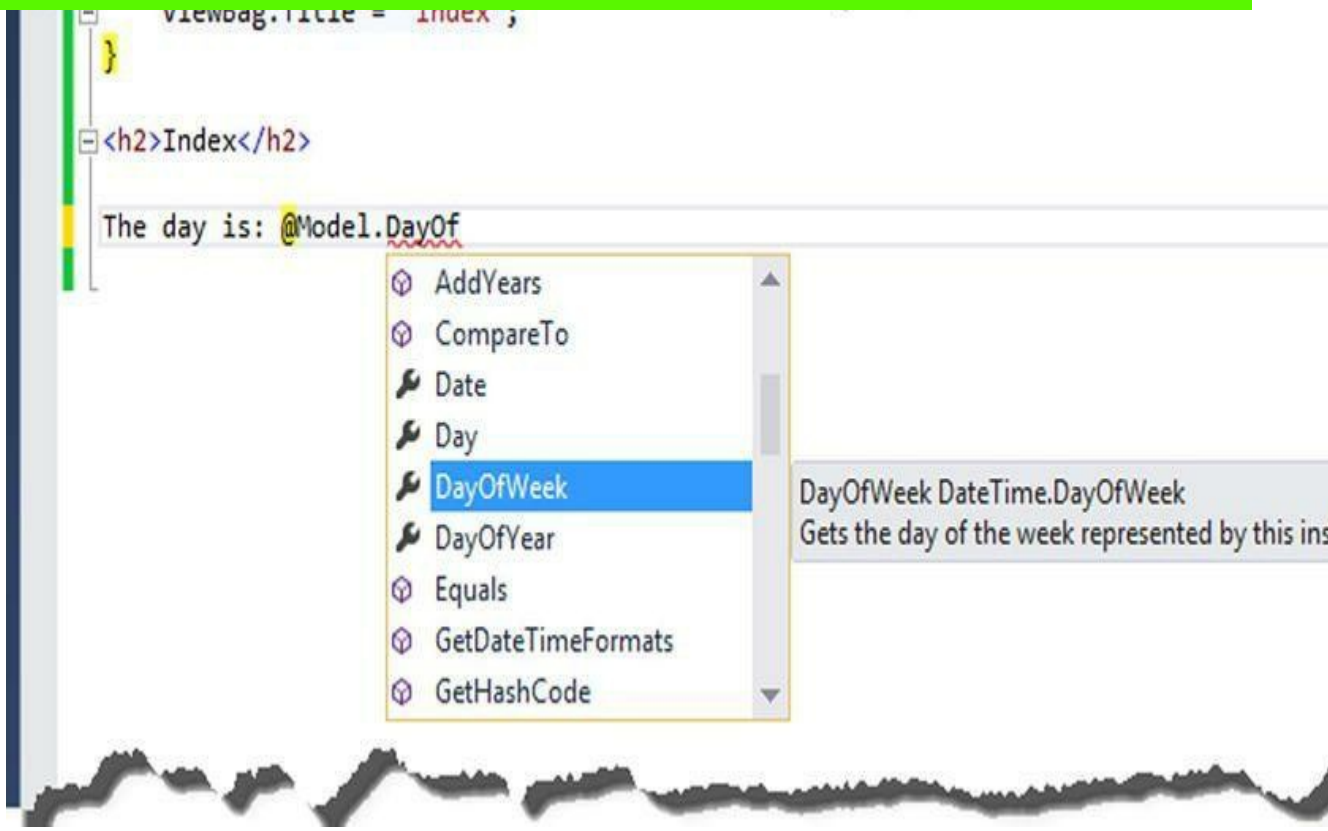


Figure 17-3. IntelliSense support for strongly typed views

UNIT TEST: VIEW MODEL OBJECTS

You can access the view model object passed from the action method to the view through the `ViewResult.ViewData.Model` property. Here is the test for the action method in Listing 17-17. Yo that I have used the `Assert.IsInstanceOfType` method to check that the view model object is an i `DateTime`:

```
...
[TestMethod]
public void ViewSelectionTest() {

  // Arrange - create the controller
  ExampleController target = new ExampleController();

  // Act - call the action method
  ViewResult result = target.Index();

  // Assert - check the result
  Assert.AreEqual("", result.ViewName);
  Assert.IsInstanceOfType(result.ViewData.Model, typeof(System.DateTime));
}
...
```

I had to change the name of the view that I check for to reflect the changes in the action method since the last u showed you, as follows:

```
...
[TestMethod]
public void ControllerTest() {
```

eBooks-IT.org

```
        ExampleController target = new ExampleController();

        // Act - call the action method
        ViewResult result = target.Index();

        // Assert - check the result
        Assert.AreEqual("", result.ViewName);
    }
    ...
```

Passing Data with the View Bag

I introduced the View Bag feature in Chapter 2. This feature allows you to define properties on a dynamic object and in a view. The dynamic object is accessed through the `Controller.ViewBag` property, as demonstrated in

Listing 17-18. Using the View Bag Feature in the ExampleController.cs File

```
using System;
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
    }
}
```

I have defined View Bag properties called `Message` and `Date` simply by assigning values to them. Before this such properties existed, and I made no preparations to create them. To read the data back in the view, I simply get th properties that I set in the action method, as Listing 17-19 shows.

Listing 17-19. Reading Data from the ViewBag in the Index.cshtml File

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

The day is: @ViewBag.Date.DayOfWeek
<p />
The message is: @ViewBag.Message
```

The `ViewBag` has an advantage over using a view model object in that it is easy to send multiple objects to the were restricted to using view models, then I would need to create a new type that had `string` and `DateTime` order to get the same effect.

When working with dynamic objects, you can enter any sequence of method and property calls in the view, like th

```
...
```

eBooks-IT.org

```
...
```

Visual Studio cannot provide IntelliSense support for any dynamic objects, including the `ViewBag`, and errors s won't be revealed until the view is rendered.

UNIT TEST: VIEWBAG

You can read values from the `ViewBag` through the `ViewResult.ViewBag` property. The following is for the action method in Listing 17-18:

```
...
[TestMethod]
public void ControllerTest() {

  // Arrange - create the controller
  ExampleController target = new ExampleController();

  // Act - call the action method
  ViewResult result = target.Index();

  // Assert - check the result
  Assert.AreEqual("Hello", result.ViewBag.Message);
}
...
```

Performing Redirections

A common result from an action method is not to produce any output directly, but to redirect the user's browser to a Most of the time, this URL is another action method in the application that generates the output you want the users to

THE POST/REDIRECT/GET PATTERN

The most frequent use of a redirect is in action methods that process HTTP `POST` requests. As I mentioned in chapter, `POST` requests are used when you want to change the state of an application. If you just return HTML the processing of a request, you run the risk that the user will click the browser's reload button and resubmit th second time, causing unexpected and undesirable results.

To avoid this problem, you can follow the pattern called Post/Redirect/Get. In this pattern, you receive a `POST` process it, and then redirect the browser so that a `GET` request is made by the browser for another URL. `GET` should not modify the state of your application, so any inadvertent resubmissions of this request won't cause ar

When you perform a redirect, you send one of two HTTP codes to the browser:

- Send the HTTP code 302, which is a *temporary* redirection. This is the most frequently used type of redirec using the Post/Redirect/Get pattern, this is the code that you want to send.
- Send the HTTP code 301, which indicates a permanent redirection. This should be used with caution, becau the recipient of the HTTP code not to request the original URL ever again and to use the new URL that is incl the redirection code. If you are in doubt, use temporary redirections; that is, send code 302.

Redirecting to a Literal URL

The most basic way to redirect a browser is to call the `Redirect` method, which returns an instance of the `RedirectResult` class, as shown in Listing 17-20.

```
using System;
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }

        public RedirectResult Redirect() {
            return Redirect("/Example/Index");
        }
    }
}
```

The URL you want to redirect to is expressed as a string and passed as a parameter to the `Redirect` method. `Redirect` method sends a temporary redirection. You can send a permanent redirection using the `RedirectPermanent` method, as shown in Listing 17-21.

Listing 17-21. Permanently Redirecting to a Literal URL in the ExampleController.cs File

```
...
public RedirectResult Redirect() {
    return RedirectPermanent("/Example/Index");
}
...
```

Tip If you prefer, you can use the overloaded version of the `Redirect` method, which takes a `bool` param specifies whether or not a redirection is permanent.

UNIT TEST: LITERAL REDIRECTIONS

Literal redirections are easy to test. You can read the URL and test whether the redirection is permanent or temp the `Url` and `Permanent` properties of the `RedirectResult` class. The following is a test method f redirection shown in Listing 17-21:

```
...
[TestMethod]
public void ControllerTest() {

  // Arrange - create the controller
  ExampleController target = new ExampleController();

  // Act - call the action method
  RedirectResult result = target.Redirect();

  // Assert - check the result
  Assert.IsTrue(result.Permanent);
  Assert.AreEqual("/Example/Index", result.Url);
}
```

Notice that I have updated the test to receive a RedirectResult when I call the action method.

Redirecting to a Routing System URL

If you are redirecting the user to a different part of your application, you need to make sure that the URL you send is your URL schema. The problem with using literal URLs for redirection is that any change in your routing schema me need to go through your code and update the URLs. Fortunately, you can use the routing system to generate valid UR RedirectToRoute method, which creates an instance of the RedirectToRouteResult, as shown 17-22.

Listing 17-22. Redirecting to a Routing System URL in the ExampleController.cs File

```
using System;
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {

            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;

            return View();
        }

        public RedirectToRouteResult Redirect() {
            return RedirectToRoute(new {
                controller = "Example",
                action = "Index",
                ID = "MyID"
            });
        }
    }
}
```

The RedirectToRoute method issues a temporary redirection. Use the RedirectToRoutePerma method for permanent redirections. Both methods take an anonymous type whose properties are then passed to the r to generate a URL. For more details of this process, see the Chapters 15 and 16.

Tip Notice that the RedirectToRoute method returns a RedirectToRouteResult object and updated the action method to return this type.

UNIT TESTING: ROUTED REDIRECTIONS

Here is the unit test for the action method in Listing 17-22:

```
...
[TestMethod]
public void ControllerTest() {

  // Arrange - create the controller
  ExampleController target = new ExampleController();
```

eBooks-IT.org

```
    // Act - call the action method
    RedirectToRouteResult result = target.Redirect();

    // Assert - check the result
    Assert.IsFalse(result.Permanent);
    Assert.AreEqual("Example", result.RouteValues["controller"]);
    Assert.AreEqual("Index", result.RouteValues["action"]);
    Assert.AreEqual("MyID", result.RouteValues["ID"]);
}
...
```

You can see that I have tested the result indirectly by looking at the routing information provided by the `RedirectToRouteResult` object, which means that I don't have to parse a URL.

Redirecting to an Action Method

You can redirect to an action method more elegantly by using the `RedirectToAction` method (for temporar redirections) or the `RedirectToActionPermanent` (for permanent redirections). These are just wrappe `RedirectToRoute` method that lets you specify values for the action method and the controller without need an anonymous type, as shown in Listing 17-23.

Listing 17-23. Redirecting Using the RedirectToAction Method in the ExampleController.cs File

```
...
public RedirectToRouteResult Redirect() {
    return RedirectToAction("Index");
}
...
```

If you just specify an action method, then it is assumed that you are referring to an action method in the current c you want to redirect to another controller, you need to provide the name as a parameter, like this:

```
...
public RedirectToRouteResult Redirect() {
    return RedirectToAction("Index", "Basic");
}
...
```

There are other overloaded versions that you can use to provide additional values for the URL generation. These ar using an anonymous type, which does tend to undermine the purpose of the convenience method, but can still make easier to read.

Note The values that you provide for the action method and controller are not verified before they are passed to system. You are responsible for making sure that the targets you specify actually exist.

PRESERVING DATA ACROSS A REDIRECTION

A redirection causes the browser to submit an entirely new HTTP request, which means that you do not have a details of the original request. If you want to pass data from one request to the next, you can use the Temp Dat

`TempData` is similar to `Session` data, except that `TempData` values are marked for deletion when the and they are removed when the request has been processed. This is an ideal arrangement for short-lived data tha persist across a redirection. Here is a simple example in an action method that uses the `RedirectToActi`

```
...
public RedirectToRouteResult RedirectToRoute() {
```

eBooks-IT.org

```
  TempData["Date"] = DateTime.Now;
  return RedirectToAction("Index");
}
...
```

When this method processes a request, it sets values in the `TempData` collection, and then redirects the user the `Index` action method in the same controller. You can read the `TempData` values back in the target act and then pass them to the view, like this:

```
...
public ViewResult Index() {
  ViewBag.Message = TempData["Message"];
  ViewBag.Date = TempData["Date"];
  return View();
}
...
```

A more direct approach would be to read these values in the view, like this:

```
@{
  ViewBag.Title = "Index";
}
<h2>Index</h2>
The day is: @(((DateTime) TempData["Date"]).DayOfWeek)
<p />
The message is: @TempData["Message"]
```

Reading the values in the view means that you do not need to use the View Bag feature in the action method. Ho must cast the `TempData` results to an appropriate type.

You can get a value from `TempData` without marking it for removal by using the `Peek` method, like this:

```
...
DateTime time = (DateTime)TempData.Peek("Date");
...
```

You can preserve a value that would otherwise be deleted by using the `Keep` method, like this:

```
...
TempData.Keep("Date");
...
```

The `Keep` method doesn't protect a value forever. If the value is read again, it will be marked for removal onc you want to store items so that they won't be removed when the request is processed then use session data inst

Returning Errors and HTTP Codes

The last of the built-in `ActionResult` classes that I will look at can be used to send specific error messages ar result codes to the client. Most applications do not require these features because the MVC Framework will automati these kinds of results. However, they can be useful if you need to take more direct control over the responses sent to

Sending a Specific HTTP Result Code

You can send a specific HTTP status code to the browser using the `HttpStatusCodeResult` class. There controller helper method for this, so you must instantiate the class directly, as shown in Listing 17-24.

```
using System;
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }

        public RedirectToRouteResult Redirect() {
            return RedirectToAction("Index");
        }

        public HttpStatusCodeResult StatusCode() {
                    return new HttpStatusCodeResult(404, "URL ca
serviced");
        }
    }
}
```

The constructor parameters for `HttpStatusCodeResult` are the numeric status code and an optional de
message. In the listing, I returned code 404, which signifies that the requested resource does not exist.

Sending a 404 Result

I can achieve the same effect as Listing 17-24 using the more convenient `HttpNotFoundResult` class, whi
from `HttpStatusCodeResult` and can be created using the controller `HttpNotFound` convenience m
shown in Listing 17-25.

Listing 17-25. Generating a 404 Result in the ExampleController.cs File

```
...
public HttpStatusCodeResult StatusCode() {
    return HttpNotFound();
}
...
```

Sending a 401 Result

Another wrapper class for a specific HTTP status code is the `HttpUnauthorizedResult`, which returns
used to indicate that a request is unauthorized. Listing 17-26 provides a demonstration.

Listing 17-26. Generating a 401 Result in the ExampleController.cs File

```
...
public HttpStatusCodeResult StatusCode() {
    return new HttpUnauthorizedResult();
}
```

There is no helper method in the `Controller` class to create instances of `HttpUnauthorizedResu` must do so directly. The effect of returning an instance of this class is usually to redirect the user to the authenticatio you saw in Chapter 12.

UNIT TEST: HTTP STATUS CODES

The `HttpStatusCodeResult` class follows the pattern you have seen for the other result types, and r available through a set of properties. In this case, the `StatusCode` property returns the numeric HTTP stat the `StatusDescription` property returns the associated descriptive string. The following test method action method in Listing 17-26:

```
...
[TestMethod]
public void ControllerTest() {

  // Arrange - create the controller
  ExampleController target = new ExampleController();

  // Act - call the action method
  HttpStatusCodeResult result = target.StatusCode();

  // Assert - check the result
  Assert.AreEqual(401, result.StatusCode);
}
...
```

Summary

Controllers are one of the key building blocks in the MVC design pattern. In this chapter, you have seen how to creat controllers by implementing the `IController` interface and more convenient controllers by deriving from the `Controller` class. You saw the role that action methods play in MVC Framework controllers and how they eas I showed you the different ways that you can receive input and generate output from an action method and demonst different kinds of `ActionResult` that make this a simple and flexible process. In the next chapter, I go deeper controller infrastructure to show you the *filters* feature, which changes how requests are processed.

eBooks-IT.org

Filters

Filters inject extra logic into MVC Framework request processing. They provide a simple and elegant way to implem
cutting concerns. This term refers to functionality that is used all over an application and doesn't fit neatly into any o
where it would break the separation of concerns pattern. Classic examples of cross-cutting concerns are logging, au
and caching. In this chapter, I show you the different categories of filters that the MVC Framework supports, how t
use custom filters, and how to control their execution. Table 18-1 provides the summary for this chapter.

***Table 18-1**. Chapter Summary*

Problem	Solution	Listing
Inject extra logic into request processing	Apply filters to the controller or its action methods	1–8
Restrict action methods to specific users and groups	Use authorization filters	9–12
Authenticate requests	Use authentication filters	13–19
Process errors when executing requests	Use exception filters	20–30
Inject general-purpose logic into the request handling process	Use action filters	31–35
Inspect or alter the results generated by action methods	Use result filters	36–41
Use filters without attributes	Use the built-in controller methods	42
Define filters that apply to all action methods in the application	Define global filters	43–46
Control the order in which filters are executed	Use the `Order` parameter	47–49
Override global and controller filters for an action method	Use an override filter	50–54

Preparing the Example Project

For this chapter I created a new MVC project called `Filters` using the `Empty` template, checking the option to
MVC folders and references. I created a `Home` controller which has the action method shown in Listing 18-1. I am
on controllers in this chapter, so I return string values from the action methods, rather than `ActionResult` obj
the effect of causing the MVC Framework to send the string value directly to the browser, by passing the Razor view

***Listing 18-1**.* The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace Filters.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "This is the Index action on the Home controlle
        }
    }
}
```

Later in the chapter, I'll be demonstrating how you can use a new MVC feature called *authentication filters*, and f

eBooks-IT.org

underlying ASP.NET platform in this book, but Apress will freely distribute the security chapters from my forthcoming *ASP.NET MVC 5 Platform* book when it is published in 2014. So, in order to demonstrate the authentication filters f is part of the MVC Framework, I am going to use the same approach I adopted in Chapter 12 and define static user c the `Web.config` file, as shown in Listing 18-2.

Listing 18-2. Defining User Credentials in the Web.config File

```
...
<system.web>
  <compilation debug="true" targetFramework="4.5.1" />
  <httpRuntime targetFramework="4.5.1" />
  <authentication mode="Forms">
    <forms loginUrl="~/Account/Login" timeout="2880">
      <credentials passwordFormat="Clear">
        <user name="user"  password="secret"/>
        <user name="admin" password="secret" />
      </credentials>
    </forms>
  </authentication>
</system.web>
...
```

I have defined two users, `user` and `admin`, and assigned them the same password, `secret`, to keep things using `forms` authentication once again and I have used the `loginUrl` attribute to specify that unauthenticated should be redirected to the `/Account/Login` URL. In Listing 18-3, you can see the contents of the `Accour` I added to the project and whose `Login` action will be targeted by the default routing configuration.

Listing 18-3. The Contents of the AccountController.cs File

```
using System.Web.Mvc;
using System.Web.Security;

namespace Filters.Controllers {
    public class AccountController : Controller {

        public ActionResult Login() {
            return View();
        }

        [HttpPost]
        public ActionResult Login(string username, string password
returnUrl) {
                 bool result = FormsAuthentication.Authenticate(u
password);
            if (result) {
                FormsAuthentication.SetAuthCookie(username, false)
                        return Redirect(returnUrl ?? Url.Action(
"Admin"));
            } else {
                    ModelState.AddModelError("", "Incorrect use
password");
                return View();
            }
        }
```

eBooks-IT.org

}

To create the view that will gather candidate credentials from the user, create a `Views/Shared` folder and ri
Select `Add` ➤ `MVC 5 View Page (Razor)`, set the name `Login.cshtml` and click the `OK` buttor
view. Edit the new view to match the content shown in Listing 18-4.

Note I am creating a shared view because I'll be adding a second authentication controller later in the chapter an
reuse the view.

Listing 18-4. The Contents of the Login.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title></title>
</head>
<body>
    @using (Html.BeginForm()) {
        @Html.ValidationSummary()
        <p><label>Username:</label><input name="username" /></p>
                        <p><label>Password:</label><input   name="p
type="password"/></p>
        <input type="submit" value="Log in" />
    }
</body>
</html>
```

Setting the Start URL and Testing the Application

As with all of the example projects, I want Visual Studio to start with the root URL for the application rather than gu
based on the file that is being edited. Select `Filters Properties` from the Visual Studio `Project` men
the `Web` tab and check the `Specific Page` option in the `Start Action` section. You don't have to pr
Just checking the option is enough. If you start the example app, you will get the response shown in Figure 18-1.

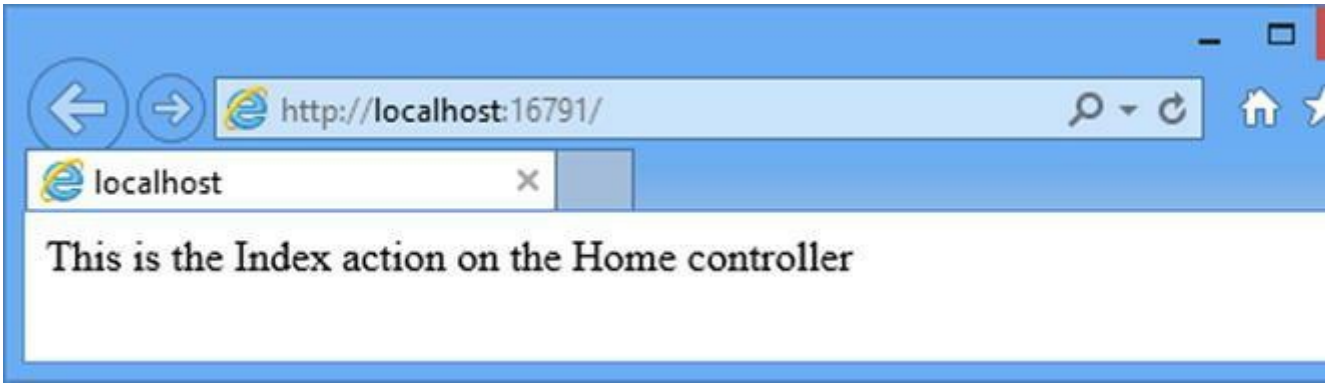


Figure 18-1. Running the example application

You have already seen an example of a filter in Chapter 12, when I applied authorization to the action methods of the administration controller. I wanted the action method to be used only by users who had authenticated themselves, wl me with a choice of approaches. I could have checked the authorization status of the request in each and every actio shown in Listing 18-5.

Listing 18-5. Explicitly Checking Authorization in Action Methods

```
namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {

        // ... instance variables and constructor

        public ViewResult Index() {
            if (!Request.IsAuthenticated) {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }

        public ViewResult Create() {
            if (!Request.IsAuthenticated) {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }

        public ViewResult Edit(int productId) {
            if (!Request.IsAuthenticated) {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }

        // ... other action methods
    }
}
```

You can see that there is a lot of repetition in this approach, which is why I decided to use a filter instead, as show 18-6.

Listing 18-6. Applying a Filter

```
namespace SportsStore.WebUI.Controllers {

    [Authorize]
    public class AdminController : Controller {

        // ... instance variables and constructor

        public ViewResult Index() {
            // ...rest of action method
        }
```

eBooks-IT.org

```
        public ViewResult Create() {
            // ...rest of action method
        }

        public ViewResult Edit(int productId) {
            // ...rest of action method
        }

        // ... other action methods
    }
}
```

Filters are .NET attributes that add extra steps to the request processing pipeline. I used the `Authorize` filter 6, which has the same effect as all of the duplicated checks in Listing 18-5.

Introducing the Filter Types

The MVC Framework supports five different types of filters. Each allows you to introduce logic at different points d processing. The filter types are described in Table 18-2.

Table 18-2. *MVC Framework Filter Types*

Filter Type	Interface	Default Implementation	Description
Authentication	`IAuthenticationFilter`	`N/A`	Runs first, before any other filter action method, but can be run a the authorization filters
Authorization	`IAuthorizationFilter`	`AuthorizeAttribute`	Runs second, after authenticatio before any other filters or the acti
Action	`IActionFilter`	`ActionFilterAttribute`	Runs before and after the action
Result	`IResultFilter`	`ActionFilterAttribute`	Runs before and after the action executed
Exception	`IExceptionFilter`	`HandleErrorAttribute`	Runs only if another filter, the acti or the action result throws an exce

Before the MVC Framework invokes an action, it inspects the method definition to see if it has attributes that impl interfaces listed in Table 18-2. If so, then at the appropriate point in the request handling process, the methods define interfaces are invoked. The framework includes default attribute classes that implement the filter interfaces. I will sh to use these default classes later in this chapter.

Tip MVC 5 introduces a new interface, `IoverrideFilter`, which I describe in the Overriding Filters sec the chapter.

The `ActionFilterAttribute` class implements both the `IActionFilter` and `IResultFil` interfaces. This class is abstract, which forces you to provide an implementation. The `AuthorizeAttribut` `HandleErrorAttribute` classes contain useful features and can be used without creating a derived class.

Applying Filters to Controllers and Action Methods

Filters can be applied to individual action methods or to an entire controller. In Listing 18-6, I applied the `Authori` the `AdminController` class, which has the same effect as applying it to each action method in the controller Listing 18-7.

```
namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {

        // ... instance variables and constructor
        [Authorize]
        public ViewResult Index() {
            // ...rest of action method
        }

        [Authorize]
        public ViewResult Create() {
            // ...rest of action method
        }

        // ... other action methods
    }
}
```

You can apply multiple filters, and mix and match the levels at which they are applied, that is, whether they are app controller or an individual action method. Listing 18-8 shows three different filters in use.

Listing 18-8. Applying Multiple Filters in a Controller Class

```
[Authorize(Roles="trader")] // applies to all actions
public class ExampleController : Controller {

    [ShowMessage]                    // applies to just this action
    [OutputCache(Duration=60)]       // applies to just this action
    public ActionResult Index() {
        // ... action method body
    }
}
```

Some of the filters in this listing take parameters. I will show you how these work as I describe the different kinds

Note If you have defined a custom base class for your controllers, any filters applied to the base class will affect classes.

Using Authorization Filters

Authorization filters are run after the authentication filters, before action filters and before the action method is invok name suggests, these filters enforce your authorization policy, ensuring that action methods can be invoked only by a users.

There is a somewhat involved relationship between the authentication and authorization filters which is easier to ex you understand how authorization filters work. I explain this relationship in the *Using Authentication Filters* section l chapter. Authorization filters implement the `IAuthorizationFilter` interface, which is shown in Listing

Listing 18-9. The IAuthorizationFilter Interface

```
namespace System.Web.Mvc {

    public interface IAuthorizationFilter {
```

eBooks-IT.org

```
        }
}
```

You can, if you are so minded, create a class that implements the `IAuthorizationFilter` interface and own security logic. See the sidebar on why this is a really bad idea.

WARNING: WRITING SECURITY CODE IS DANGEROUS

Programming history is littered with the wreckage of applications whose programmers thought they knew how security code. That's actually a skill that few people possess. There is usually some forgotten wrinkle or untest that leaves a gaping hole in the application's security. If you do not believe me, just Google the term `securit` start reading through the top results.

Wherever possible, I like to use security code that is widely tested and proven. In this case, the MVC Framewo provided a full-featured authorization filter, which can be derived to implement custom authorization policies. I t whenever I can, and I recommend that you do the same. At the least, you can pass some of the blame to Micro your secret application data is spread far and wide on the Internet.

A much safer approach is to create a subclass of the `AuthorizeAttribute` class which takes care of all stuff and makes it easy to write custom authorization code. The best way to demonstrate this is to create a custom f this end, I have added an `Infrastructure` folder to the example project and created a new class file within it `CustomAuthAttribute.cs`. You can see the content of this file in Listing 18-10.

Listing 18-10. The Contents of the CustomAuthAttribute.cs File

```
using System.Web;
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class CustomAuthAttribute : AuthorizeAttribute {
        private bool localAllowed;

        public CustomAuthAttribute(bool allowedParam) {
            localAllowed = allowedParam;
        }

        protected override bool AuthorizeCore(HttpContextBase http
{
            if (httpContext.Request.IsLocal) {
                return localAllowed;
            } else {
                return true;
            }
        }
    }
}
```

This is a simple authorization filter. It allows you to prevent access to local requests (a local request is one where and the application server are running on the same device, such as your development PC).

I have used the simplest approach to creating an authorization filter, which is to subclass the `AuthorizeAtt` class and then override the `AuthorizeCore` method. This ensures that I benefit from the features built in to `AuthorizeAttribute`. The constructor for the filter takes a `bool` value, indicating whether local request permitted.

The interesting part of the filter class is the implementation of the `AuthorizeCore` method, which is how th Framework checks to see if the filter will authorize access for a request. The argument to this method is an `HttpContextBase` object, through which I can get information about the request being processed. By taking

eBooks-IT.org

true from the AuthorizeCore method if I want to authorize a request and return false if I do not.

KEEPING AUTHORIZATION ATTRIBUTES SIMPLE

The AuthorizeCore method is passed an HttpContextBase object, which provides access to in about the request, but not about the controller or action method that the authorization attribute has been applied reason that developers implement the IAuthorizationFilter interface directly is to get access to th AuthorizationContext passed to the OnAuthorization method, through which a much wi information can be obtained, including routing details and the current controller and action method.

I do not recommend this approach, and not just because I think writing your own security code is dangerous. A authorization is a cross-cutting concern, building logic into your authorization attributes which is tightly coupled structure of your controllers undermines the separation of concerns and causes testing and maintenance probler authorization attributes simple and focused on authorization based on the request. Let the context of what is bein come from where the attribute is applied.

Applying the Custom Authorization Filter

To use the custom authorization filter, I simply apply an attribute to the action methods or controllers that I want to p illustrated by Listing 18-11, which demonstrates the application of the filter to the Index action method in the Hor

Listing 18-11. Applying a Custom Authorization Filter in the HomeController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [CustomAuth(false)]
        public string Index() {
            return "This is the Index action on the Home controlle
        }
    }
}
```

I have set the constructor argument for the filter to false, which means that local requests will be denied acces Index action method. You can test this by starting the application. The routing configuration will target the Inde method when the root URL is requested by the browser. If the browser making the request is on the machine runnin Studio, then you will see the result in Figure 18-2. The filter has denied authorization for the request and the MVC Fr responded in the only way it knows how: by prompting the user for credentials. Of course, a username or password change the fact that the request is coming from the local machine and there is nothing you can do at this point to get authentication challenge.

However, the filter will authorize the request if you change the constructor argument for the filter to true and re application. (You can't test by making a request from another machine because IIS Express, which runs the applicat configured to reject any connections that are not local.)

eBooks-IT.org

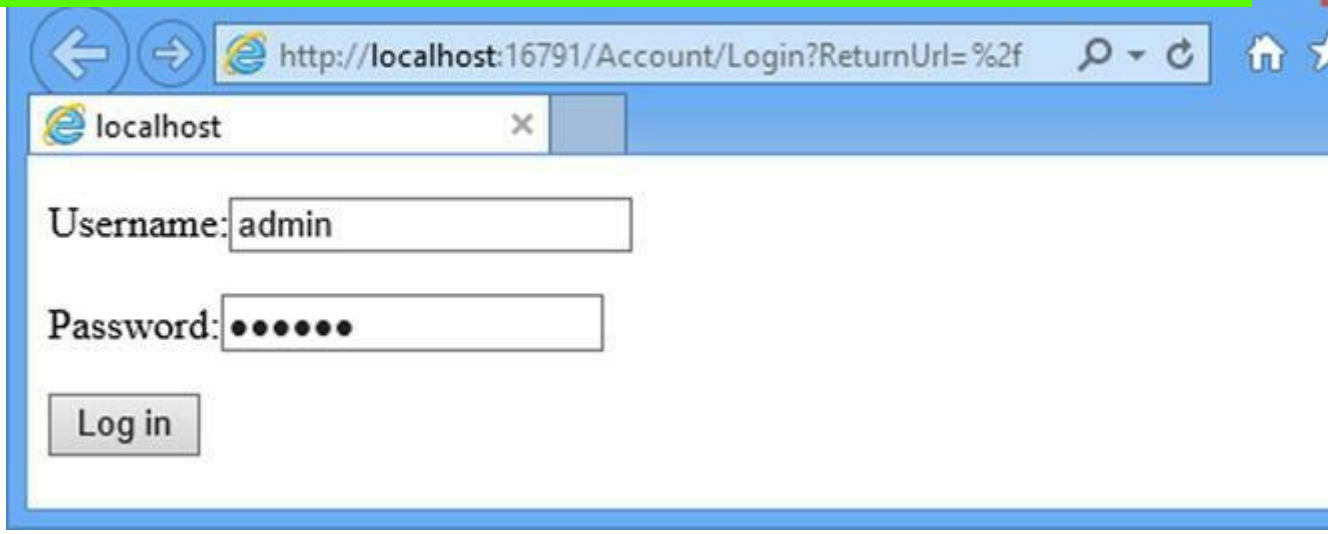


Figure 18-2. Re-prompting for credentials for a local request by the custom authorization filter

Using the Built-in Authorization Filter

Although I used the `AuthorizeAttribute` class as the base for the custom filter, it has its own implementa `AuthorizeCore` method which is useful for performing general purpose authorization tasks. When using the `AuthorizeAttribute` directly, I can specify an authorization policy using two public properties of this clas in Table 18-3.

***Table 18-3**. AuthorizeAttribute Properties*

Name	Type	Description
`Users`	`string`	A comma-separated list of usernames that are allowed to access the action method.
`Roles`	`string`	A comma-separated list of role names. To access the action method, users must be in at least one of these roles.

Listing 18-12 shows how I can use the built-in filter to protect an action method with one of these properties.

***Listing 18-12**.* Using the Built-in Authorization Filter

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controlle
        }
    }
}
```

I have specified that the `admin` user is authorized to invoke the `Index` action method, but there is an implicit c well: the request is authenticated. If I do not specify any users or roles, then any authenticated user can use the actio For most applications, the authorization policy that `AuthorizeAttribute` provides is sufficient. If you war implement something special, you can derive from this class as just I did earlier in the chapter or supplement your co with authentication filters, which I describe the next section.

eBooks-IT.org

Authentication filters are new in MVC version 5 and provide the means to provide fine-grain control over how users authenticated for controllers and actions in an application.

Authentication filters have a relatively complex lifecycle. They are run before any other filter, which lets you defin authentication policy that will be applied before any other type of filter is used. Authentication filters can also be com authorization filters to provide authentication challenges for requests that don't comply to the authorization policy. Au filters are also run after an action method has been executed but before the `ActionResult` is processed. I expl this works and provide some examples along the way.

Understanding the IAuthenticationFilter Interface

Authentication filters implement the `IAuthenticationFilter` interface, which is shown in Listing 18-13.

Listing 18-13. The IAuthenticationFilter Interface

```
namespace System.Web.Mvc.Filters {

    public interface IAuthenticationFilter {
        void OnAuthentication(AuthenticationContext context);
             void  OnAuthenticationChallenge(AuthenticationChalleng
context);
    }
}
```

The `OnAuthenticationChallenge` method is called by the MVC Framework whenever a request ha authentication or authorization policies for an action method. The `OnAuthenticationChallenge` metho `AuthenticationChallengeContext` object, which is derived from the `ControllerContext` described in Chapter 17 and which defines the additional properties shown in Table 18-4.

Table 18-4. *The properties defined by the AuthenticationChallengeContext class*

Name	Description
`ActionDescriptor`	Returns an `ActionDescriptor` that describes the action method to which the filter has been applied
`Result`	Sets an `ActionResult` that expresses the result of the authentication challenge

The most important property is `Result`, because it allows the authentication filter to pass an `ActionResul` MVC Framework, a process known as *short-circuiting* that I will describe shortly. The best way of explaining how a authentication filter works is through an example. To my mind, the most interesting aspect of authentication filters is allow a single controller to define action methods that are authenticated in different ways, so my first step is to add a controller that simulates Google logins. In Listing 18-14, you can see the definition of the `GoogleAccountController`.

Listing 18-14. The Contents of the GoogleAccountController.cs File

```
using System.Web.Mvc;
using System.Web.Security;

namespace Filters.Controllers {
    public class GoogleAccountController : Controller {

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

eBooks-IT.org

```
        public ActionResult Login(string username, string password
returnUrl) {
            if (username.EndsWith("@google.com") && password == "
{
                FormsAuthentication.SetAuthCookie(username, false)
                return Redirect(returnUrl ?? Url.Action("Index", "
            } else {
                    ModelState.AddModelError("", "Incorrect use
password");
                return View();
            }
        }
    }
}
```

I don't want to implement real Google logins because it means delving into the dark world of third-party authentica a topic in its own right. Instead, I have created a terrible hack that will authenticate any user name that ends with `@google.com` as long as it is provided with the password `secret`.

At the moment, my Google authentication controller isn't hooked up to the application, and that's where the auther comes in. I created a new class file called `GoogleAuthAttribute.cs`, shown in Listing 18-15, in the `Infrastructure` folder. The `FilterAttribute` class, from which my `GoogleAuth` filter is deri base for all filter classes.

Listing 18-15. The Contents of the GoogleAuthAttribute.cs File

```
using System;
using System.Web.Mvc;
using System.Web.Mvc.Filters;
using System.Web.Routing;

namespace Filters.Infrastructure {

            public class GoogleAuthAttribute : FilterAt
IAuthenticationFilter {

        public void OnAuthentication(AuthenticationContext context
            // not implemented
        }

                                                        public
OnAuthenticationChallenge(AuthenticationChallengeContext context)
            if (context.Result == null) {
                        context.Result = new RedirectToRouteRe
RouteValueDictionary {
                    {"controller", "GoogleAccount"},
                    {"action", "Login"},
                    {"returnUrl", context.HttpContext.Request.RawU
                });
            }
        }
    }
}
```

My implementation of the `OnAuthenticationChallenge` method checks to see if the `Result` pro `AuthenticationChallengeContext` argument has been set. This allows me to avoid challenging the

eBooks-IT.org

Handling the Final Challenge Request section later in this chapter.

What's important for this section is that I use the `OnAuthenticationChallenge` method to challenge credentials by redirecting their browser to my `GoogleAccount` controller with a `RedirectToRoute`. Authentication filters can use all of the `ActionResult` types that I described in Chapter 17, but the `Control` convenience methods for creating them are not available, which is why I had to use a `RouteValueDiction` specify the segment values so that a route to the challenge action method can be generated.

Implementing the Authentication Check

My authentication filter is ready to challenge users for their fake Google credentials, and now I can wire up the rema behavior. The controller will call the `OnAuthentication` method before running any other kind of filter, pro opportunity to perform a broad authentication check. You don't have to implement the `OnAuthentication` am going to do so in order to check that I am dealing with a Google account.

The `OnAuthentication` method is passed an `AuthenticationContext` object that, like the `AuthenticationChallengeContext` class, is derived from `ControllerContext` and provid all of the information I described in Chapter 17. The `AuthenticationContext` class also defines the prop in Table 18-5.

***Table 18-5**. The properties defined by the AuthenticationContext class*

Name	Description
`ActionDescriptor`	Returns an `ActionDescriptor` that describes the action method to which the filter has been applied
`Principal`	Returns an `IPrincipal` implementation that identifies the current user, if they have already been authentica
`Result`	Sets an `ActionResult` that expresses the result of the authentication check

If the `OnAuthentication` sets a value for the `Result` property of the context object, then the MVC Fr call the `OnAuthenticationChallenge` method. If the `OnAuthenticationChallenge` meth set a value for the `Result` property on its context object, then the one from the `OnAuthentication` meth executed.

I use the `OnAuthentication` method to create a result that reports an authentication error to the user, wh be overridden by the `OnAuthenticationChallenge` method to challenge the user for credentials instead me to be sure that they see a meaningful response, even if no challenge can be issued (although I must admit that I h encounter a situation where this has happened). In Listing 18-16, you can see how I have implemented the `OnAuthentication` method so that it checks that the request has been authenticated by using any Google cr

***Listing 18-16**.* Implementing the OnAuthentication Method in the GoogleAuthAttribute.cs File

```
using System;
using System.Security.Principal;
using System.Web.Mvc;
using System.Web.Mvc.Filters;
using System.Web.Routing;

namespace Filters.Infrastructure {

            public    class    GoogleAuthAttribute    :    FilterAt
IAuthenticationFilter {

        public void OnAuthentication(AuthenticationContext context
            IIdentity ident = context.Principal.Identity;
                                        if    (!ident.IsAuthentica
!ident.Name.EndsWith("@google.com")) {
                context.Result = new HttpUnauthorizedResult();
```

eBooks-IT.org

```
        }

                                                                public
OnAuthenticationChallenge(AuthenticationChallengeContext context)
                        if (context.Result == null || context.Re
HttpUnauthorizedResult) {
                            context.Result = new RedirectToRouteRe
RouteValueDictionary {
                    {"controller", "GoogleAccount"},
                    {"action",  "Login"},
                    {"returnUrl", context.HttpContext.Request.RawU
                });
            }
        }
    }
}
```

My implementation of the `OnAuthentication` method checks to see if the request has been authenticated username that ends with `@google.com`. If the request is not authenticated or the request is authenticated using kind of credential, then I set the `Result` property of the `AuthenticationContext` object to a new `HttpUnauthorizedResult`.

The `HttpUnauthorizedResult` is set as the `Result` value for the `AuthenticationChallengeContext` object that is passed to the `OnAuthenticationChal` method and you can see that I have updated this method to challenge the user when this happens, coordinating the ac two methods in the filter. The next step is to apply the filter to the controller, which you can see in Listing 18-17.

Listing 18-17. Applying the Authentication Filter in the HomeController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controlle
        }

        [GoogleAuth]
        public string List() {
            return "This is the List action on the Home controller
        }
    }
}
```

I have defined a new action method called `List`, which I decorated with the `GoogleAuth` filter. The result to the `Index` method is secured through the built-in support for forms authentication but that access to the `List` method is secured through my custom fake Google authentication system.

You can see the effect by starting the application. By default the browser will target the `Index` action method, w trigger the standard authentication and require you to log in using one of the usernames that I defined in the `Web.c` If you then request the `/Home/List` URL, then your existing credentials will be rejected and you will have to au using a Google username.

You can combine authentication and authorization filters on the same action methods to narrow the scope of your se
The MVC Framework will call the OnAuthentication method of the authentication filter, just as in the prev
and move on to run the authorization filter if the request passes the authentication check. If the request doesn't pass
authorization filter, then the OnAuthenticationChallenge method of the authentication filter will be ca
you can challenge the user for the required credentials. In Listing 18-18, you can see how I have combined the Goo
and Authorize filters to restrict access to the List action in the Home controller.

Listing 18-18. Combining Authentication and Authorization Filters in the HomeController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controlle
        }

        [GoogleAuth]
        [Authorize(Users = "bob@google.com")]
        public string List() {
            return "This is the List action on the Home controller
        }
    }
}
```

The Authorize filter restricts access to the bob@google.com account. If the action method is targete
Google account, then the authentication filter OnAuthenticationChallenge method will be passed an
AuthenticationChallengeContext object whose Result property is set to an instance of the
HttpUnauthorizedResult class (which is why I used the same class in the OnAuthentication

The filters in the Home controller restrict access to the Index method to the user admin, who is authenticat
AccountController, and restrict access to the List method to the bob@google.com user, who
authenticated through the GoogleAccount controller.

Handling the Final Challenge Request

The MVC Framework calls the OnAuthenticationChallenge method one final time after the action m
been executed, but before the ActionResult is returned and executed. This provides authentication filters an
respond to the fact that the action has completed or even alter the result (something that is also possible with result f
describe later in the chapter).

It is for this reason that I check the Result property of the AuthenticationChallengeContex
the OnAuthenticationChallenge method. If I did not, I end up challenging the user for credentials on
which makes little sense given that the action method has already been executed by this point.

The only reason I have found for responding to this last method call is to clear the authentication for the request, v
useful when important action methods require temporarily elevated credentials that you want entered each and every
action is to be executed. In Listing 18-19, you can see how I have implemented this feature.

Listing 18-19. Handling the Final Challenge Call in the GoogleAuthAttribute.cs File

```
using System;
using System.Security.Principal;
```

eBooks-IT.org

```
using System.Web.Mvc.Filters;
using System.Web.Routing;
using System.Web.Security;

namespace Filters.Infrastructure {

            public class GoogleAuthAttribute : FilterAt
IAuthenticationFilter {

        public void OnAuthentication(AuthenticationContext context
            IIdentity ident = context.Principal.Identity;
                                        if (!ident.IsAuthenticat
!ident.Name.EndsWith("@google.com")) {
                context.Result = new HttpUnauthorizedResult();
            }
        }

                                                        public
OnAuthenticationChallenge(AuthenticationChallengeContext context)
                        if (context.Result == null || context.Re
HttpUnauthorizedResult) {
                            context.Result = new RedirectToRouteRe
RouteValueDictionary {
                    {"controller", "GoogleAccount"},
                    {"action", "Login"},
                    {"returnUrl", context.HttpContext.Request.RawU
                });
            } else {
                FormsAuthentication.SignOut();
            }
        }
    }
}
```

You can see the effect by starting the application and requesting the `Home/List` URL. You will be prompted t credentials and you will be able to execute the action method if you authenticate as bob@google.com. But you prompted for credentials again if you reload the browser, essentially targeting the `List` method a second time.

Using Exception Filters

Exception filters are run only if an unhandled exception has been thrown when invoking an action method. The excep come from the following locations:

- Another kind of filter (authorization, action, or result filter)
- The action method itself
- When the action result is executed (see Chapter 17 for details on action results)

Creating an Exception Filter

Exception filters implement the `IExceptionFilter` interface, which is shown in Listing 18-20.

```
namespace System.Web.Mvc {

    public interface IExceptionFilter {
        void OnException(ExceptionContext filterContext);
    }
}
```

The `OnException` method is called when an unhandled exception arises. The parameter for this method is ar `ExceptionContext` object, which is derived from `ControllerContext` and provides a number of u properties that you can use to get information about the request, as shown in Table 18-6.

***Table 18-6**. Useful ControllerContext Properties*

Name	Type	Description
`Controller`	`ControllerBase`	Returns the controller object for this request
`HttpContext`	`HttpContextBase`	Provides access to details of the request and access to the response
`IsChildAction`	`bool`	Returns `true` if this is a child action (see Chapter 20)
`RequestContext`	`RequestContext`	Provides access to the `HttpContext` and the routing data, both of which are availabl properties
`RouteData`	`RouteData`	Returns the routing data for this request

In addition to the properties inherited from the `ControllerContext` class, the `ExceptionContex` defines some additional properties which are useful with dealing with exceptions, as shown in Table 18-7.

***Table 18-7**. Additional ExceptionContext Properties*

Name	Type	Description
`ActionDescriptor`	`ActionDescriptor`	Provides details of the action method
`Result`	`ActionResult`	The result for the action method; a filter can cancel the request by setting this proper non- `null` value
`Exception`	`Exception`	The unhandled exception
`ExceptionHandled`	`bool`	Returns `true` if another filter has marked the exception as handled

The exception that has been thrown is available through the `Exception` property. An exception filter can repo handled the exception by setting the `ExceptionHandled` property to `true`. All of the exception filters appli action are invoked even if this property is set to `true`, so it is good practice to check whether another filter has alr with the problem, to avoid attempting to recover from a problem that another filter has resolved.

Note If none of the exception filters for an action method set the `ExceptionHandled` property to `true` Framework uses the default ASP.NET exception handling procedure which will display the dreaded "yellow screen o

The `Result` property is used by the exception filter to tell the MVC Framework what to do. The two main uses exception filters are to log the exception and to display a message to the user. To demonstrate how this all fits togeth created a new class file called `RangeExceptionAttribute.cs` in the `Infrastructure` folder. T of this file are shown in Listing 18-21.

***Listing 18-21**.* The Contents of the RangeExceptionAttribute.cs File

```
using System;
using System.Web.Mvc;

namespace Filters.Infrastructure {
            public   class   RangeExceptionAttribute   : FilterAttri
```

```
        public void OnException(ExceptionContext filterContext) {
            if (!filterContext.ExceptionHandled &&
                filterContext.Exception is ArgumentOutOfRangeExcep
                    filterContext.Result

RedirectResult("~/Content/RangeErrorPage.html");
                    filterContext.ExceptionHandled = true;
            }
        }
    }
}
```

This exception filter handles `ArgumentOutOfRangeException` instances by redirecting the user's br file called `RangeErrorPage.html` in the `Content` folder. Notice that I have derived the `RangeExceptionAttribute` class from the `FilterAttribute` class, in addition to implementing `IExceptionFilter` interface. In order for a .NET attribute class to be treated as an MVC filter, the class ha implement the `IMvcFilter` interface. You can do this directly, but the easiest way to create a filter is to derive from `FilterAttribute`, which implements the required interface and provides some useful basic features, s handling the default order in which filters are processed (which I will come to later in this chapter).

Applying the Exception Filter

I need to do some groundwork before I can test the exception filter. First, I need to create a `Content` folder in th project and create the `RangeErrorPage.html` file within it. This is the file that I will direct users to when t is handled and you can see the contents of the file in Listing 18-22.

Listing 18-22. The Contents of the RangeErrorPage.html File

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>One of the arguments was out of the expected range.</spa
</body>
</html>
```

Next, I need to add an action method to the `Home` controller which will throw the exception I want to demonstra see the addition in Listing 18-23.

Listing 18-23. Adding a New Action in the HomeController.cs File

```
using System;
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
```

eBooks-IT.org

```
        }

        [GoogleAuth]
        [Authorize(Users = "bob@google.com")]
        public string List() {
            return "This is the List action on the Home controller
        }

        public string RangeTest(int id) {
            if (id > 100) {
                return String.Format("The id value is: {0}", id);
            } else {
                throw new ArgumentOutOfRangeException("id", id, ""
            }
        }
    }
}
```

You can see the default exception handling if you start the application and navigate to the `/Home/RangeTes` The default routing that Visual Studio creates for an MVC project has a segment variable called `id` which will be set this URL, triggering the response shown in Figure 18-3. (See Chapters 15 and 16 for details of routing and URL segm

eBooks-IT.org

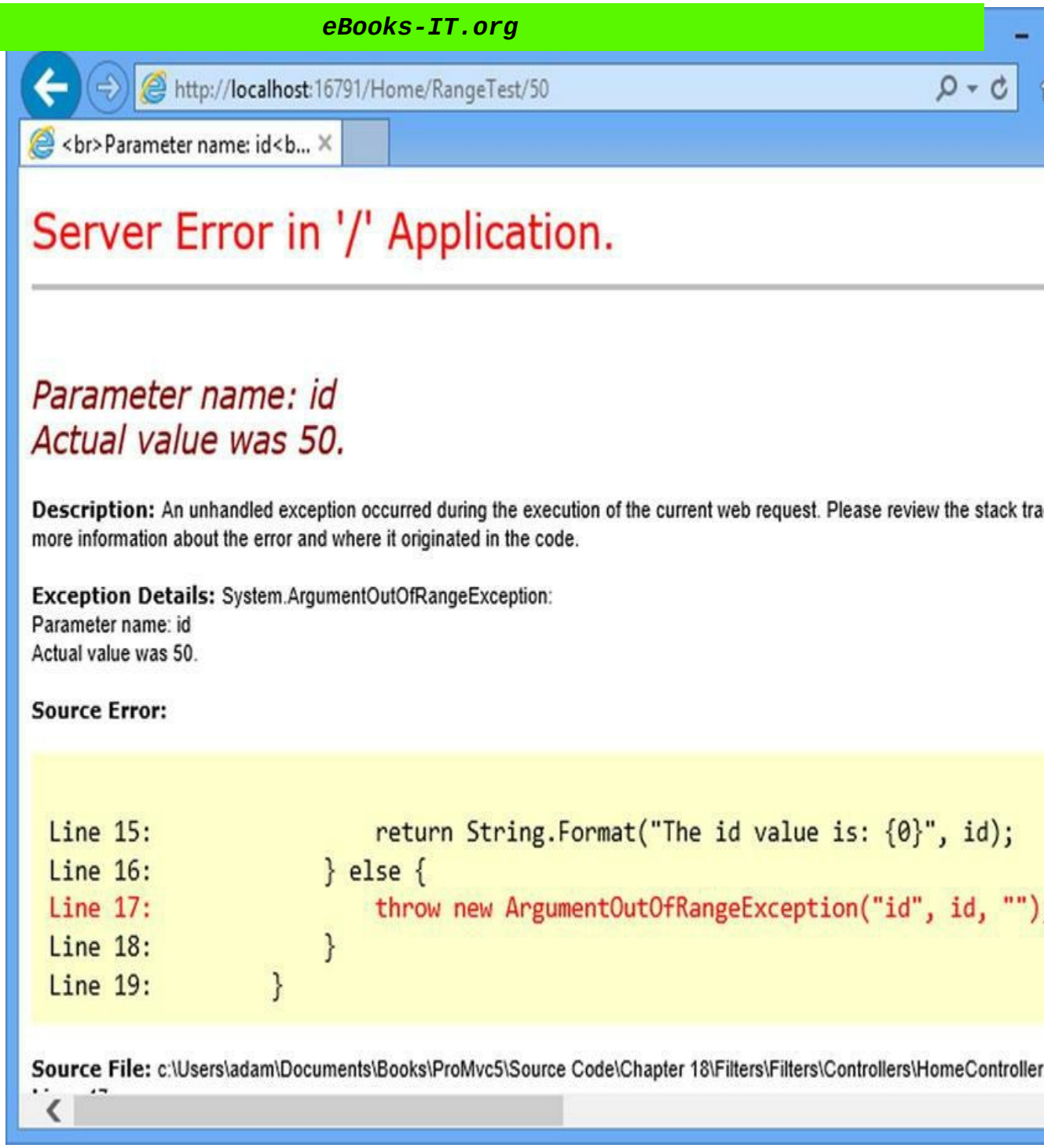


Figure 18-3. The default exception handling response

Note Visual Studio will detect the exception and break the debugger to give you control of the execution of the a
`Press` F5 or click the `Continue` button to continue the execution of the application and see the default excepti
behavior.

I can apply the exception filter to either controllers or individual actions, as shown in Listing 18-24.

Listing 18-24. Applying the Filter in the HomeController.cs File

```
...
[RangeException]
public string RangeTest(int id) {
    if (id > 100) {
        return String.Format("The id value is: {0}", id);
```

eBooks-IT.org

```
            throw new ArgumentOutOfRangeException("id");
        }
    }
...
```

You can see the effect if you restart the application and navigate to the `/Home/RangeTest/50` URL again
Figure 18-4.

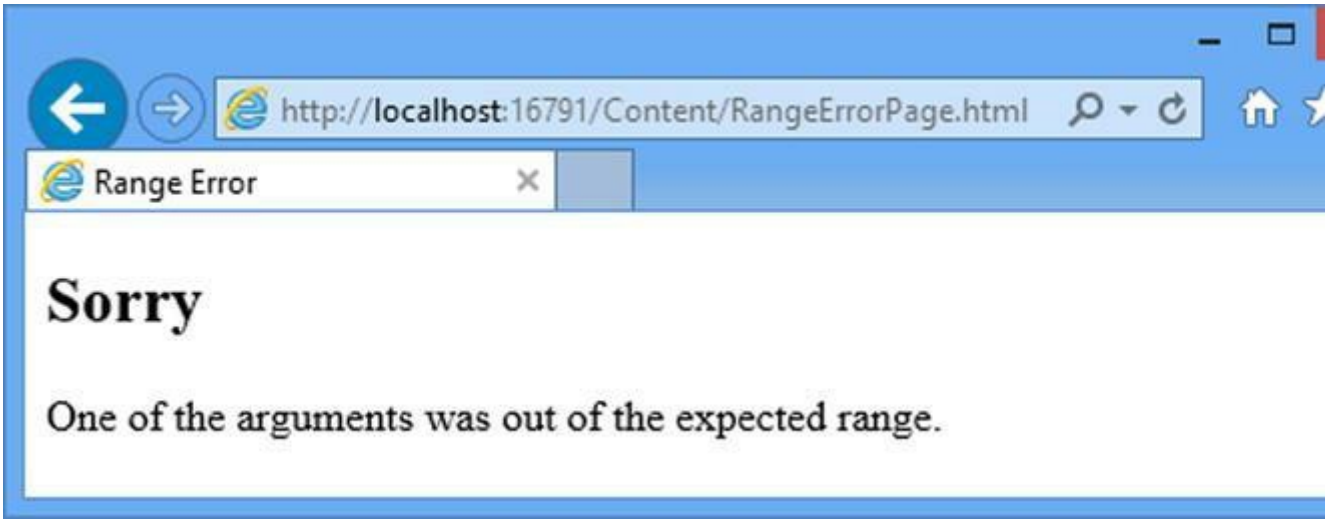


Figure 18-4. The effect of applying the exception filter

Using a View to Respond to an Exception

Depending on the exception you are dealing with, displaying a page of static content can be the simplest and safest th
There is little chance of the process going wrong and causing additional problems. However, while you can be confi
user will see the message, this approach is not especially useful to the user, who gets a generic warning and is dropp
application.

An alternative approach is to use a view to display details of the problem and present the user with some contextua
and options they can follow to sort things out. To demonstrate this, I have made some changes to the
`RangeExceptionAttribute` class, as shown in Listing 18-25.

Listing 18-25. Returning a View from an Exception Filter in the RangeExceptionAttribute.cs File

```
using System;
using System.Web.Mvc;

namespace Filters.Infrastructure {
            public   class   RangeExceptionAttribute   :   FilterAt
IExceptionFilter {

        public void OnException(ExceptionContext filterContext) {

            if (!filterContext.ExceptionHandled &&
                                                filterContext.Excep
ArgumentOutOfRangeException) {

                int val = (int)(((ArgumentOutOfRangeException)
                    filterContext.Exception).ActualValue);
                filterContext.Result = new ViewResult {
                    ViewName = "RangeError",
                    ViewData = new ViewDataDictionary<int>(val)};
```

eBooks-IT.org

```
            }
        }
    }
}
```

I create a `ViewResult` object and set the values of the `ViewName` and `ViewData` properties to specify the view and the model object that will be passed to it. This is messy code because I am working with the `ViewRe` object directly, rather than relying on the `View` method defined by the `Controller` class that is used in action am not going to go into this code because I cover views in depth in Chapter 20 and the built-in exception filter, whic the next section, can be used to achieve the same effect more elegantly. I just want you to see how things work behi scenes.

The `ViewResult` object specifies a view called `RangeError` and passes the `int` value of the argument the exception as the view model object.

To display details of the error, I created the `Views/Shared` folder to the Visual Studio project and created th `RangeError.cshtml` file within it, the contents of which you can see in Listing 18-26.

Listing 18-26. The Contents of the RangeError.cshtml view File

```
@model int

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>The value @Model was out of the expected range.</span>
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
</body>
</html>
```

The view file uses standard HTML and Razor tags to present a (slightly) more useful message to the user. The exa application is pretty limited, so there is not anything useful I can direct the user to do to resolve the problem. But I ha `ActionLink` helper method to create a link that targets another action method, just to demonstrate that you have of view features available. You can see the result if you restart the application and navigate to the `/Home/Range` URL, as shown in Figure 18-5.

eBooks-IT.org

Figure 18-5. Using a view to display an error message from an exception filter

Avoiding the Wrong Exception Trap

The benefits of using a view to display an error are that you can use layouts to make the error message consistent w your application and generate dynamic content that will help the user understand what is going wrong and what they it.

The drawback is that you must thoroughly test your view to make sure that you do not just generate another exce this a lot, where the testing focus is on the main features of the application and does not properly cover the different situations that can arise. As a simple demonstration, I have added a Razor code block to the `RangeError.csh` that will throw an exception, as shown in Listing 18-27.

Listing 18-27. Adding Code That Will Throw an Exception to the RangeError.cshtml File

```
@model int

@{
    var count = 0;
    var number = Model / count;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>The value @Model was out of the expected range.</span>
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
</body>
</html>
```

When the view is rendered, the code will generate a `DivideByZeroException`. If you start the applicat navigate to the `/Home/RangeTest/50` URL again, you will see the exception thrown while trying to render t not the one thrown by the controller, as shown in Figure 18-6.

eBooks-IT.org

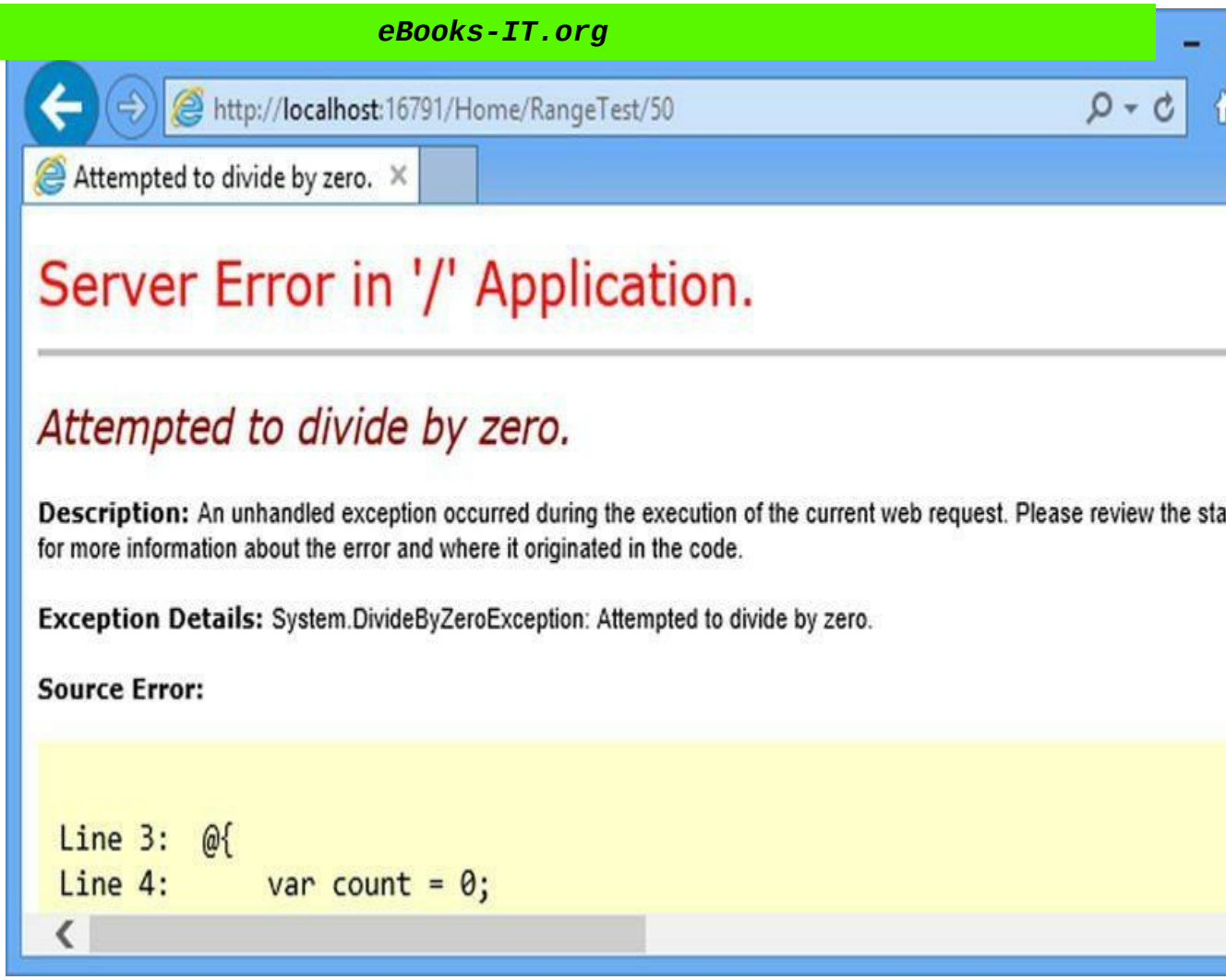


Figure 18-6. An exception thrown while rendering a view

This isn't a realistic scenario, but it demonstrates what happens when there are problems in the view. The user se bewildering error that does not relate to the problem he or she encountered in the application. When using an excepti relies on a view, be careful to test that view thoroughly.

Using the Built-in Exception Filter

I showed you how to create an exception filter because I think understanding what happens behind the scenes in the Framework is a good thing. But you do not often need to create your own filters in real projects because Microsoft h the `HandleErrorAttribute` in the MVC Framework, which is a built-in implementation of the `IExceptionFilter` interface. With it, you can specify an exception and the names of a view and layout usin properties described in Table 18-8.

Table 18-8. *HandleErrorAttribute Properties*

Name	Type	Description
`ExceptionType`	`Type`	The exception type handled by this filter. It will also handle exception types that inherit from the specifi ignore all others. The default value is `System.Exception`, which means that, by default, it will ha exceptions.
`View`	`string`	The name of the view template that this filter renders. If you do not specify a value, it takes a default val `Error`, so by default, it renders `/Views/<currentControllerName>/Error.cshtml` or `/Views/Shared/Error.cshtml`.
`Master`	`string`	The name of the layout used when rendering this filter's view. If you do not specify a value, the view uses layout page.

When an unhandled exception of the type specified by `ExceptionType` is encountered, this filter will render

Preparing to Use the Built-in Exception Filter

The HandleErrorAttribute filter works only when custom errors are enabled in the Web.config fi done by adding a customErrors attribute inside the <system.web> node, as shown in Listing 18-28.

Listing 18-28. Enabling Custom Error in the Web.config File

```
...
<system.web>
  <compilation debug="true" targetFramework="4.5.1" />
  <httpRuntime targetFramework="4.5.1" />
  <authentication mode="Forms">
    <forms loginUrl="~/Account/Login" timeout="2880">
      <credentials passwordFormat="Clear">
        <user name="user"  password="secret"/>
        <user name="admin" password="secret" />
      </credentials>
    </forms>
  </authentication>
  <customErrors mode="On" defaultRedirect="/Content/RangeErrorPage
</system.web>
...
```

The default value for the mode attribute is RemoteOnly, which means that connections made from the local always receive the standard yellow page of death errors, which is a problem because IIS Express will only allow loca connections. By setting the mode attribute to On, I am specifying that my error handling policy should always be a irrespective of where the connections originate. The defaultRedirect attribute specifies a default content p be displayed if all else fails.

Applying the Built-in Exception Filter

You can see how I applied the HandleError filter to the Home controller in Listing 18-29.

Listing 18-29. Using the HandleErrorAttribute Filter in the HomeController.cs File

```
...
[HandleError(ExceptionType = typeof(ArgumentOutOfRangeException),
= "RangeError")]
public string RangeTest(int id) {
    if (id > 100) {
        return String.Format("The id value is: {0}", id);
    } else {
        throw new ArgumentOutOfRangeException("id", id, "");
    }
}
...
```

I have recreated the situation I had with the custom filter, which is that an ArgumentOutOfRangeExce be dealt with by displaying the RangeError view to the user.

When rendering a view, the HandleErrorAttribute filter passes a HandleErrorInfo view mo which is a wrapper around the exception that provides additional information that you use in your view. Table 18-9 d properties defined by the HandleErrorInfo class.

Name	Type	Description
ActionName	string	Returns the name of the action method that generated the exception
ControllerName	string	Returns the name of the controller that generated the exception
Exception	Exception	Returns the exception

You can see how I have updated the RangeError.cshtml view to use this model object in Listing 18-30.

Listing 18-30. Using a HandleErrorInfo Model Object in the RangeError.cshtml File

```
@model HandleErrorInfo

@{
    ViewBag.Title = "Sorry, there was a problem!";
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
                                                        <span>The
@(((ArgumentOutOfRangeException)Model.Exception).ActualValue)
        was out of the expected range.</span>
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
</body>
</html>
```

I have to cast the value of the Model.Exception property to the ArgumentOutOfRangeExcep
be able to read the ActualValue property because the HandleErrorInfo class is a general-purpose mo
that is used to pass any exception to a view.

Using Action Filters

Action filters are filters that can be used for any purpose. The built-in class for creating this kind of filter, IActio
is shown in Listing 18-31.

Listing 18-31. The IActionFilter Interface

```
namespace System.Web.Mvc {

    public interface IActionFilter {
        void OnActionExecuting(ActionExecutingContext filterContex
        void OnActionExecuted(ActionExecutedContext filterContext)
    }
}
```

eBooks-IT.org

method is invoked. It calls the OnActionExecuted method *after* the action method has been invoked. *before*

Implementing the OnActionExecuting Method

The OnActionExecuting method is called before the action method is invoked. You can use this opportunit the request and elect to cancel the request, modify the request, or start some activity that will span the invocation of The parameter to this method is an ActionExecutingContext object, which subclasses the ControllerContext class and defines the two additional properties described in Table 18-10.

***Table 18-10**. ActionExecutingContext Properties*

Name	Type	Description
ActionDescriptor	ActionDescriptor	Provides details of the action method
Result	ActionResult	The result for the action method; a filter can cancel the request by setting this proper non- null value

You can use a filter to cancel the request by setting the Result property of the parameter to an action result. T this, I have created my own action filter class file called CustomActionAttribute.cs in the Infras folder of the example project, as shown in Listing 18-32.

***Listing 18-32**.* The Contents of the CustomActionAttribute.cs File

```
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class CustomActionAttribute : FilterAttribute, IActionF

                    public  void  OnActionExecuting(ActionExecutin
filterContext) {
            if (filterContext.HttpContext.Request.IsLocal) {
                filterContext.Result = new HttpNotFoundResult();
            }
        }

         public void OnActionExecuted(ActionExecutedContext filter
{
            // not yet implemented
        }
    }
}
```

In this example, I use the OnActionExecuting method to check whether the request has been made fron machine. If it has, I return a 404—Not Found response to the user.

Note You can see from Listing 18-32 that you do not need to implement both methods defined in the IActio interface to create a working filter. Be careful not to throw a NotImplementedException, which Visual to a class when you implement an interface. The MVC Framework calls both methods in an action filter and if an ex thrown then you will trigger the exception filters. If you do not need to add any logic to a method, then just leave it e

You apply an action filter as you would any other attribute. To demonstrate the action filter I created in Listing 18- added a new action method to the Home controller, as shown in Listing 18-33.

***Listing 18-33**.* Adding a New Action in the HomeController.cs File

```
using System;
```

eBooks-IT.org

```
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controlle
        }

        [GoogleAuth]
        [Authorize(Users = "bob@google.com")]
        public string List() {
            return "This is the List action on the Home controller
        }

        [HandleError(ExceptionType = typeof(ArgumentOutOfRangeExce
            View = "RangeError")]
        public string RangeTest(int id) {
            if (id > 100) {
                return String.Format("The id value is: {0}", id);
            } else {
                throw new ArgumentOutOfRangeException("id", id, ""
            }
        }

        [CustomAction]
        public string FilterTest() {
            return "This is the FilterTest action";
        }
    }
}
```

You can test the filter by starting the application and navigating to the `/Home/FilterTest` URL. The requ
browser will, of course, be a local connection and that will cause the custom action filter to generate a 404 error for
as shown in Figure 18-7.

eBooks-IT.org

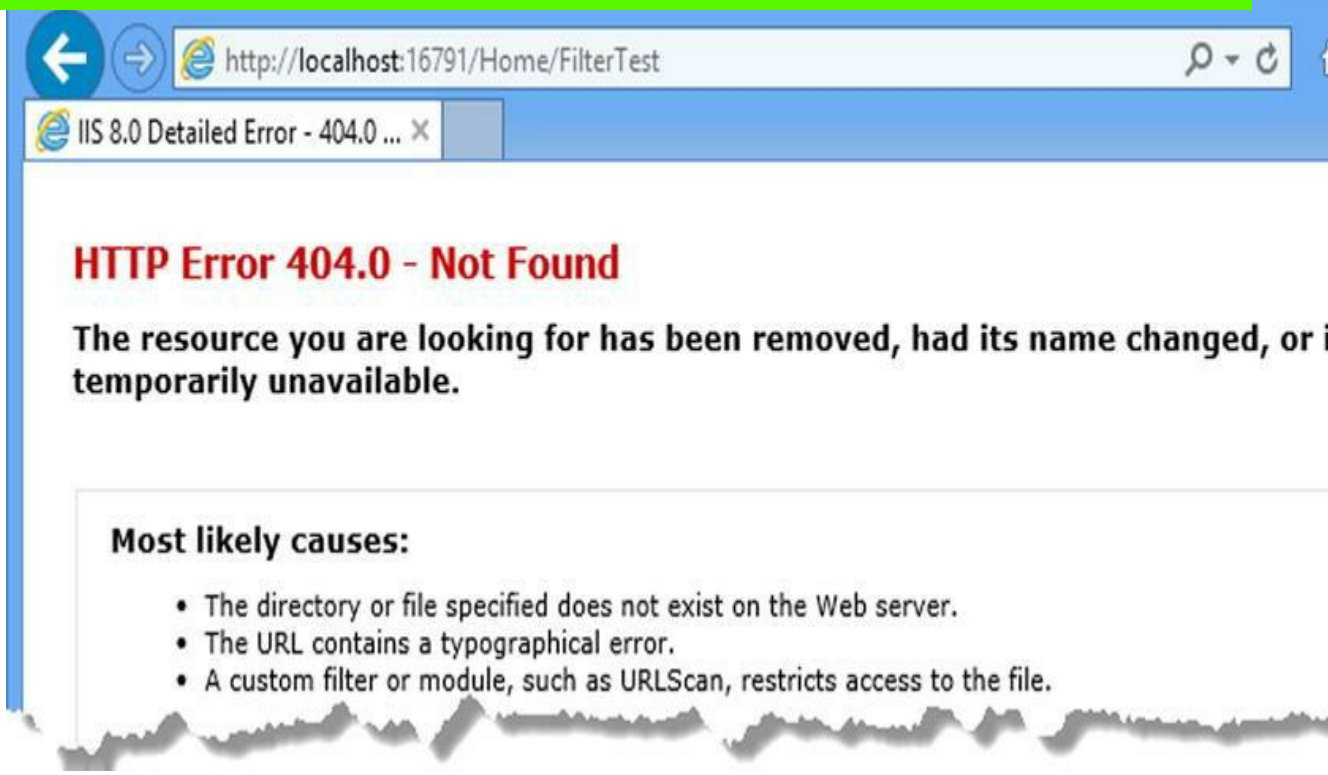


Figure 18-7. The effect of using an action filter

Tip If you want to be sure that it is the filter that is producing the error, simply remove the attribute from the `FilterTest` action method in the `Home` controller and try again.

Implementing the OnActionExecuted Method

You can also use the filter to perform some task that spans the execution of the action method. As a simple example, created a class file called `ProfileActionAttribute.cs` in the `Infrastructure` folder, and use a class that measures the amount of time that an action method takes to execute. You can see the code for this filter 34.

Listing 18-34. The Contents of the ProFileActionAttribute.cs File

```
using System.Diagnostics;
using System.Web.Mvc;

namespace Filters.Infrastructure {

    public class ProfileActionAttribute : FilterAttribute, IAction
        private Stopwatch timer;

                    public   void   OnActionExecuting(ActionExecutin
filterContext) {
            timer = Stopwatch.StartNew();
        }

         public void OnActionExecuted(ActionExecutedContext filter
{
            timer.Stop();
            if (filterContext.Exception == null) {
                filterContext.HttpContext.Response.Write(
```

```
</div> ",                                    timer.Elapsed.TotalSeconds));
                }
            }
        }
}
```

In this example, I use the `OnActionExecuting` method to start a timer (using the high-resolution `Stopw` class in the `System.Diagnostics` namespace). The `OnActionExecuted` method is invoked when t method has completed. Listing 18-35 shows how I applied the attribute to the `Home` controller. (I removed the prev created so that local requests are not redirected.)

Listing 18-35. Applying the Action Filter in the HomeController.cs File

```
...
[ProfileAction]
public string FilterTest() {
    return "This is the ActionFilterTest action";
}
...
```

If you start the application and navigate to the `/Home/FilterTest` URL, you will see the results illustrated 18-8.

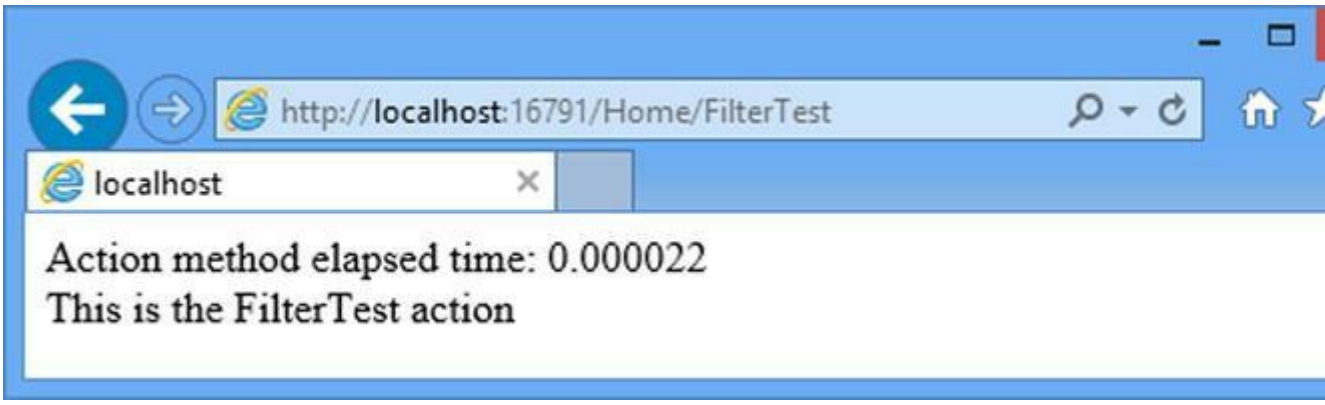


Figure 18-8. Using an action filter to measure performance

Tip Notice that the profile information is shown in the browser before the result of the action method. This is be action filter is executed after the action method has completed but before the result is processed.

The parameter that is passed to the `OnActionExecuted` method is an `ActionExecutedContext` class defines some useful properties, as shown in Table 18-11. The `Exception` property returns any exception t method has thrown, and the `ExceptionHandled` property indicates if another filter has dealt with it.

Table 18-11. *ActionExecutedContext Properties*

Name	Type	Description
`ActionDescriptor`	`ActionDescriptor`	Provides details of the action method
`Canceled`	`bool`	Returns `true` if the action has been canceled by another filter
`Exception`	`Exception`	Returns an exception thrown by another filter or by the action method
`ExceptionHandled`	`bool`	Returns `true` if the exception has been handled
`Result`	`ActionResult`	The result for the action method; a filter can cancel the request by setting this proper non- `null` value

eBooks-IT.org

for the
property) since the time that the filter's OnActionExecuting method was invoked. The OnActionExe
method will still be called, but only so that it can tidy up and release any resources the filter was using.

Using Result Filters

Result filters are general-purpose filters which operate on the results produced by action methods. Result filters imple
IResultFilter interface, which is shown in Listing 18-36.

Listing 18-36. The IResultFilter Interface

```
namespace System.Web.Mvc {

    public interface IResultFilter {
        void OnResultExecuting(ResultExecutingContext filterContex
        void OnResultExecuted(ResultExecutedContext filterContext)
    }
}
```

In Chapter 17, I explained how action methods return action results, allowing separation between the intent of an a
and its execution. When I apply a result filter to an action method, the OnResultExecuting method is invok
action method has returned an action result but *before* the action result is executed. The OnResultExecuted
invoked after the action result is executed.

The parameters to these methods are ResultExecutingContext and ResultExecutedConte
respectively, and they are similar to their action filter counterparts. They define the same properties, which have the s
(See Table 18-11.) To demonstrate a simple result filter, I created a class file called ProfileResultAttrib
the Infrastructure folder and used it to define the class shown in Listing 18-37.

Listing 18-37. The Contents of the ProFileResultAttribute.cs File

```
using System.Diagnostics;
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class ProfileResultAttribute : FilterAttribute, IResult
        private Stopwatch timer;

                    public  void  OnResultExecuting(ResultExecutin
filterContext) {
            timer = Stopwatch.StartNew();
        }

         public void OnResultExecuted(ResultExecutedContext filter
{
            timer.Stop();
            filterContext.HttpContext.Response.Write(
                          string.Format("<div>Result elapsed time
</div>",
                       timer.Elapsed.TotalSeconds));
        }
    }
}
```

This result filter is the complement to the action filter I created in the previous section and measures the amount o

Listing 18-38. Applying the Result Filter in the HomeController.cs File

```
...
[ProfileAction]
[ProfileResult]
public string FilterTest() {
    return "This is the ActionFilterTest action";
}
...
```

Figure 18-9 shows the effect of starting the application and navigating to the `/Home/FilterTest` URL. Yo that both filters have added data to the response sent to the browser. The output from the result filter is shown after from the action method, of course, since the `OnResultExecuted` method cannot be executed by the MVC F until the result has been properly dealt with—which, in this case, means inserting a `string` value into the result.

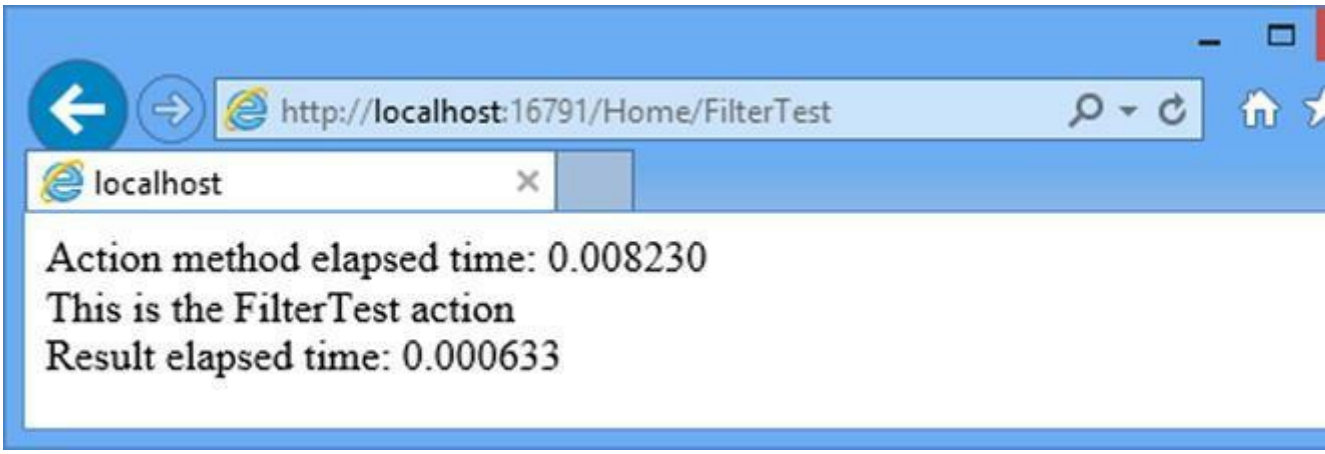


Figure 18-9. The effect of applying a result filter

Using the Built-in Action and Result Filter Class

The MVC Framework includes a built-in class that can be derived to create a class that is both an action and result fi class, called `ActionFilterAttribute`, is shown in Listing 18-39.

Listing 18-39. The ActionFilterAttribute Class

```
public abstract class ActionFilterAttribute : FilterAt
IActionFilter,
    IResultFilter{

            public virtual void OnActionExecuting(ActionExecutin
filterContext) {
        }

             public virtual void OnActionExecuted(ActionExecute
filterContext) {
        }

            public virtual void OnResultExecuting(ResultExecutin
filterContext) {
        }
```

```
filterContext) {
        }
    }
}
```

The only benefit to using this class is that you do not need to override and implement the methods that you do not Otherwise, there is no advantage over implementing the filter interfaces directly.

To demonstrate the use of the `ActionFilterAttribute` class, I added a class file called `ProfileAllAttribute.cs` to the `Infrastructure` folder of the example project and used it to de shown in Listing 18-40.

Listing 18-40. The Contents of the ActionFilterAttribute.cs File

```
using System.Diagnostics;
using System.Web.Mvc;

namespace Filters.Infrastructure {
    public class ProfileAllAttribute : ActionFilterAttribute {
        private Stopwatch timer;

            public override void OnActionExecuting(ActionExecutin
filterContext) {
            timer = Stopwatch.StartNew();
        }

             public override void OnResultExecuted(ResultExecute
filterContext) {
            timer.Stop();
            filterContext.HttpContext.Response.Write(
                    string.Format("<div>Total elapsed time: {0:F6}
                        timer.Elapsed.TotalSeconds));
        }
    }
}
```

The `ActionFilterAttribute` class implements the `IActionFilter` and `IResultFilter` which means that the MVC Framework will treat derived classes as both types of filters, even if not all of the metho overridden. In the example, I have implemented only the `OnActionExecuting` method from the `IAction` interface and the `OnResultExecuted` method from the `IResultFilter` interface. This allows me to c profiling theme and measure the time it takes for the action method to execute and the result to be processed as a sin Listing 18-41 shows how I applied the filter to the `Home` controller.

Listing 18-41. Applying the Filter in the HomeController.cs File

```
...
[ProfileAction]
[ProfileResult]
[ProfileAll]
public string FilterTest() {
    return "This is the FilterTest action";
}
...
```

You can see the effect of all of these filters if you start the application and navigate to the `/Home/FilterTe` Figure 18-10 shows the result.

eBooks-IT.org

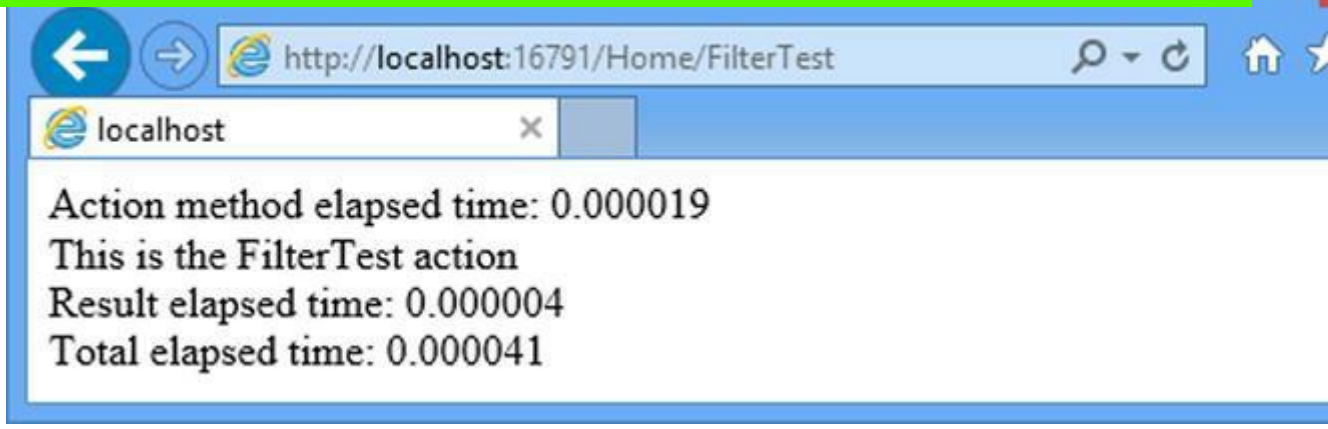


Figure 18-10. The effect of adding the ProFileAll filter

Using Other Filter Features

The previous examples have given you all the information you need to work effectively with filters. In the following s will show you some of the advanced MVC Framework filtering capabilities, which are interesting but not as widely u

Filtering Without Attributes

The normal way of using filters is to apply attributes, as I have demonstrated in the previous sections. However, ther alternative. The `Controller` class implements the `IAuthenticationFilter`, `IAuthorizatio` `IActionFilter`, `IResultFilter`, and `IExceptionFilter` interfaces. It also provides empty v implementations of each of the `On` *XXX* methods you have already seen, such as `OnAuthorization` and `OnException`. In Listing 18-42 I have updated the `Home` controller to use this feature and create a self-profilin class.

Listing 18-42. Using the Controller Filter Methods in the HomeController.cs File

```
using System;
using System.Diagnostics;
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class HomeController : Controller {
        private Stopwatch timer;

        [Authorize(Users = "admin")]
        public string Index() {
            return "This is the Index action on the Home controlle
        }

        [GoogleAuth]
        [Authorize(Users = "bob@google.com")]
        public string List() {
            return "This is the List action on the Home controller
        }

         [HandleError(ExceptionType = typeof(ArgumentOutOfRangeExc
View = "RangeError")]
```

eBooks-IT.org

```
            if (id > 100) {
                return String.Format("The id value is: {0}", id);
            } else {
                throw new ArgumentOutOfRangeException("id", id, ""
            }
        }

        public string FilterTest() {
            return "This is the FilterTest action";
        }

          protected override void OnActionExecuting(ActionExecutin
filterContext) {
            timer = Stopwatch.StartNew();
        }

           protected override void OnResultExecuted(ResultExecute
filterContext) {
            timer.Stop();
            filterContext.HttpContext.Response.Write(
                    string.Format("<div>Total elapsed time: {0}</d
                        timer.Elapsed.TotalSeconds));
        }
    }
}
```

I removed the filters from the `FilterTest` action method because they are no longer required. The `Home` c
add the profile information to the response for any action method. Figure 18-11 shows the effect of starting the appl
navigating to the `/Home/RangeTest/200` URL, which targets the `RangeTest` action without causing th
set up to demonstrate the `HandleError` filter.

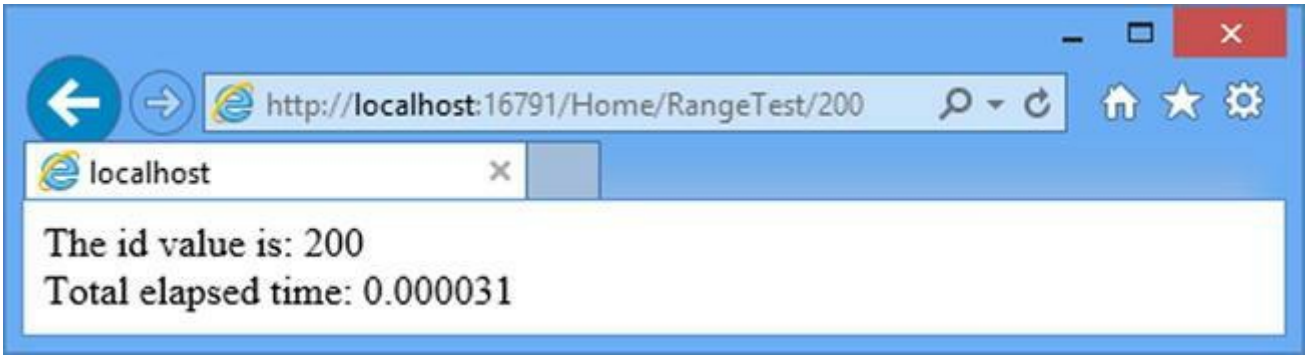


Figure 18-11. The effect of implementing filter methods directly in the controller

This technique is most useful when you are creating a base class from which multiple controllers in your project a
The whole point of filtering is to put code that is required *across* the application in one reusable location, so using the
a class that will not be used as a base for controllers does not make much sense.

Tip I prefer to use attributes. I like the separation between the controller logic and the filter logic. If you are look
to apply a filter to all of your controllers, continue reading to see how to do that with global filters.

Using Global Filters

Global filters are applied to all of the action methods in all of the controllers in your application. There is a conventior
up global filters, which is created by Visual Studio automatically when you use the `MVC` project template but which

eBooks-IT.org

Application-wide configuration is done in classes added to the App_Start folder, which is why I defined rout 15 and 16 in the App_Start/RouteConfig.cs file. To create the equivalent for filters, I added a new cl FilterConfig.cs to the App_Start folder with the content shown in Listing 18-43.

Listing 18-43. The Content of the FilterConfig.cs File

```
using System.Web;
using System.Web.Mvc;

namespace Filters {
    public class FilterConfig {
          public static void RegisterGlobalFilters(GlobalFilterCo
filters) {
            filters.Add(new HandleErrorAttribute());
        }
    }
}
```

This is the same content that Visual Studio would have created for the MVC template. The FilterConfig c static method called RegisterGlobalFilters that receives the collection of global filters, expressed as a GlobalFilterCollection object, to which new filters can be added.

There are two conventions to note in this file. The first is that the FilterConfig class is defined in the Fil namespace and not Filters.App_Start, which is what Visual Studio will use when it creates the file. The the HandleError filter, which I described earlier in the chapter, is always defined as a global filter by calling th method on the GlobalFilterCollection object.

Note You don't have to set up the HandleError filter globally, but it defines the default MVC exception ha This will render the /Views/Shared/Error.cshtml view when an unhandled exception arises. This ex handling policy is disabled by default for development. See the "Creating an Exception Filter" section earlier in the ch note on how to enable it in the Web.config file.

I am going to apply my ProfileAll filter globally and I use the same method call that sets up the Handle filter, as shown in Listing 18-44.

Listing 18-44. Adding a Global Filter to the FilterConfig.cs File

```
using System.Web;
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters {
    public class FilterConfig {
          public static void RegisterGlobalFilters(GlobalFilterCo
filters) {
            filters.Add(new HandleErrorAttribute());
            filters.Add(new ProfileAllAttribute());
        }
    }
}
```

Tip Notice that I register the filter globally by creating an instance of the filter class, which means that I need to class name, including the Attribute suffix. The rule is that you omit Attribute when applying the filter a but include it when directly creating an instance of its class.

The next step is to ensure that the FilterConfig.RegisterGlobalFilters method is called fr

Listing 18-45. Setting Up Global Filters in the Global.asax File

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

namespace Filters {
    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filte
        }
    }
}
```

To demonstrate the global filter, I have created a new controller called `Customer`, as shown in Listing 18-46. I a new controller because I want to use code to which I have not previously applied a filter.

Listing 18-46. The Contents of the CustomerController.cs File

```
using System.Web.Mvc;

namespace Filters.Controllers {
    public class CustomerController : Controller {

        public string Index() {
            return "This is the Customer controller";
        }
    }
}
```

This is a simple controller whose `Index` action returns a `string`. Figure 18-12 illustrates the effect of the gl which I achieved by starting the application and navigating to the `/Customer` URL. Even though I have not appli directly to the controller, the global filter adds the profiling information shown in the figure.

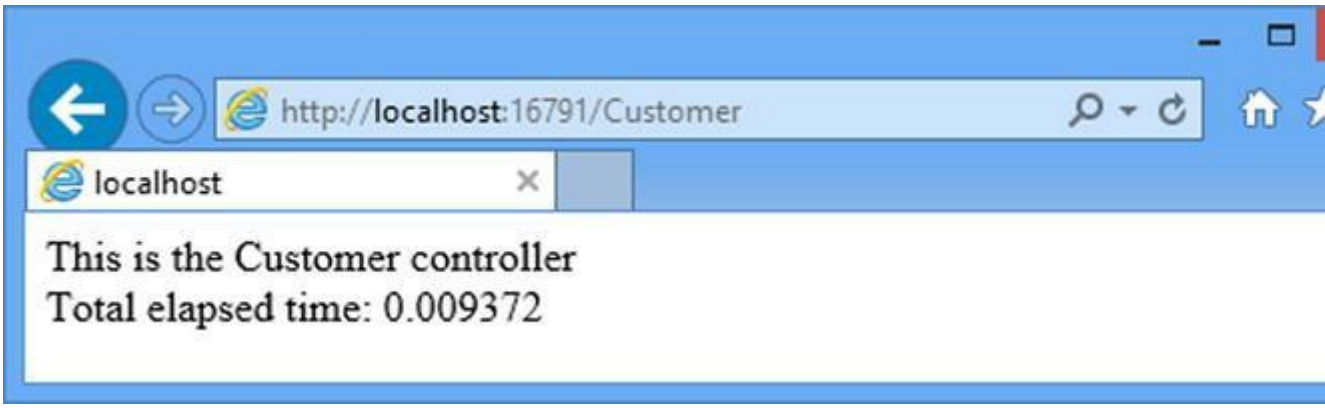


Figure 18-12. The Effect of a Global Filter

Ordering Filter Execution

eBooks-IT.org

filters, a
and then result filters. The framework executes exception filters at any stage if there is an unhandled exception. How
each type category, you can take control of the order in which individual filters are used. Listing 18-47 shows a class
`SimpleMessageAttribute.cs` that I added to the `Infrastructure` folder to define a simple filt
demonstrate ordering filter execution.

Listing 18-47. The Contents of the SimpleMessageAttribute.cs File

```
using System;
using System.Web.Mvc;

namespace Filters.Infrastructure {

    [AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
    public class SimpleMessageAttribute : FilterAttribute, IAction

        public string Message { get; set; }

                    public  void  OnActionExecuting(ActionExecutin
filterContext) {
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[Before Action: {0}]<div>", Me
        }

         public void OnActionExecuted(ActionExecutedContext filter
{
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[After Action: {0}]<div>", Mes
        }
    }
}
```

This filter writes a message to the response when the `OnActionExecuting` and `OnActionExecute`
are invoked, part of which is specified using the `Message` property (which I will set when I apply the filter). I ca
multiple instances of this filter to an action method, as shown in Listing 18-48. (Notice that in the `AttributeUs`
attribute in Listing 18-47, I set the `AllowMultiple` property to `true`).

Listing 18-48. Applying Multiple Filters to an Action in the CustomerController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class CustomerController : Controller {

        [SimpleMessage(Message="A")]
        [SimpleMessage(Message="B")]
        public string Index() {
            return "This is the Customer controller";
        }
    }
}
```

I created two filters with different messages: the first has a message of `A`, and the other has a message of `B`. I co
two different filters, but this approach allows me to demonstrate that you can configure global filters through propert

eBooks-IT.org

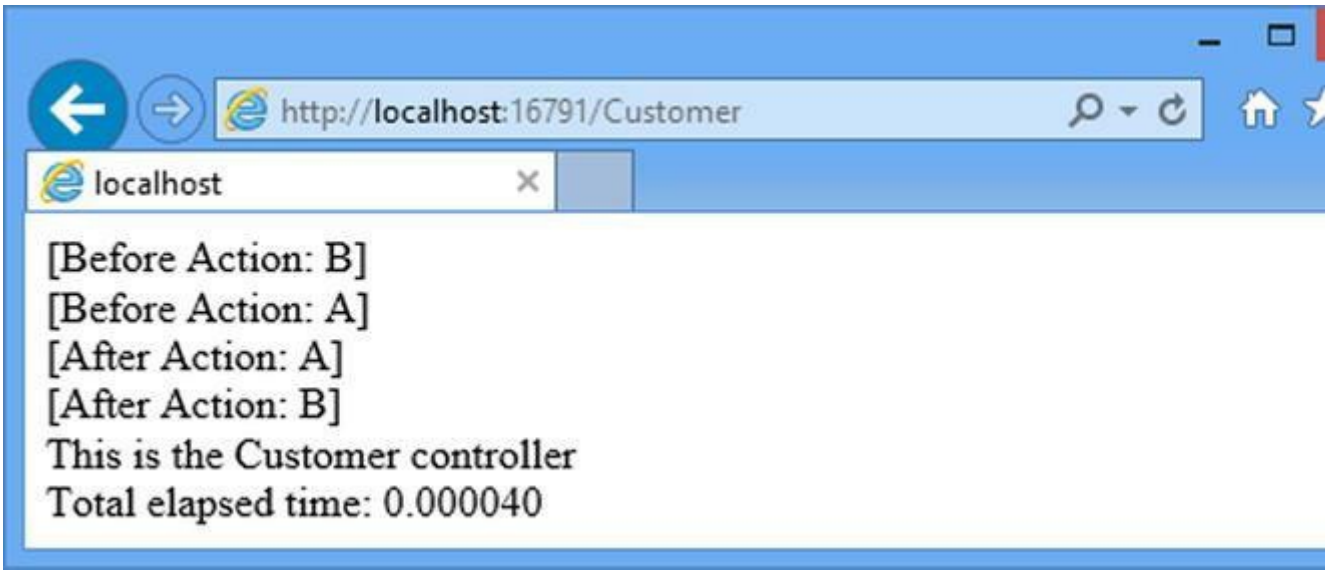


Figure 18-13. Multiple filters on the same action method

The MVC Framework executes the `B` filter before the `A` filter, but it could have been the other way around. The M Framework does not guarantee any particular order or execution. Most of the time, the order does not matter. When can use the `Order` property, as shown in Listing 18-49.

Listing 18-49. Using the Order Property in a Filter

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    public class CustomerController : Controller {

        [SimpleMessage(Message = "A", Order = 1)]
        [SimpleMessage(Message = "B", Order = 2)]
        public string Index() {
            return "This is the Customer controller";
        }
    }
}
```

The `Order` parameter takes an `int` value, and the MVC Framework executes the filters in ascending order. In have given the `A` filter the lowest value, so the framework executes it first, as shown in Figure 18-14.

eBooks-IT.org

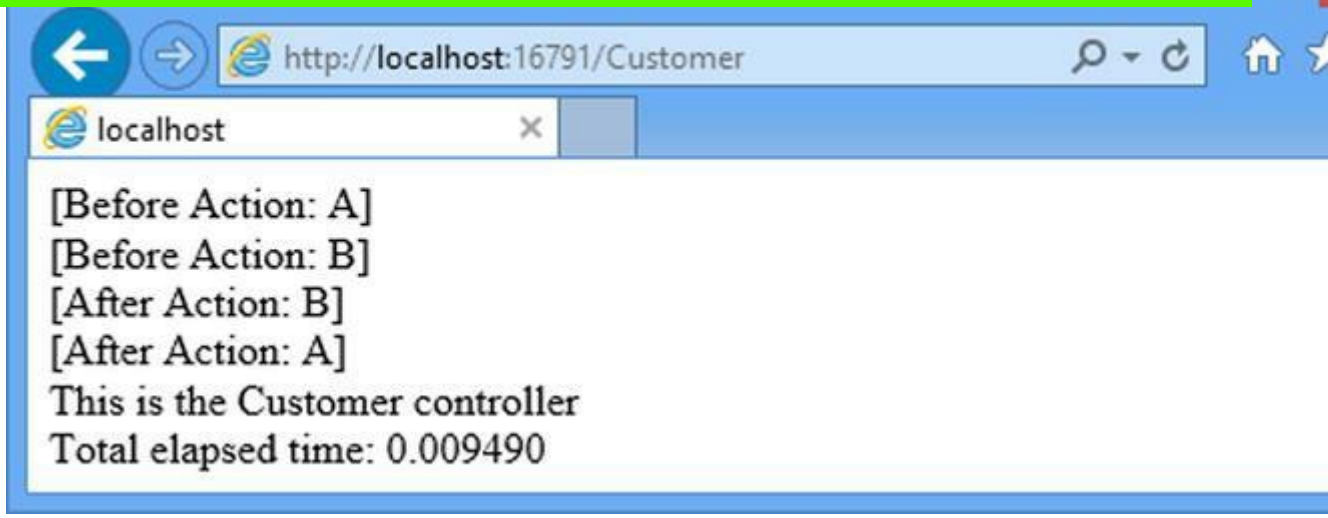


Figure 18-14. Specifying the order of filter execution

Note Notice that the `OnActionExecuting` methods are executed in the order I specified, but the `OnActionExecuted` methods are executed in the reverse order. The MVC Framework builds up a stack of f executes them before the action method, and then unwinds the stack afterward. This unwinding behavior cannot be

If I do not specify a value for the `Order` property, it is assigned a default value of `-1`. This means that if you m that some have `Order` values and others do not, the ones without these values will be executed first, since they ha `Order` value.

If multiple filters of the same type (say, action filters) have the same `Order` value (say `1`), then the MVC Frame determines the execution order based on where the filter has been applied. Global filters are executed first, then filters the controller class, and then filters applied to the action method.

Note The order of execution is reversed for exception filters. If exception filters are applied with the same `Ord` the controller and to the action method, the filter on the action method will be executed first. Global exception filters `Order` value are executed last.

Overriding Filters

There will be occasions when you want to apply a filter globally or at the controller level, but use a different filter for action method. To demonstrate what I mean, I have updated the `SimpleMessage` filter so that it can be applie controller, as shown in Listing 18-50.

Listing 18-50. Adding Controller-Level Application in the SimpleMessageAttribute.cs File

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

namespace Filters.Infrastructure {

    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Meth
        AllowMultiple = true)]
    public class SimpleMessageAttribute : FilterAttribute, IAction

        public string Message { get; set; }
```

eBooks-IT.org

```
filterContext) {
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[Before Action: {0}]<div>", Me
        }

         public void OnActionExecuted(ActionExecutedContext filter
{
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[After Action: {0}]<div>", Mes
        }
    }
}
```

This change means that the filter can be applied to individual action methods or to the entire controller class. In Lis you can see how I have changed the way that the filter is applied to the `Customer` controller.

Listing 18-51. Updating the CustomerController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    [SimpleMessage(Message = "A")]
    public class CustomerController : Controller {

        public string Index() {
            return "This is the Customer controller";
        }

        [SimpleMessage(Message = "B")]
        public string OtherAction() {
            return "This is the Other Action in the Customer contr
        }
    }
}
```

I have applied the `SimpleMessage` filter to the controller class, meaning that the message `A` will be added to when either of the action methods is invoked. I have added a new `OtherAction` method, to which I have appli `SimpleMessage` filter again, but this time with the message `B`.

The problem is that, by default, the `OtherAction` method is affected by both applications of the filter: at the method level. You can see how this works by starting the application and navigating to `/Customer/OtherAc` shown in Figure 18-15.

eBooks-IT.org

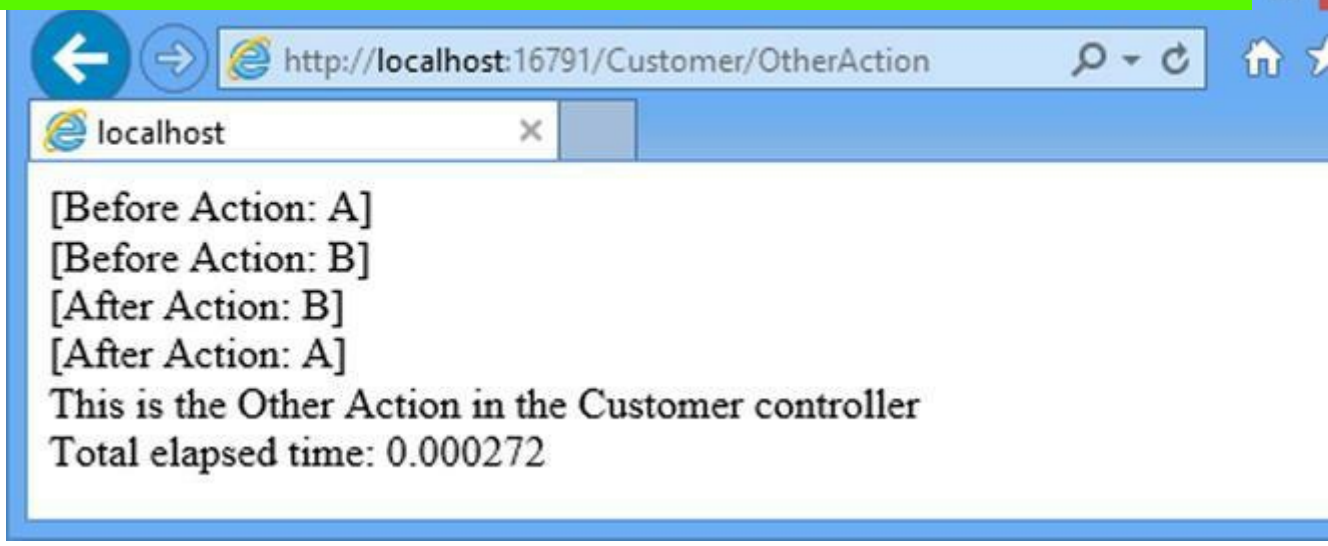


Figure 18-15. The default filter behavior

If you want an action method to just be affected by the filters that have been directly applied to it, then you can us *override*. This tells the MVC Framework to ignore any filters that have been defined at a higher-level, such as the co globally. Filter overrides are attributes that implement the `IOverrideFilter` interface, which is shown in Lis

Listing 18-52. The IOverrideFilter interface

```
namespace System.Web.Http.Filters {
    public interface IOverrideFilter : IFilter {

        Type FiltersToOverride { get; }
    }
}
```

The `FiltersToOverride` method returns the type of filter that will be overridden. I am interested in actio this example and to that end I created the `CustomOverrideActionFiltersAttribute.cs` file in `Infrastructure`. As Listing 18-53 shows, I implemented the `FiltersToOverride` method so that attribute overrides the `IActionFilter` type.

Caution The MVC Framework comes with some built-in filter overrides in the `System.Web.Mvc.Filt` namespace: `OverrideAuthenticationAttribute`, `OverrideActionFiltersAttribu` on. As I write this, these filters do not work. This is because they are derived from `Attribute` and not `FilterAttribute`. I assume that this will be resolved in a later release, but in the meantime you should creat filter override attributes like the one I demonstrate below.

Listing 18-53. The Contents of the CustomOverrideActionFiltersAttribute.cs File

```
using System;
using System.Web.Mvc;
using System.Web.Mvc.Filters;

namespace Filters.Infrastructure {

    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Meth
        Inherited = true, AllowMultiple = false)]
    public class CustomOverrideActionFiltersAttribute : FilterAttr
            IOverrideFilter {
        public Type FiltersToOverride {
```

eBooks-IT.org

```
        }
    }
}
```

I can apply this filter to my controller to prevent the global and controller level action filters from taking effect, as Listing 18-54.

Listing 18-54. Applying a Filter Override in the CustomerController.cs File

```
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers {
    [SimpleMessage(Message = "A")]
    public class CustomerController : Controller {

        public string Index() {
            return "This is the Customer controller";
        }

        [CustomOverrideActionFilters]
        [SimpleMessage(Message = "B")]
        public string OtherAction() {
            return "This is the Other Action in the Customer contr
        }
    }
}
```

As Figure 18-16 shows, only the `SimpleMessage` attribute which I have applied directly to the `OtherAc` method is run.

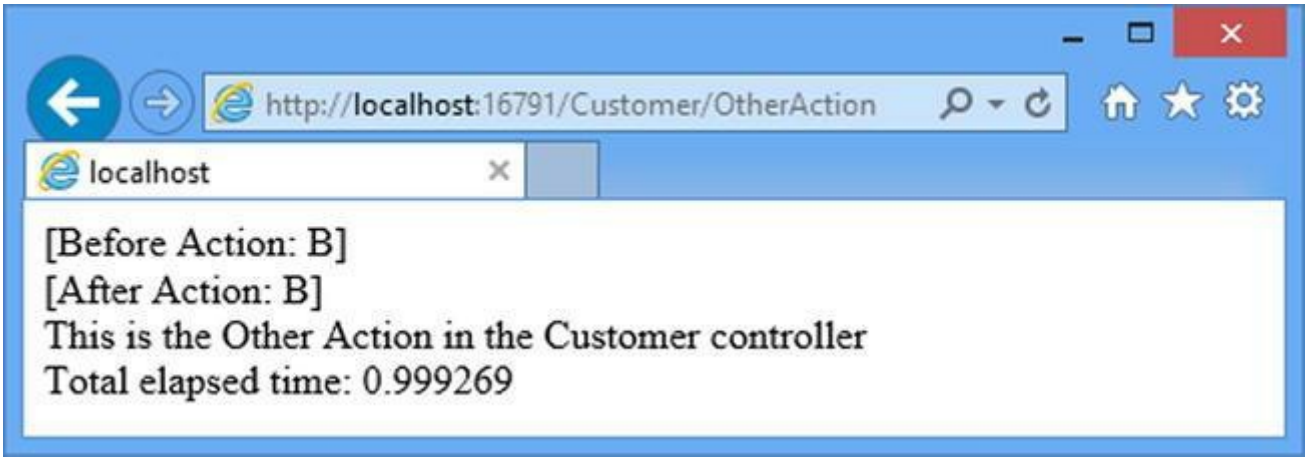


Figure 18-16. The effect of overriding action filters

Summary

In this chapter, you have seen how to encapsulate logic that addresses cross-cutting concerns as filters. I showed yo different kinds of filters available and how to implement each of them. You saw how filters can be applied as attribut controllers and action methods, and how they can be applied as global filters. Filters are a means of extending the log applied when a request is processed, without needing to include that logic in the action method. In the next chapter, I how to change and extend the way that the MVC Framework deals with controllers.

eBooks-IT.org

Controller Extensibility

In this chapter, I am going to show you some of the advanced MVC features for working with controllers. I start th exploring the parts of the request handling process that lead to the execution of an action method and demonstrating ways to take control of it. Figure 19-1 shows the basic flow of control between components.

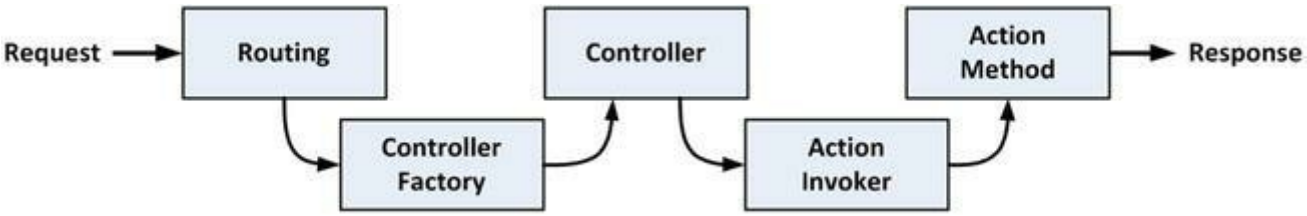


Figure 19-1. Invoking an action method

My focus for the first part of this chapter is the *controller factory* and the *action invoker*. The names of these con suggest their purpose. The controller factory is responsible for creating instances of controllers to service a request a invoker is responsible for finding and invoking the action method in the controller class. The MVC Framework inclu implementations of both of these components, and I will show you how to configure and control their behavior. I wi you how to replace these components entirely and use custom logic. Table 19-1 provides the summary for this chap

***Table 19-1**. Chapter Summary*

Problem	Solution
Create a custom controller factory	Implement the `IControllerFactory` interface
Prioritize namespaces in the default controller factory	Use the `DefaultNamespaces` collection
Create a custom controller activator	Implement the `IControllerActivator` interface
Create a custom action invoker	Implement the `IActionInvoker` interface
Specify an action name that is different from the action method name	Use the `ActionName` attribute
Control the selection of action methods	Apply action method selectors
Prevent a method from being used as an action	Use the `NoAction` attribute
Create a custom action method selector	Derive from the `ActionMethodSelectorAttribute` class
Respond to requests for non-existent action methods	Override the `HandleUnknownAction` method in the controller
Control how controllers use the session feature	Return a value from the `SessionStateBehavior` enumeration in the `IControllerFactory` implementation or apply the `SessionState` attribute to the control class
Prevent controllers from blocking worker threads when waiting for input	Create an asynchronous controller

Preparing the Example Project

eBooks-IT.org

the option to add the core MVC references and folders. I need some simple controllers to work with in this chapter, demonstrate the different kinds of extensibility features that are available. To get set up, I created the `Result.cs` `Models` folder and used it to define the `Result` class shown in Listing 19-1.

Listing 19-1. The Contents of the Result.cs File

```
namespace ControllerExtensibility.Models {
    public class Result {
        public string ControllerName { get; set; }
        public string ActionName { get; set; }
    }
}
```

The next step is to create the `/Views/Shared` folder and add a new view called `Result.cshtml`. Th that all of the action methods in the controllers will render, and you can see the contents of this file in Listing 19-2.

Listing 19-2. The Contents of the Result.cshtml File

```
@model ControllerExtensibility.Models.Result

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Result</title>
</head>
<body>
    <div>Controller: @Model.ControllerName</div>
    <div>Action: @Model.ActionName</div>
</body>
</html>
```

This view uses the `Result` class that I defined in Listing 19-2 as its model and displays the values of the `ControllerName` and `ActionName` properties. Finally, I need to create some basic controllers. Listing 19 `Product` controller.

Listing 19-3. The Contents of the Product Controller.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class ProductController : Controller {

        public ViewResult Index() {
            return View("Result", new Result {
                ControllerName = "Product",
                ActionName = "Index"
            });
```

```
        public ViewResult List() {
            return View("Result", new Result {
                ControllerName = "Product",
                ActionName = "List"
            });
        }
    }
}
```

Listing 19-4 shows the `Customer` controller.

Listing 19-4. The Contents of the CustomerController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class CustomerController : Controller {

        public ViewResult Index() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "Index"
            });
        }

        public ViewResult List() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "List"
            });
        }
    }
}
```

These controllers do not perform any useful actions other than to report that they have been called via the `Result.cshtml` view.

Setting the Start URL

I want Visual Studio to start with the root URL for the application rather than guess the URL based on the file that is
Select `ControllerExtensibility Properties` from the Visual Studio `Project` menu, switch
tab and check the `Specific Page` option in the `Start Action` section. You don't have to provide a va
checking the option is enough.

Creating a Custom Controller Factory

As with much of the MVC Framework, the best way to understand how controller factories work is to create a cust
implementation. I do not recommend that you do this in a real project, as there are easier ways to create custom beha
extending the built-in factory. But this is a nice way to demonstrate how the MVC framework creates instances of c
Controller factories are defined by the `IControllerFactory` interface, which is shown in Listing 19-5.

```
using System.Web.Routing;
using System.Web.SessionState;

namespace System.Web.Mvc {
    public interface IControllerFactory {

        IController CreateController(RequestContext requestContext
            string controllerName);

          SessionStateBehavior GetControllerSessionBehavior(Reques
requestContext,
            string controllerName);

        void ReleaseController(IController controller);
    }
}
```

In the sections that follow, I create a simple custom controller factory and walk you through implementations for methods in the `IControllerFactory` interface. To begin, I created an `Infrastructure` folder and class file called `CustomControllerFactory.cs`, which I used to create the custom controller factory s Listing 19-6.

Listing 19-6. The Contents of the CustomControllerFactory.cs File

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.SessionState;
using ControllerExtensibility.Controllers;

namespace ControllerExtensibility.Infrastructure {

    public class CustomControllerFactory: IControllerFactory {

        public IController CreateController(RequestContext request
            string controllerName) {

            Type targetType = null;
            switch (controllerName) {
                case "Product":
                    targetType = typeof(ProductController);
                    break;
                case "Customer":
                    targetType = typeof(CustomerController);
                    break;
                default:
                            requestContext.RouteData.Values["contro
"Product";
                    targetType = typeof(ProductController);
                    break;
            }

            return targetType == null ? null :
                (IController)DependencyResolver.Current.GetService
```

```
                                                public      SessionState
GetControllerSessionBehavior(RequestContext
            requestContext, string controllerName) {

            return SessionStateBehavior.Default;
        }

        public void ReleaseController(IController controller) {
            IDisposable disposable = controller as IDisposable;
            if (disposable != null) {
                disposable.Dispose();
            }
        }
    }
}
```

The most important method in the interface is `CreateController`, which the MVC Framework calls whe controller to service a request. The parameters to this method are a `RequestContext` object, which allows th inspect details of the request, and a string, which contains the `controller` value from the routed URL. The `RequestContext` class defines the properties described in Table 19-2.

***Table 19-2**. RequestContext Properties*

Name	Type	Description
`HttpContext`	`HttpContextBase`	Provides information about the HTTP request
`RouteData`	`RouteData`	Provides information about the route that matches the request

One of the reasons that I do not recommend creating a custom controller this way is that finding controller classes application and instantiating them is complicated. You need to be able to locate controllers dynamically and consisten with all sorts of potential problems, such as disambiguating between classes with the same name in different namesp constructor exceptions and a whole lot more.

There are only two controllers in the example project and I am going to instantiate them directly, which means har class names into the controller factory, something which is obviously not a good idea for a real project, but which le step enormous amounts of complexity.

The purpose of the `CreateController` method is to create instances of controller classes that can handle request. There are no restrictions on how you do this. The only rule is that you *must* return an object that implements `IController` interface as the method result.

The conventions that you have seen so far in this book exist because that's how the default controller factory has As an example, I have implemented one of these conventions in my code: that when I receive a request for a controll `Controller` to the class name, so that a request for `Product` leads to the `ProductController` cla instantiated.

You are free to follow the MVC Framework conventions when you write a controller factory or to discard them a your own to suit your project needs. I do not think it is sensible to create your own conventions just for the sake of useful to understand just how flexible the MVC Framework can be.

Dealing with the Fallback Controller

Custom controller factories must return an implementation of the `IController` interface as the result from the `CreateController` method, otherwise, an error will be displayed to the user. This means that you need to h position for when the request you are processing does not target any of the controllers in your project. You can crea you like for dealing with this situation: you could define a special controller that renders an error message, for examp have and map the request to a controller class that is known to always exist.

When I get a request that does not map to either of the controllers in the project, I target the `ProductContr` class. This may not be the most useful thing to do in a real project, but it demonstrates that the controller factory has

eBooks-IT.org

operate.

By default, the MVC Framework selects a view based on the `controller` value in the routing data, not the na controller class. So, in my example, if I want the fallback position to work with views that follow the convention of organized by controller name, I need to change the value of the `controller` routing property, like this:

```
...
requestContext.RouteData.Values["controller"] = "Product";
...
```

This change will cause the MVC Framework to search for views associated with the fallback controller and not th that the routing system has identified based on the URL that the user requested.

There are two important points here: the first is that not only does the controller factory have sole responsibility fo requests to controllers, but it can *change* the request to alter the behavior of subsequent steps in the request processi This is pretty potent stuff and a critical characteristic of the MVC Framework.

The second point is that while you are free to follow whatever conventions you want in your controller factory, yo know what the conventions are for other parts of the MVC Framework. And, because those other components can b with custom code as well (as I demonstrate for views in Chapter 20), it makes sense to follow as many of the conve possible to allow components to be developed and used independently of one another.

Instantiating Controller Classes

There are no rules about how you instantiate your controller classes, but it is good practice to use the dependency re introduced in Chapter 6. This allows you to keep your custom controller factory focused on mapping requests to con classes, and leaves issues like dependency inject to be handled separately and for the entire application. You can see h the `DependencyResolver` class to create controllers instances:

```
...
return targetType == null ? null :
    (IController)DependencyResolver.Current.GetService(targetType)
...
```

The static `DependencyResolver.Current` property returns an implementation of the `IDependencyResolver` interface, which defines the `GetService` method. You pass a `System.Ty` this method and get an instance of it in return. There is a strongly typed version of the `GetService` method, bu not know what type I am dealing with in advance, I have to use the version that returns an `Object` and then perfo case to `IController`.

Note Notice that I am not using the dependency resolver to address tight-coupling issues between classes. Instea asking it to create instances of types that I specify so that it can examine the dependencies that the controller classes and resolve them. I have not set up Ninject in this chapter, which means that the default resolver will be used and tha creates instances by looking for parameterless constructors and invoking them. However, by building my controller f the `DependencyResolver` class, I ensure that I can seamlessly take advantage of more advanced dependen like Ninject if one is added to the project.

Implementing the Other Interface Methods

Two other methods are in the `IControllerFactory` interface:

- The `GetControllerSessionBehavior` method is used by the MVC Framework to determine should be maintained for a controller. I will come back to this in the "Using Sessionless Controllers" sectio chapter.
- The `ReleaseController` method is called when a controller object created by the `CreateCo` method is no longer needed. In my example implementation, I check to see if the class implements the `IDi` interface. If it does, I call the `Dispose` method to release any resources that can be freed.

eBooks-IT.org

suitable for most projects and can be used verbatim (although you should read the section on sessionless controllers chapter to make sure you understand the options available).

Registering a Custom Controller Factory

I tell the MVC Framework to use the custom controller factory through the `ControllerBuilder` class. You register custom factory controllers when the application is started, which means using the `Application_Sta` in the `Global.asax.cs` file, as shown in Listing 19-7.

Listing 19-7. Registering a Custom Controller Factory in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using ControllerExtensibility.Infrastructure;

namespace ControllerExtensibility {

    public class MvcApplication : System.Web.HttpApplication {

        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ControllerBuilder.Current.SetControllerFactory(new
                CustomControllerFactory());
        }
    }
}
```

Once the controller factory has been registered, it will be responsible for handling all of the requests that the applic receives. You can see the effect of the custom factory by starting the application. The browser will request the root will be mapped to the `Home` controller by the routing system. The custom factory will handle the request for the `H` controller by creating an instance of the `ProductController` class, which will produce the result shown in

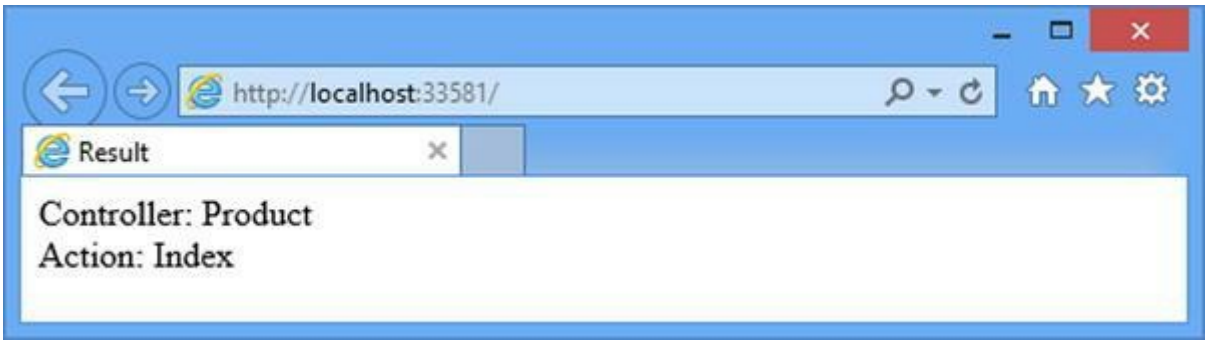


Figure 19-2. Using the custom controller factory

Working with the Built-in Controller Factory

I showed you how to create a custom controller factory because it is the most effective way of demonstrating what factory does and how it functions. For most applications, however, the built-in controller factory class, called

eBooks-IT.org

system looks at the routing data to find the value of the `Controller` property and tries to find a class in the Web applic meets the following criteria:

- The class must be `public`.
- The class must be concrete (not `abstract`).
- The class must *not* take generic parameters.
- The name of the class must end with `Controller`.
- The class must implement the `IController` interface.

The `DefaultControllerFactory` class maintains a list of such classes in the application so that it do perform a search every time a request arrives. If a suitable class is found, then an instance is created using the contr (I will come back to this in the upcoming "Customizing DefaultControllerFactory Controller Creation" section), and t controller is complete. If there is no matching controller, then the request cannot be processed any further.

Notice how the `DefaultControllerFactory` class follows the convention-over-configuration pattern need to register your controllers in a configuration file, because the factory will find them for you. All you need to do classes that meet the criteria that the factory is seeking.

If you want to create custom controller factory behavior, you can configure the settings of the default factory or of the methods. This way, you are able to build on the useful convention-over-configuration behavior without having it, a task which, I noted earlier, is complicated and painful. In the sections that follow, I show you different ways to controller creation.

Prioritizing Namespaces

In Chapter 16, I showed you how to prioritize one or more namespaces when creating a route. This was to address controller problem, where controller classes have the same name but reside in different namespaces. It is the `DefaultControllerFactory` that processes the list of namespaces and prioritizes them.

Tip Global prioritization is overridden by route-specific prioritization. This means you can define a global policy, individual routes as required. See Chapter 16 for details on specifying namespaces for individual routes.

If you have an application that has a lot of routes, it can be more convenient to specify priority namespaces global they are applied to all of your routes. Listing 19-8 shows how to do this in the `Application_Start` method `Global.asax` file. (This is where I put these statements, but you can also use the `RouteConfig.cs` file `App_Start` folder if you prefer.)

Listing 19-8. Global Namespace Prioritization in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using ControllerExtensibility.Infrastructure;

namespace ControllerExtensibility {

    public class MvcApplication : System.Web.HttpApplication {

        protected void Application_Start() {

            AreaRegistration.RegisterAllAreas();
```

```
                ControllerBuilder.Current.DefaultNamespaces.Add("MyCon
                ControllerBuilder.Current.DefaultNamespaces.Add("MyPro
            }
        }
}
```

I use the static `ControllerBuilder.Current.DefaultNamespaces.Add` method to add that should be given priority. The order in which I add the namespaces does not imply any kind of search order or re All of the namespaces defined by the `Add` method are treated equally and the priority is relative to those namespaces not been specified by the `Add` method. This means that the controller factory will search the entire application if it c suitable controller class in the namespaces defined by the `Add` method.

Tip Notice that I used an asterisk character (*) in the second statement shown in bold in Listing 19-8. This allow specify that the controller factory should look in the `MyProject` namespace and any child namespaces that `MyP` contains. Although this looks like regular expression syntax, it isn't; you can end your namespaces with `.*`, but you any other regular expression syntax with the `Add` method.

Customizing DefaultControllerFactory Controller Instantiation

There are a number of ways to customize how the `DefaultControllerFactory` class instantiates contr By far, the most common reason for customizing the controller factory is to add support for DI. There are several di of doing this. The most suitable technique depends on how you are using DI elsewhere in your application.

Using the Dependency Resolver

The `DefaultControllerFactory` class will use a dependency resolver to create controllers if one is ava covered dependency resolvers in Chapter 6 and showed you the `NinjectDependencyResolver` class, w implements the `IDependencyResolver` interface to provide Ninject DI support. I also demonstrated how to `DependencyResolver` class earlier in this chapter when I created my own custom controller factory. The `DefaultControllerFactory` will call the `IDependencyResolver.GetService` method controller instance, which gives you the opportunity to resolve and inject any dependencies.

Using a Controller Activator

You can also introduce DI into controllers by creating a *controller activator*. You create this activator by implementin `IControllerActivator` interface, as shown in Listing 19-9.

Listing 19-9. The IControllerActivator Interface

```
namespace System.Web.Mvc {
    using System.Web.Routing;

    public interface IControllerActivator {
                IController  Create(RequestContext  requestContex
controllerType);
    }
}
```

The interface contains one method, called `Create`, which is passed a `RequestContext` object describing and a `Type` that specifies which controller class should be instantiated.

To demonstrate an implementation of this interface, I added a new class file called `CustomControllerActivator.cs` in the `Infrastructure` folder and used it to define the clas

Listing 19-10. The Contents of the CustomControllerActivator.cs File

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using ControllerExtensibility.Controllers;

namespace ControllerExtensibility.Infrastructure {
    public class CustomControllerActivator : IControllerActivator

        public IController Create(RequestContext requestContext,
            Type controllerType) {

            if (controllerType == typeof(ProductController)) {
                controllerType = typeof(CustomerController);
            }

(IController)DependencyResolver.Current.GetService(controllerType)
        }
    }
}
```

This `IControllerActivator` implementation is simple. If the `ProductController` class is re responds with an instance of the `CustomerController` class. This is not something you would want to do project, but it demonstrates how you can use the `IControllerActivator` interface to intercept requests b controller factory and the dependency resolver.

To use a custom activator, I need to pass an instance of the implementation class to the `DefaultControllerFactory` constructor and register the result in the `Application_Start` m `Global.asax` file, as shown in Listing 19-11.

Listing 19-11. Registering a Custom Activator in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using ControllerExtensibility.Infrastructure;

namespace ControllerExtensibility {

    public class MvcApplication : System.Web.HttpApplication {

        protected void Application_Start() {

            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ControllerBuilder.Current.SetControllerFactory(new
                                                DefaultControllerFac
CustomControllerActivator()));
        }
```

eBooks-IT.org

}

You can see the effect of the custom activator if you start the application and navigate to the `/Product` URL. target the `Product` controller and the `DefaultControllerFactory` will ask the activator to instantia `ProductFactory` class, but my activator intercepts this request and creates an instance of the `CustomerController` class instead, as shown in Figure 19-3.

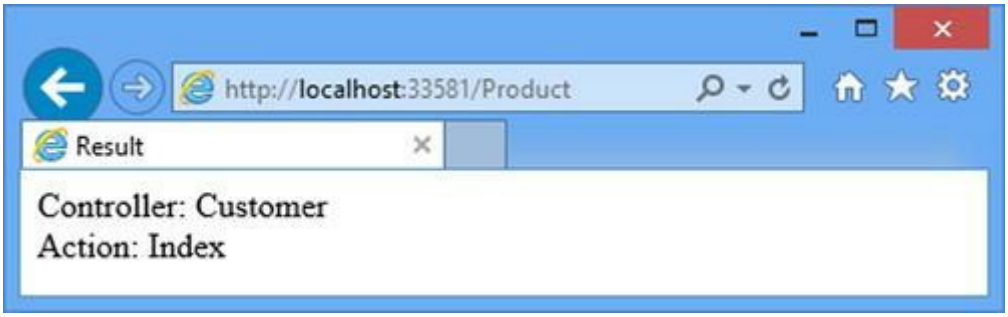


Figure 19-3. Intercepting instantiation requests using a custom controller activator

Overriding DefaultControllerFactory Methods

You can override methods in the `DefaultControllerFactory` class to customize the creation of contro 19-3 describes the three methods you can override, each of which performs a different role.

***Table 19-3**. Overridable DefaultContollerFactory Methods*

Method	Result	Description
`CreateController`	`IController`	The implementation of the `CreateController` method from the `IControl` interface. By default, this method calls `GetControllerType` to determine which instantiated, and then gets a controller object by passing the result to the `GetControllerInstance` method.
`GetControllerType`	`Type`	Maps requests to controller types. This is where most of the conventions listed earlie are enforced.
`GetControllerInstance`	`IController`	Creates an instance of a specified type.

Creating a Custom Action Invoker

Once the controller factory has created an instance of a class, the framework needs a way of invoking an action on t If you derived your controller from the `Controller` class, then this is the responsibility of an *action invoker*, w subject of this section.

Tip If you create a controller directly from the `IController` interface, then you are responsible for execut yourself. (See details of creating controllers in this way.) Action invokers are part of the functionality included in the `Controller` class.

An action invoker implements the `IActionInvoker` interface, which is shown in Listing 19-12.

***Listing 19-12**.* The IActionInvoker Interface

```
namespace System.Web.Mvc {

    public interface IActionInvoker {

            bool InvokeAction(ControllerContext controllerContext,
```

eBooks-IT.org

```
    }
}
```

The interface has only a single member: `InvokeAction`. The parameters are a `ControllerContext` (which I described in Chapter 17) and a string that contains the name of the action to be invoked. The result type is a value of `true` indicates that the action was found and invoked and `false` indicates that the controller has no ma

Notice that I have not used the word *method* in this description. The association between actions and methods is s optional. Although this is the approach that the built-in action invoker takes, you are free to handle actions any way th choose. Listing 19-13 shows an implementation of the `IActionInvoker` interface that takes a different appro defined in a class file called `CustomActionInvoker.cs` in the `Infrastructure` folder.

Listing 19-13. The Contents of the CustomActionInvoker.cs File

```
using System.Web.Mvc;

namespace ControllerExtensibility.Infrastructure {

    public class CustomActionInvoker : IActionInvoker {

        public bool InvokeAction(ControllerContext controllerConte
                string actionName) {

            if (actionName == "Index") {
                controllerContext.HttpContext.
                            Response.Write("This is output from th
action");
                return true;
            } else {
                return false;
            }
        }
    }
}
```

This action invoker doesn't care about the methods in the controller class. In fact, it deals with actions itself. If th for the `Index` action, then the invoker writes a message directly to the `Response`. If the request is for any ot then it returns `false`, which causes a `404—Not found` error to be displayed to the user.

The action invoker associated with a controller is obtained through the `Controller.ActionInvoker` means that different controllers in the same application can use different action invokers. To demonstrate this, I have controller to the example project called `ActionInvoker`, the definition of which you can see in Listing 19-14.

Listing 19-14. The Contents of the ActionInvokerController.cs File

```
using ControllerExtensibility.Infrastructure;
using System.Web.Mvc;

namespace ControllerExtensibility.Controllers {

    public class ActionInvokerController : Controller {
        public ActionInvokerController() {
            this.ActionInvoker = new CustomActionInvoker();
        }
    }
}
```

eBooks-IT.org

n see I
works by starting the application and navigating to the /ActionInvoker/Index URL. The custom action i
generate the response shown in Figure 19-4. If you navigate to a URL that targets any other action on the same cont
see the 404 error page.

Figure 19-4. The effect of a custom action invoker

I am not suggesting that you implement your own action invoker. And, if you do, I do not suggest you follow this
Why? First, the built-in support has some useful features, as you will see shortly. Second, the example has some pro
of extensibility, poor separation of responsibilities, and a lack of support for views of any kind. But the example show
MVC Framework fits together and demonstrates, once again, that almost every aspect of the request processing pipe
customized or replaced entirely.

Using the Built-in Action Invoker

The built-in action invoker, which is the `ControllerActionInvoker` class, has some sophisticated techn
matching requests to actions. And, unlike my implementation in the previous section, the default action invoker opera
methods. To qualify as an action, a method must meet the following criteria:

- The method must be `public`.
- The method must *not* be `static`.
- The method must *not* be present in `System.Web.Mvc.Controller` or any of its base classes.
- The method must *not* have a special name.

The first two criteria are simple enough. For the next, excluding any method that is present in the `Controlle`
bases means that methods such as `ToString` and `GetHashCode` are excluded, as are the methods that impl
`IController` interface. This is sensible, because the inner workings of controllers should not be exposed to th
world. The last criterion means that constructors, property and event accessors are excluded. In fact, no class mem
the `IsSpecialName` flag from `System.Reflection.MethodBase` will be used to process an ac

Note Methods that have generic parameters (such as `MyMethod<T>()`) meet all of the criteria, but the MV
will throw an exception if you try to invoke such a method to process a request.

By default, the `ControllerActionInvoker` finds a method that has the same name as the requested a
example, if the `action` value that the routing system produces is `Index`, then the `ControllerActionI`
will look for a method called `Index` that fits the action criteria. If it finds such a method, it will be invoked to hand
This behavior is exactly what you want almost all of the time, but as you might expect, the MVC Framework provid
opportunities to fine-tune the process.

Using a Custom Action Name

Usually, the name of an action method determines the action that it represents. The `Index` action method services
the `Index` action. You can override this behavior using the `ActionName` attribute, which I have applied to the
`Customer` controller as shown in Listing 19-15.

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class CustomerController : Controller {

        public ViewResult Index() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "Index"
            });
        }

        [ActionName("Enumerate")]
        public ViewResult List() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "List"
            });
        }
    }
}
```

In this listing, I have applied the attribute to the `List` method, passing in a parameter value of `Enumerate`. V
action invoker receives a request for the `Enumerate` action, it will now use the `List` method to service it. You
effect of the `ActionName` attribute by starting the application and navigating to the `/Customer/Enumera`
You can see that the results shown by the browser in Figure 19-5 are those from the `List` method.

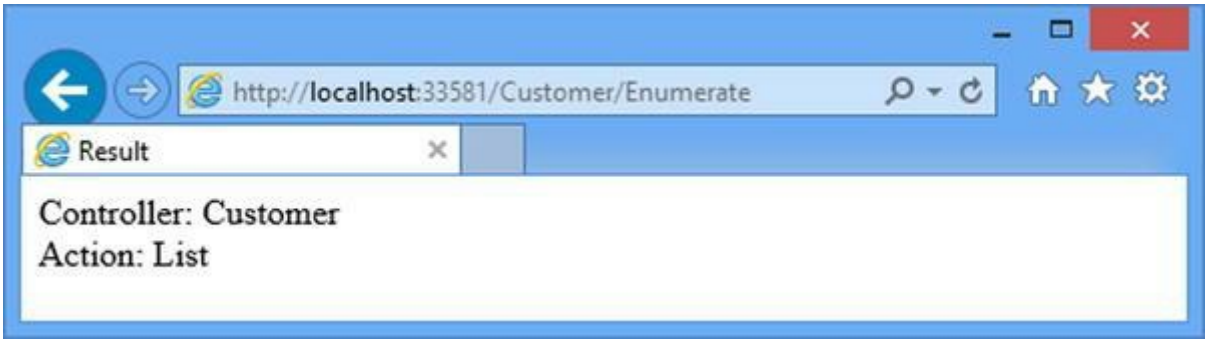


Figure 19-5. The effect of the ActionName attribute

Applying the attribute overrides the name of the action. This means that URLs which directly target the `List` me
longer work, as shown in Figure 19-6.

eBooks-IT.org

Figure 19-6. Using the method name as the action when the ActionName attribute has been applied

There are two main reasons why you might want to override a method name in this way:

- You can then accept an action name that wouldn't be legal as a C# method name (
  `[ActionName("User-Registration")]`).
- If you want to have two different C# methods that accept the same set of parameters and should handle th
  name, but in response to different HTTP request types (for example, one with `[HttpGet]` and t
  `[HttpPost]`), you can give the methods different C# names to satisfy the compiler, but then use `[Act`
  to map them both to the same action name.

Using Action Method Selection

It is often the case that a controller will contain several actions with the same name. This can be because there are m
methods, each with different parameters, or because you used the `ActionName` attribute so that multiple metho
the same action.

In these situations, the MVC Framework needs some help selecting the appropriate action with which to process a
mechanism for doing this is called *action method selection*. It allows you to define kinds of requests that an action is
process. You have already seen an example of action method selection when I restricted an action using the `HttpP`
attribute when I built the SportsStore application. I had two methods called `Checkout` in the `Cart` controller ar
`HttpPost` attribute to indicate that one of them was to be used only for HTTP `POST` requests, as shown in List

Listing 19-16. Using the HttpPost Attribute

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;
        private IOrderProcessor orderProcessor;
```

```
proc) {
            repository = repo;
            orderProcessor = proc;
        }

        // ...other action methods omitted for brevity...

        public ViewResult Checkout() {
            return View(new ShippingDetails());
        }

        [HttpPost]
                public ViewResult Checkout(Cart cart, Shippin
shippingDetails) {
            if (cart.Lines.Count() == 0) {
                    ModelState.AddModelError("", "Sorry, your
empty!");
            }

            if (ModelState.IsValid) {
                orderProcessor.ProcessOrder(cart, shippingDetails)
                cart.Clear();
                return View("Completed");
            } else {
                return View(shippingDetails);
            }
        }
    }
}
```

The action invoker uses action method selectors to resolve ambiguity when selecting an action. In Listing 19-16, t candidates for the `Checkout` action. The invoker gives preference to the actions that have selectors. In this case, `HttpPost` selector is evaluated to see if the request can be processed. If it can, then this is the method that will b not, then the *other* method, the one without the attribute, will be used.

There are built-in attributes that work as selectors for the different kinds of HTTP requests: `HttpPost` for PO `HttpGet` for `GET` requests, `HttpPut` for `PUT` requests, and so on. Another built-in attribute is `NonActio` indicates to the action invoker that a method that would otherwise be considered a valid action method should not be can see how I have applied the `NonAction` attribute in Listing 19-17, where I have defined a new action method `Customer` controller.

Listing 19-17. Using the NonAction Selector in the CustomerController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class CustomerController : Controller {

        public ViewResult Index() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "Index"
            });
        }
```

eBooks-IT.org

```
        public ViewResult List() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "List"
            });
        }

        [NonAction]
        public ActionResult MyAction() {
            return View();
        }
    }
}
```

The `MyAction` method in the listing will not be considered as an action method, even though it meets all of the the invoker looks for. This is useful for ensuring that you do not expose the workings of your controller classes as a course, normally such methods should simply be marked `private`, which will prevent them from being invoked however, `[NonAction]` is useful if for some reason you must mark such a method as `public`. Requests for target `NonAction` methods will generate `404—Not Found` errors, as shown in Figure 19-7.

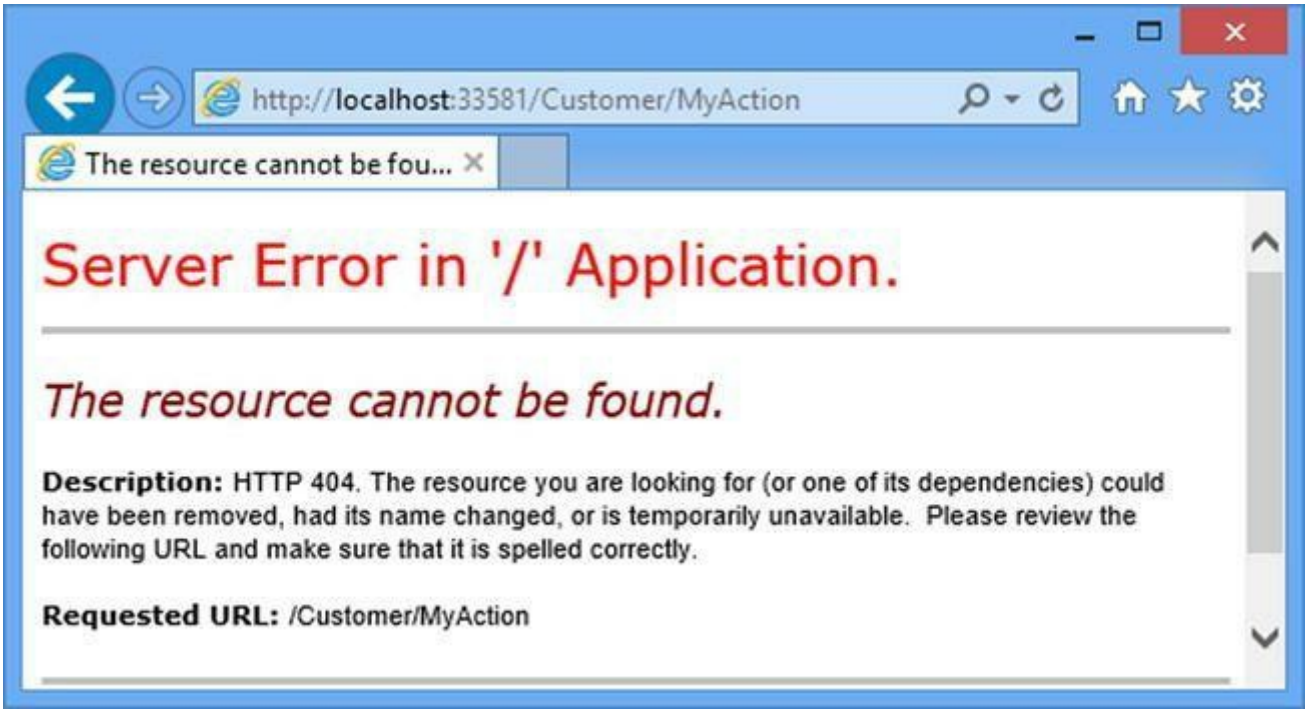


Figure 19-7. The effect of requesting a URL that targets a NonAction method

Creating a Custom Action Method Selector

Action method selectors are derived from the `ActionMethodSelectorAttribute` class, which is sho 19-18.

Listing 19-18. The ActionMethodSelectorAttribute Class

```
using System.Reflection;

namespace System.Web.Mvc {
        [AttributeUsage(AttributeTargets.Method,  AllowMultiple  =
```

eBooks-IT.org

```
    public abstract class ActionMethodSelectorAttribute : Attribut

                public abstract bool IsValidForRequest(Controlle
controllerContext,
            MethodInfo methodInfo);
    }
}
```

The `ActionMethodSelectorAttribute` is abstract and defines one abstract method: `IsValidForRequest`. The parameters for this method are a `ControllerContext` object, which all inspect the request, and a `MethodInfo` object, which you can use to get information about the method to which has been applied. You return `true` from `IsValidForRequest` if the method is able to process a request, a otherwise. I created a simple custom action method selector in a class file, `LocalAttribute.cs`, that I add `Infrastructure` folder of the example project, as shown in Listing 19-19.

Listing 19-19. The Contents of the LocalAttribute.cs File

```
using System.Reflection;
using System.Web.Mvc;

namespace ControllerExtensibility.Infrastructure {
    public class LocalAttribute : ActionMethodSelectorAttribute {

                public override bool IsValidForRequest(Controlle
controllerContext,
                MethodInfo methodInfo) {
            return controllerContext.HttpContext.Request.IsLocal;
        }
    }
}
```

I have overridden the `IsValidForRequest` method so that it returns `true` when the request originates f machine. To demonstrate the custom action method selector, I created a `Home` controller in the example project, as Listing 19-20.

Listing 19-20. The Contents of the HomeController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Infrastructure;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            return View("Result", new Result {
                ControllerName = "Home", ActionName = "Index"
            });
        }

        [ActionName("Index")]
        public ActionResult LocalIndex() {
            return View("Result", new Result {
                ControllerName = "Home", ActionName = "LocalIndex"
```

```
        }
    }
}
```

I have used the `ActionName` attribute to create a situation in which there are two `Index` action methods. A the action invoker doesn't have any way to figure out which one should be used when a request for the `/Home/I` arrives and will generate the error shown in Figure 19-8 when such a request is received.

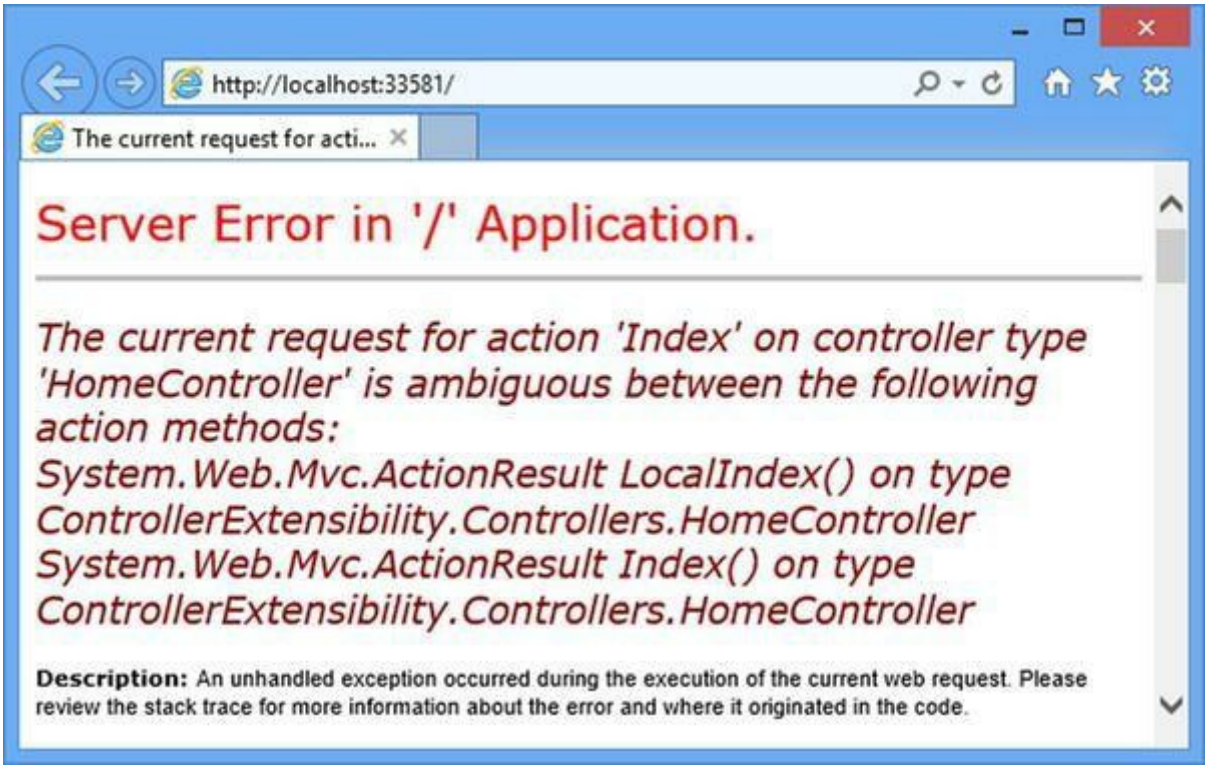


Figure 19-8. The error shown when there are ambiguous action method names

To resolve this situation, I can apply the custom method selection attribute to one of the ambiguous methods, as sl Listing 19-21.

Listing 19-21. Applying the Method Selection Attribute to the HomeController.cs File

```
...
[Local]
[ActionName("Index")]
public ActionResult LocalIndex() {
    return View("Result", new Result {
        ControllerName = "Home", ActionName = "LocalIndex"
    });
}
...
```

If you restart the application and navigate to the root URL from a browser running on the local machine, you will s MVC Framework takes the method selection attribute into account to resolve the ambiguity between the methods in t class, as shown in Figure 19-9.

eBooks-IT.org

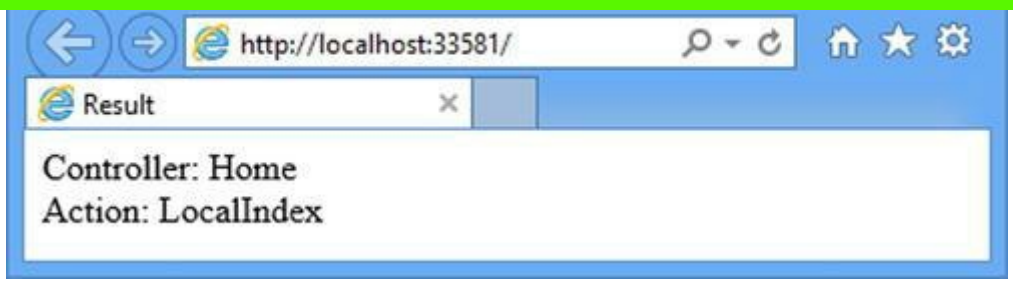


Figure 19-9. Using a method selection attribute to resolve action method ambiguity

THE ACTION METHOD DISAMBIGUATION PROCESS

Now that you have seen inside the action method selector base class, you can understand how the action invoke action method. The invoker starts the process with a list of possible candidates, which are the controller metho the action method criteria. Then it goes through the following process:

- The invoker discards any method based on name. Only methods that have the same name as the target ac suitable `ActionName` attribute are kept on the list.
- The invoker discards any method that has an action method selector attribute that returns `false` for the cu
- If there is exactly one action method with a selector left, then this is the method that is used. If there is method with a selector, then an exception is thrown, because the action invoker cannot disambiguate betwee methods.
- If there are no action methods with selectors, then the invoker looks at those without selectors. If there is ex method, then this is the one that is invoked. If there is more than one method without a selector, an except because the invoker can't choose between them.

Handling Unknown Actions

If the action invoker is unable to find an action method to invoke, it returns `false` from its `InvokeAction` this happens, the `Controller` class calls its `HandleUnknownAction` method. By default, this method `404—Not Found` response to the client. This is the most useful thing that a controller can do for most applicati can choose to override this method in your controller class if you want to do something special. Listing 19-22 provid demonstration of overriding the `HandleUnknownAction` method in the `Home` controller.

Listing 19-22. Overriding the HandleUnknownAction Method in the HomeController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Infrastructure;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class HomeController : Controller {

        // ...other action methods omitted for brevity...

        protected override void HandleUnknownAction(string actionN
            Response.Write(string.Format("You requested the {0}
actionName));
        }
    }
}
```

eBooks-IT.org

respons

Figure 19-10.

Figure 19-10. Dealing with requests for action methods that do not exist

Improving Performance with Specialized Controllers

The MVC Framework provides two special kinds of controllers that may improve the performance of your MVC we applications. Like all performance optimizations, these controllers represent compromises, either in ease of use or wi functionality. In the follow sections, I demonstrate both kinds of controllers and outline their benefits and shortcomin

Using Sessionless Controllers

By default, controllers support *session state*, which can be used to store data values across requests, making life easi MVC programmer. Creating and maintaining session state is an involved process. Data must be stored and retrieved, sessions themselves must be managed so that they expire appropriately. Session data consumes server memory or sp other storage location, and needing to synchronize the data across multiple Web servers makes it harder to run your a a server farm.

In order to simplify session state, ASP.NET will process only one query for a given session at a time. If the client multiple overlapping requests, they will be queued up and processed sequentially by the server. The benefit is that yo to worry about multiple requests modifying the same data. The downside is that you do not get the request throughp like.

Not all controllers need the session state features. In such cases, you can improve the performance of your applica avoiding work involved in maintaining session state. You do this by using *sessionless controllers.* These are just like r controllers, with two exceptions: the MVC Framework will not load or store session state when they are used to pro request, and overlapping requests can be processed simultaneously.

Managing Session State in a Custom IControllerFactory

At the start of this chapter, I showed you that the `IControllerFactory` interface contained a method calle `GetControllerSessionBehavior`, which returns a value from the `SessionStateBehavio` enumeration. That enumeration contains four values that control the session state configuration of a controller, as de Table 19-4.

***Table 19-4**. The Values of the SessionStateBehavior Enumeration*

Value	Description
`Default`	Use the default ASP.NET behavior, which is to determine the session state configuration from `HttpContext`.
`Required`	Full read-write session state is enabled.
`ReadOnly`	Read-only session state is enabled.
`Disabled`	Session state is disabled entirely.

A controller factory that implements the `IControllerFactory` interface directly sets the session state be controllers by returning `SessionStateBehavior` values from the `GetControllerSessionBeh` method. The parameters to this method are a `RequestContext` object and a string containing the name of the

eBooks-IT.org

demonstration, I have changed the implementation of the GetControllerSessionBehavior method in CustomControllerFactory class that I created earlier in the chapter, as shown in Listing 19-23.

Listing 19-23. Defining Session State Behavior for a Controller in the CustomControllerFactory.cs File

```
...
public SessionStateBehavior GetControllerSessionBehavior(RequestCo
        requestContext, string controllerName) {
    switch (controllerName) {
        case "Home":
            return SessionStateBehavior.ReadOnly;
        case "Product":
            return SessionStateBehavior.Required;
        default:
            return SessionStateBehavior.Default;
    }
}
...
```

Managing Session State Using DefaultControllerFactory

When you are using the built-in controller factory, you can control the session state by applying the SessionSta to individual controller classes, as shown in Listing 19-24 where I have created a new controller called FastCont

Listing 19-24. Using the SessionState Attribute in the FastController.cs File

```
using System.Web.Mvc;
using System.Web.SessionState;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {

    [SessionState(SessionStateBehavior.Disabled)]
    public class FastController : Controller {

        public ActionResult Index() {
            return View("Result", new Result {
                ControllerName = "Fast ",ActionName = "Index"
            });
        }
    }
}
```

The SessionState attribute is applied to the controller class and affects all of the actions in the controller. T parameter to the attribute is a value from the SessionStateBehavior enumeration. In the example, I disal state entirely, which means that if I try to set a session value in the controller, like this:

```
...
Session["Message"] = "Hello";
...
```

or try to read back from the session state in a view, like this:

```
...
```

eBooks-IT.org

...

The MVC Framework will throw an exception when the action is invoked or the view is rendered.

Tip When session state is `Disabled`, the `HttpContext.Session` property returns `null`.

If you have specified the `ReadOnly` behavior, then you can read values that have been set by other controllers, still get a runtime exception if you try to set or modify a value. You can get details of the session through the `HttpContext.Session` object but trying to alter any values causes an error.

Tip If you are simply trying to pass data from the controller to the view, consider using the View Bag feature ins not affected by the `SessionState` attribute.

Using Asynchronous Controllers

The underlying ASP.NET platform maintains a pool of .NET threads that are used to process client requests. This po the *worker thread pool*, and the threads are called *worker threads*. When a request is received, a worker thread is tak pool and given the job of processing the request. When the request has been processed, the worker thread is returne so that it is available to process new requests as they arrive. There are two key benefits of using thread pools for AS applications:

- By reusing worker threads, you avoid the overhead of creating a new one each time you process a request.
- By having a fixed number of worker threads available, you avoid the situation where you are processing more requests than your server can handle.

The worker thread pool works best when requests can be processed in a short period of time. This is the case for applications. However, if you have actions that depend on other servers and take a long time to complete, then you c point where all of your worker threads are tied up waiting for other systems to complete their work.

Note In this section, I assume that you are familiar with the Task Parallel Library (TPL). If you want to learn ab see my book on the topic, called *Pro .NET Parallel Programming in C#*, which is published by Apress.

Your server is capable of doing more work (after all, you are just waiting, which takes up little of your resources), you have tied up all of your worker threads, incoming requests are being queued up. You will be in the odd state of y application grinding to a halt while the server is largely idle.

Caution At this point, some readers are thinking that they can write a worker thread pool that is tailored to their *Do not do it*. Writing concurrent code is easy. Writing concurrent code *that works* is difficult. If you are new to con programming, then you lack the required skills. My advice is to stick with the default pool. If you are experienced in programming, then you already know that the benefits will be marginal compared with the effort of coding and testin thread pool.

The solution to this problem is to use an *asynchronous controller*. This increases the overall performance of your a but does not bring any benefits to the execution of your asynchronous operations.

Note Asynchronous controllers are useful only for actions that are I/O- or network-bound and *not* CPU-intensive problem you are trying to solve with asynchronous controllers is a mismatch between the pool model and the type of are processing. The pool is intended to ensure that each request gets a decent slice of the server resources, but you e set of worker threads that are doing nothing. If you use additional background threads for CPU-intensive actions, the dilute the server resources across too many simultaneous requests.

Creating the Example

To begin the exploration of asynchronous controllers, I am going to show you an example of the kind of problem tha intended to solve. Listing 19-25 shows a regular synchronous controller called `RemoteData` that I added to the e project.

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class RemoteDataController : Controller {

        public ActionResult Data() {
            RemoteService service = new RemoteService();
            string data = service.GetRemoteData();
            return View((object)data);
        }
    }
}
```

This controller contains an action method, `Data`, which creates an instance of the model class `RemoteServ` calls the `GetRemoteData` method on it. This method is an example of a time-consuming, low-CPU activity. Th `RemoteService` class, which I defined in a class file called `RemoteService.cs` in the `Models` fold in Listing 19-26.

Listing 19-26. The Contents of the RemoteService.cs File

```
using System.Threading;

namespace ControllerExtensibility.Models {
    public class RemoteService {

        public string GetRemoteData() {
            Thread.Sleep(2000);
            return "Hello from the other side of the world";
        }
    }
}
```

Okay, I admit it: I faked the `GetRemoteData` method. In the real world, this method could be retrieving con across a slow network connection, but to keep things simple, I used the `Thread.Sleep` method to simulate a t delay. The last addition I need is a new view. I created the `Views/RemoteData` folder and added the `Data` view file to it, the contents of which are shown in Listing 19-27.

Listing 19-27. The Contents of the Data.cshtml File

```
@model string

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Data</title>
</head>
<body>
```

eBooks-IT.org

```
        Data: @Model
    </div>
</body>
</html>
```

When you run the application and navigate to the `/RemoteData/Data` URL, the action method is invoked, `RemoteService` object is created, and the `GetRemoteData` method is called. After two seconds (simula operation), the data is returned from the `GetRemoteData` method, passed to the view and rendered as Figure 1

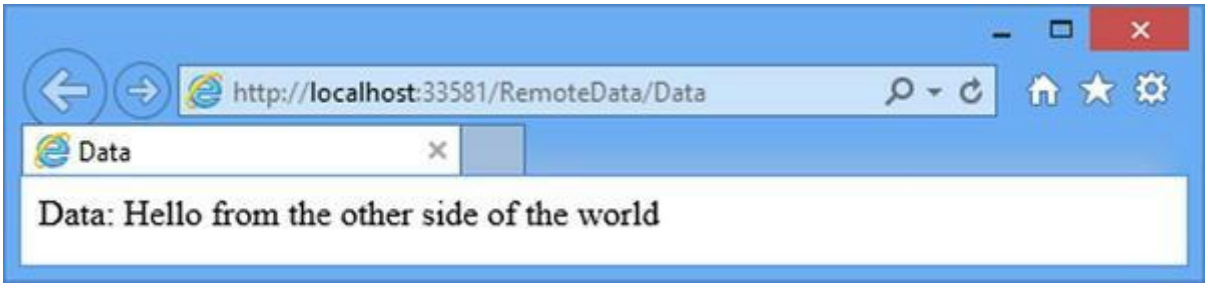


Figure 19-11. Navigating to the /RemoteData/Data URL

The problem here is that the worker thread that was handling the request was idle for two seconds. It wasn't doin useful, and it was not available for handling other requests while it was waiting.

Caution Using an asynchronous controller frees up the worker thread so that it can process other queries. It do the user from experiencing a two-second wait. After all, that fake data still has to be obtained and processed. There a techniques you can use to make such requests asynchronously in the browser, which allows you to at least keep the informed about the progress of getting the data and allow them to continue working with another part of the applicat *Pro ASP.NET MVC 5 Client* book, published by Apress in 2014, for details.

Creating an Asynchronous Controller

Having shown you the problem I was going to solve, I can now move on to create the asynchronous controller. The ways to create an asynchronous controller. One is to implement the `System.Web.Mvc.Async.IAsyncController` interface, which is the asynchronous equivalent of `IController`. I am not going to demonstrate that approach, because it requires so much explanation of the .N programming facilities.

Tip Not all actions in an asynchronous controller need to be asynchronous. You can include synchronous metho and they will behave as expected.

I want to stay focused on the MVC Framework, which is why I will demonstrate the second approach: to use the and `async` keywords in a regular controller.

In previous versions of the .NET Framework, creating asynchronous controllers was a complex process and requ the controller from a special class and splitting actions into two methods. The new `await` and `async` keywords described in Chapter 4, have simplified this process a lot: you create a new `Task` object and `await` its response, Listing 19-28.

Tip The old method of creating asynchronous action methods is still supported, although the approach I describe more elegant and the one I recommend. One artifact of the old approach is that you can't use action method names t `Async` (e.g., `IndexAsync`) or `Completed` (e.g., `IndexCompleted`).

Listing 19-28. Creating an Asynchronous Controller in the RemoteDataController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;
using System.Threading.Tasks;
```

eBooks-IT.org

```
namespace ControllerExtensibility.Controllers {
    public class RemoteDataController : Controller {

        public async Task<ActionResult> Data() {
            string data = await Task<string>.Factory.StartNew(() =
                return new RemoteService().GetRemoteData();
            });

            return View((object)data);
        }
    }
}
```

I have refactored the action method so that it returns a `Task<ActionResult>`, applied the `async` and a keywords, and created a `Task<string>`, which is responsible for calling the `GetRemoteData` method.

Consuming Asynchronous Methods in a Controller

You can also use an asynchronous controller to consume asynchronous methods elsewhere in your application. To d this, I have added an asynchronous method to the `RemoteService` class, as shown in Listing 19-29.

Listing 19-29. Adding an Asynchronous Method in the RemoteService.cs File

```
using System.Threading;
using System.Threading.Tasks;

namespace ControllerExtensibility.Models {
    public class RemoteService {

        public string GetRemoteData() {
            Thread.Sleep(2000);
            return "Hello from the other side of the world";
        }

        public async Task<string> GetRemoteDataAsync() {
            return await Task<string>.Factory.StartNew(() => {
                Thread.Sleep(2000);
                return "Hello from the other side of the world";
            });
        }
    }
}
```

The result from the `GetRemoteDataAsync` method is a `Task<string>`, which yields the same me synchronous method when it is completed. In Listing 19-30, you can see how I have consumed this asynchronous m new action method that I added to the `RemoteData` controller.

Listing 19-30. Consuming Asynchronous Methods in the RemoteData Controller

```
using System.Web.Mvc;
using ControllerExtensibility.Models;
using System.Threading.Tasks;
```

eBooks-IT.org

```
    public class RemoteDataController : Controller {

        public async Task<ActionResult> Data() {
            string data = await Task<string>.Factory.StartNew(() =
                return new RemoteService().GetRemoteData();
            });

            return View((object)data);
        }

        public async Task<ActionResult> ConsumeAsyncMethod() {
            string data = await new RemoteService().GetRemoteDataA
            return View("Data", (object)data);
        }
    }
}
```

You can see that both action methods follow the same basic pattern and that the difference is where the `Task` ob created. The result of calling either action method is that the worker thread is not tied up while I wait for the `GetRemoteData` call to complete, which means that the thread is available to process other requests which car improve the performance of your MVC Framework application.

Summary

In this chapter, you have seen how the MVC Framework creates controllers and invokes methods. I have explored a customized the built-in implementations of the key interfaces, and created custom versions to demonstrate how they have learned how action method selectors can be used to differentiate between action methods and seen some specia controllers that can be used to increase the request processing capability of your applications.

The underlying theme of this chapter is extensibility. Almost every aspect of the MVC Framework can be modifiec entirely. For most projects, the default behaviors are entirely sufficient. But having a working knowledge of how the Framework fits together helps you to make informed design and coding decisions (and it is just plain interesting).

In the next chapter, I turn to views. I explain how they work and, as you will have come to expect by now, how t and customize the default behaviors.

eBooks-IT.org

Views

In Chapter 17, you saw how action methods can return `ActionResult` objects. As you learned, the most com action result is `ViewResult`, which causes a view to be rendered and returned to the client. You have seen view in many examples already, so you know roughly what they do. In this chapter, I focus and clarify that knowledge. I showing you how the MVC Framework handles `ViewResults` using *view engines*, including demonstrating hov custom view engine. Next, I will describe techniques for working effectively with the built-in Razor View Engine. Th cover how to create and use partial views, child actions, and Razor sections, which are all essential topics for effecti development. Table 20-1 provides the summary for this chapter.

Table 20-1. *Chapter Summary*

Problem	Solution	Listing
Create a custom view engine	Implement the `IViewEngine` and `IView` interfaces	1–8
Customize the Razor view engine	Derive from the `RazorViewEngine` class	9–15
Define regions of content for use in a layout	Use Razor sections	16
Apply sections in a layout	Use the `RenderSection` and `RenderBody` helpers	17–22
Define reusable fragments of markup	Use partial views	23–26
Define reusable business logic	Use child actions	27–29

Creating a Custom View Engine

I am going to dive in at the deep end and create a custom view engine. You do not need to do this for most projects MVC Framework includes the Razor view engine, whose syntax I described in Chapter 5 and which I have been usir the examples so far in this book.

Tip Older versions of the MVC Framework supported views created using the same markup and view engine as Web Forms, which is why you will sometimes see references to `.aspx` files in debugging and error messages.

The value in creating a custom view engine is to demonstrate how the request processing pipeline works and comp knowledge of how the MVC Framework operates. This includes understanding just how much freedom view engines translating a `ViewResult` into a response to the client. View engines implement the `IViewEngine` interface shown in Listing 20-1.

Listing 20-1. The IViewEngine Interface from the MVC Framework

```
namespace System.Web.Mvc {
    public interface IViewEngine {

                        ViewEngineResult   FindPartialView(Controlle
controllerContext,
            string partialViewName, bool useCache);

        ViewEngineResult FindView(ControllerContext controllerCont
```

```
        void ReleaseView(ControllerContext controllerContext, IVie
    }
}
```

The role of a view engine is to translate requests for views into `ViewEngineResult` objects. The first two the interface, `FindView` and `FindPartialView`, are passed parameters that describe the request and the processed it (a `ControllerContext` object), the name of the view and its layout, and whether the view eng to reuse a previous result from its cache. These methods are called when a `ViewResult` is being processed. Th method, `ReleaseView`, is called when a view is no longer needed.

Note The MVC Framework support for view engines is implemented by the `ControllerActionInvo` which is the built-in implementation of the `IActionInvoker` interface, as described in Chapter 17. You will n automatic access to the view engines feature if you have implemented your own action invoker or controller factory the `IActionInvoker` or `IControllerFactory` interfaces.

The `ViewEngineResult` class allows a view engine to respond to the MVC Framework when a view is re Listing 20-2 shows the `ViewEngineResult` class.

Listing 20-2. The ViewEngineResult Class from the MVC Framework

```
using System.Collections.Generic;

namespace System.Web.Mvc {
    public class ViewEngineResult {

        public ViewEngineResult(IEnumerable<string> searchedLocati
            if (searchedLocations == null) {
                throw new ArgumentNullException("searchedLocations
            }
            SearchedLocations = searchedLocations;
        }

        public ViewEngineResult(IView view, IViewEngine viewEngine
            if (view == null) { throw new ArgumentNullException("v
                                if (viewEngine == null) { thr
ArgumentNullException("viewEngine");}
            View = view;
            ViewEngine = viewEngine;
        }

        public IEnumerable<string> SearchedLocations { get; privat
        public IView View { get; private set; }
        public IViewEngine ViewEngine { get; private set; }
    }
}
```

You express a result by choosing one of the two constructors. If your view engine is able to provide a view for a you create a `ViewEngineResult` using this constructor:

```
...
public ViewEngineResult(IView view, IViewEngine viewEngine)
...
```

The parameters to this constructor are an implementation of the `IView` interface and a view engine (so that the `ReleaseView` method can be called later). If your view engine cannot provide a view for a request, then you us

```
...
public ViewEngineResult(IEnumerable<string> searchedLocations)
...
```

The parameter for this version is an enumeration of the places you searched to find a view. This information is dis user if no view can be found, as I will demonstrate later.

Note You are not alone if you think that the `ViewEngineResult` class is a little awkward. Expressing ou different versions of a class constructor is an odd approach and does not really fit with the rest of the MVC Framew

The last building block of the view engine system is the `IView` interface, which is shown in Listing 20-3.

Listing 20-3. The IView Interface from the MVC Framework

```
using System.IO;

namespace System.Web.Mvc {
    public interface IView {
        void Render(ViewContext viewContext, TextWriter writer);
    }
}
```

An `IView` implementation is passed to the constructor of a `ViewEngineResult` object, which is then re the view engine methods. The MVC Framework then calls the `Render` method. The `ViewContext` paramete information about the request from the client and the output from the action method. The `TextWriter` paramete writing output to the client.

The `ViewContext` object defines properties that give you access to information about the request and details MVC Framework has processed it so far. I have described the most useful of these properties in Table 20-2.

Table 20-2. *Useful ViewContext Properties*

Name	Description
`Controller`	Returns the `IController` implementation that processed the current request
`RequestContext`	Returns details of the current request
`RouteData`	Returns the routing data for the current request
`TempData`	Returns the temp data associated with the request
`View`	Returns the implementation of the `IView` interface that will process the request. Obviously, this will be the curre are creating a custom view implementation.
`ViewBag`	Returns an `object` that represents the view bag
`ViewData`	Returns a dictionary of the view model data, which also contains the view bag and meta data for the model. See Ta details.

The most interesting of these properties is `ViewData`, which returns a `ViewDataDictionary` object. `ViewDataDictionary` class defines a number of useful properties that give access to the view model, the v the view model metadata. I have described the most useful of these properties in Table 20-3.

Table 20-3. *Useful ViewDataDictionary Properties*

Name	Description
`Keys`	Returns a collection of key values for the data in the dictionary, which can be used to access view bag properties
`Model`	Returns the view model object for the request
`ModelMetadata`	Returns a `ModelMetadata` object that can be used to reflect on the model type

As I said earlier, the simplest way to see how this works—how IViewEngine, IView, and ViewEngi fit together—is to create a view engine. I am going to create a simple view engine that returns one kind of view. This render a result that contains information about the request and the view data produced by the action method. This ap me demonstrate the way that view engines operate without getting bogged down in parsing view templates.

Preparing the Example Project

The example project for this part of the chapter is called Views and I created it using the Empty template, check to add the core MVC folders and references. I created a Home controller, which you can see in Listing 20-4.

Listing 20-4. The Contents of the HomeController.cs File

```
using System;
using System.Web.Mvc;

namespace Views.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Message = "Hello, World";
            ViewBag.Time = DateTime.Now.ToShortTimeString();
            return View("DebugData");
        }

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

I have not created any views for this project because I am going to implement a custom view engine rather than re Razor.

Creating a Custom IView

I am going to start by creating an implementation of the IView interface. I added an Infrastructure folde example project and created a new class file within it called DebugDataView.cs, which is shown in Listing 2

Listing 20-5. The Contents of the DebugDataView.cs

```
using System.IO;
using System.Web.Mvc;

namespace Views.Infrastructure {
    public class DebugDataView : IView {

        public void Render(ViewContext viewContext, TextWriter wri

            Write(writer, "---Routing Data---");
            foreach (string key in viewContext.RouteData.Values.Ke
                Write(writer, "Key: {0}, Value: {1}",
                    key, viewContext.RouteData.Values[key]);
            }
```

eBooks-IT.org

```
            foreach (string key in viewContext.ViewData.Keys) {
                Write(writer, "Key: {0}, Value: {1}", key,
                    viewContext.ViewData[key]);
            }
        }

        private void Write(TextWriter writer, string template,
object[] values) {
            writer.Write(string.Format(template, values) + "<p/>")
        }
    }
}
```

This view demonstrates the use of the two parameters to the `Render` method: I take values from the `ViewCo` write a response to the client using the `TextWriter`. First I write out the routing data information and then the

Tip The view data feature is a holdover from earlier versions of the MVC Framework that were released before C support for dynamic objects (which I described in Chapter 4). View data was a less flexible precursor to the view ba used directly any more, except when writing custom `IView` implementations when it provides easy access to the p defined on the view bag object.

Creating an IViewEngine Implementation

Remember that the purpose of the view engine is to produce a `ViewEngineResult` object that contains eithe or a list of the places that searched for a suitable view. Now that I have an `IView` implementation to work with, I view engine. I added a class file called `DebugDataViewEngine.cs` in the `Infrastructure` folder, of which are shown in Listing 20-6.

Listing 20-6. The Contents of the DebugDataViewEngine.cs File

```
using System.Web.Mvc;

namespace Views.Infrastructure {
    public class DebugDataViewEngine : IViewEngine {

                    public  ViewEngineResult  FindView(Controlle
controllerContext,
                string viewName, string masterName, bool useCache)

            if (viewName == "DebugData") {
                return new ViewEngineResult(new DebugDataView(), t
            } else {
                return new ViewEngineResult(new string[]
                    { "No view (Debug Data View Engine)" });
            }
        }

               public  ViewEngineResult  FindPartialView(Controlle
controllerContext,
                string partialViewName, bool useCache) {

            return new ViewEngineResult(new string[]
                { "No view (Debug Data View Engine)" });
        }
```

```
view) {
            // do nothing
        }
    }
}
```

I am going to support only a single view, which is called DebugData. When I see a request for that view, I w new instance of the custom IView implementation, like this:

```
...
return new ViewEngineResult(new DebugDataView(), this);
...
```

If I were implementing a more serious view engine, I would use this opportunity to search for templates, taking in layout and provided caching settings. As it is, this simple example only requires a new instance of the DebugData class. If I receive a request for a view other than DebugData, I return a ViewEngineResult, like this:

```
...
return new ViewEngineResult(new string[] { "No view (Debug Da
Engine)" });
...
```

The IViewEngine interface presumes that the view engine has places it needs to look to find views. This is a assumption, because views are typically template files that are stored as files in the project. In this case, I do not hav look, so I just return a dummy location which will indicate that I was asked for a view that cannot be delivered.

The custom view engine doesn't support partial views, so I return a result from the FindPartialView me indicates I do not have a view to offer. I return to the topic of partial views and how they are handled in the Razor en the chapter. I have not implemented the ReleaseView method, because there are no resources that I need to re custom IView implementation, which is the usual purpose of this method.

Registering a Custom View Engine

I register view engines in the Application_Start method of Global.asax, as shown in Listing 20-7

Listing 20-7. Registering a Custom View Engine Using Global.asax

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using Views.Infrastructure;

namespace Views {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ViewEngines.Engines.Add(new DebugDataViewEngine());
        }
    }
}
```

eBooks-IT.org

MVC Framework supports the idea of there being several engines installed in a single application. When a ViewRe
being processed, the action invoker obtains the set of installed view engines and calls their FindView methods in

The action invoker stops calling FindView methods as soon as it receives a ViewEngineResult objec
contains an IView. This means that the order in which engines are added to the ViewEngines.Engines
significant if two or more engines are able to service a request for the same view name. If you want your view to tak
precedence, then you can insert it at the start of the collection, like this:

```
...
ViewEngines.Engines.Insert(0, new DebugDataViewEngine());
...
```

Testing the View Engine

I am now in a position to test the custom view engine. When the application is started, the browser will automatically
the root URL for the project, which will be mapped to the Index action in the Home controller. The action metho
View method to return a ViewResult that specifies the DebugData view. You can see the result of this i

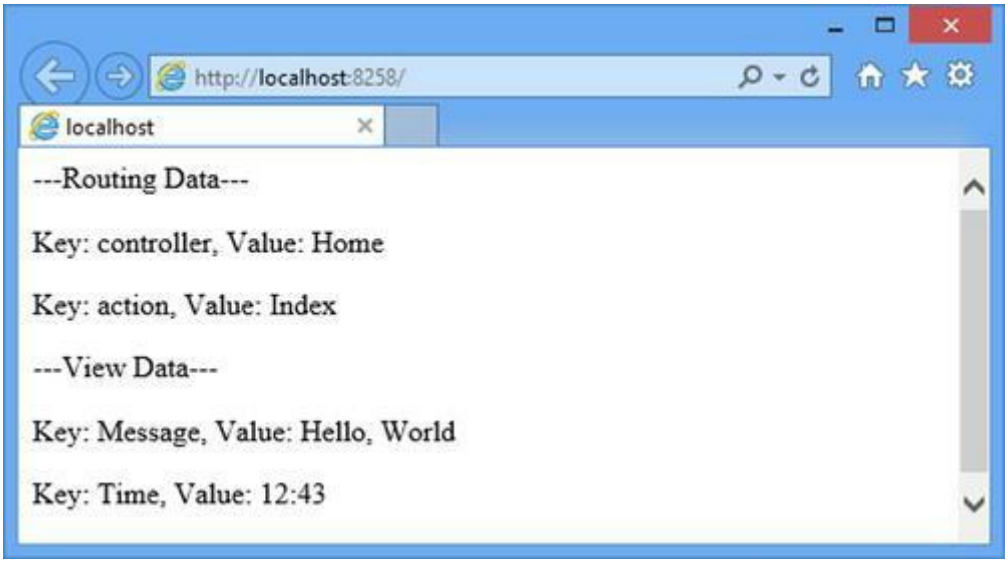


Figure 20-1. Using a custom view engine

This is the result of the FindView method being called for a view that I am able to process. If you navigate to
/Home/List URL, the MVC Framework will invoke the List action method, which calls the View method
default view, which is not one that is supported. You can see the result in Figure 20-2.

eBooks-IT.org

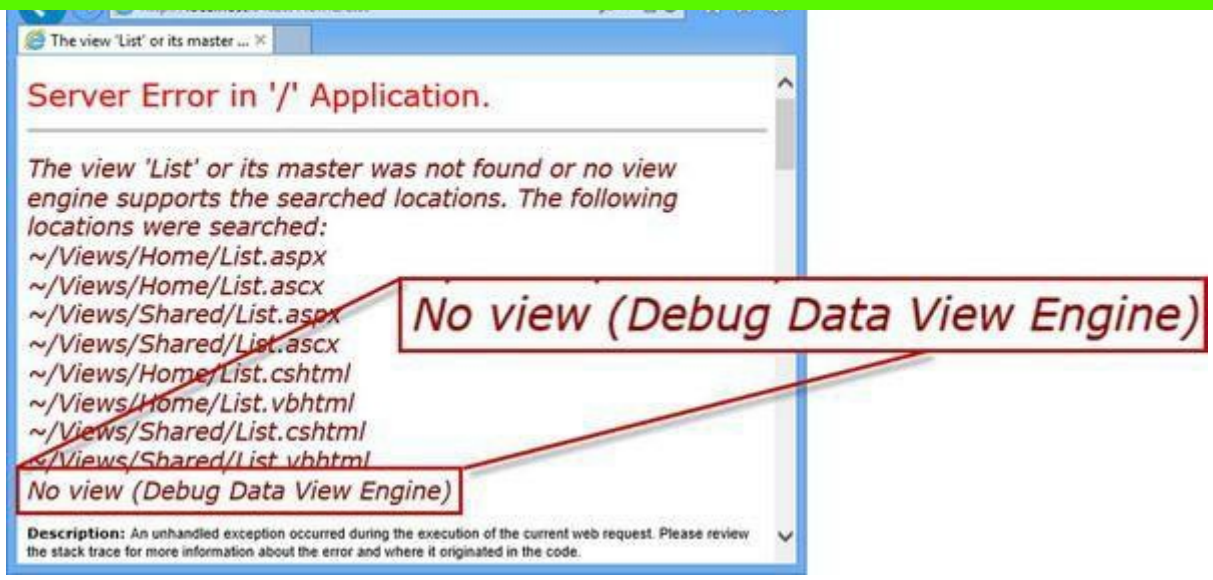


Figure 20-2. Requesting an unsupported view

You can see that my message is reported as one of the locations that have been searched for a view. Notice that R ASPX views appear on the list as well. This is because those view engines are still being used. If I want to ensure th custom view engine is used, then I have to call the `Clear` method before I register my engine in the `Global.a` shown in Listing 20-8.

Listing 20-8. Removing the Other View Engines in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using Views.Infrastructure;

namespace Views {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ViewEngines.Engines.Clear();
            ViewEngines.Engines.Add(new DebugDataViewEngine());
        }
    }
}
```

If you restart the application and navigate to `/Home/List` again, only the custom view engine will be used, as Figure 20-3.

eBooks-IT.org

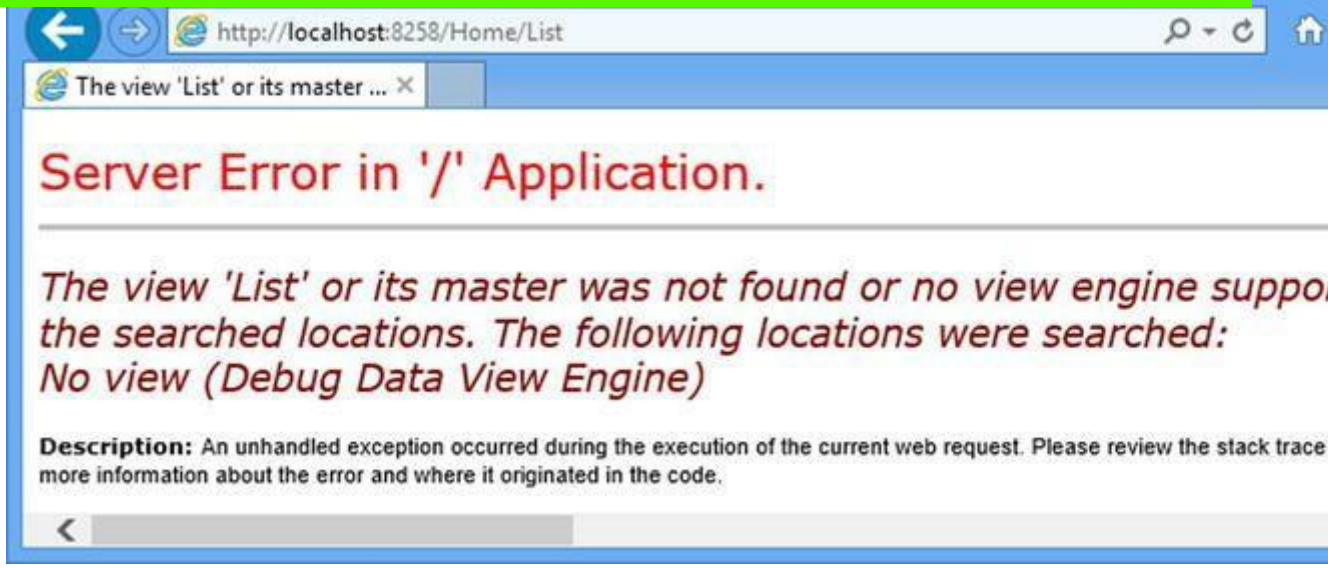


Figure 20-3. Using only the custom view engine in the example application

Working with the Razor Engine

In the previous section, I was able to create a custom view engine by implementing just two interfaces. Admittedly, I with something simple that generated ugly views, but you saw how the concept of MVC extensibility continues throu request processing pipeline.

The complexity in a view engine comes from the system of view templates that includes code fragments and supp and is compiled to optimize performance. I did not do any of these things in the simple custom view engine—and the need to—because the built-in Razor engine takes care of all of that for me. The functionality that almost all MVC app require is available in Razor. Only a vanishingly small number of projects need to go to the trouble of creating a custo engine. I gave you a primer on the Razor syntax in Chapter 5. In this chapter, I am going to show you how to use ot to create and render Razor views. You will also learn how to customize the Razor engine.

Preparing the Example Project

For this part of the chapter, I have created a new MVC project using the `Empty` template option, checking the opti core MVC folders and references. I called the project `WorkingWithRazor` and I added a `Home` controller, v shown in Listing 20-9.

Listing 20-9. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace WorkingWithRazor.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            string[] names = { "Apple", "Orange", "Pear" };
            return View(names);
        }
    }
}
```

I also created a view called `Index.cshtml` in the `Views/Home` folder. You can see the contents of the v Listing 20-10.

```
@model string[]

@{
    ViewBag.Title = "Index";
}

This is a list of fruit names:

@foreach (string name in Model) {
    <span><b>@name</b></span>
}
```

Understanding Razor View Rendering

The Razor View Engine compiles the views in your applications to improve performance. The views are translated in and then compiled, which is why you are able to include C# code fragments so easily. It is instructive to look at the s that Razor views generate, because it helps to put many of the Razor features in context.

The views in an MVC application are not compiled until the application is started, so to see the classes that are crea you need to start the application and navigate to the `/Home/Index` action. The initial request to MVC application compilation process for all views. You can see the output from the request in Figure 20-4.

Figure 20-4. The output from the Index action method on the Home controller

Conveniently, classes generated from the view files are written to the disk as C# code files and then compiled, whi you can see the C# statements that represent a view. You can find the generated files in `c:\Users\<` *yourLogin* `\AppData\Local\Temp\Temporary ASP.NET Files` on Windows 7 and Windows 8.

Finding the code file generated for a particular view requires a bit of poking around. There are usually a number of cryptic names, and the names of the `.cs` files do not correspond to the names of the classes they contain. As an ex found the generated class for the view in Listing 20-10 in a file called `App_Web_ihpp0d0l.0.cs` in the `root\7bbfc2bc\bd7485cd` folder. I have tidied up the class from my system to make it easier to read, as Listing 20-11.

Listing 20-11. The Generated C# Class for a Razor View

```
namespace ASP {
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using System.Net;
    using System.Web;
    using System.Web.Helpers;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.WebPages;
    using System.Web.Mvc;
```

eBooks-IT.org

```
    using System.Web.Mvc.Ajax;
    using System.Web.Mvc.Html;
    using System.Web.Optimization;
    using System.Web.Routing;

                    public     class     _Page_Views_Home_Index_csht
System.Web.Mvc.WebViewPage<string[]> {

        public _Page_Views_Home_Index_cshtml() {
        }

        public override void Execute() {
            ViewBag.Title = "Index";
                        WriteLiteral("\r\n\r\nThis  is  a  list  o
names:\r\n\r\n");

            foreach (string name in Model) {
                WriteLiteral("    <span><b>");
                Write(name);
                WriteLiteral("</b></span>\r\n");
            }
        }
    }
}
```

First, note that the class is derived from `WebViewPage<T>`, where `T` is the model type: `WebViewPage<string[]>` for this example. This is how strongly typed views are handled. Also notice the class that has been generated: `_Page_Views_Home_Index_cshtml`. You can see how the path of the v been encoded in the class name. This is how Razor maps requests for views into instances of compiled classes.

In the `Execute` method, you can see how the statements and elements in the view have been handled. The cod that I prefixed with the `@` symbol are expressed directly as C# statements. The HTML elements are handled with the `WriteLiteral` method, which writes the contents of the parameter to the result as they are given. This is oppo `Write` method, which is used for C# variables and encodes the string values to make them safe for use in an HTM

Both the `Write` and `WriteLiteral` methods write content to a `TextWriter` object. This is the same passed to the `IView.Render` method, which you saw at the start of the chapter. The goal of a compiled Razor generate the static and dynamic content and send it to the client via the `TextWriter`. This is useful to keep in m turn to HTML helper methods later in later chapters.

Configuring the View Search Locations

The Razor View Engine follows a standard convention when looking for a view. For example, if you request the `In` associated with the `Home` controller, Razor looks through this list of views:

- `~/Views/Home/Index.cshtml`
- `~/Views/Home/Index.vbhtml`
- `~/Views/Shared/Index.cshtml`
- `~/Views/Shared/Index.vbhtml`

As you now know, Razor is not really looking for the view files on disk, because they have already been compiled classes. Razor looks for the compiled class that represents these views. The `.cshtml` files are templates containi statements (the kind I am using), and the `.vbhtml` files contain Visual Basic statements.

You can change the view files that Razor searches for by creating a subclass of `RazorViewEngine`. This c Razor `IViewEngine` implementation. It builds on a series of base classes that define a set of properties that dete view files are searched for. These properties are described in Table 20-4.

eBooks-IT.org

Property	Description	Default Value
ViewLocationFormats MasterLocationFormats PartialViewLocationFormats	The locations to look for views, partial views, and layouts	~/Views/{1}/{0}.cshtml, ~/Views/{1}/{0}.vbhtml, ~/Views/Shared/{0}.cshtml ~/Views/Shared/{0}.vbhtml
AreaViewLocationFormats AreaMasterLocationFormats AreaPartialViewLocationFormats	The locations to look for views, partial views, and layouts for an area	~/Areas/{2}/Views/{1}/{ ~/Areas/{2}/Views/{1}/{0} ~/Areas/{2}/Views/Shared/ ~/Areas/{2}/Views/Shared/

These properties predate the introduction of Razor, which why each set of three properties has the same values. E is an array of strings, which are expressed using the composite string formatting notation. The following are the para that correspond to the placeholders:

- {0} represents the name of the view.
- {1} represents the name of the controller.
- {2} represents the name of the area.

To change the search locations, you create a new class that is derived from RazorViewEngine and change for one or more of the properties described in Table 20-4.

To demonstrate how to change the locations that are searched, I added an Infrastructure folder to the p created a class file called CustomLocationViewEngine.cs, which is shown in Listing 20-12.

Listing 20-12. The Contents of the CustomLocationViewEngine.cs File

```
using System.Web.Mvc;

namespace WorkingWithRazor.Infrastructure {
    public class CustomLocationViewEngine : RazorViewEngine {

        public CustomLocationViewEngine() {
            ViewLocationFormats = new string[]
                {"~/Views/{1}/{0}.cshtml", "~/Views/Common/{0}.csht
        }
    }
}
```

I have set a new value for the ViewLocationFormats. The new array contains entries only for .csht addition, I have changed the location I look for shared views to be Views/Common, rather than Views/Sha register the derived view engine using the ViewEngines.Engines collection in the Application_St of Global.asax, as shown in Listing 20-13.

Listing 20-13. Registering the Custom View Engine in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using WorkingWithRazor.Infrastructure;
```

eBooks-IT.org

```
namespace WorkingWithRazor {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ViewEngines.Engines.Clear();
            ViewEngines.Engines.Add(new CustomLocationViewEngine()
        }
    }
}
```

Remember that the action invoker goes to each view engine in turn to see if a view can be found. By the time that add the view to the collection, it will already contain the standard Razor View Engine. To avoid competing with that implementation, I call the `Clear` method to remove any other view engines that may have been registered, and then `Add` method to register the custom implementation.

To demonstrate the changed locations, I created the `/Views/Common` folder and added a view file called `List.cshtml`. You can see the contents of this file in Listing 20-14.

Listing 20-14. The Contents of the /Views/Common/List.cshtml File

```
@{
    ViewBag.Title = "List";
}

<h3>This is the /Views/Common/List.cshtml View</h3>
```

To display this view, I added a new action method to the `Home` controller, as shown in Listing 20-15.

Listing 20-15. Adding a New Action Method in the HomeController.cs File

```
using System.Web.Mvc;

namespace WorkingWithRazor.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            string[] names = { "Apple", "Orange", "Pear" };
            return View(names);
        }

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

When I start the application and navigate to the `/Home/List` URL, the custom locations will be used to locate `List.cshtml` view file in the `/Views/Common` folder, as shown in Figure 20-5.

eBooks-IT.org

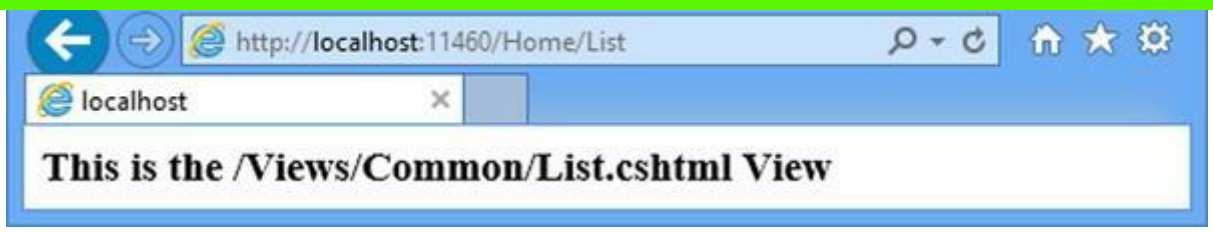


Figure 20-5. The effect of custom locations in the view engine

Adding Dynamic Content to a Razor View

The whole purpose of views is to allow you to render parts of your domain model as a user interface. To do that, yo able to add *dynamic content* to views. Dynamic content is generated at runtime, and can be different for each and ev This is opposed to *static content*, such as HTML, which you create when you are writing the application and is the s and every request. You can add dynamic content to views in the different ways described in Table 20-5.

***Table 20-5**. Adding Dynamic Content to a View*

Technique	When to Use
Inline code	Use for small, self-contained pieces of view logic, such as `if` and `foreach` statements. This is the fundamental tool for cr content in views, and some of the other approaches are built on it. I introduced this technique in Chapter 5 and you have seen examples in the chapters since.
HTML helper methods	Use to generate single HTML elements or small collections of them, typically based on view model or view data values. The M includes a number of useful HTML helper methods, and it is easy to create your own. HTML helper methods are the topic of
Sections	Use for creating sections of content that will be inserted into layout at specific locations.
Partial views	Use for sharing subsections of view markup between views. Partial views can contain inline code, HTML helper methods, and r other partial views. Partial views do not invoke an action method, so they cannot be used to perform business logic.
Child actions	Use for creating reusable UI controls or widgets that need to contain business logic. When you use a child action, it invokes an renders a view, and injects the result into the response stream.

Two of these options, inline code and HTML helper methods, are covered elsewhere in this book and I describe th the sections that follow.

Using Layout Sections

The Razor engine supports the concept of *sections*, which allow you to provide regions of content within a layout. R give greater control over which parts of the view are inserted into the layout and where they are placed. To demonst sections feature, I have edited the `/Views/Home/Index.cshtml` file, as shown in Listing 20-16.

***Listing 20-16**. Defining a Section in the Index.cshtml File*

```
@model string[]

@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Header {
    <div class="view">
        @foreach (string str in new [] {"Home", "List", "Edit"}) {
            @Html.ActionLink(str, str, null, new { style = "marg
})
```

```
    </div>
}

<div class="view">
    This is a list of fruit names:

    @foreach (string name in Model) {
        <span><b>@name</b></span>
    }
</div>

@section Footer {
    <div class="view">
        This is the footer
    </div>
}
```

Sections are defined using the Razor `@section` tag followed by a name for the section. In this listing, I create
called `Header` and `Footer`. The content of a section contains the usual mix of HTML markup and Razor tags.
defined in the view, but applied in a layout with the `@RenderSection` helper method. To demonstrate how thi
created the `/Views/Shared/_Layout.cshtml` file, the contents of which you can see in Listing 20-17

Listing 20-17. Using Sections in the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderBody()

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Footer")

    <div class="layout">
        This is part of the layout
    </div>
</body>
```

Tip My custom view engine locations are still in use, but I have specified the view explicitly in the `Index.cs` which means that my layout will be obtained from the `/Views/Shared` folder, even though shared views are l `/Views/Common` folder.

When Razor parses the layout, the `RenderSection` helper method is replaced with the contents of the secti with the specified name. The parts of the view that are not contained with a section are inserted into the layout using `RenderBody` helper.

You can see the effect of the sections by starting the application, as shown in Figure 20-6. I added some basic CS help make it clear which sections of the output are from the view and which are from the layout. This result is not p neatly demonstrates how you can put regions of content from the view into specific locations in the layout.

Figure 20-6. Using sections in a view to locate content in a layout

Note A view can define only the sections that are referred to in the layout. The MVC Framework will throw an e you attempt to define sections in the view for which there is no corresponding `@RenderSection` helper call in

Mixing the sections in with the rest of the view is unusual. The convention is to define the sections at either the sta of the view, to make it easier to see which regions of content will be treated as sections and which will be captured b `RenderBody` helper. Another approach, which I tend to use, is to define the view solely in terms of sections, inc for the body, as shown in Listing 20-18.

Listing 20-18. Defining a View in Terms of Razor Sections in the Index.cshtml File

```
@model string[]

@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Header {
    <div class="view">
        @foreach (string str in new [] {"Home", "List", "Edit"}) {
             @Html.ActionLink(str, str, null, new { style = "marg
})
        }
    </div>
}

@section Body {
```

eBooks-IT.org

```
        This is a list of fruit names:

        @foreach (string name in Model) {
            <span><b>@name</b></span>
        }
    </div>
}

@section Footer {
    <div class="view">
        This is the footer
    </div>
}
```

I find this makes for clearer views and reduces the chances of extraneous content being captured by `RenderBo` this approach, I have to replace the call to the `RenderBody` helper with `RenderSection("Body")`, as Listing 20-19.

Listing 20-19. Using RenderSection("Body") in the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Body")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Footer")

    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>
```

Testing For Sections

eBooks-IT.org

section if a view does not need or want to provide specific content. I have modified the _Layout.cshtml file default see if a Footer section is defined, as shown in Listing 20-20.

Listing 20-20. Checking Whether a Section Is Defined in the _Layout.cshtml File

```
...
@if (IsSectionDefined("Footer")) {
    @RenderSection("Footer")
} else {
    <h4>This is the default footer</h4>
}
...
```

The IsSectionDefined helper takes the name of the section you want to check and returns true if the rendering defines that section. In the example, I used this helper to determine if I should render some default content view does not define the Footer section.

Rendering Optional Sections

By default a view has to contain all of the sections for which there are RenderSection calls in the layout. If s missing, then the MVC Framework will report an exception to the user. To demonstrate this, I have added a new RenderSection call to the _Layout.cshtml file for a section called scripts, as shown in Listing a section that Visual Studio adds to the layout by default when you create an MVC project using the MVC template.

Listing 20-21. A RenderSection Call for Which There Is No Corresponding Section in the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Body")

    <div class="layout">
        This is part of the layout
    </div>

    @if (IsSectionDefined("Footer")) {
        @RenderSection("Footer")
    } else {
```

```
    }

    @RenderSection("scripts")

    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>
```

When you start the application and the Razor engine attempts to render the layout and the view, you will see the er
Figure 20-7.

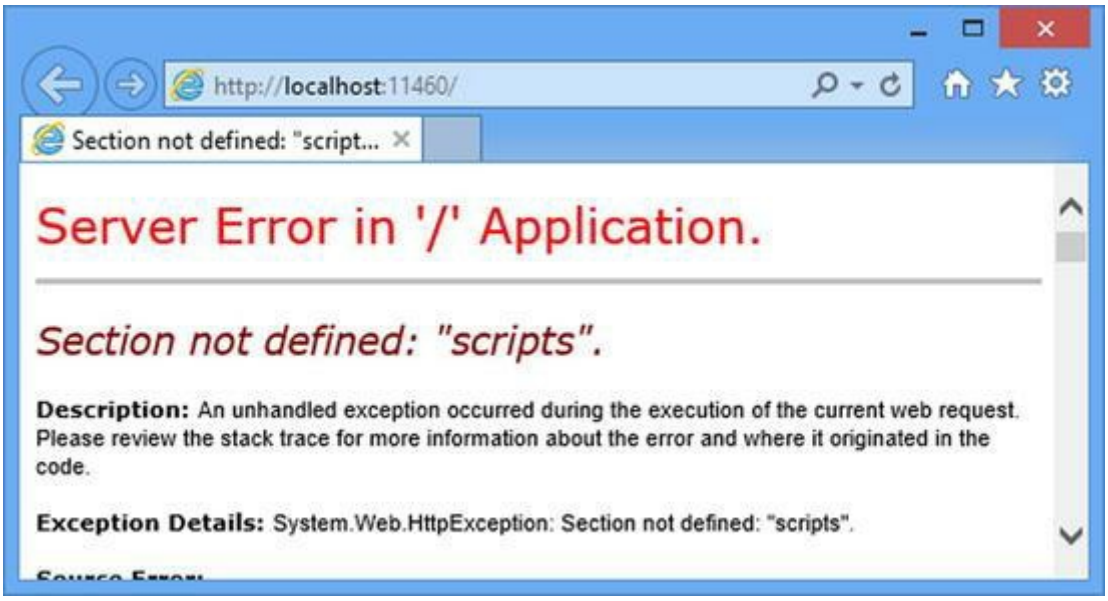


Figure 20-7. The error shown when there is a missing section

You can use the `IsSectionDefined` method to avoid making `RenderSection` calls for sections tha
does not define, but a more elegant approach is to use optional sections, which you do by passing an additional `fal`
the `RenderSection` method, as shown in Listing 20-22.

Listing 20-22. Making a Section Optional

```
...
@RenderSection("scripts", false)
...
```

This creates an optional section, the contents of which will be inserted into the result if the view defines it and wh
throw an exception otherwise.

Using Partial Views

You will often need to use the same fragments of Razor tags and HTML markup in several different places in the app
Rather than duplicate the content, you can use *partial views*, which are separate view files that contain fragments of
markup that can be included in other views. In this section, I show you how to create and use partial views, explain
work, and demonstrate the techniques available for passing view data to a partial view.

Creating a Partial View

eBooks-IT.org

folder, s

➤ View from the popup menu. Visual Studio will display the Add View dialog window, which you have seen i chapters. Set View Name to MyPartial, Template to Empty (without model) and check as partial view option, as shown in Figure 20-8.

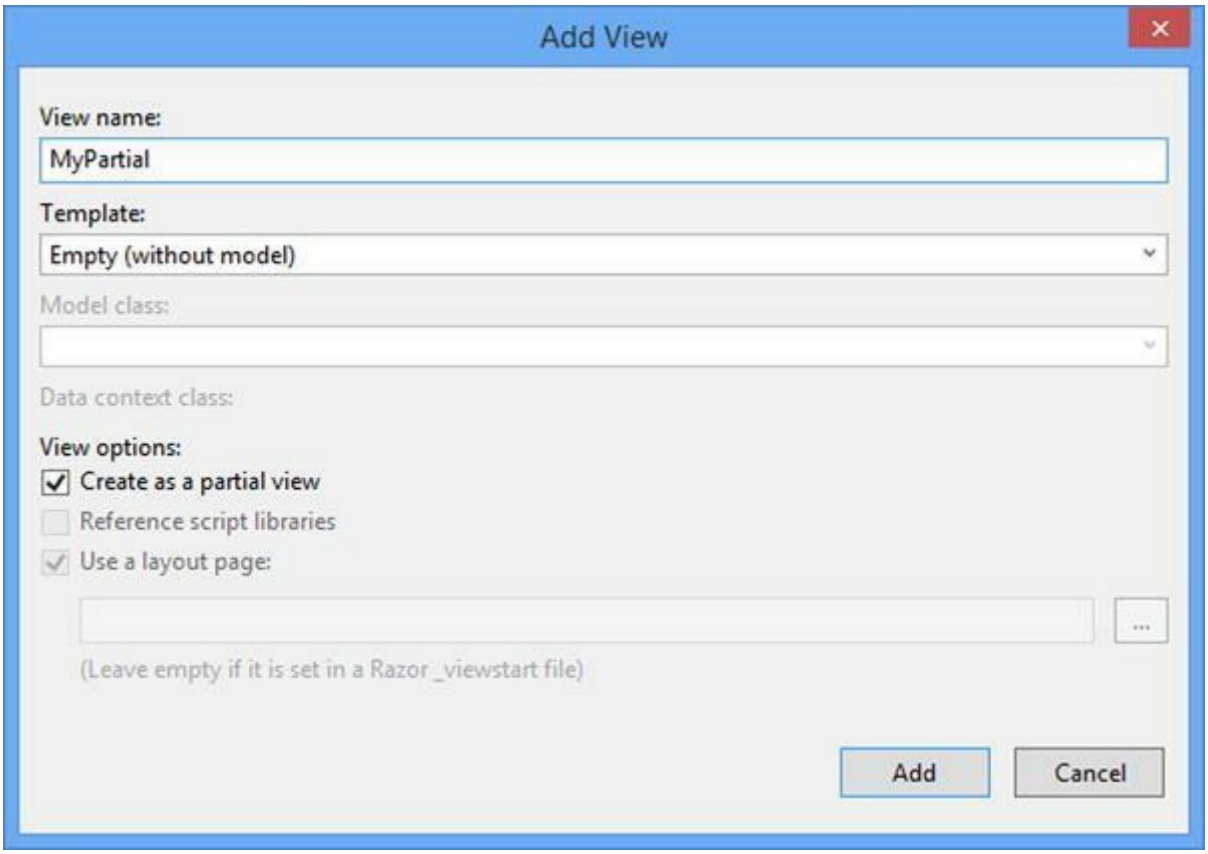


Figure 20-8. Creating a partial view

Click the Add button and Visual Studio will create the partial view, which is initially empty. I added the content sh Listing 20-23.

Listing 20-23. The Content of the MyPartial.cshtml File

```
<div>
    This is the message from the partial view.
    @Html.ActionLink("This is a link to the Index action", "Index"
</div>
```

Tip The scaffolding feature only sets the initial content for a file. What makes a view a partial is its content (it on fragment of HTML, rather than a complete HTML document, and doesn't reference layouts) and the way that it is us describe shortly). Once you are familiar with the different kinds of view, you can just use Add ➤ MVC 5 View (Razor) and set the contents you require directly.

I want to demonstrate that you can mix HTML markup and Razor tags in a partial view, so I have defined a simple a call to the ActionLink helper method. A partial view is consumed by calling the Html.Partial helper within another view. To demonstrate this, I have made the changes to the ~/Views/Common/List.cshtm shown in Listing 20-24.

Listing 20-24. Consuming a Partial View in the List.cshtml File

```
@{
    ViewBag.Title = "List";
```

```
}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyPartial")
```

I specify the name of the partial view file without the file extension. The view engine will look for the partial view specified in the usual locations, which means the `/Views/Home` and `/Views/Shared` folders for this exa called the `Html.Partial` method in a view that is being rendered for the `Home` controller. (I set the `Layou` `null` so that I do not have to specify the sections I defined in the `_Layout.cshtml` file used earlier in the c

Tip The Razor View Engine looks for partial views in the same way that it looks for regular views (in the `~/Views/<controller>` and `~/Views/Shared` folders). This means that you can create specialized partial views that are controller-specific and override partial views of the same name in the `Shared` folder. This m an odd thing to do, but one of the most common uses of partial views is to render content in layouts, and this feature handy.

You can see the effect of consuming the partial view by starting the application and navigating to the `/Home/L` shown in Figure 20-9.

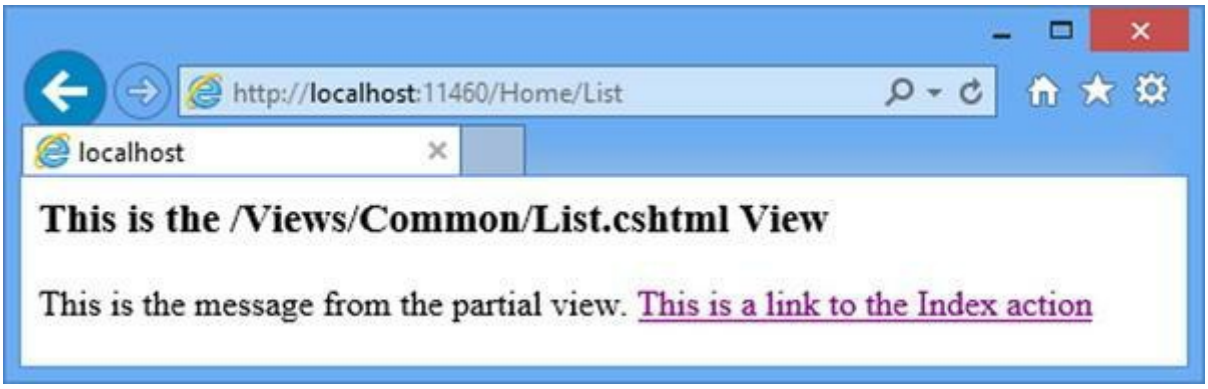


Figure 20-9. The effect of consuming a partial view

Tip The call I made to the `ActionLink` helper method in the partial view takes its controller information fro that is being processed. That means that when I specified the `Index` method, the `a` element will refer to the `Hom` since that was the controller that led to the partial view being rendered. If I consume the partial view in a view being another controller, then the `ActionLink` would generate a reference to that controller instead. I come back to th HTML helper methods in Chapter 21.

Using Strongly Typed Partial Views

You can create strongly typed partial views, and then pass view model objects to be used when the partial view is re demonstrate this feature, I created a new strongly typed partial view called `MyStronglyTypedPartial.` the `/Views/Shared` folder. This time, rather than use the scaffold option, I selected `Add` ➤ `MVC 5 Vie` `(Razor)`, set the name to `MyStronglyTypedPartial` and clicked the `OK` button to create the view. A in the previous section, there is nothing about the file itself that denotes a partial view, just the content and the way it application. I removed the default content that Visual Studio adds to new view files and replaced it with the markup s Listing 20-25.

Listing 20-25. The Contents of the MyStronglyTypedPartial.cshtml File

```
@model IEnumerable<string>

<div>
    This is the message from the partial view.
```

eBooks-IT.org

```
        @foreach (string str in Model) {
            <li>@str</li>
        }
    </ul>
</div>
```

I use a Razor `@foreach` loop to display the contents of the view model object as items in an HTML list. To de use of this partial view, I updated the `/Views/Common/List.cshtml` file, as shown in Listing 20-26.

Listing 20-26. Consuming a Strongly Typed Partial View in the List.cshtml File

```
@{
    ViewBag.Title = "List";
    Layout = null;
}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyStronglyTypedPartial", new    []    {"Apple",    "
"Pear"})
```

The difference from the previous example is that I pass an additional argument to the `Partial` helper method v the view model object. You can see the strongly typed partial view in use by starting the application and navigating to `/Home/List` URL, as shown in Figure 20-10.

Figure 20-10. Consuming a strongly typed partial view

Using Child Actions

Child actions are action methods invoked from within a view. This lets you avoid repeating controller logic that you v several places in the application. Child actions are to actions as partial views are to views. You can use a child action you want to display some data-driven widget that appears on multiple pages and contains data unrelated to the main a running. I used this technique in the SportsStore example to include a data-driven navigation menu on every page, wi to supply the navigation data directly from every action method. The navigation data was supplied independently by t action.

Creating a Child Action

Any action can be used as a child action. To demonstrate the child action feature, I have added a new action method controller, as shown in Listing 20-27.

```
using System;
using System.Web.Mvc;

namespace WorkingWithRazor.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            string[] names = { "Apple", "Orange", "Pear" };
            return View(names);
        }

        public ActionResult List() {
            return View();
        }

        [ChildActionOnly]
        public ActionResult Time() {
            return PartialView(DateTime.Now);
        }
    }
}
```

The action method is called `Time` and it renders a partial view by calling the `PartialView` method (which Chapter 17). The `ChildActionOnly` attribute ensures that an action method can be called *only* as a child met within a view. An action method doesn't need to have this attribute to be used as a child action, but I tend to use it to action methods from being invoked as a result of a user request.

Having defined an action method, I need to create the partial view that will be rendered when the action is invoked are typically associated with partial views, although this is not compulsory. Listing 20-28 shows the `/Views/Home/Time.cshtml` view that I created for this demonstration. This is a strongly typed partial v view model is a `DateTime` object.

Listing 20-28. The Contents of the Time.cshtml File

```
@model DateTime

<p>The time is: @Model.ToShortTimeString()</p>
```

Rendering a Child Action

Child actions are invoked using the `Html.Action` helper. With this helper, the action method is executed, the `ViewResult` is processed, and the output is injected into the response to the client. Listing 20-29 shows the cha made to the `/Views/Common/List.cshtml` file to render the child action.

Listing 20-29. Calling a Child Action in the List.cshtml File

```
@{
    ViewBag.Title = "List";
    Layout = null;
}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyStronglyTypedPartial",   new   []   {"Apple",   "
```

```
@Html.Action("Time")
```

You can see the effect of the child action by starting the application and navigating to the `/Home/List` URL a shown in Figure 20-11.

Figure 20-11. Using a child action

When I called the `Action` helper in Listing 20-29, I provided a single parameter that specified the name of the a to invoke. This causes the MVC Framework to look for an action method in the controller that is handling the curren call action methods in other controllers, provide the controller name, like this:

```
...
@Html.Action("Time", "MyController")
...
```

You can pass parameters to action methods by providing an anonymously typed object whose properties correspo names of the child action method parameters. So, for example, if I have this child action:

```
...
[ChildActionOnly]
public ActionResult Time(DateTime time) {
    return PartialView(time);
}
...
```

then I can invoke it from a view as follows:

```
...
@Html.Action("Time", new { time = DateTime.Now })
...
```

Summary

In this chapter, I explored the details of the MVC view system and the Razor View Engine. You have seen how to cr view engine, how to customize the behavior of the default Razor engine and the different techniques available for ins dynamic content into a view. In the next chapter, I focus on helper methods, which assist in generating content that into your views.

eBooks-IT.org

Helper Methods

In this chapter, I look at the *helper methods*, which allow you to package up chunks of code and markup so that they reused throughout an MVC Framework application. I start by showing you how to create your own helper methods. Framework comes with a wide range of built-in helper methods, and I explore them in this chapter and the next two starting with the helper methods that you can use to create HTML `form`, `input` and `select` elements. Table the summary for this chapter.

***Table 21-1**. Chapter Summary*

Problem	Solution
Create a region of reusable markup within a view	Create an inline helper
Create markup that can be used within multiple views	Create an external helper
Generate a `form` element	Use the `Html.BeginForm` and `Html.EndForm` helpers
Generate a `form` element using a specific route	Use the `Html.BeginRouteForm` helper
Generate `input` elements	Use the input helpers
Generate `input` elements from model objects	Use the strongly typed input helpers
Generate `select` elements	Use the `DropDownList` and `ListBox` helpers and their strongly typed counterparts

Preparing the Example Project

For this chapter, I created a new Visual Studio MVC project called `HelperMethods` using the `Empty` templa the option to add the core `MVC` folders and references. I added a `Home` controller, which you can see in Listing 21-

***Listing 21-1**.* The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace HelperMethods.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {

            ViewBag.Fruits = new string[] {"Apple", "Orange", "Pea
             ViewBag.Cities = new string[] { "New York", "London",
};

            string message = "This is an HTML element: <input>";

            return View((object)message);
        }
    }
```

In the Index action method, I pass a pair of string arrays to the view via the view bag and set the model ol string. I added a view called Index.cshtml to the Views/Home folder. You can see the contents of t Listing 21-2. This is a strongly typed view (where the model type is string) and I have not used a layout.

Listing 21-2. The Contents of the Index.cshtml File

```
@model string

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Here are the fruits:
        @foreach (string str in (string[])ViewBag.Fruits) {
            <b>@str </b>
        }
    </div>
    <div>
        Here are the cities:
        @foreach (string str in (string[])ViewBag.Cities) {
            <b>@str </b>
        }
    </div>
    <div>
        Here is the message:
        <p>@Model</p>
    </div>
</body>
</html>
```

Setting the Start URL

I want Visual Studio to start with the root URL for the application rather than guess the URL based on the file that is Select HelperMethods Properties from the Visual Studio Project menu, switch to the Web tab a Specific Page option in the Start Action section. You don't have to provide a value. Just checking enough.

Testing the Example Application

You can see how the view is rendered by starting the application. The default routing configuration added to the proj Studio will map the root URL requested automatically by the browser to the Index action on the Home controller Figure 21-1.

eBooks-IT.org

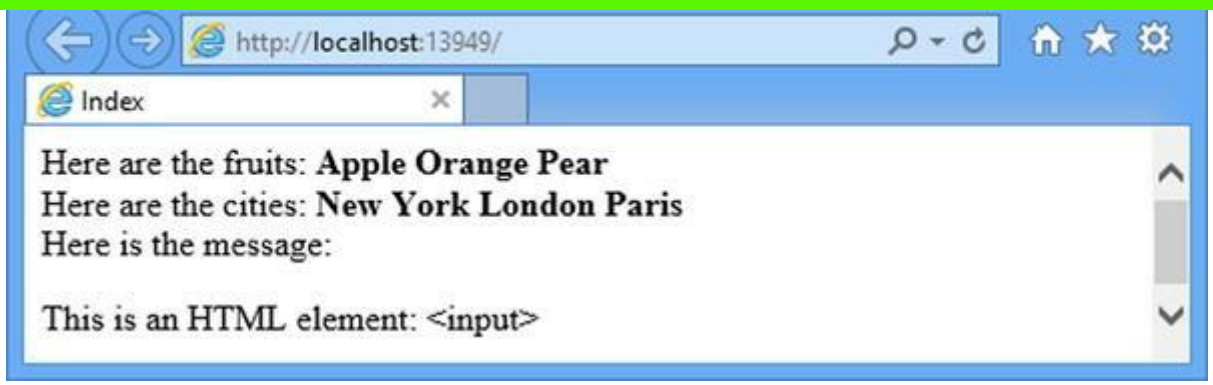


Figure 21-1. Running the example application

Creating Custom Helper Methods

I am going to follow the pattern I have established over the last few chapters and introduce you to helper methods by
own custom implementation. In the sections that follow, I will show you two different techniques for creating custo
methods.

Creating an Inline Helper Method

The simplest kind of helper method is an *inline helper*, which is defined within a view. I can create an inline helper to
example view using the `@helper` tag, as shown in Listing 21-3.

Listing 21-3. Creating an Inline Helper Method in the Index.cshtml File

```
@model string

@{
    Layout = null;
}

@helper ListArrayItems(string[] items) {
    foreach(string str in items) {
        <b>@str </b>
    }
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Here are the fruits: @ListArrayItems(ViewBag.Fruits)
    </div>
    <div>
        Here are the cities: @ListArrayItems(ViewBag.Cities)
    </div>
    <div>
```

```
        <p>@Model</p>
    </div>
</body>
</html>
```

Inline helpers have names and parameters similar to regular C# methods. In the example, I defined a helper called `ListArrayItems`, which takes a string array as a parameter. Although an inline helper looks like a method, th return value. The contents of the helper body are processed and put into the response to the client.

Tip Notice that I did not have to cast the dynamic properties from the `ViewBag` to string arrays when using t helper. One of the nice features of this kind of helper method is that it is happy to evaluate types at runtime.

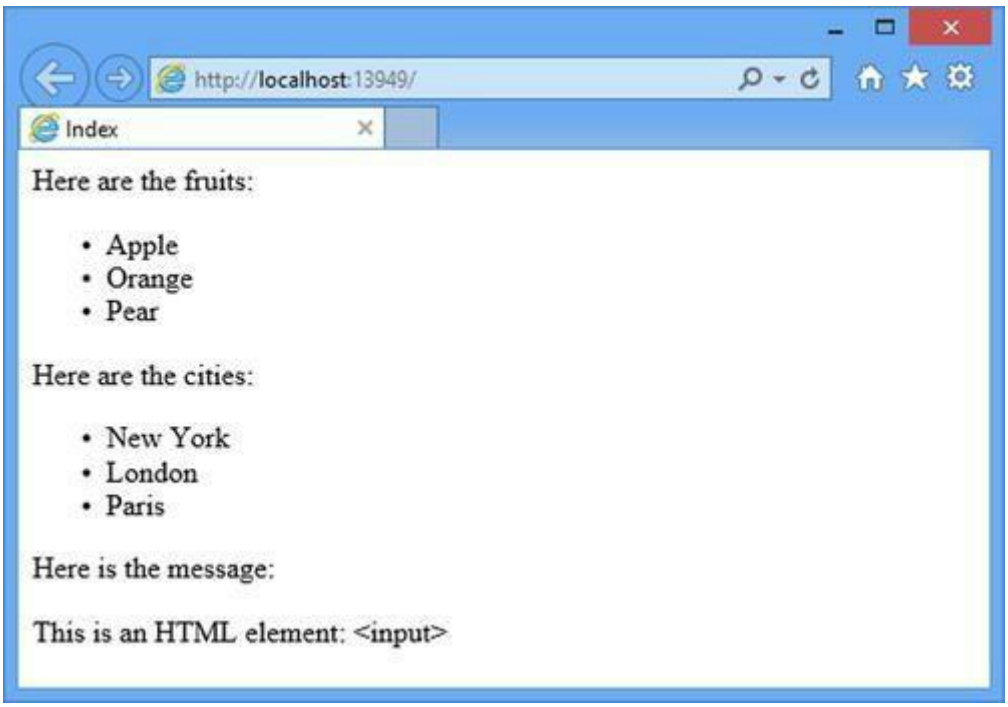
The body of an inline helper follows the same syntax as the rest of a Razor view. Literal strings are regarded as sta and statements that require processing by Razor are prefixed with the `@` character. The helper in the example mixes s and Razor tags to enumerate the items in the array, which produces the same output as the original view but has redu amount of duplication in the view.

The benefit of this approach is that I only have to make one change if I want to change the way that the array con displayed. As a simple example, in Listing 21-4 you can see how I have switched from just writing out the values to HTML unnumbered list elements.

Listing 21-4. Changing the Contents of a Helper Method in the Index.cshtml File

```
...
@helper ListArrayItems(string[] items) {
    <ul>
        @foreach(string str in items) {
            <li>@str</li>
        }
    </ul>
}
...
```

I only had to make the change in one place, which may seem like a trivial advantage in such a simple project, but t useful way to keep your views simple and consistent in a real project. You can see the result of this change in Figure



eBooks-IT.org

Tip Notice that I had to prefix the `foreach` keyword with `@` in this example but not in Listing 21-4. This is b first element in the helper body changed to become an HTML element, which means I have to use `@` to tell Razor tha C# statement. In the previous example there was no HTML element, so Razor assumed the contents were code. It ca keep track of these quirks, but the Visual Studio will flag up errors like this for you.

Creating an External Helper Method

Inline helpers are convenient, but they can be used only from the view in which they are declared and, if they are too they can take over that view and make it hard to read.

The alternative is to create an external HTML helper method, which is expressed as a C# extension method. Extern methods can be used more widely, but are a little more awkward to write, because C# doesn't naturally handle HTM generation elegantly. To demonstrate this feature, I added an `Infrastructure` folder to the example project a new `CustomHelpers.cs` class file within it. You can see the contents of this file in Listing 21-5.

Listing 21-5. The Contents of the CustomHelpers.cs File

```
using System.Web.Mvc;

namespace HelperMethods.Infrastructure {
    public static class CustomHelpers {

        public static MvcHtmlString ListArrayItems(this HtmlHelp
string[] list) {

            TagBuilder tag = new TagBuilder("ul");

            foreach(string str in list) {
                TagBuilder itemTag = new TagBuilder("li");
                itemTag.SetInnerText(str);
                tag.InnerHtml += itemTag.ToString();
            }

            return new MvcHtmlString(tag.ToString());
        }
    }
}
```

The helper method I created performs the same function as the inline helper in the previous example. It takes an ar and generates an HTML `ul` element, containing a `li` element for each `string` in the array.

The first parameter to the helper method is an `HtmlHelper` object, prefixed with the `this` keyword to tell t compiler that I am defining an extension method. The `HtmlHelper` provides access to information that can be u creating content, through the properties described in Table 21-2.

Table 21-2. *Useful Properties Defined by the HtmlHelper Class*

Property	Description
`RouteCollection`	Returns the set of routes defined by the application
`ViewBag`	Returns the view bag data passed from the action method to the view that has called the helper method
`ViewContext`	Returns a `ViewContext` object, which provides access to details of the request and how it has been handled (an describe below)

The `ViewContext` property is the most useful when you want to create content which adapts to the request processed. In Table 21-3, I have described some of the most commonly used properties defined by the `ViewCon`

Property	Description
`Controller`	Returns the controller processing the current request
`HttpContext`	Returns the `HttpContext` object that describes the current request
`IsChildAction`	Returns `true` if the view that has called the helper is being rendered by a child action (see Chapter 20 for details of
`RouteData`	Returns the routing data for the request
`View`	Returns the instance of the `IView` implementation that has called the helper method

The information you can get about the request is fairly comprehensive, but for the most part helper methods are si used to keep formatting consistent. You can use the built-in helper methods for generating requests-specific content these helpers later in the chapter) and you can use partial views or child actions for more complex tasks (I provide gu which approach to use in next section of this chapter).

I do not need any information about the request in the example helper, but I do need to construct some HTML elem easiest way to create HTML in a helper method is to use the `TagBuilder` class, which allows you to build up H without needing to deal with all of the escaping and special characters. The `TagBuilder` class is part of the `System.Web.WebPages.Mvc` assembly but uses a feature called *type forwarding* to appear as though it is `System.Web.Mvc` assembly. Both assemblies are added to MVC projects by Visual Studio, so you can use the `TagBuilder` class easily enough, but it does not appear in the Microsoft Developer Network (MSDN) API docu

I create a new `TagBuilder` instance, passing in name the HTML element I want to construct as the construc parameter. I do not need to use the angle brackets (< and >) with the `TagBuilder` class, which means I can cr element, like this:

```
...
TagBuilder tag = new TagBuilder("ul");
...
```

The most useful members of the `TagBuilder` class are described in Table 21-4.

***Table 21-4**. Some Members of the TagBuilder Class*

Member	Description
`InnerHtml`	A property that lets you set the contents of the element as an HTML string. The value assigned to this p be encoded, which means that is can be used to nest HTML elements.
`SetInnerText(string)`	Sets the text contents of the HTML element. The `string` parameter is encoded to make it safe to c
`AddCssClass(string)`	Adds a CSS class to the HTML element
`MergeAttribute(string, string, bool)`	Adds an attribute to the HTML element. The first parameter is the name of the attribute, and the second The `bool` parameter specifies if an existing attribute of the same name should be replaced.

The result of an HTML helper method is an `MvcHtmlString` object, the contents of which are written direc response to the client. For the example helper, I pass the result of the `TagBuilder.ToString` method to th of a new `MvcHtmlString` object, like this:

```
...
return new MvcHtmlString(tag.ToString());
...
```

This statement generates the HTML fragment that contains the `ul` and `li` elements and returns them to the view that it can be inserted into the response.

Using a Custom External Helper Method

Using a custom external helper method is a little different to using an inline one. In Listing 21-6, you can see the char made to the `/Views/Home/Index.cshtml` file to replace the inline helper with the external one.

```
@model string
@using HelperMethods.Infrastructure

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
                                                Here      are      the
@Html.ListArrayItems((string[])ViewBag.Fruits)
    </div>
    <div>
                                                Here      are      the
@Html.ListArrayItems((string[])ViewBag.Cities)
    </div>
    <div>
        Here is the message:
        <p>@Model</p>
    </div>
</body>
</html>
```

I need to ensure that the namespace that contains the helper extension method is in scope. I have done this using a tag, but if you are developing a lot of custom helpers then you will want to add the namespaces that contain them to /Views/Web.config file so that they are always available in your views.

I refer to the helper using @Html.<helper>, where <helper> is the name of the extension method. In use @Html.ListArrayItems. The Html part of this expression refers to a property defined by the view which returns an HtmlHelper object, which is the type to which I applied the extension method in Listing 21-5

I pass data to the helper method as I would for an inline helper or a C# method, although I must take care to cast dynamic properties of the ViewBag object to the type defined by the external helper (in this case a string arra syntax is not as elegant as using inline helpers, but it is part of the price that you must pay to create a helper that can any view in your project.

KNOWING WHEN TO USE HELPER METHODS

Now that you have seen how helper methods work, you might be wondering when you should use them in pref partial views or child actions, especially as there is overlap between what these features are capable of.

I only use helper methods to reduce the amount of duplication in views, just as I did in this example, and only fo of content. For more complex markup and content I use partial views and I use a child action when I need to pe manipulation of model data. I recommend that you follow the same approach and keep your use of helper metho as possible. (If my helpers contain more than a handful of C# statements—or more C# statements that HTML e I tend to switch to a child action.)

Managing String Encoding in a Helper Method

The MVC Framework makes an effort to protect you from malicious data by automatically encoding it so that it can an HTML page safely. You can see an example of this in the Home controller in the example application where I pas

Listing 21-7. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace HelperMethods.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {

            ViewBag.Fruits = new string[] { "Apple", "Orange", "Pe
             ViewBag.Cities = new string[] { "New York", "London",
};

            string message = "This is an HTML element: <input>";

            return View((object)message);
        }
    }
}
```

The model object contains a valid HTML element, but when the value is rendered by Razor, the following HTML i

```
...
<div>
    Here is the message:
    <p>This is an HTML element:  &lt;input &gt;</p>
</div>
...
```

This is a basic security precaution that prevents data values from being interpreted as valid markup by the browser foundation for a common form of attack in which malicious users will try to subvert the behavior of an application b add their own HTML markup or JavaScript code. Razor encodes data values automatically when they are used in a v helper methods need to be able to generate HTML. As a consequence, they are given a higher level of trust by the vie this can require some careful attention.

Demonstrating the Problem

To demonstrate the problem, I have created a new helper method in the `CustomHelpers` class, as shown in L This helper takes a string as a parameter and generates the same HTML that I included in the `Index` view.

Listing 21-8. Defining a New Helper Method in the CustomHelpers.cs File

```
using System;
using System.Web.Mvc;

namespace HelperMethods.Infrastructure {
    public static class CustomHelpers {

         public static MvcHtmlString ListArrayItems(this HtmlHelp
string[] list) {

            TagBuilder tag = new TagBuilder("ul");
```

eBooks-IT.org

```
                TagBuilder itemTag = new TagBuilder("li");
                itemTag.SetInnerText(str);
                tag.InnerHtml += itemTag.ToString();
            }

            return new MvcHtmlString(tag.ToString());
        }

        public static MvcHtmlString DisplayMessage(this HtmlHelp
string msg) {
            string result = String.Format("This is the message
</p>", msg);
            return new MvcHtmlString(result);
        }
    }
}
```

I use the `String.Format` method to generate the HTML markup and pass the result as the argument to the `MvcHtmlString` constructor. In Listing 21-9, you can see how I have changed the `/View/Home/Inde` view to use the new helper method. (I also made some changes to emphasize the content that comes from the helper

Listing 21-9. Using the DisplayMessage Helper Method in the Index.cshtml File

```
@model string
@using HelperMethods.Infrastructure

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <p>This is the content from the view:</p>
    <div style="border: thin solid black; padding: 10px">
        Here is the message:
        <p>@Model</p>
    </div>

    <p>This is the content from the helper method:</p>
    <div style="border: thin solid black; padding: 10px">
        @Html.DisplayMessage(Model)
    </div>
</body>
</html>
```

You can see the effect the new helper method has by starting the application, as shown in Figure 21-3.

eBooks-IT.org

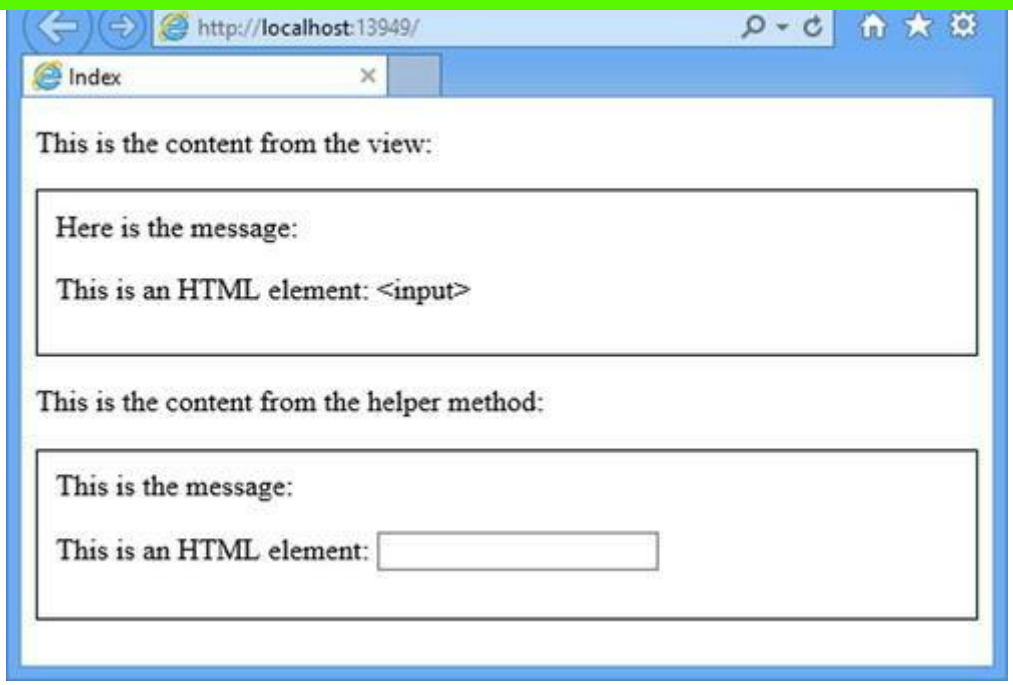


Figure 21-3. Comparing how data values are encoded

The helper method is trusted to generate safe content, which is unfortunate because it leads to the browser display
`input` element, which is the kind of behavior that can be exploited to subvert an application.

Encoding Helper Method Content

There are a couple of different ways to solve this problem and the choice between them depends on the nature of the
your helper method produces. The simplest solution is to change the return type of the helper method to `string`,
Listing 21-10. This alerts the view engine that your content is not safe and should be encoded before it is added to th

Listing 21-10. Ensuring that Razor Encodes Content in the CustomHelpers.cs File

```
using System.Web.Mvc;
using System;

namespace HelperMethods.Infrastructure {
    public static class CustomHelpers {

        public static MvcHtmlString ListArrayItems(this HtmlHelp
string[] list) {

            TagBuilder tag = new TagBuilder("ul");

            foreach(string str in list) {
                TagBuilder itemTag = new TagBuilder("li");
                itemTag.SetInnerText(str);
                tag.InnerHtml += itemTag.ToString();
            }

            return new MvcHtmlString(tag.ToString());
        }

        public static string DisplayMessage(this HtmlHelper html
```

eBooks-IT.org

```
            return string.Format("This is the message: <p>{0}</p>"
        }
    }
}
```

This technique causes Razor to encode all of the content that is returned by the helper, which is a problem when y generating HTML elements (as I am in the example helper), but which is convenient otherwise. You can see the effec 21-4.

Figure 21-4. Ensuring that the view engine encodes the response from a helper method

I have solved the problem with the `input` element, but my `p` elements have been encoded as well, which is not In these situations, I need to be more selective and encode just the data values, as shown in Listing 21-11.

Listing 21-11. Selectively Encoding Data Values in the CustomHelpers.cs File

```
...
public static MvcHtmlString DisplayMessage(this HtmlHelper html,
msg) {
    string encodedMessage = html.Encode(msg);
      string result = String.Format("This is the message: <p>{
encodedMessage);
    return new MvcHtmlString(result);
}
...
```

The `HtmlHelper` class defines an instance method called `Encode`, which solves the problem and encodes so that it can be safely included in a view. The problem with this technique is that you have to remember to use it. I encode all of the data values at the start of the method as a reminder and I suggest that you adopt a similar approach.

You can see the result of this change in Figure 21-5, where you will see that the content generated by the external method matches that generated by using the model value directly in the view.

eBooks-IT.org

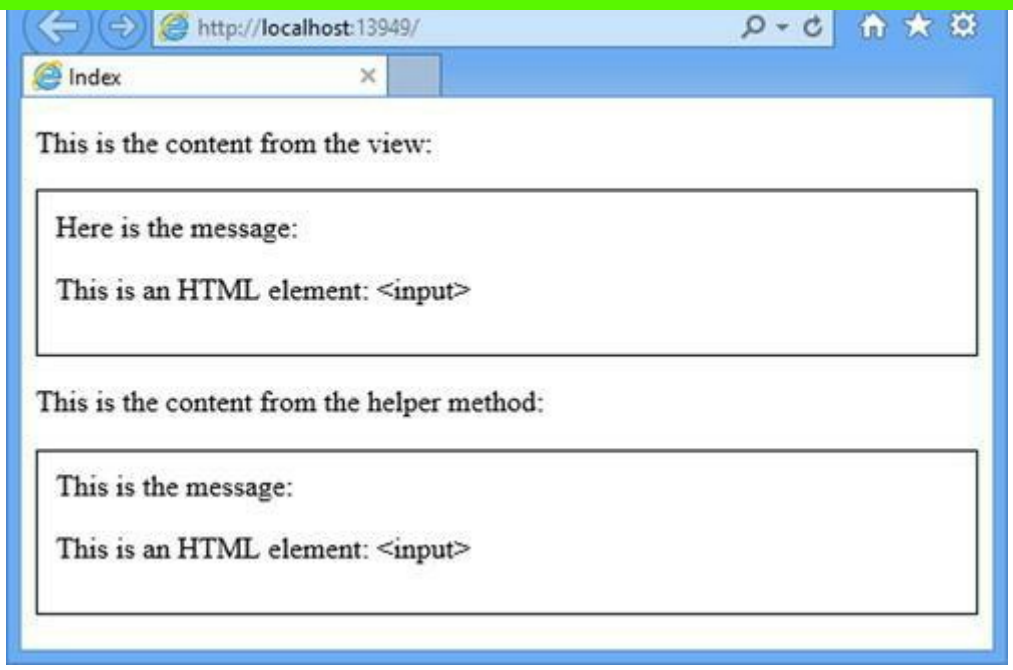


Figure 21-5. The effect of selectively encoding content in an external helper method

Using the Built-In Form Helper Methods

The MVC Framework includes a selection of built-in helper methods that help you manage the creating of HTML `fo`
In the following sections, I will put these helpers in context and show you how they are used.

Creating Form Elements

One of the most common forms of interaction in a web application is the HTML form, which is the subject of a num
different helper methods. To demonstrate the form-related helpers, I made some additions to the example project. I s
creating a new class file called `Person.cs` in the `Models` folder. You can see the contents of this file in Listin
`Person` type will be the view model class when I demonstrate the form-related helpers, and the `Address` and
will help me showcase some more advanced features.

Listing 21-12. The Contents of the Person.cs Model

```
using System;

namespace HelperMethods.Models {

    public class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    public class Address {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
```

```
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }

    public enum Role {
        Admin,
        User,
        Guest
    }
}
```

I also added new action methods to the Home controller to use the model objects, as shown in Listing 21-13.

Listing 21-13. Adding Action Methods in the HomeController.cs File

```
using System.Web.Mvc;
using HelperMethods.Models;

namespace HelperMethods.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {

            ViewBag.Fruits = new string[] { "Apple", "Orange", "Pe
             ViewBag.Cities = new string[] { "New York", "London",
};

            string message = "This is an HTML element: <input>";

            return View((object)message);
        }

        public ActionResult CreatePerson() {
            return View(new Person());
        }

        [HttpPost]
        public ActionResult CreatePerson(Person person) {
            return View(person);
        }
    }
}
```

This is the standard two-method approach to dealing with HTML forms, where I rely on model binding so that the Framework will create a Person object from the form data and pass it to the action method with the HttpPos explained the HttpPost attribute in Chapter 19 and model binding is the topic of Chapter 24).

I am not processing the form data in any way because I am focused on how to generate elements in the view. The HttpPost action method just calls the View method and passes the Person object that it received as a paran has the effect of redisplaying the form data to the user.

I am going to start with a standard manual HTML form and show you how to replace different parts of it using he You can see the initial version of the form in Listing 21-14, which shows the CreatePerson.cshtml view added to the /Views/Home folder.

Listing 21-14. The Contents of the CreatePerson.cshtml File

```
@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}

<h2>CreatePerson</h2>

<form action="/Home/CreatePerson" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
</form>
```

This view contains a standard manually created form in which I have set the value of the `value` attribute of the elements using the model object.

Tip Notice that I have set the `name` attribute on all of the `input` elements so that it corresponds to the model the `input` element displays. The `name` attribute is used by the MVC Framework default model binder to work ou `input` elements contain values for the model type properties when processing a post request. If you omit the `nam` your form will not work properly. I describe model binding fully in Chapter 24, including how you can change this b

I have created the `Views/Shared` folder and added a layout file called `_Layout.cshtml` with the con in Listing 21-15. This is a simple layout with some CSS for the `input` elements in the form.

Listing 21-15. The Contents of the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style type="text/css">
        label { display: inline-block; width: 100px; }
        div.dataElem { margin: 5px; }
    </style>
</head>
<body>
    @RenderBody()
</body>
</html>
```

You can see the basic form functionality by starting the application and navigating to the `/Home/CreatePe` as shown in Figure 21-6. Because the form data is not used by the application in any way, clicking the `Submit` bu

eBooks-IT.org

Figure 21-6. Using the simple HTML form in the example application

In Listing 21-16, you can see the HTML that the example MVC application has sent to the browser. I will use this changes caused by helper methods.

Listing 21-16. The HTML Sent to the Browser for the Example Form

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>CreatePerson</title>
    <style type="text/css">
        label { display: inline-block; width: 100px; }
        div.dataElem { margin: 5px; }
    </style>
</head>
<body>

<h2>CreatePerson</h2>

<form action="/Home/CreatePerson" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" />
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" />
    </div>
    <input type="submit" value="Submit" />
</form>
```

eBooks-IT.org

```
</html>
```

Note Using the helper methods to generate HTML elements like `form` and `input` is not compulsory. If you p can code them using static HTML tags and populate values using view data or view model objects, just as I did in thi HTML that I generate from the helper methods in the following sections is clean and there are no special attribute val tricks that mean you have to use them. But they make it easy to ensure that the HTML is in sync with the application example, changes in routing configuration will be reflected automatically in your forms. The helpers are there for con rather than because they create essential or special HTML, and you do not have to use them if they do not suit your c style.

Creating Form Elements

Two of the most useful (and most commonly used) helpers are `Html.BeginForm` and `Html.EndForm`. T create HTML `form` tags and generate a valid `action` attribute for the form that is based on the routing mechanis application.

There are 13 different versions of the `BeginForm` method, allowing you to be increasingly specific about how `form` element will be generated. I only need the most basic for the example application, which takes no arguments `form` element whose `action` attribute ensures that the form will be posted back to the same action method whic current view being generated. You can see how I have applied this overload of `BeginForm` and the `EndForm` Listing 21-17. The `EndForm` helper has only one definition and it just closes the form element by adding `</form` view.

Listing 21-17. Using the BeginForm and EndForm Helper Methods in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@{Html.BeginForm();}
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
@{Html.EndForm();}
```

Notice that I had to treat the call to the helper methods as a C# statements. This is because of the way that the hel write their tags to the output. It is a pretty ugly result, but it doesn't matter because these helpers are rarely used in t much more common approach is shown in Listing 21-18, which wraps the call to the `BeginForm` helper method expression. At the end of the `using` block, the .NET runtime calls the `Dispose` method on the object returned `BeginForm` method, which calls the `EndForm` method for you. (You can see how this works by downloading code for the MVC Framework and taking a look at the `System.Web.Mvc.Html.FormExtensions` c

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginForm()) {
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}
```

This approach, known as a self-closing form, is the one I use in my own projects. I like the way that the code blo the form and makes it clear which elements will appear between the opening and closing `form` tags.

The other 12 variations for the `BeginForm` method allow you to change different aspects of the `form` eleme created. There is a lot of repetition in these overloads, as they allow you to be incrementally more specific about the provide. In Table 21-5 I have listed the most important overloads, which are the ones that you will use on a regular b MVC application. The other overloads of the `BeginForm` method—which I have omitted—are provided for com the versions of the MVC Framework that were released before C# had support for creating dynamic objects.

***Table 21-5**. The Overloads of the BeginForm Helper Method*

Overload	Description
`BeginForm()`	Creates a form which posts back to the action method it originated from
`BeginForm(`**action , controller**`)`	Creates a form which posts back to the action method and controller, specified as strings
`BeginForm(action, controller,` **method**`)`	As for the previous overload, but allows you to specify the value for the method attribute us the `System.Web.Mvc.FormMethod` enumeration
`BeginForm(action, controller, method,` **attributes**`)`	As for the previous overload, but allows you to specify attributes for the form element an ob properties are used as the attribute names
`BeginForm(action, controller,` **routeValues** `, method, attributes)`	As for the previous overload, but allows you to specify values for the variable route segment application routing configuration as an object whose properties correspond to the routing va

I have shown you the simplest version of the `BeginForm` method, which is all I need for the example app, but 19 you can see the most complex, in which I specify additional information for how the `form` element should be c

***Listing 21-19**.* Using the Most Complex Overload of the BeginForm Method in the CreatePerson.cs File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
```

eBooks-IT.org

```
}
<h2>CreatePerson</h2>

@using (Html.BeginForm("CreatePerson", "Home",
    new { id = "MyIdValue" }, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    
    <input type="submit" value="Submit" />
}
```

In this example, I have explicitly specified some details that would have been inferred automatically by the MVC F such as the action name and the controller. I also specified that the form should be submitted using the HTTP POST which would have been used anyway.

The more interesting arguments are the ones that set values for the route variable and set attributes on the form ele values arguments to specify a value for the `id` segment variable in the default route added by Visual Studio to the `/App_Start/RouteConfig.cs` file when the project was created and I defined `class` and `data` att attributes are custom attributes which you can add to elements to make processing HTML content.) Here is the HTM that this call to `BeginForm` produces:

```
...
<form   action="/Home/CreatePerson/MyIdValue"   class="personClass"
formType="person"
    method="post">
...
```

You can see that the value for the `id` attribute has been appended to the target URL and that the class and data att been applied to the element. Notice that I specified an attribute called `data_formType` in the call to `BeginF` ended up with a `data-formType` attribute in the output. You cannot specify property names in a dynamic obje contain hyphens, so I use an underscore that is then automatically mapped to a hyphen in the output, neatly side-step mismatch between the C# and HTML syntaxes. (And, of course, I had to prefix the property name `class` with a (can use a C#-reserved keyword as a property name for the `class` attribute.)

Specifying the Route Used by a Form

When you use the `BeginForm` method, the MVC Framework finds the first route in the routing configuration tha to generate a URL that will target the required action and controller. In essence, you leave the route selection to be fi you. If you want to ensure that a particular route is used, then you can use the `BeginRouteForm` helper meth demonstrate this feature, I have added a new route to the `/App_Start/RouteConfig.cs` file, as shown 20.

Listing 21-20. Adding a New Route to the RouteConfig.cs File

```
using System;
```

```
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace HelperMethods {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Ind
                    id = UrlParameter.Optional }
            );

            routes.MapRoute(
                name: "FormRoute",
                url: "app/forms/{controller}/{action}"
            );
        }
    }
}
```

If I call the `BeginForm` method with this routing configuration, I will end up with a `form` element whose `a` attribute contains a URL which is created from the default route. In Listing 21-21, you can see how I have specified route should be used through the `BeginRouteForm` method.

Listing 21-21. Specifying Which Route Should Be Used in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
```

eBooks-IT.org

}

This produces the following `form` tag, whose `action` attribute corresponds to the structure of the new route

```
...
<form action="/app/forms/Home/CreatePerson" class="personClass"
    data-formType="person" method="post">
...
```

Tip There are a range of different overloads for the `BeginRouteForm` method allowing you to specify diff of details for the `form` element, just as with the `BeginForm` method. These follow the same structure as their `BeginForm` counterparts. See the API documentation for details.

Using Input Helpers

An HTML form is of no use unless you also create some `input` elements. Table 21-6 shows the basic helper meth available to create `input` elements and gives examples of the HTML they produce. For all of these helper methods argument is used to set the value of the `id` and `name` attributes for the input element and the second argument is u value attribute.

***Table 21-6**. Basic Input HTML Helpers*

HTML Element	Example

Check box	`Html.CheckBox("myCheckbox", false)`
	Output:
	`<input id="myCheckbox" name="myCheckbox" type="checkbox" value="true" />`
	`<input name="myCheckbox" type="hidden" value="false" />`

Hidden field	`Html.Hidden("myHidden", "val")`
	Output:
	`<input id="myHidden" name="myHidden" type="hidden" value="val" />`

Radio button	`Html.RadioButton("myRadiobutton", "val", true)`
	Output:
	`<input checked="checked" id="myRadiobutton" name="myRadiobutton"type="radio" val`

Password	`Html.Password("myPassword", "val")`
	Output:
	`<input id="myPassword" name="myPassword" type="password" value="val" />`

Text area	`Html.TextArea("myTextarea", "val", 5, 20, null)`
	Output:
	`<textarea cols="20" id="myTextarea" name="myTextarea" rows="5">val</textarea>`

Text box	`Html.TextBox("myTextbox", "val")`
	Output:
	`<input id="myTextbox" name="myTextbox" type="text" value="val" />`

Each of these helpers is overloaded. The table shows the simplest version, but you can provide an additional `obj` argument that you use to specify HTML attributes, just as I did with the `form` element in the previous section.

Note Notice that the checkbox helper (`Html.CheckBox`) renders *two* `input` elements. It renders a chec a hidden `input` element of the same name. This is because browsers do not submit a value for checkboxes when selected. Having the hidden control ensures that the MVC Framework will get a value from the hidden field when this

You can see how I have used these basic `input` helper methods in Listing 21-22.

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBox("personId", @Model.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBox("firstName", @Model.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBox("lastName", @Model.LastName)
    </div>
    <input type="submit" value="Submit" />
}
```

You can see the HTML `input` elements that this view produces in Listing 21-23. The output is similar to the ori element, but you can see some hints of the MVC Framework have appeared in the form of some `data` attributes w been added to support form validation, which I describe in Chapter 25.

Listing 21-23. The Input Elements Created by the Basic Input Helper Methods

```
...
<form   action="/app/forms/Home/CreatePerson"   class="personClass"
formType="person"
        method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input data-val="true" data-val-number="The field PersonId
a number."
                    data-val-required="The  PersonId  field  is  re
id="personId"
            name="personId" type="text" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input id="firstName" name="firstName" type="text" value="
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input id="lastName" name="lastName" type="text" value=""
    </div>
    <input type="submit" value="Submit" />
```

...

Generating the Input Element from a Model Property

The helper methods I used in the previous section are fine, but I still have to ensure that the value I pass as the first a corresponds to the model value I pass as the second argument. If they are not consistent, then the MVC Framework able to reconstruct the model object from the form data because the `name` attributes and the forms values of the `i` elements will not match. For each of the methods I listed in Table 21-6, there is an alternative overload which takes a string argument, which I have used in Listing 21-24.

Listing 21-24. Generating the Input Element from the Model Property Name in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBox("PersonId")
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBox("firstName")
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBox("lastName")
    </div>
    <input type="submit" value="Submit" />
}
```

The `string` argument is used to search the view data, `ViewBag`, and view model to find a corresponding da can be used as the basis for the input element. So, for example, if you call `@Html.TextBox("DataValue` MVC Framework tries to find some item of data that corresponds with the key `DataValue`. The following locat checked:

- `ViewBag.DataValue`
- `@Model.DataValue`

The first value that is found is used to set the `value` attribute of the generated HTML. (The last check, for `@Model.DataValue`, works only if the view model for the view contains a property or field called `DataVa`

If I specify a string like `DataValue.First.Name`, the search becomes more complicated. The MVC Fra try different arrangements of the dot-separated elements, such as the following:

- `ViewBag.DataValue.First.Name`

- `ViewBag.DataValue["First.Name"]`
- `ViewBag.DataValue["First"]["Name"]`

Many permutations will be checked. Once again, the first value that is found will be used, terminating the search. T obvious performance consideration to this technique, but bear in mind that usually only a few items are in the view b not take much time to search through them.

Using Strongly Typed Input Helpers

For each of the basic input helpers that I described in Table 21-6, there are corresponding *strongly typed* helpers. Yo these helpers in Table 21-7 along with samples of the HTML they produce. These helpers can be used only with stro views. (Some of these helpers generate attributes that help with client-side form validation. I have omitted these from brevity.)

***Table 21-7**. Strongly Typed Input HTML Helpers*

HTML Element	Example

Check box	`Html.CheckBoxFor(x => x.IsApproved)`
	Output:
	`<input id="IsApproved" name="IsApproved" type="checkbox" value="true" />`
	`<input name="IsApproved" type="hidden" value="false" />`

Hidden field	`Html.HiddenFor(x => x.FirstName)`
	Output:
	`<input id="FirstName" name="FirstName" type="hidden" value="" />`

Radio button	`Html.RadioButtonFor(x => x.IsApproved, "val")`
	Output:
	`<input id="IsApproved" name="IsApproved" type="radio" value="val" />`

Password	`Html.PasswordFor(x => x.Password)`
	Output:
	`<input id="Password" name="Password" type="password" />`

Text area	`Html.TextAreaFor(x => x.Bio, 5, 20, new{})`
	Output:
	`<textarea cols="20" id="Bio" name="Bio" rows="5">Bio value</textarea>`

Text box	`Html.TextBoxFor(x => x.FirstName)`
	Output:
	`<input id="FirstName" name="FirstName" type="text" value="" />`

The strongly typed input helpers work on lambda expressions. The value that is passed to the expression is the vie object, and you can select the field or property that will be used to set the `value` attribute. You can see how I have kind of helper in the `CreatePerson.cshtml` view from the example application in Listing 21-25.

***Listing 21-25**.* Using the Strongly Typed Input Helper Methods in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
```

eBooks-IT.org

```
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <input type="submit" value="Submit" />
}
```

The HTML generated by these helpers is not any different, but I use the strongly typed helper methods in my own because they reduce the chances of causing an error by mistyping a property name.

Creating Select Elements

Table 21-8 shows the helper methods that can be used to create `select` elements. These can be used to select a s from a drop-down list or present a multiple-item `select` element that allows several items to be selected. As with form elements, there are versions of these helpers that are weakly and strongly typed.

***Table 21-8**. HTML Helpers That Render Select Elements*

HTML Element	Example

Drop-down list	`Html.DropDownList("myList", new SelectList(new [] {"A", "B"}), "Choose")`
	Output:
	`<select id="myList" name="myList">`
	`  <option value="">Choose</option>`
	`  <option>A</option>`
	`  <option>B</option>`
	`</select>`

Drop-down list	`Html.DropDownListFor(x => x.Gender, new SelectList(new [] {"M", "F"}))`
	Output:
	`<select id="Gender" name="Gender">`
	`  <option>M</option>`
	`  <option>F</option>`
	`</select>`

Multiple-select	`Html.ListBox("myList", new MultiSelectList(new [] {"A", "B"}))`
	Output:
	`<select id="myList" multiple="multiple" name="myList">`
	`  <option>A</option>`
	`  <option>B</option>`
	`</select>`

	`Html.ListBoxFor(x => x.Vals, new MultiSelectList(new [] {"A", "B"}))` Output:

```
        <option>B</option>
    </select>
```

The `select` helpers take `SelectList` or `MultiSelectList` parameters. The difference between t is that `MultiSelectList` has constructor options that let you specify that more than one item should be sele page is rendered initially.

Both of these classes operate on `IEnumerable` sequences of objects. In Table 21-8, I created inline arrays th the list items I wanted displayed, but a nice feature of `SelectList` and `MultiSelectList` is that they v values from objects, including the model object, for the list items. You can see how I have created a `select` elem `Role` property of the `Person` model in Listing 21-26.

Listing 21-26. Creating a Select Element for the Person.Role Property in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.DropDownListFor(m => m.Role,

SelectList(Enum.GetNames(typeof(HelperMethods.Models.Role))))
    </div>
    <input type="submit" value="Submit" />
}
```

I defined the `Role` property so that it is a value from the `Role` enumeration defined in the same class file. Beca `SelectList` and `MultiSelectList` objects operate on `IEnumerable` objects, I have to use the `Enum.GetNames` method to be able to use the `Role enum` as the source for the `select` element. You c HTML that the latest version of the view creates, including the `select` element, in Listing 21-27.

Listing 21-27. The HTML Generated by the CreatePerson View

```
<!DOCTYPE html>
```

eBooks-IT.org

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>CreatePerson</title>
    <style type="text/css">
        label { display: inline-block; width: 100px;}
        .dataElem { margin: 5px;}
    </style>
</head>
<body>

<h2>CreatePerson</h2>

<form action="/app/forms/Home/CreatePerson" class="personClass"
            data-formType="person" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input data-val="true" data-val-number="The field PersonId
a   number."   data-val-required="The   PersonId   field   is   re
id="PersonId" name="PersonId" type="text" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input id="FirstName" name="FirstName" type="text" value="
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input id="LastName" name="LastName" type="text" value=""
    </div>
    <div class="dataElem">
        <label>Role</label>
            <select data-val="true" data-val-required="The Role f
required."
                id="Role" name="Role">
            <option selected="selected">Admin</option>
            <option>User</option>
            <option>Guest</option>
        </select>
    </div>
    <input type="submit" value="Submit" />
</form>
</body>
</html>
```

Summary

In this chapter, I introduced the concept of helper methods, which you can use in views to generate chunks of conte reusable way. I started by showing you how to create custom inline and external helpers and then showed you the he available to create HTML `form`, `input` and `select` elements. In the next chapter, I continue on this theme an how to use *templated helpers*.

eBooks-IT.org

Templated Helper Methods

The HTML helpers that I looked at in the previous chapter, such as `Html.CheckBoxFor` and `Html.Text` generate a specific type of element, which means that I have to decide in advance what kinds of elements should be represent model properties and to manually update the views if the type of a property changes.

In this chapter, I demonstrate the *templated helper methods*, with which I specify the property I want displayed ar MVC Framework figure out what HTML elements are required. This is a more flexible approach to displaying data to although it requires some initial care and attention to set up. Table 22-1 provides the summary for this chapter.

Table 22-1. *Chapter Summary*

Problem	Solution
Generate an element that can be used to edit a model property	Use the `Html.Editor` and `Html.EditorFor` help
Generate labels for and read-only representations of model properties	Use the `Html.Label` and `Html.Display` helpers
Generate elements for a complete model object	Use the `DisplayForModel`, `EditorForModel` a `LabelForModel` helpers
Hide an element from the user when generating elements using a whole-model helper or prevent it from being edited	Apply the `HiddenInput` attribute to the property
Set the label that will be used to display model properties	Use the `DisplayName` and `Display` attributes
Specify the way in which model properties are displayed	Use the `DataType` attribute
Specify the template used to display a model property	Use the `UIHint` attribute
Define model metadata separately from the model type	Create a buddy class and use the `MetadataType` attri
Change the elements that are generated for a model property	Create a custom template

Preparing the Example Project

In this chapter I am going to continue using the `HelperMethod` project that I created in Chapter 21. In that pro a `Person` model class along with a couple of supporting types. As a reminder, I have listed these in Listing 22-1.

Listing 22-1. The Contents of the Person.cs File

```
using System;

namespace HelperMethods.Models {

    public class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
```

```
    public class Address {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }

    public enum Role {
        Admin,
        User,
        Guest
    }
}
```

The example project contains a simple Home controller that I use to display forms and receive form posts. You c definition of the HomeController class in Listing 22-2.

Listing 22-2. The Contents of the HomeController.cs File

```
using System.Web.Mvc;
using HelperMethods.Models;

namespace HelperMethods.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {

            ViewBag.Fruits = new string[] { "Apple", "Orange", "Pe
             ViewBag.Cities = new string[] { "New York", "London",
};

            string message = "This is an HTML element: <input>";

            return View((object)message);
        }

        public ActionResult CreatePerson() {
            return View(new Person());
        }

        [HttpPost]
        public ActionResult CreatePerson(Person person) {
            return View(person);
        }
    }
}
```

It is the two CreatePerson action methods that I will be using in this chapter, both of which render the /Views/Home/CreatePerson.cshtml view file. In Listing 22-3, you can see the CreatePerso the end of the last chapter, with a simple change.

Listing 22-3. The Contents of the CreatePerson.cshtml File

```
@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson</h2>

@using (Html.BeginRouteForm("FormRoute", new { }, FormMethod.Post,
    new { @class = "personClass", data_formType = "person" })) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.DropDownListFor(m => m.Role,

SelectList(Enum.GetNames(typeof(HelperMethods.Models.Role))))
    </div>
    <input type="submit" value="Submit" />
}
```

I made one addition, which I have marked in bold. By default, the helper methods will add `data` attributes to the elements they generate to support the kind of form validation I showed you when I created the `SportsStore` a do not want those attributes in this chapter, so I have used the `Html.EnableClientValidation` meth them for the `CreatePerson` view. The client validation feature is still enabled for the rest of the application and how validation works in detail (including the purpose of the `data` attributes) in Chapter 25.

Using Templated Helper Methods

The first templated helper methods that I am going to look at are `Html.Editor` and `Html.EditorFor`. T method takes a string argument that specifies the property for which editor element is required. The helper follows th process that I described in Chapter 20 to locate a corresponding property in the view bag and model object. The `Ed` method is the strongly typed equivalent, which allows you to use a lambda expression to specify a model property th the editor element for.

In Listing 22-4, you can see how I have applied both the `Editor` and `EditorFor` helper methods to the `CreatePerson` view. As I mentioned in the last chapter, I prefer to use the strongly typed helpers because they chances of causing an error by mistyping the property name, but I have used both types in this listing just to demons can mix and match as you see fit.

Listing 22-4. Using the Editor and EditorFor Helper Methods in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person
```

```
@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.Editor("PersonId")
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.Editor("FirstName")
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.EditorFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.EditorFor(m => m.Role)
    </div>
    <div class="dataElem">
        <label>Birth Date</label>
        @Html.EditorFor(m => m.BirthDate)
    </div>
    <input type="submit" value="Submit" />
}
```

The HTML elements that are created by the `Editor` and `EditorFor` methods are the same. The only differ way that you specify the property that the editor elements are created for. You can see the effect of the changes that by starting the example application and navigating to the `/Home/CreatePerson` URL, as shown in Figure 22

eBooks-IT.org

Figure 22-1. Using the Editor and EditorFor helper methods in a form

Other than the addition of the `BirthDate` property, this does not look different from the kind of form that I w Chapter 21. However, there is a pretty substantial change, which you can see if you use a different browser. In Figu can see the same URL displayed in the Opera browser (which you can get from `www.opera.com`).

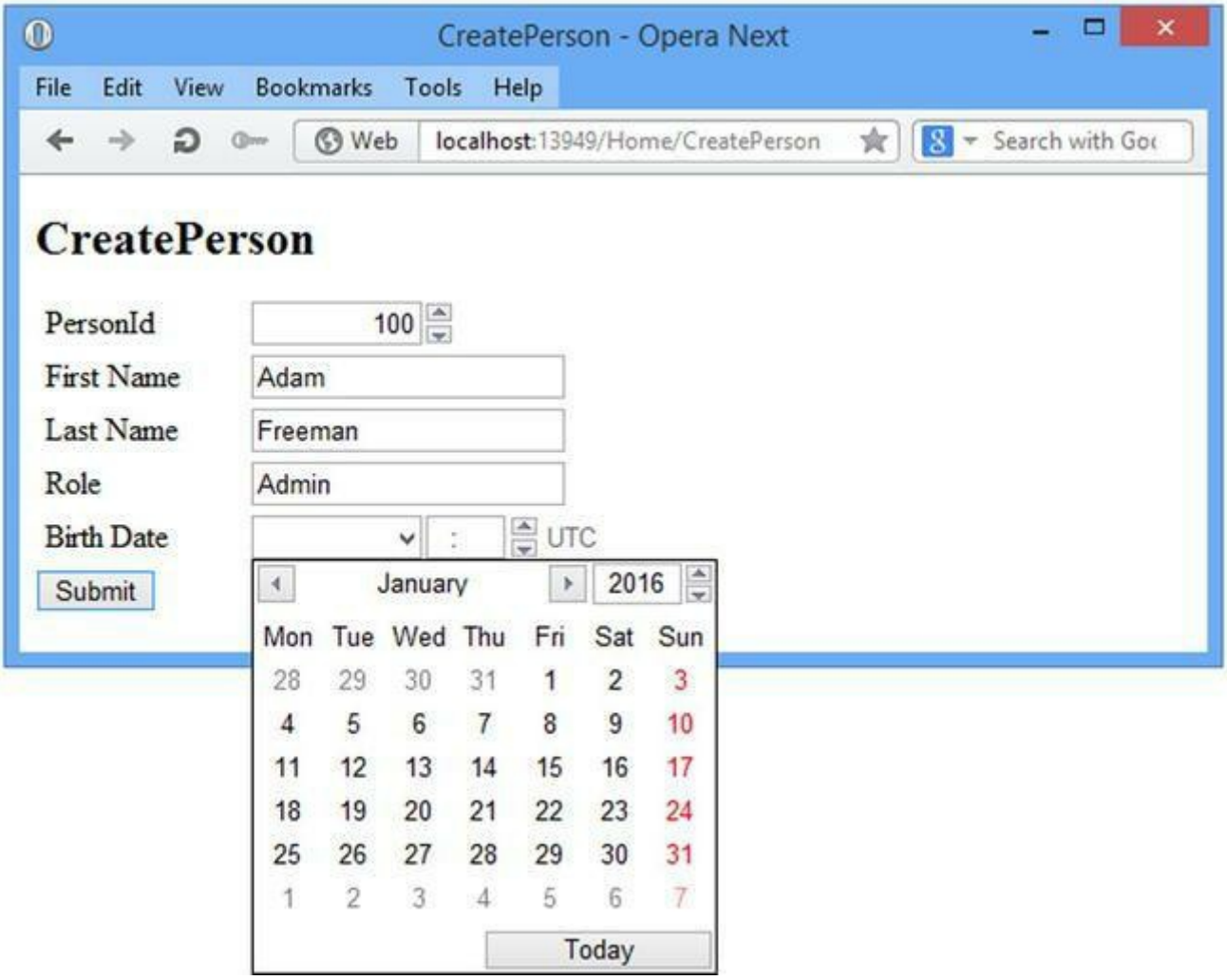


Figure 22-2. Displaying a form created using the Editor and EditorFor helper methods

eBooks-IT.org

hId el

spinner arrows (allowing you to increment and decrement the value) and the BirthDate element is presented wi picker.

The HTML5 specification defines different types of `input` element that can be used to edit common data types, numbers and dates. The `Helper` and `HelperFor` methods use the type of the property I want to edit to select new `input` element types. You can see this in Listing 22-5, where I have shown the HTML that was generated for

Listing 22-5. The HTML Input Elements Created by the Editor and EditorFor Helper Methods

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>CreatePerson</title>
    <style type="text/css">
        label { display: inline-block; width: 100px;}
        .dataElem { margin: 5px;}
    </style>
</head>
<body>

<h2>CreatePerson</h2>

<form action="/app/forms/Home/CreatePerson" class="personClass"
        data-formType="person" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input class="text-box single-line" id="PersonId" name="Pe
            type="number" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
                    <input  class="text-box  single-line"  id="Fi
name="FirstName"
            type="text" value="" />
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input class="text-box single-line" id="LastName" name="La
            type="text" value="" />
    </div>
    <div class="dataElem">
        <label>Role</label>
        <input class="text-box single-line" id="Role" name="Role"
            type="text" value="Admin" />
    </div>
    <div class="dataElem">
        <label>Birth Date</label>
                    <input  class="text-box  single-line"  id="Bi
name="BirthDate"
            type="datetime" value="01/01/0001 00:00:00" />
    </div>
    <input type="submit" value="Submit" />
</form>
```

eBooks-IT.org

```
</html>
```

The `type` attribute specifies which kind of `input` element should be displayed by the browser. The helper me specified the `number` and `datetime` types for the `PersonId` and `BirthDate` properties and the `tex` is the default for the other properties. The reason that we only see these types in Opera is because the HTML5 featur widely supported, even in the latest version of Internet Explorer and Chrome.

Tip Most web UI toolkits include date pickers that you can use instead of relying on the HTML5 `input` elemer you have not already selected such a toolkit for a project, then I suggest you start with jQuery UI (`http://jqueryui.com`), which is an open-source toolkit, built on jQuery.

You can see that by using the templated helper methods I have been able to tailor the form elements to the content, in an especially useful way, in part because not all browsers can display the HTML5 `input` element types and in p some properties, such as `Role`, are not displayed in a helpful way. I will show you how to provide the MVC Frame additional information that will improve the HTML that the helper methods produce. But I am going to show you the templated helpers that are available before I get into the detail. You can see the complete set of helpers in Table 22-2 demonstrate each of them in the sections that follow.

***Table 22-2**. The MVC Templated HTML Helpers*

Helper	Example	Description
`Display`	`Html.Display("FirstName")`	Renders a read-only view of the specified model property, choosing an HTML ele to the property's type and metadata
`DisplayFor`	`Html.DisplayFor(x => x.FirstName)`	Strongly typed version of the previous helper
`Editor`	`Html.Editor("FirstName")`	Renders an editor for the specified model property, choosing an HTML element a property's type and metadata
`EditorFor`	`Html.EditorFor(x => x.FirstName)`	Strongly typed version of the previous helper
`Label`	`Html.Label("FirstName")`	Renders an HTML `<label>` element referring to the specified model proper
`LabelFor`	`Html.LabelFor(x => x.FirstName)`	Strongly typed version of the previous helper

Generating Label and Display Elements

To demonstrate the other helper methods, I am going to add a new action method and view to the example that will c only view of the data submitted from the HTML form. First, I have updated the `HttpPost` version of the `Crea` action in the `Home` controller, as shown in Listing 22-6.

***Listing 22-6**.* Specifying a Different View in the HomeController .cs File

```
using System.Web.Mvc;
using HelperMethods.Models;

namespace HelperMethods.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {

            ViewBag.Fruits = new string[] { "Apple", "Orange", "Pe
             ViewBag.Cities = new string[] { "New York", "London",
};

            string message = "This is an HTML element: <input>";
```

```
        }

        public ActionResult CreatePerson() {
            return View(new Person());
        }

        [HttpPost]
        public ActionResult CreatePerson(Person person) {
            return View("DisplayPerson", person);
        }
    }
}
```

I added the `DisplayPerson.cshtml` view file to the `/Views/Home` folder, and you can see the co file in Listing 22-7.

Listing 22-7. The Contents of the DisplayPerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "DisplayPerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>DisplayPerson</h2>
<div class="dataElem">
    @Html.Label("PersonId")
    @Html.Display("PersonId")
</div>
<div class="dataElem">
    @Html.Label("FirstName")
    @Html.Display("FirstName")
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.LastName)
    @Html.DisplayFor(m => m.LastName)
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.Role)
    @Html.DisplayFor(m => m.Role)
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.BirthDate)
    @Html.DisplayFor(m => m.BirthDate)
</div>
```

You can see the output that this new view produces by starting the application, navigating to the `/Home/CreatePerson` URL, filling in the form and clicking the `Submit` button. The result is shown in F you can see that I have taken a small step backward, because the `Label` and `LabelFor` helpers have just used names as the content for the labels.

eBooks-IT.org

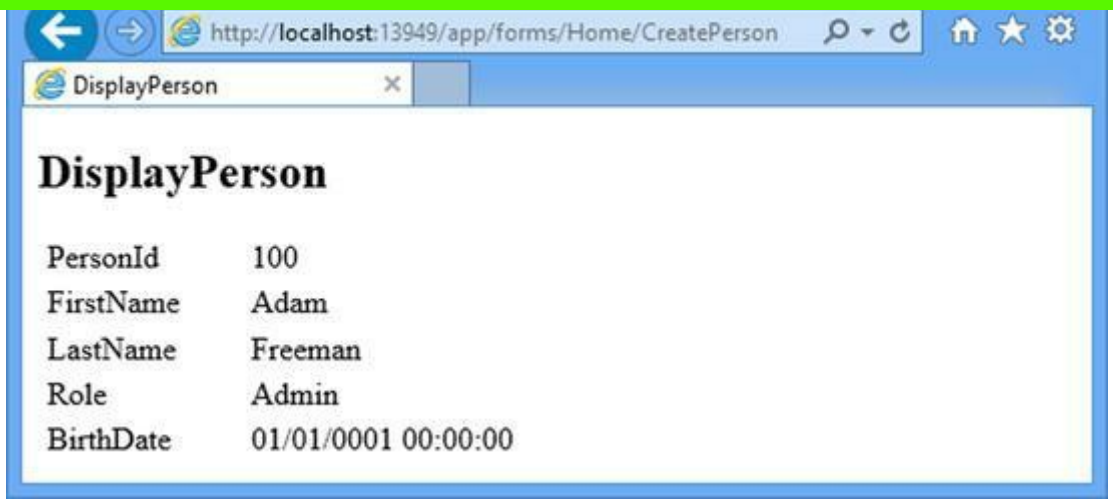


Figure 22-3. Using helpers to generate a read-only view of the Person object

You can see the output that these helper methods produce in Listing 22-8. Notice that the `Display` and `Disp` methods do not generate an HTML element by default. They just emit the value of the property they operate on.

Listing 22-8. The HTML Generated from the DisplayPerson View

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>DisplayPerson</title>
    <style type="text/css">
        label { display: inline-block; width: 100px;}
        .dataElem { margin: 5px;}
    </style>
</head>
<body>

<h2>DisplayPerson</h2>
<div class="dataElem">
    <label for="PersonId">PersonId</label>
    100
</div>
<div class="dataElem">
    <label for="FirstName">FirstName</label>
    Adam
</div>
<div class="dataElem">
    <label for="LastName">LastName</label>
    Freeman
</div>
<div class="dataElem">
    <label for="Role">Role</label>
    Admin
</div>
<div class="dataElem">
    <label for="BirthDate">BirthDate</label>
```

eBooks-IT.org

```
</div>

</body>
</html>
```

Although these helpers may not seem especially useful at the moment, I will show you how to change their behavi order to produce output that is much more the kind of thing that you would want to display to users.

Using Whole-Model Templated Helpers

I have been using templated helpers which generate output for a single property, but the MVC Framework also defin operate on the entire objects, a process known as *scaffolding*. There are scaffolding helpers available, as shown in T

***Table 22-3**. The MVC Scaffolding Templated Helper methods*

Helper	Example	Description
`DisplayForModel`	`Html.DisplayForModel()`	Renders a read-only view of the entire model object
`EditorForModel`	`Html.EditorForModel()`	Renders editor elements for the entire model object
`LabelForModel`	`Html.LabelForModel()`	Renders an HTML `<label>` element referring to the entire model object

Tip This is not the same kind of scaffolding that Microsoft added to Visual Studio to create MVC components lik and views, but the basic idea is the same in which output is generated based on the characteristics of a data type. In Visual Studio, the output from the scaffolding is a class or Razor file and for the templated helpers, the output is HTM

In Listing 22-9, you can see how I have used the `LabelForModel` and `EditorForModel` helper meth simplify the `CreatePerson.cshtml` view.

***Listing 22-9**. Using the Scaffolding Helper Methods in the CreatePerson.cshtml File*

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}

<h2>CreatePerson: @Html.LabelForModel()</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    @Html.EditorForModel()

    <input type="submit" value="Submit" />
}
```

You can see the effect of the scaffold helpers in Figure 22-4. Once again, you can see what the helpers are trying things are not quite right yet. The `LabelForModel` helper has not generated a useful label. Although more prop the `Person` model object are being shown than I defined manually in previous examples, not everything is visible (`Address` property) and what is visible is not always useful (such as the `Role` property, which would be more u expressed as a `select` instead of an `input` element).

eBooks-IT.org

Figure 22-4. Using the scaffolding helpers to create an editor for the Person model object

Part of the problem is that the HTML that the scaffolding helpers generate doesn't correspond to the CSS styles th the `/Views/Shared/_Layout.cshtml` file in the previous chapter. Here is an example of the HTML g edit the `FirstName` property:

```
...
<div class="editor-label" >
    <label for="FirstName">FirstName</label>
</div>
<div class="editor-field" >
      <input class="text-box single-line" id="FirstName" name="Fi
type="text" value="" />
</div>
...
```

I can tidy up the appearance of a view by adding styles to the layout that correspond to the CSS `class` values a `div` and `input` elements by the scaffolding helpers. In Listing 22-10, you can see the changes I made to the `_Layout.cshtml` file.

Listing 22-10. Making Changes to the CSS Styles Defined in the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style type="text/css">
        label { display: inline-block; width: 100px; }
        div.dataElem { margin: 5px; }
        h2 > label { width: inherit; }
        .editor-label, .editor-field { float: left; margin-top: 10
```

eBooks-IT.org

```
        .editor-label { clear: left;}
        .editor-field { margin-left: 10px;  }
        input[type=submit] { float: left; clear: both; margin-top:
        .column { float: left; margin: 10px;}
    </style>
</head>
<body>
    @RenderBody()
</body>
</html>
```

These new styles produce something that is more in keeping with the layout I used in earlier examples, as shown i

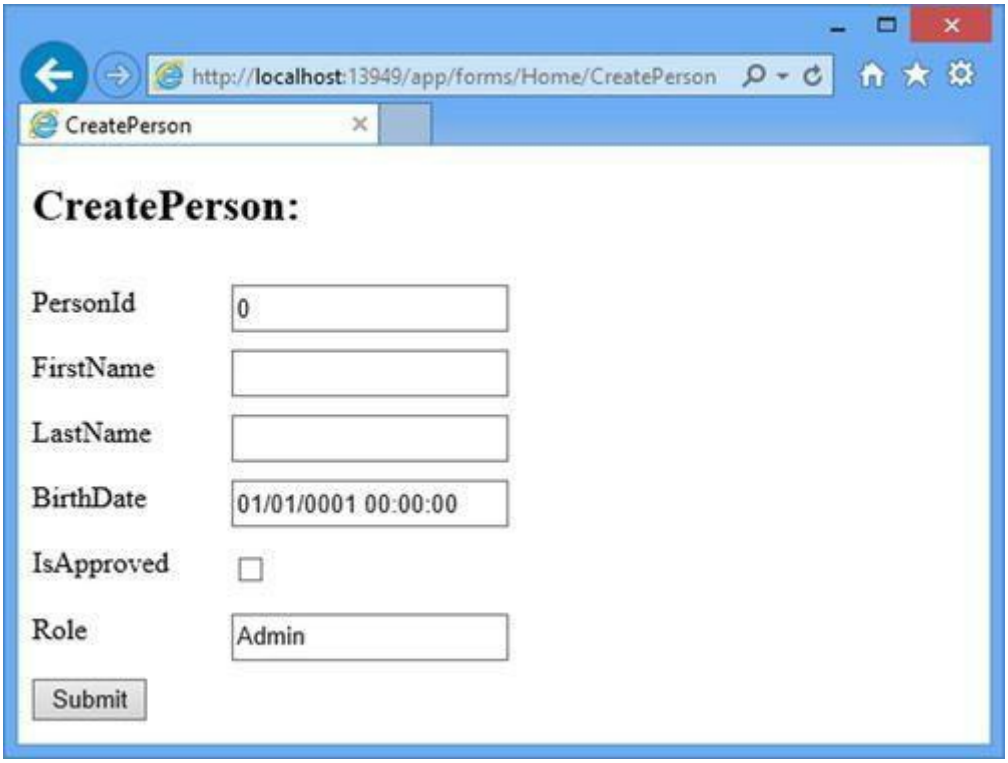


Figure 22-5. The effect of styling elements using the classes defined by the scaffolding helper methods

Using Model Metadata

As you have seen, the templated helpers have no special knowledge about the application and its model data types, an with HTML that is not what exactly what I desire. I want the benefits that come with simpler views, but I need to im quality of the output that the helper methods generate before I can use them seriously.

I cannot blame the templated helpers in these situations; the HTML that is generated is based on a best guess abou This is, of course, the problem I have with all scaffolding, which has to make a best-effort attempt based on a gener understanding of the application. Fortunately, the templated helpers can be improved by using model metadata to pro about how to handle model types. Metadata is expressed using C# attributes, where attributes and parameter values p range of instructions to the view helpers. The metadata is applied to the model class, which the helper methods cons generate HTML elements. In the following sections, I show you how to use metadata to provide directions to the hel labels, displays, and editors.

Using Metadata to Control Editing and Visibility

In the `Person` class, the `PersonId` property is one that I do not want the user to be able to see or edit. Most r have at least one such property, often related to the underlying storage mechanism—a primary key that is managed b

eBooks-IT.org

attribute, which causes the helper to render a hidden input field. You can see how I have applied the HiddenA
to the Person class in Listing 22-11.

Listing 22-11. Using the HiddenInput Attribute in the Person.cs File

```
using System;
using System.Web.Mvc;

namespace HelperMethods.Models {

    public class Person {
        [HiddenInput]
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    // ... other types omitted for brevity ...
}
```

When this attribute has been applied, the Html.EditorFor and Html.EditorForModel helpers w
read-only view of the decorated property (which is the term used to describe a property to which an attribute has be
shown in Figure 22-6, which shows the effect of starting the application and navigating to the /Home/CreateP
URL.

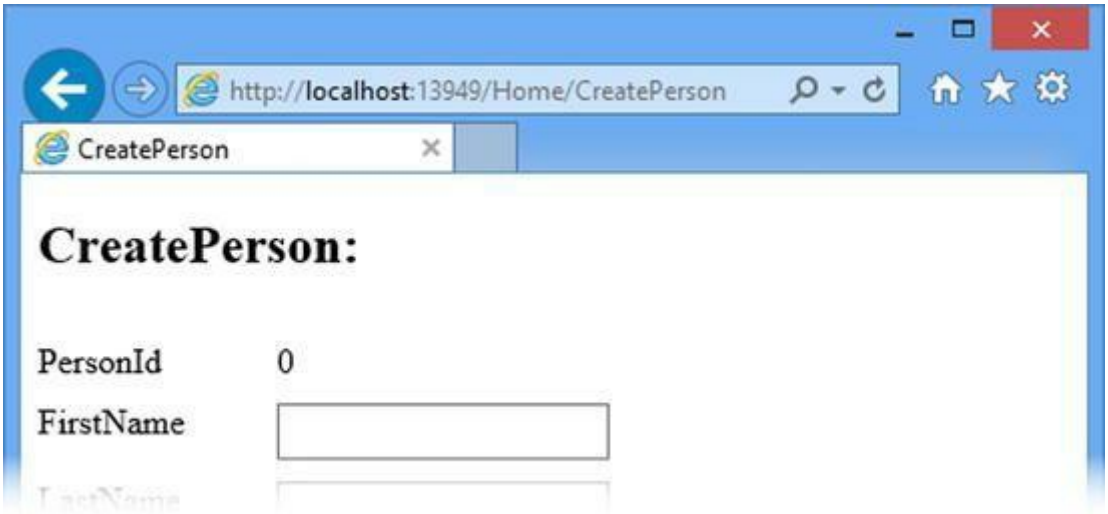


Figure 22-6. Creating a read-only representation of a property in an editor

The value of the PersonId property is shown, but the user cannot edit it. The HTML that is generated for the
follows:

```
...
<div class="editor-field">
    0
    <input id="PersonId" name="PersonId" type="hidden" value="0" /
</div>
...
```

eBooks-IT.org

. eleme
property, which is helpful for HTML forms because it ensures that a value for the property is submitted along with t
form. This is something I will return to when I look at model binding in Chapter 24 and model validation in Chapter 2
to hide a property entirely, then I can set the value of the `DisplayValue` property in the `DisplayName` at
`false`, as shown in Listing 22-12.

Listing 22-12. Using the HiddenInput Attribute to Hide a Property in the Person.cshtml File

```
...
public class Person {
    [HiddenInput(DisplayValue=false)]
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
    public Address HomeAddress { get; set; }
    public bool IsApproved { get; set; }
    public Role Role { get; set; }
}
...
```

When I use the `Html.EditorForModel` helper on a `Person` object, a hidden `input` will be created
value for the `PersonId` property will be included in any form submissions, but the label and the value will be om
the effect of hiding the `PersonId` property from the user, as shown by Figure 22-7.

Figure 22-7. Hiding model object properties from the user

If I have chosen to render HTML for individual properties, I can still create the hidden input for the `PersonId`
using the `Html.EditorFor` helper, like this:

```
...
@Html.EditorFor(m => m.PersonId)
...
```

The `HiddenInput` property is detected, and if `DisplayValue` is `true`, then the following HTML is g

```
...
<input id="PersonId" name="PersonId" type="hidden" value="1" />
...
```

EXCLUDING A PROPERTY FROM SCAFFOLDING

eBooks-IT.org

bute. W
HiddenInput attribute includes a value for the property in a hidden input element, the ScaffoldC
attribute can mark a property as being entirely off limits for the scaffolding process. Here is an example of the a

```
...
[ScaffoldColumn(false)]
public int PersonId { get; set; }
...
```

When the scaffolding helpers see the ScaffoldColumn attribute applied in this way, they skip over the pr
entirely; no hidden input elements will be created, and no details of this property will be included in the gene
The appearance of the generated HTML will be the same as if I had used the HiddenInput attribute, but n
be returned for the property during a form submission. This has an effect on model binding, which I discuss in
The ScaffoldColumn attribute doesn't have an effect on the per-property helpers, such as EditorFo
@Html.EditorFor(m => m.PersonId) in a view, then an editor for the PersonId propert
generated, even when the ScaffoldColumn attribute is present.

Using Metadata for Labels

By default, the Label, LabelFor, LabelForModel, and EditorForModel helpers use the names
as the content for the label elements they generate. For example, if I render a label like this:

```
...
@Html.LabelFor(m => m.BirthDate)
...
```

the HTML element that is generated will be as follows:

```
...
<label for="BirthDate">BirthDate</label>
...
```

Of course, the names given to properties are often not suitable for display to the user. To that end, I can apply the
DisplayName attribute from the System.ComponentModel.DataAnnotations namespace,
value I want as a value for the Name property. Listing 22-13 demonstrates this attribute applied to the Person cl

Listing 22-13. Using the DisplayName Attribute to Define a Label in the Person.cs File

```
using System;
using System.Web.Mvc;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel;

namespace HelperMethods.Models {

    [DisplayName("New Person")]
    public class Person {
        [HiddenInput(DisplayValue=false)]
        public int PersonId { get; set; }

        [Display(Name="First")]
        public string FirstName { get; set; }

        [Display(Name = "Last")]
        public string LastName { get; set; }
```

eBooks-IT.org

```
        public DateTime BirthDate { get; set; }

        public Address HomeAddress { get; set; }

        [Display(Name="Approved")]
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    // ...other types omitted for brevity...
}
```

When the label helpers render a label element for the `BirthDate` property, they will detect the `Display` att the value of the `Name` parameter for the inner text, like this:

```
...
<label for="BirthDate">Birth Date</label>
...
```

The helpers also recognize the `DisplayName` attribute, which can be found in the `System.Componen` namespace. This attribute has the advantage of being able to be applied to classes, which allows me to use the `Html.LabelForModel` helper. You can see how I have applied this attribute to the `Person` class in the lis apply the `DisplayName` attribute to properties as well, but I tend to use this attribute only for model classes, fo other than habit.) You can see the effect of the `Display` and `DisplayName` attributes in Figure 22-8.

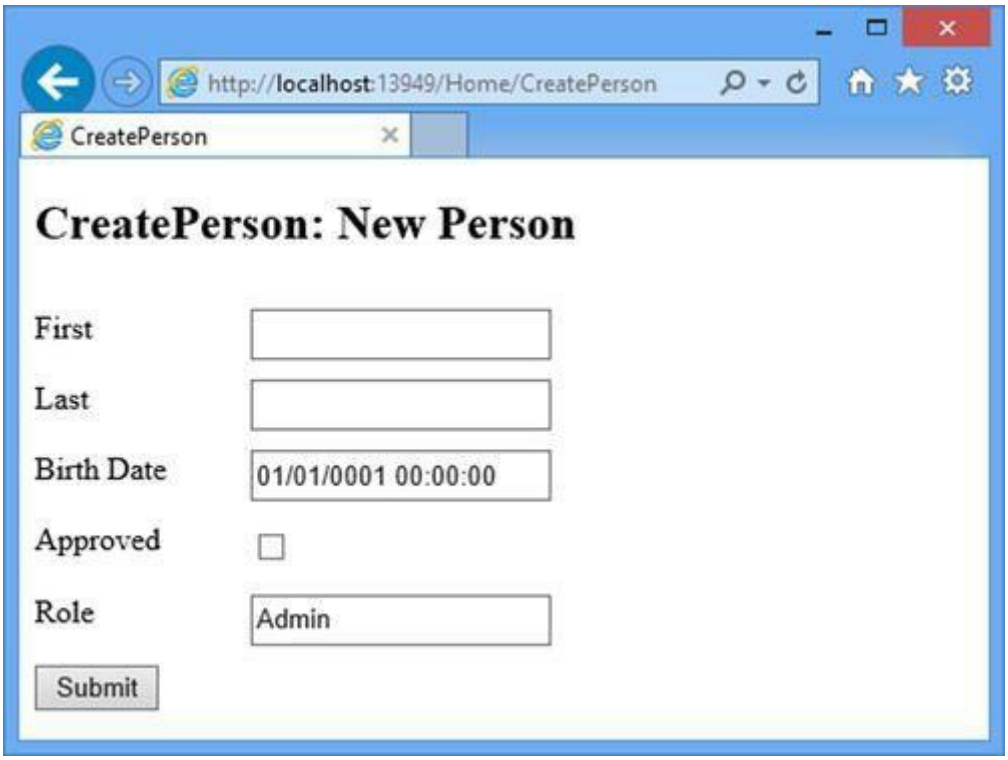


Figure 22-8. Using the Display and DisplayName attributes to control labels

Using Metadata for Data Values

I can also use metadata to provide instructions about how a model property should be displayed. I can use this to dea fact that the `BirthDate` property is displayed with a time when I really just want a date, for example. I control t data values are displaying using the `DataType` attribute, which you can see applied to the `Person` class in List

```
...
[DisplayName("New Person")]
public class Person {
    [HiddenInput(DisplayValue=false)]
    public int PersonId { get; set; }

    [Display(Name="First")]
    public string FirstName { get; set; }

    [Display(Name = "Last")]
    public string LastName { get; set; }

    [Display(Name = "Birth Date")]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; set; }

    public Address HomeAddress { get; set; }

    [Display(Name="Approved")]
    public bool IsApproved { get; set; }
    public Role Role { get; set; }
}
...
```

The `DataType` attribute takes a value from the `DataType` enumeration as a parameter. In the example I ha the `DataType.Date` value, which causes the templated helpers to render the value of the `BirthDate` prop without a time component, as shown in Figure 22-9.

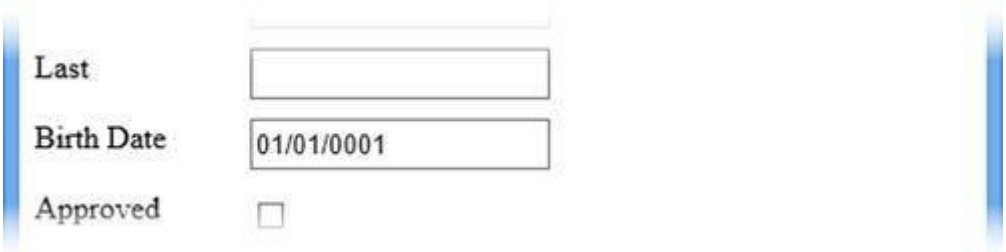


Figure 22-9. Using the DataType attribute to control the display of a DateTime value

Tip The change is more pronounced when the application is viewed using a Web browser that has better support HTML5 `input` element types.

Table 22-4 describes the most useful values of the `DataType` enumeration.

***Table 22-4**. The Values of the DataType Enumeration*

Value	Description
`DateTime`	Displays a date and time (this is the default behavior for `System.DateTime` values)
`Date`	Displays the date portion of a `DateTime`
`Time`	Displays the time portion of a `DateTime`
`Text`	Displays a single line of text
`PhoneNumber`	Displays a phone number
`MultilineText`	Renders the value in a `textarea` element
`Password`	Displays the data so that individual characters are masked from view

eBooks-IT.org

EmailAddress	Displays the data as an e-mail address (using an a element with a mailto href)	

The effect of these values depends on the type of the property they are associated with and the helper being used. the MultilineText value will lead those helpers that create editors for properties to create an HTML texta but will be ignored by the display helpers. This makes sense. The textarea element allows the user to edit a valu doesn't have a beating when displaying the data in a read-only form. Equally, the Url value has an effect only on th helpers, which render an HTML a element to create a link.

Using Metadata to Select a Display Template

As their name suggests, templated helpers use display templates to generate HTML. The template that is used is base of the property being processed and the kind of helper being used. I can use the UIHint attribute to specify the te to render HTML for a property, as shown in Listing 22-15.

Listing 22-15. Using the UIHint Attribute in the Person.cshtml File

```
...
[DisplayName("New Person")]
public class Person {
    [HiddenInput(DisplayValue=false)]
    public int PersonId { get; set; }

    [Display(Name="First")]
    [UIHint("MultilineText")]
    public string FirstName { get; set; }

    [Display(Name = "Last")]
    public string LastName { get; set; }

    [Display(Name = "Birth Date")]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; set; }

    public Address HomeAddress { get; set; }

    [Display(Name="Approved")]
    public bool IsApproved { get; set; }
    public Role Role { get; set; }
}
...
```

In the listing, I specified the MultilineText template, which renders an HTML textarea element for FirstName property when used with one of the editor helpers, such as EditorFor or EditorForMod 22-5 shows the set of built-in templates that the MVC Framework includes.

Table 22-5. *The Built-In MVC Framework View Templates*

Value	Effect (Editor)	Effect (Display)
Boolean	Renders a checkbox for bool values. For nullable bool? values, a select element is created with options for True, False, and Not Set.	As for the editor helpers, but with the add disabled attribute, which renders re controls
Collection	Renders the appropriate template for each of the elements in an IEnumerable sequence. The items in the sequence do not have to be of the same type.	As for the editor helper
Decimal	Renders a single-line textbox input element and formats the data value	Renders the data value formatted to two d

eBooks-IT.org

DateTime	Renders an input element whose type attribute is datetime and which contains the complete date and time	Renders the complete value of a Date
Date	Renders an input element whose type attribute is date and that contains the date component (but not the time)	Renders the date component of a Dat
EmailAddress	Renders the value in a single-line textbox input element	Renders a link using an HTML a elem href attribute that is formatted as a mai
HiddenInput	Creates a hidden input element	Renders the data value and creates a hidde element
Html	Renders the value in a single-line textbox input element	Renders a link using an HTML a elem
MultilineText	Renders an HTML textarea element that contains the data value	Renders the data value
Number	Renders an input element whose type attribute is set to number	Renders the data value
Object	See explanation after this table	See explanation after this table
Password	Renders the value in a single-line textbox input element so that the characters are not displayed but can be edited	Renders the data value—the characters ar obscured
String	Renders the value in a single-line textbox input element	Renders the data value
Text	Identical to the String template	Identical to the String template
Tel	Renders an input element whose type attribute is set to tel	Renders the data value
Time	Renders an input element whose type attribute is time and which contains the time component (but not the date)	Renders the time component of a Dat
Url	Renders the value in a single-line textbox input element	Renders a link using an HTML a elem HTML and the href attribute are both s value.

Caution Care must be taken when using the UIHint attribute. I will receive an exception if I select a template operate on the type of the property I have applied it to, for example, applying the Boolean template to a strin

The Object template is a special case. It is the template used by the scaffolding helpers to generate HTML for object. This template examines each of the properties of an object and selects the most suitable template for the prop Object template takes metadata such as the UIHint and DataType attributes into account.

Applying Metadata to a Buddy Class

It is not always possible to apply metadata to an entity model class. This is usually the case when the model classes a automatically, like sometimes with ORM tools such as the Entity Framework (although not the way I used Entity Fra the SportsStore application). Any changes applied to automatically generated classes, such as applying attributes, wil next time the classes are updated or regenerated.

The solution to this problem is to ensure that the model class is defined as partial and to create a second pa class that contains the metadata. Many tools that generate classes automatically create partial classes by defaul the Entity Framework. Listing 22-16 shows the Person class modified such that it could have been generated aut There is no metadata, and the class is defined as partial.

Listing 22-16. A Partial Model Class in the Person.cs File

```
using System;
using System.ComponentModel.DataAnnotations;

namespace HelperMethods.Models {

    [MetadataType(typeof(PersonMetaData))]
    public partial class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
```

eBooks-IT.org

```
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    // ...other types omitted from listing for brevity...
}
```

I tell the MVC Framework about the buddy class through the `MetadataType` attribute, which takes the type class as its argument. Buddy classes must be defined in the same namespace and must also be `partial` classes. demonstrate how this works, I have added a new folder to the example project called `Models/Metadata`. In created a new class file called `PersonMetadata.cs`, the contents of which are shown in Listing 22-17.

Listing 22-17. The Contents of the PersonMetadata.cs File

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace HelperMethods.Models {

    [DisplayName("New Person")]
    public partial class PersonMetaData {
        [HiddenInput(DisplayValue=false)]
        public int PersonId { get; set; }

        [Display(Name="First")]
        public string FirstName { get; set; }

        [Display(Name = "Last")]
        public string LastName { get; set; }

        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime BirthDate { get; set; }

        [Display(Name="Approved")]
        public bool IsApproved { get; set; }
    }
}
```

The buddy class only needs to contain properties to which metadata applies. I do not have to replicate all of the pr `Person` class, for example.

Tip Take particular care to change the namespace that Visual Studio adds to the new class file. The buddy class same namespace as the model class, which is `HelperMethods.Models` for the example project.

Working with Complex Type Properties

The template process relies on the `Object` template that I described in the previous section. Each property is insp template is selected to render HTML to represent the property and its data value.

You may have noticed that the `HomeAddress` property was not rendered as part of the `Person` class wher `EditorForModel` helper. This happens because the `Object` template operates only on *simple types*, which

eBooks-IT.org

System.ComponentModel.TypeDescriptor class. The supported types include the intrinsic C# ty int, bool, and double, and many common framework types, including Guid and DateTime.

The result is that scaffolding helpers are not recursive. Given an object to process, a scaffolding templated helper generate HTML only for simple property types and will ignore any properties that are complex objects.

Although it can be inconvenient, this is a sensible policy. The MVC Framework does not know how the model obj created. If the Object template was recursive, then it could easily end up triggering an ORM lazy-loading feature, lead it to read and render every object in the underlying database. If I want to render HTML for a complex property, explicitly by making a separate call to a templated helper method. You can see how I have done this in Listing 22-18, the changes I have made to the CreatePerson.cshtml view.

Listing 22-18. Dealing with a Property That Is a Complex Type in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson: @Html.LabelForModel()</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="column">
        @Html.EditorForModel()
    </div>
    <div class="column">
        @Html.EditorFor(m => m.HomeAddress)
    </div>
    <input type="submit" value="Submit" />
}
```

To display the HomeAddress property, I added a call to the strongly typed EditorFor helper method. (I some div elements to provide structure to the HTML that is generated, relying on the CSS style I defined for the c class back in Listing 22-10.) You can see the result in Figure 22-10.

eBooks-IT.org

Figure 22-10. Displaying a complex property

Tip The `HomeAddress` property is typed to return an `Address` object, and I can apply all of the same m `Address` class as I did to the `Person` class. The `Object` template is invoked explicitly when I use the `Edi` helpers on the `HomeAddress` property, and so all of the metadata conventions are honored.

Customizing the Templated View Helper System

I have shown you how to use metadata to shape the way that the templated helpers render data, but this is the MVC and so there are some advanced options that completely customize the templated helpers. In the following sections, I you how to supplement or replace the built-in support to create specific results.

Creating a Custom Editor Template

One of the easiest ways of customizing the templated helpers is to create a custom template. This allows me to rende HTML I want for a model property.

To demonstrate how this feature works, I am going to create a custom template for the `Role` property in the `Pe` This property is typed to be a value from the `Role` enumeration, but the way that this is rendered by default is prob because the templated helpers just create a regular `input` element that allows the user to enter any value and not ju defined by the enumeration.

The MVC Framework looks for custom editor templates in the `/Views/Shared/EditorTemplates` created this folder in the example project and then created a new strongly typed partial view called `Role.cshtm` You can see the contents of this file in Listing 22-19.

Listing 22-19. The Contents of the Role.cshtml File

```
@model HelperMethods.Models.Role

@Html.DropDownListFor(m => m,
    new SelectList(Enum.GetNames(Model.GetType()), Model.ToString(
```

The model type for this view is the `Role` enumeration and I use the `Html.DropDownListFor` helper m create a `select` with `option` elements for the values in the enumeration. I passed an additional value to the

eBooks-IT.org

DropDownListFor method and the SelectList object operate on String values, so I have to make ct. The cast the enumeration values and the view model value.

When I use any of the templated helper methods to generate an editor for the `Role` type, my `/Views/Shared/EditorTemplates/Role.cshtml` file will be used, ensuring that I present the consistent and usable representation of the data type. You can see the effect of the custom template in Figure 22-11.

Figure 22-11. The effect of a custom template for the Role enumeration

UNDERSTANDING THE TEMPLATE SEARCH ORDER

The `Role.cshtml` template works because the MVC Framework looks for custom templates for a given C it uses one of the built-in templates. In fact, there is a specific sequence that the MVC Framework follows to fi template:

1. The template passed to the helper. For example, `Html.EditorFor(m => m.SomeProperty, "MyTemplate")` would lead to `MyTemplate` being used.
2. Any template that is specified by metadata attributes, such as `UIHint`
3. The template associated with any data type specified by metadata, such as the `DataType` attribute
4. Any template that corresponds to the.NET class name of the data type being processed
5. The built-in `String` template if the data type being processed is a simple type
6. Any template that corresponds to the base classes of the data type
7. If the data type implements `IEnumerable`, then the built-in `Collection` template will be used.
8. If all else fails, the `Object` template will be used, subject to the rule that scaffolding is not recursive.

Some of these steps rely on the built-in templates, which are described in Table 22-5. At each stage in the templ process, the MVC Framework looks for a template called `EditorTemplates/` <*name*> for editor helpe `DisplayTemplates/` <*name*> for display helper methods. For the `Role` template, I satisfied step 4 i process; I created a template called `Role.cshtml` and placed it in the `/Views/Shared/EditorTemplates` folder.

Custom templates are found using the same search pattern as regular views, which means I can create a contro custom template and place it in the `~/Views/` <*controller*> `/EditorTemplates` folder to override t found in the `~/Views/Shared/EditorTemplates` folder.

Creating a Generic Template

I am not limited to creating type-specific templates. I can, for example, create a template that works for all enumerat specify that this template be selected using the `UIHint` attribute. If you look at the template search sequence in th "Understanding the Template Search Order" sidebar, you will see that templates specified using the `UIHint` attribu precedence over type-specific ones.

To demonstrate how this works, I have created a new view file called `Enum.cshtml` in the

Listing 22-20. The Contents of the Enum.cshtml

```
@model Enum

@Html.DropDownListFor(m => m, Enum.GetValues(Model.GetType())
    .Cast<Enum>()
    .Select(m => {
        string enumVal = Enum.GetName(Model.GetType(), m);
        return new SelectListItem() {
            Selected = (Model.ToString() == enumVal),
            Text = enumVal,
            Value = enumVal
        };
    }))
```

The model type for this template is `Enum`, which allows me to work with any enumeration. For variety, I have us LINQ to generate the strings that are required to create the `select` and `option` elements (although this is not a for a generic template: I just like LINQ).

I can then apply the `UIHint` attribute. The example project defines a metadata buddy class, so I have applied th the `PersonMetadata` class, as shown in Listing 22-21. (As a reminder, you can find this class defined in the `/Models/Metadata/PersonMetadata.cs` file.)

Listing 22-21. Using the UIHint Attribute to Specify a Custom Template in the PersonMetadata.cs File

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace HelperMethods.Models {

    [DisplayName("New Person")]
    public partial class PersonMetaData {
        [HiddenInput(DisplayValue=false)]
        public int PersonId { get; set; }

        [Display(Name="First")]
        public string FirstName { get; set; }

        [Display(Name = "Last")]
        public string LastName { get; set; }

        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime BirthDate { get; set; }

        [Display(Name="Approved")]
        public bool IsApproved { get; set; }

        [UIHint("Enum")]
        public Role Role { get; set; }
    }
}
```

eBooks-IT.org

displayed using a `select` element. I prefer to create model type-specific custom templates, but it can be more co
have one template that you can apply widely.

Replacing the Built-in Templates

If I create a custom template that has the same name as one of the built-in templates, the MVC Framework will use t
version in preference to the built-in one. Listing 22-22 shows the contents of the `Boolean.cshtml` file that I
`/Views/Shared/EditorTemplates` folder. This view replaces the built-in `Boolean` template whic
render `bool` and `bool?` values.

Listing 22-22. The Contents of the Boolean.cshtml File

```
@model bool?

@if (ViewData.ModelMetadata.IsNullableValueType && Model == null)
    @:(True) (False) <b>(Not Set)</b>
} else if (Model.Value) {
    @:<b>(True)</b> (False) (Not Set)
} else {
    @:(True) <b>(False)</b> (Not Set)
}
```

In this view, I display all of the possible values and highlight the one that corresponds to the model object. You ca
effect of this template in Figure 22-12.

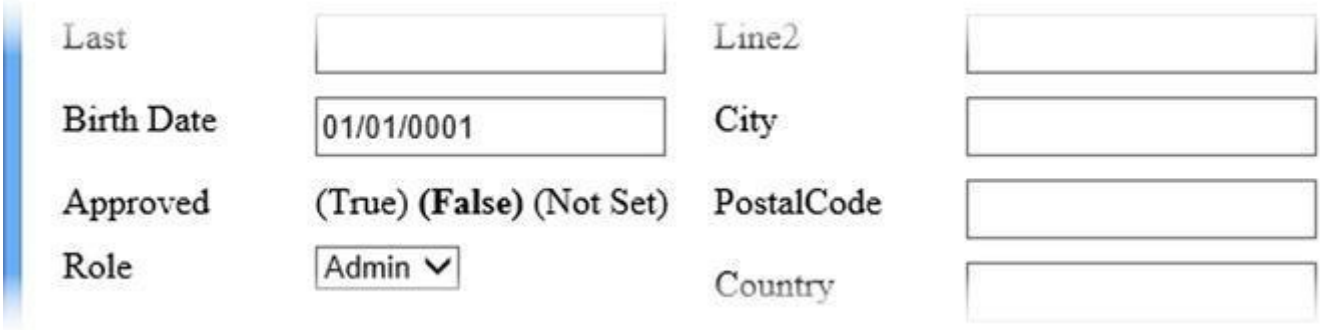


Figure 22-12. *The effect of overriding a built-in editor template*

You can see the flexibility that custom templates offer, even if the example I have shown is not especially useful a
the user change the property value. As you have seen, there are a number of different ways that you can control how
properties are displayed and edited, and you can pick the approach that suits your programming style and application

Summary

In this chapter, I have shown you the system of model templates that are accessible through the templated view help
can take a little while to set up the metadata and to create custom templates, but the result is closely tailored to your
and gives you complete flexibility in how your view model data is displayed and edited.

eBooks-IT.org

URL and Ajax Helper Methods

In this chapter, I am going to complete my coverage of the MVC Framework helper methods by showing you those are able to generate URLs, links, and Ajax-enabled elements. Ajax is a key feature of any rich Web application and the Framework includes some useful features that are based on the jQuery library. I'll show you how this works and den you can use it to create Ajax-enabled forms and links. Table 23-1 provides the summary for this chapter.

***Table 23-1**. Chapter Summary*

Problem	Solution
Generate links and URLs	Use the `Url.Content`, `Url.Action`, `Url.RouteUrl`, `Html.ActionLin` `Html.RouteLink` helpers.
Submit form data via Ajax	Use the unobtrusive Ajax package and the `Ajax.BeginForm` helper.
Ensure that non-JavaScript browsers do not display HTML fragments	Set the `Url` Ajax option.
Provide the user with feedback during an Ajax request	Use the `LoadingElementId` and `LoadingElementDuration` Ajax option
Prompt the user before making an Ajax request	Use the `Confirm` Ajax option.
Create an Ajax enabled link	Use the `Ajax.ActionLink` helper.
Receive notifications about the progress and outcome of Ajax requests	Use the Ajax callback options.
Use JSON data in Ajax requests	Use the `JsonResult` action result.
Detect Ajax requests in the controller	Use the `Request.IsAjaxRequest` method.

Note You will need to clear the browser history as you go from one example in this chapter to the next. This is j build features incrementally and isn't something you would need to worry about in a real project. I have added notes at key points in the chapter, but if you don't get the result you are expecting from an example then the first thing to t the history.

Preparing the Example Project

I am going to continue using the `HelperMethods` project that I created in Chapter 21 and added to in Chapter chapter, I have created a new controller called `People`, shown in Listing 23-1. This controller defines a collection model objects that I will use to demonstrate different helper features.

***Listing 23-1**.* The Contents of the PeopleController.cs File

```
using System;
using System.Linq;
using System.Web.Mvc;
using HelperMethods.Models;

namespace HelperMethods.Controllers {
    public class PeopleController : Controller {
```

eBooks-IT.org

```
                new Person {FirstName = "Adam", LastName = "Freeman"
Role.Admin},
                new Person {FirstName = "Jacqui", LastName = "Griffyt
= Role.User},
                new Person {FirstName = "John", LastName = "Smith",
Role.User},
                new Person {FirstName = "Anne", LastName = "Jones",
Role.Guest}
        };

        public ActionResult Index() {
            return View();
        }

        public ActionResult GetPeople() {
            return View(personData);
        }

        [HttpPost]
        public ActionResult GetPeople(string selectedRole) {
            if (selectedRole == null || selectedRole == "All") {
                return View(personData);
            } else {
                            Role selected = (Role)Enum.Parse(typeo
selectedRole);
                return View(personData.Where(p => p.Role == select
            }
        }
    }
}
```

I have not used any new techniques in this controller. The `Index` action method returns the default view. I will
`GetPeople` action methods to handle a simple form. The new features in this chapter appear in the views, whic
demonstrate different helper methods.

Defining Additional CSS Styles

I also need to add some new CSS styles to the project, which I have done in the `Views/Shared/_Layout`
file, as shown in Listing 23-2. I will define the elements the new styles apply to as I go through the chapter.

Listing 23-2. Adding Styles to the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style type="text/css">
        label { display: inline-block; width: 100px; }
        div.dataElem { margin: 5px; }
        h2 > label { width: inherit; }
        .editor-label, .editor-field { float: left; margin-top: 10
        .editor-field input { height: 20px; }
```

eBooks-IT.org

```
        .editor field { margin left: 10px; }
        input[type=submit] { float: left; clear: both; margin-top:
        .column { float: left; margin: 10px; }
            table, td, th { border: thin solid black; border-c
collapse;
            padding: 5px; background-color: lemonchiffon;
            text-align: left; margin: 10px 0; }
        div.load { color: red; margin: 10px 0; font-weight: bold;
        div.ajaxLink { margin-top: 10px; margin-right: 5px; float:
    </style>
</head>
<body>
    @RenderBody()
</body>
</html>
```

Installing the NuGet Packages

The MVC Framework relies on the Microsoft *Unobtrusive Ajax* package to make and process Ajax requests. To inst package, select Package Manager Console from the Visual Studio `Tools` ➤ `Library Package Manage` enter the following command:

```
Install-Package jQuery -version 1.10.2
Install-Package Microsoft.jQuery.Unobtrusive.Ajax -version 3.0.0
```

NuGet will install the package—and the jQuery library it depends on—into the project, creating a `Scripts` fold contains a number of JavaScript files.

Creating Basic Links and URLs

One of the most fundamental tasks in a view is to create a link or URL that the user can follow to another part of the In previous chapters, you saw most of the helper methods that you can use to create links and URLs, but I want to t to recap before moving on to some of the more advanced helpers that are available. Table 23-2 describes the availabl helpers and it shows examples of each of them.

Table 23-2. HTML Helpers That Render URLs

Description	Example

Application-relative URL	`Url.Content("~/Content/Site.css")`
	Output:
	`/Content/Site.css`

Link to named action/controller	`Html.ActionLink("My Link", "Index", "Home")`
	Output:
	`<a href="/">My Link</a>`

URL for action	`Url.Action("GetPeople", "People")`
	Output:
	`/People/GetPeople`

URL using route data	`Url.RouteUrl(new {controller = "People", action="GetPeople"})`
	Output:
	`/People/GetPeople`

Output:
<a href="/People/GetPeople">My Link</a>

Html.RouteLink("My Link", "FormRoute", new {controller = "People", action="GetPeople"})

Link to named route

Output:
<a href="/app/forms/People/GetPeople">My Link</a>

Tip As a reminder, the benefit of using these helpers to generate links and URLs is that the output is derived from configuration, which means that a change in routes is automatically reflected in the links and URLs.

To demonstrate these helpers in action, I added an `Index.cshtml` view file to the `Views/People` fold contents of which you can see in Listing 23-3.

Listing 23-3. The Contents of the Index.cshtml File

```
@{
    ViewBag.Title = "Index";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>Basic Links & URLs</h2>
<table>
    <thead><tr><th>Helper</th><th>Output</th></tr></thead>
    <tbody>
        <tr>
            <td>Url.Content("~/Content/Site.css")</td>
            <td>@Url.Content("~/Content/Site.css")</td>
        </tr>
        <tr>
            <td>Html.ActionLink("My Link", "Index", "Home")</td>
            <td>@Html.ActionLink("My Link", "Index", "Home")</td>
        </tr>
        <tr>
            <td>Url.Action("GetPeople", "People")</td>
            <td>@Url.Action("GetPeople", "People")</td>
        </tr>
        <tr>
                        <td>Url.RouteUrl(new  {controller  =  "
action="GetPeople"})</td>
                       <td>@Url.RouteUrl(new  {controller  =  "
action="GetPeople"})</td>
        </tr>
        <tr>
            <td>Html.RouteLink("My Link", new {controller = "Peopl
                action="GetPeople"})</td>
            <td>@Html.RouteLink("My Link", new {controller = "Peop
                action="GetPeople"})</td>
        </tr>
        <tr>
             <td>Html.RouteLink("My Link", "FormRoute", new {cont
"People",
                action="GetPeople"})</td>
            <td>@Html.RouteLink("My Link", "FormRoute", new {cont
"People",
```

eBooks-IT.org

```
        </tr>
    </tbody>
</table>
```

This view contains the same set of helper calls that I listed in Table 23-2 and presents the results in an HTML table the effect by starting the application and navigating to the `/People/Index` URL, as shown in Figure 23-1. I ha this example because it makes it easy to experiment with routing changes and immediately see the effect.

Figure 23-1. Using helpers to create links and URLs

Using MVC Unobtrusive Ajax

Ajax (or, if you prefer, AJAX) is shorthand for *Asynchronous JavaScript and XML*. The XML part is not as significa to be, but the asynchronous part is what makes Ajax useful. It is a model for requesting data from the server in the b without having to reload the Web page. The MVC Framework contains built-in support for *unobtrusive* Ajax, which you use helper methods to define your Ajax features, rather than having to add blocks of code throughout your views

Tip The MVC Framework unobtrusive Ajax feature is based on jQuery. If you are familiar with the way that jQu Ajax, then you will understand the MVC Ajax features.

Creating the Synchronous Form View

I am going to begin this section by creating the view for the `GetPeople` action in the controller, which I created `/Views/People/GetPeople.cshtml` file. You can see the contents of this file in Listing 23-4.

Listing 23-4. The Contents of the GetPeople.cshtml View File

```
@using HelperMethods.Models
@model IEnumerable<Person>
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>Get People</h2>
```

eBooks-IT.org

```
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
    <tbody>
        @foreach (Person p in Model) {
            <tr>
                <td>@p.FirstName</td>
                <td>@p.LastName</td>
                <td>@p.Role</td>
            </tr>
        }
    </tbody>
</table>

@using (Html.BeginForm()) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

This is a strongly typed view whose model type is `IEnumerable<Person>`. I enumerate the `Person` o model to create rows in an HTML table and use the `Html.BeginForm` helper to create a form that posts back and controller that the view was generated by. The form contains a call to the `Html.DropDownList` helper, to create a `select` element that contains `option` elements for each of the values defined by the `Role` enume the value `All`. (I have used LINQ to create the list of values for the `option` elements by concatenating the value `enum` with an array that contains a single `All` string.)

The form contains a button that submits the form. The effect is that you can use the form to filter the `Person` defined in the controller in Listing 23-1, as shown in Figure 23-2. To test, start the application and navigate to the `/People/GetPeople` URL.

eBooks-IT.org

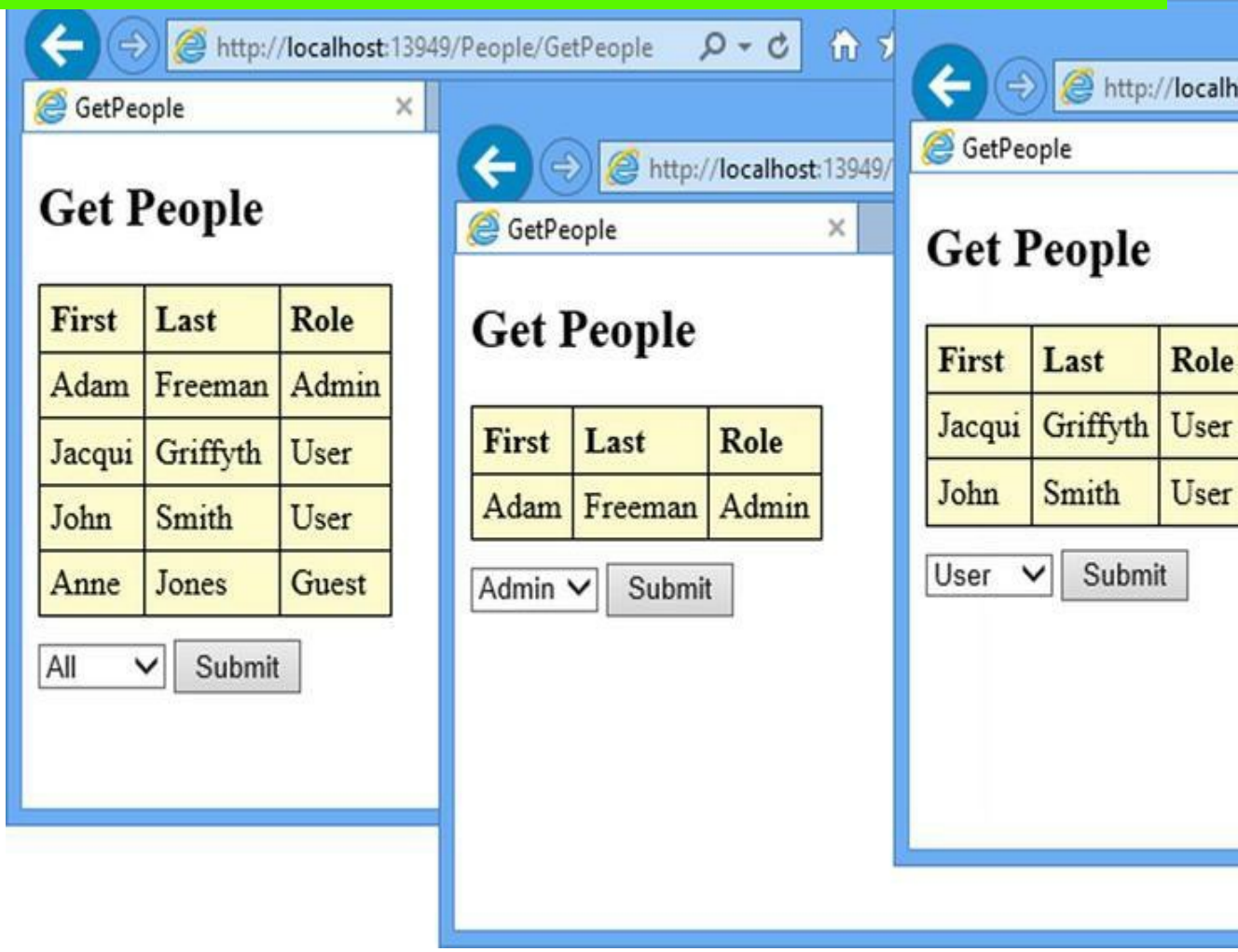


Figure 23-2. A simple synchronous form

This is a simple demonstration of a fundamental limitation in HTML forms, which is that the entire page is reloade form is submitted. It means that the entire content of the Web page has to be regenerated and loaded from the server be an expensive operation for complex views) and while this is happening, users cannot perform any other task with application. They have to wait until the new page is generated, loaded, and then displayed by the browser.

For a simple application like this one, where the browser and server are running on the same machine, the delay is noticeable. But for real applications over real internet connections, synchronous forms can make using a Web applica frustrating for the user and expensive in terms of server bandwidth and processing power.

Preparing the Project for Unobtrusive Ajax

The unobtrusive Ajax feature is set up in two places in the application. First, in the `Web.config` file (the one in of the project) the `configuration/appSettings` element contains an entry for the `UnobtrusiveJavaScriptEnabled` property, which must be set to `true`, as shown in Listing 23-5. is set to `true` by default when Visual Studio creates the project.)

Listing 23-5. Enabling the Unobtrusive Ajax Feature in the Web.config File

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
```

eBooks-IT.org

```
    <add key="unobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5.1" />
    <httpRuntime targetFramework="4.5.1" />
  </system.web>
</configuration>
```

In addition to checking the `Web.config` setting, I need to add references to the jQuery JavaScript libraries tha the unobtrusive Ajax functionality from the NuGet package I added at the start of the chapter. You can reference the individual views, but a more common approach is to do this in a layout file so that it affects all of the views that use Listing 23-6, you can see how I have added references for two JavaScript libraries to the `/Views/Shared/_Layout.cshtml` file.

Listing 23-6. Adding References for the Unobtrusive Ajax JavaScript Libraries to the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style type="text/css">
        label { display: inline-block; width: 100px; }
        div.dataElem { margin: 5px; }
        h2 > label { width: inherit; }
        .editor-label, .editor-field { float: left; margin-top: 10
        .editor-field input { height: 20px; }
        .editor-label { clear: left; }
        .editor-field { margin-left: 10px; }
        input[type=submit] { float: left; clear: both; margin-top:
        .column { float: left; margin: 10px; }
            table, td, th { border: thin solid black; border-c
collapse;
            padding: 5px; background-color: lemonchiffon;
            text-align: left; margin: 10px 0; }
        div.load { color: red; margin: 10px 0; font-weight: bold;
        div.ajaxLink { margin-top: 10px; margin-right: 5px; float:
    </style>
    <script src="~/Scripts/jquery-1.10.2.js"></script>
    <script src="~/Scripts/jquery.unobtrusive-ajax.js"></script>
</head>
<body>
    @RenderBody()
</body>
</html>
```

The files that I have referenced with the `script` elements were added to the `Scripts` folder by the NuGet p `jquery-1.10.2.js` file contains the core jQuery library and the `jquery.unobtrusive-ajax.j` contains the Ajax functionality (which relies on the main jQuery library).

Creating an Unobtrusive Ajax Form

eBooks-IT.org

usive A
the sections that follow, I go through the process of replacing a regular synchronous form with an Ajax equivalent an
how the unobtrusive Ajax feature works.

Preparing the Controller

The goal is that only the data in the HTML `table` element is replaced when the user clicks on the `Submit` butto
example application. That means that the first thing that I need to do is refactor the action methods in the `People`
that I can get just the data I want through a child action. You can see the changes I have made to the `People` con
Listing 23-7.

Listing 23-7. Refactoring the Action Methods in the PeopleController.cs File

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

namespace HelperMethods.Controllers {
    public class PeopleController : Controller {
        private Person[] personData = {
             new Person {FirstName = "Adam", LastName = "Freeman"
Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyt
= Role.User},
              new Person {FirstName = "John", LastName = "Smith",
Role.User},
              new Person {FirstName = "Anne", LastName = "Jones",
Role.Guest}
        };

        public ActionResult Index() {
            return View();
        }

            public PartialViewResult GetPeopleData(string selecte
"All") {
            IEnumerable<Person> data = personData;
            if (selectedRole != "All") {
                         Role selected = (Role)Enum.Parse(typeo
selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            return PartialView(data);
        }

        public ActionResult GetPeople(string selectedRole = "All")
            return View((object)selectedRole);
        }
    }
}
```

I have added a `GetPeopleData` action that selects the `Person` objects that I need to display, and passes t

eBooks-IT.org

GetPeopleData action method, I have been able to simplify the GetPeople action method and remove th HttpPost version entirely. The purpose of this method is to pass the selected role as a string to its view.

I created a new partial view file, /Views/People/GetPeopleData.cshtml, for the new GetPeopleData action method. You can see the contents of the view in Listing 23-8. This view is responsible generating the tr elements that will populate the table using the enumeration of Person objects that are passed fr method.

Listing 23-8. The Contents of the GetPeopleData.cshtml File

```
@using HelperMethods.Models
@model IEnumerable<Person>

@foreach (Person p in Model) {
    <tr>
        <td>@p.FirstName</td>
        <td>@p.LastName</td>
        <td>@p.Role</td>
    </tr>
}
```

I also had to update the /Views/People/GetPeople.cshtml view, which you can see in Listing 2

Listing 23-9. Updating the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>Get People</h2>
<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
    <tbody>
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Html.BeginForm()) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

I have changed the view model type to string, which I pass to the Html.Action helper method to invok GetPeopleData child action. This renders the partial view and generates the table rows.

Creating the Ajax Form

I still have a synchronous form in the application after these changes, but I have separated out the functionality in the that I can request just the table rows through the GetPeopleData action. This new action method will be the Ajax request and the next step is to update the GetPeople.cshtml view so that posting the form is handled

Listing 23-10. Creating an Unobtrusive Ajax Form in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody"
    };
}
<h2>Get People</h2>
<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
    <tbody id="tableBody" >
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm("GetPeopleData", ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

At the heart of the MVC Framework support for Ajax forms is the `Ajax.BeginForm` helper method, which `AjaxOptions` object as its argument. I like to create the `AjaxOptions` objects at the start of the view in a block, but you can create them inline when you call `Ajax.BeginForm` if you prefer.

The `AjaxOptions` class, which is in the `System.Web.Mvc.Ajax` namespace, defines properties tha configure how the asynchronous request to the server is made and what happens to the data that comes back. These are described in Table 23-3.

Table 23-3. AjaxOptions Properties

Property	Description
`Confirm`	Sets a message to be displayed to the user in a confirmation window before making the Ajax request
`HttpMethod`	Sets the HTTP method that will be used to make the request—must be either `Get` or `Post`
`InsertionMode`	Specifies the way in which the content retrieved from the server is inserted into the HTML. The three cl expressed as values from the `InsertionMode` enum: `InsertAfter`, `InsertBefore` and Rep the default).
`LoadingElementId`	Specifies the ID of an HTML element that will be displayed while the Ajax request is being performed
`LoadingElementDuration`	Specifies the duration of the animation used to reveal the element specified by `LoadingElementI`
`UpdateTargetId`	Sets the ID of the HTML element into which the content retrieved from the server will be inserted
`Url`	Sets the URL that will be requested from the server

In the listing, I have set the `UpdateTargetId` property to `tableBody`. This is the `id` I assigned to the HTML element in the view in Listing 23-10. When the user clicks the `Submit` button, an asynchronous request w the `GetPeopleData` action method and the HTML fragment that is returned is used to replace the existing elem `tbody`.

eBooks-IT.org request

See the "Working with Ajax Callbacks" section later in this chapter for details.

That's all there is to it: I replace the `Html.BeginForm` method with `Ajax.BeginForm` and ensure tha target for the new content. Everything else happens automatically and the result is an asynchronous form.

It can be hard to detect when you are testing with the browser and the server on the same machine, but you can t browser is making Ajax requests for fragments of HTML by using the browser F12 tools. These tools allow you to r network requests that the browser makes and in Figure 23-3, you can see the Internet Explorer tools showing a call t `GetPeopleData` action method.

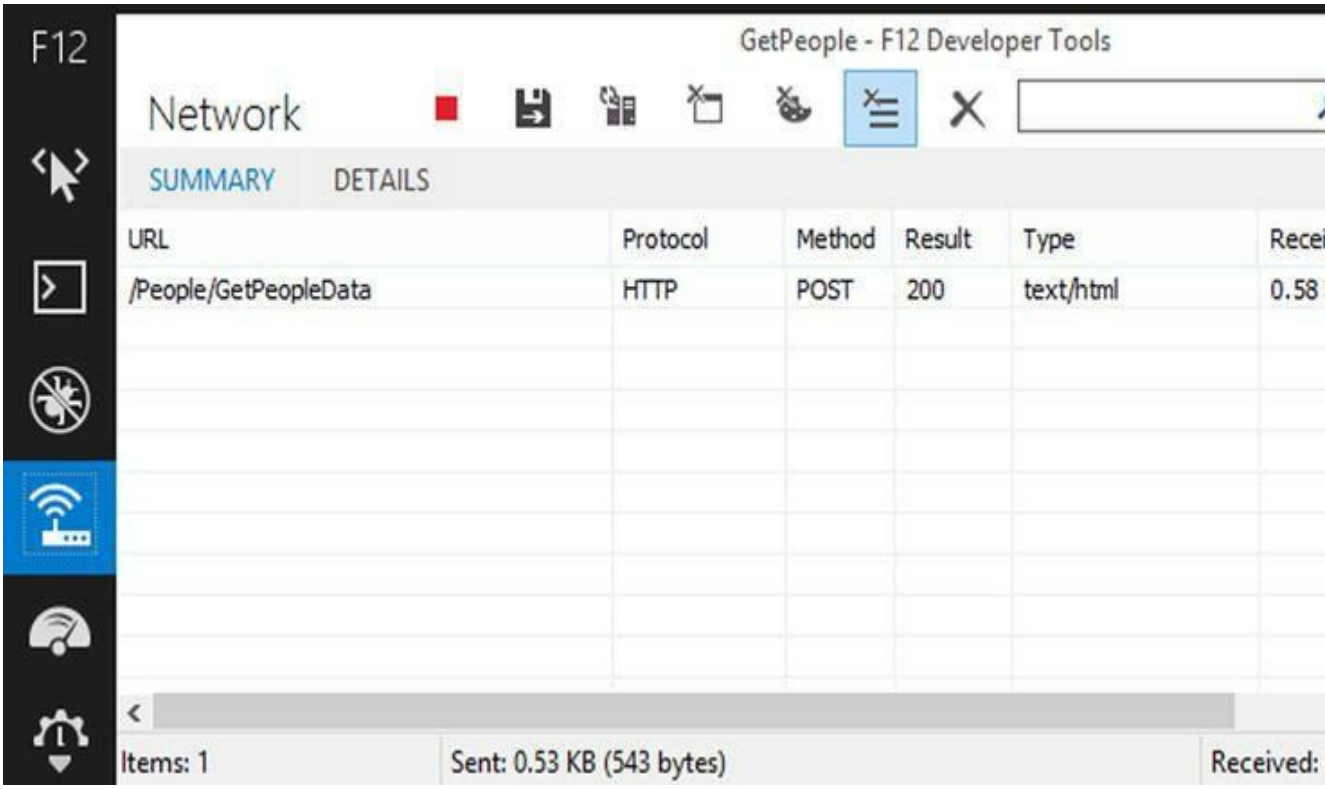


Figure 23-3. Confirming that Ajax requests are being made

Understanding How Unobtrusive Ajax Works

When I call the `Ajax.BeginForm` helper method, the options that I specify using the `AjaxOptions` obje transformed into attributes applied to the `form` element. The view in Listing 23-10 produces the following `form` e

```
...
<form        action="/People/GetPeopleData"data-ajax="true"        da
mode="replace"
    data-ajax-update="#tableBody" id="form0" method="post">
...
```

When the HTML page rendered from the `GetPeople.cshtml` view is loaded by the browser, the JavaScri `jquery.unobtrusive-ajax.js` library scans the HTML elements and identifies the Ajax form by looki elements that have a `data-ajax` attribute with a value of `true`.

The other attributes whose names start with `data-ajax` contain the values I specified using the `AjaxOpti` These configuration options are used to configure jQuery, which has built-in support for managing Ajax requests.

Tip You don't have to use the MVC Framework support for unobtrusive Ajax. There are plenty of alternatives av including using jQuery directly. That said, pick a technique and stick to it. I recommend against mixing the MVC Fra unobtrusive Ajax support with other techniques and libraries in the same view, as there can be some unfortunate inte as duplicated or dropped Ajax requests.

eBooks-IT.org

I can fine-tune the behavior of the Ajax requests by setting values for the properties of the `AjaxOptions` object the `Ajax.BeginForm` helper method. In the following sections, I explain what each of these options does and be useful.

Ensuring Graceful Degradation

When I set up the Ajax-enabled form in Listing 23-10, I passed in the name of the action method that I wanted to be asynchronously. In the example, this was the `GetPeopleData` action, which generates a partial view containin of HTML.

One problem with this approach is that it doesn't work well if the user has disabled JavaScript (or is using a brow doesn't support it). In such cases, when the user submits the form, the browser display discards the current HTML replaces it with the fragment returned by the target action method. The effect can be seen in Figure 23-4.

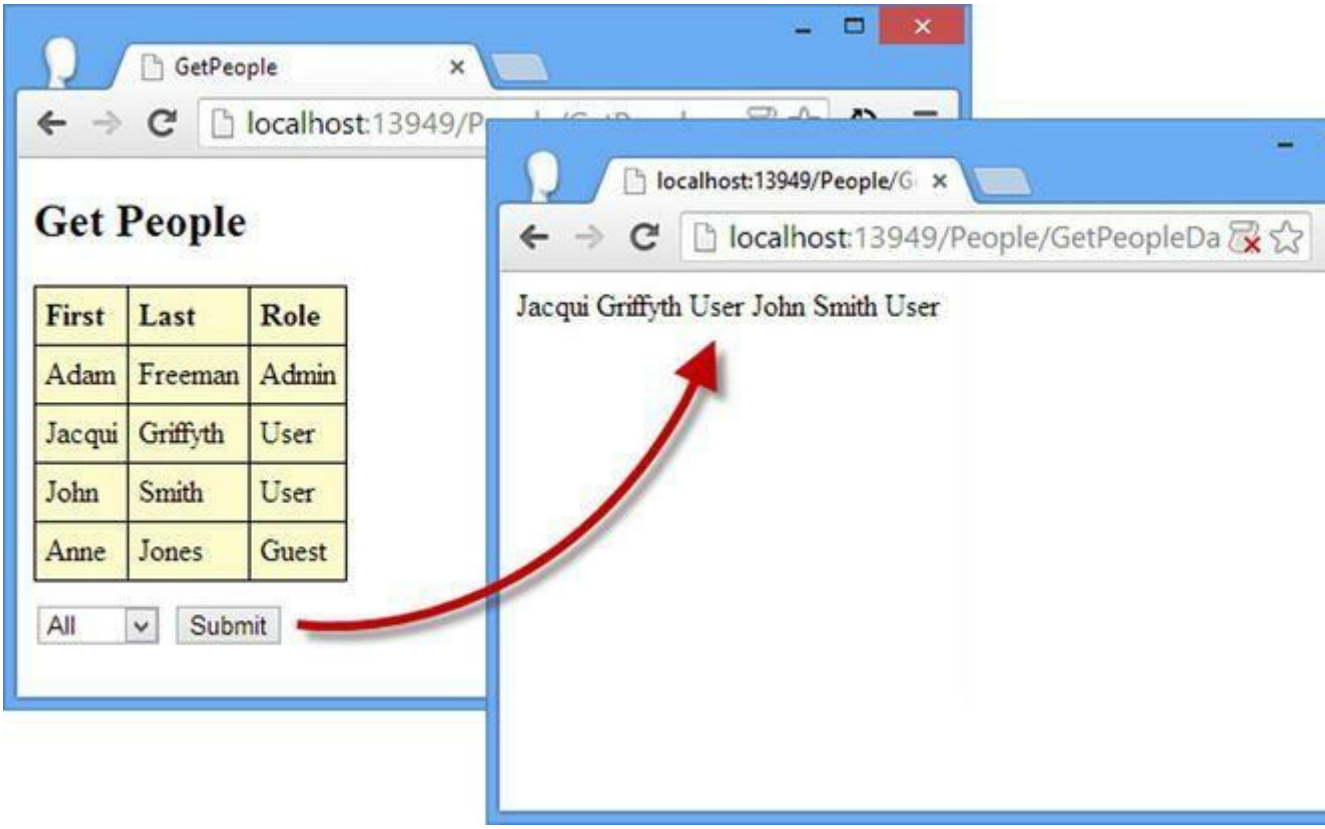


Figure 23-4. The effect of using the Ajax.BeginForm helper without browser JavaScript support

Note I used Google Chrome for this figure because it makes it easy to toggle JavaScript.

The simplest way to address this problem is to use the `AjaxOptions.Url` property to specify the target U asynchronous request rather than specifying the action name as an argument to the `Ajax.BeginForm` method Listing 23-11.

Listing 23-11. Ensuring Gracefully Degrading Forms in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
```

```
    };

}
<h2>Get People</h2>
<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

I have used the `Url.Action` helper method to create a URL that will invoke the `GetPeopleData` action the version of the `Ajax.BeginForm` method that takes only an `AjaxOptions` parameter. This has the eff creating a `form` element that posts back to the originating action method if JavaScript isn't enabled, like this:

```
...
<form action="/People/GetPeople" data-ajax="true" data-ajax-mode="
        data-ajax-update="#tableBody"data-ajax-url="/People/GetPeo
id="form0"
    method="post">
...
```

If JavaScript is enabled, then the unobtrusive Ajax library will take the target URL from the `data-ajax-url` which refers to the child action. If JavaScript is disabled, then the browser will use the regular form posting techniqu takes the target URL from the `action` attribute, which points back at the action method that will generate a comp page.

Caution You might be wondering why I am making such a big deal about users who have disabled JavaScript. A does that? In fact, it is prevalent in two groups of users. The first group consists of those users who take their IT se seriously and disable anything that could be used as the basis for an attack, something that JavaScript has been know years. The second group is users in large corporations, which apply incredibly restrictive policies in the name of IT s (although, in my experience, corporate PCs are so poorly set up that security is nonexistent and the restrictions just a users). You can ignore graceful degradation if you feel you can ignore IT security experts and people who work for companies. But, since these can be affluent and tech-savvy users, I always take the time to make sure I support then

Providing the User with Feedback While Making an Ajax Request

One drawback of using Ajax is that it isn't obvious to the user that something is happening, because the request to th made in the background. I can inform the user that a request is being performed by using the `AjaxOptions.LoadingElementId` and `AjaxOptions.LoadingElementDuration` p Listing 23-12 shows how I have applied these properties in the `GetPeople.cshtml` view file.

Listing 23-12. Giving Feedback to the User in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
```

eBooks-IT.org

```
    Layout = "/Views/Shared/_Layout.cshtml",
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000
    };

}
<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

The `AjaxOptions.LoadingElementId` property specifies the `id` attribute value of a hidden HTML
will be shown to the user while an Ajax request is performed. To demonstrate this feature, I added a `div` element to
I hid from the user by setting the CSS `display` property to `none`. I gave this `div` element an `id` attribute of
and used this `id` as the value for the `LoadingElementId` property and the unobtrusive Ajax feature will disp
element to the user for the duration of the request, as shown in Figure 23-5. The `LoadingElementDurati`
specifies the duration of the animation that is used to reveal the `loading` element to the user. I specified a value o
which denotes one second.

eBooks-IT.org

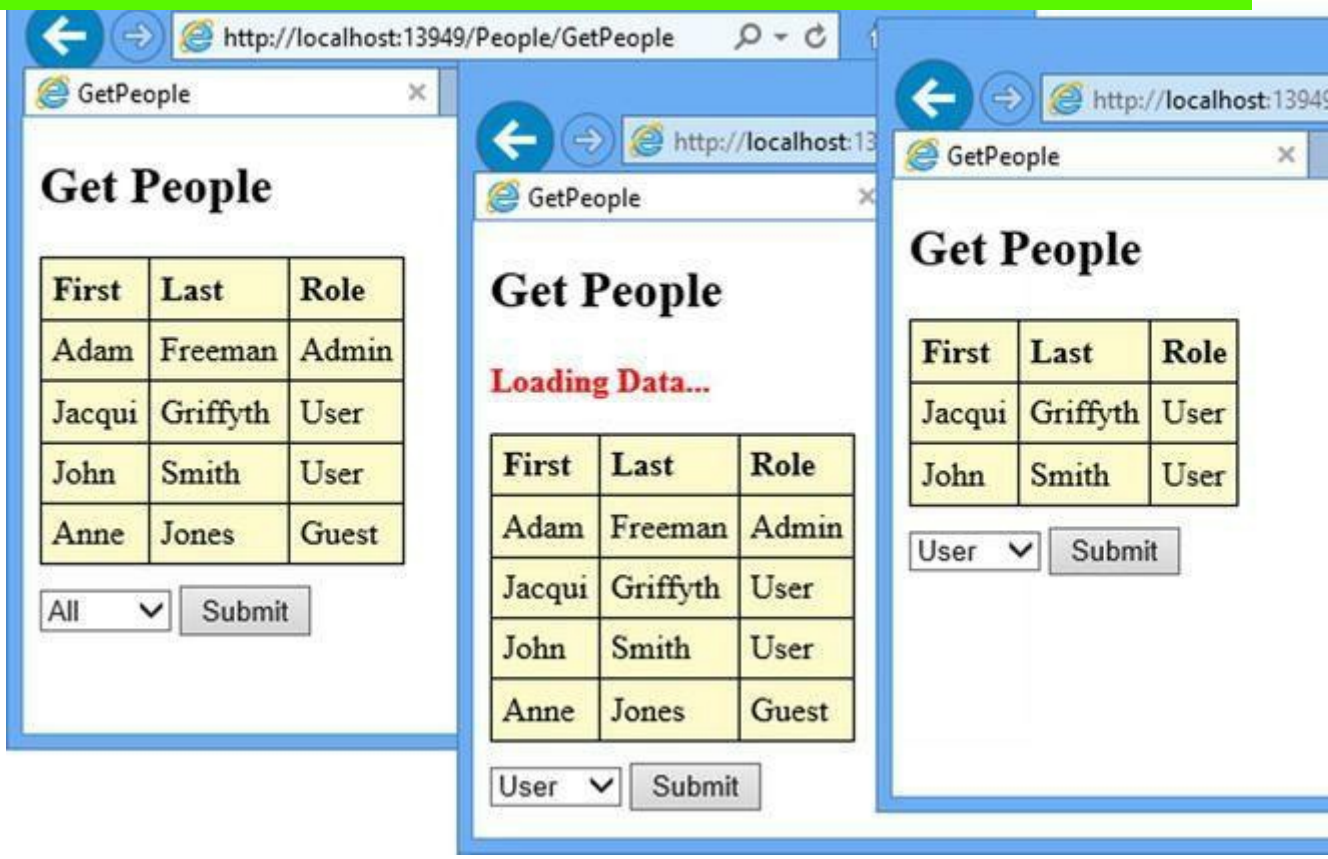


Figure 23-5. Providing the user with feedback during an Ajax request

Prompting the User Before Making a Request

The `AjaxOptions.Confirm` property lets me specify a message that will be used to prompt the user before asynchronous request. The user can elect to proceed with or cancel the request. Listing 23-13 shows how I have ap property to the `GetPeople.cshtml` file.

Listing 23-13. Promoting the User Before Making an Asynchronous Request in the GetPeople.cshtml File

```
...
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}
...
```

With this addition, the user is prompted each time they submit the form, as shown in Figure 23-6. The user is pro *every* request, which means that this feature should be used sparingly to avoid irritating the user.

eBooks-IT.org

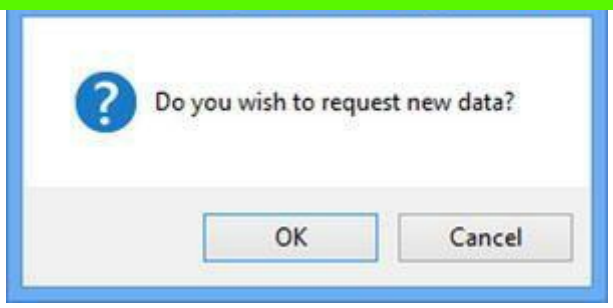


Figure 23-6. Prompting the user before making a request

Creating Ajax Links

In addition to forms, unobtrusive Ajax can be used to create a elements that will be followed asynchronously. The m
this is similar to the way that Ajax forms work. You can see how I have added Ajax links to the GetPeople.cs
in Listing 23-14.

Listing 23-14. Creating Ajax-Enabled Links in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}
<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

eBooks-IT.org

```
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeopleData",
                new {selectedRole = role},
                new AjaxOptions {UpdateTargetId = "tableBody"})
        </div>
    }
</div>
```

I have used a `foreach` loop to call the `Ajax.ActionLink` helper for each of the values defined by the R enumeration, creating a set of Ajax-enabled `a` elements. The `a` elements that are produced have the same kind of da you saw when working with forms, like this:

```
...
<a data-ajax="true" data-ajax-mode="replace" data-ajax-update="#ta
    href="/People/GetPeopleData?selectedRole=Guest">Guest</a>
...
```

The routing configuration in the project does not have an entry for the `selectedRole` variable, so the URL t generated for the `href` attribute specifies the role that the link represents using the query string component of the U

You can see the links I added to the view in Figure 23-7. Clicking one of these links will call the `GetPeopleD` method and replace the contents of the `tbody` element with the HTML fragment that is returned. This creates the s filtering the data that I achieved using the Ajax-enabled form earlier in the chapter.

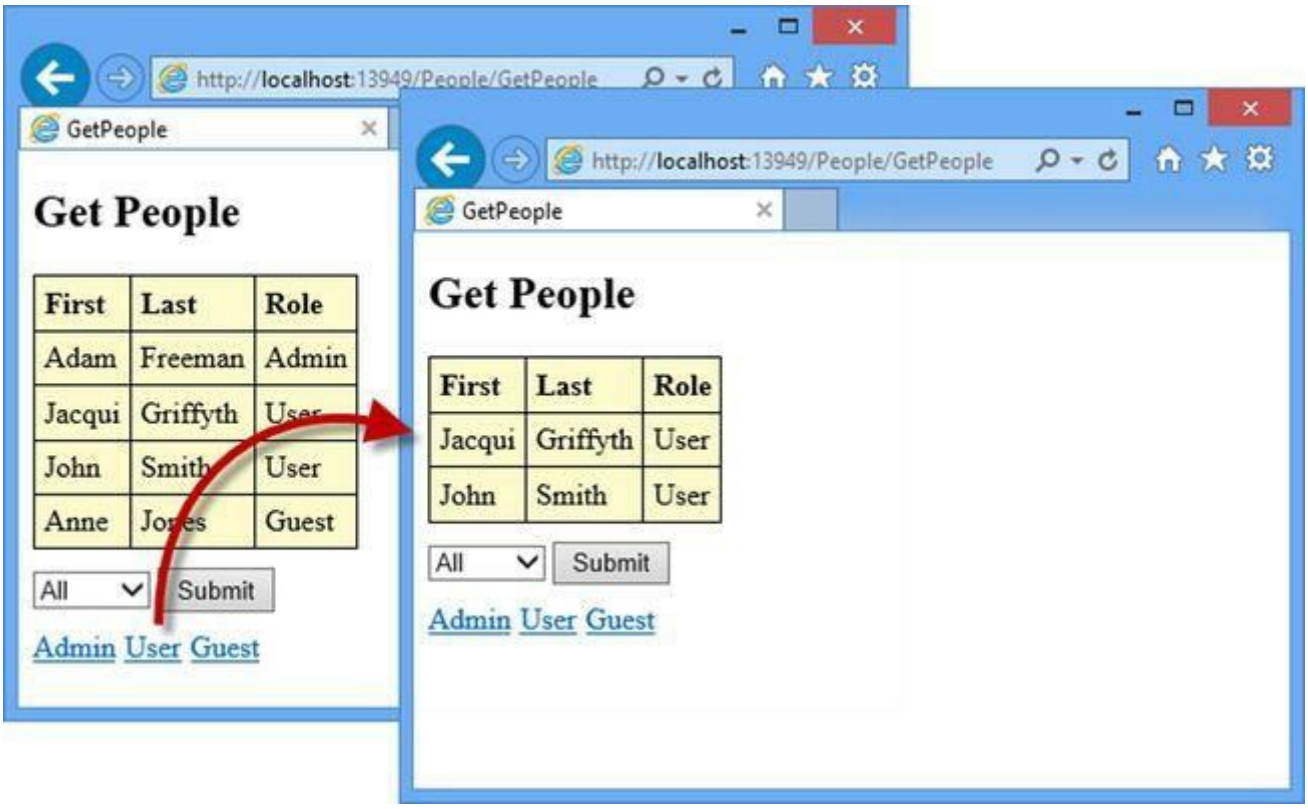


Figure 23-7. Adding Ajax-enabled links to a view

Tip You may have to clear your browser history to see the changes for this example.

Ensuring Graceful Degradation for Links

I face the same problem with the Ajax-enabled links as I did with the forms. When there is no JavaScript support on clicking one of the links will just display the HTML fragment that the `GetPeopleData` action method generates

eBooks-IT.org

have specified the GetPeople action to the Ajax.ActionLink helper method, as shown in Listing 23-15

Listing 23-15. Creating Graceful Ajax-Enabled Links in the GetPeople.cshtml File

```
...
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new {selectedRole = role},
                new AjaxOptions {
                    UpdateTargetId = "tableBody",
                     Url = Url.Action("GetPeopleData", new {select
role})
                })
        </div>
    }
</div>
...
```

This is why I created a new `AjaxOptions` object for each of the links rather than using the one I created in t code block for the `form` element. Independent `AjaxOptions` allow me to specify a different value for the `Ur` for each link and support graceful degradation for non-JavaScript browsers.

Working with Ajax Callbacks

The `AjaxOptions` class defines a set of properties that specify JavaScript functions that will be called at variou the Ajax request life cycle. These properties are described in Table 23-4.

Table 23-4. AjaxOptions Callback Properties

Property	jQuery Event	Description
`OnBegin`	`beforeSend`	Called immediately prior to the request being sent
`OnComplete`	`complete`	Called if the request is successful
`OnFailure`	`error`	Called if the request fails
`OnSuccess`	`success`	Called when the request has completed, irrespective of whether the request succeeded or failed

Each of the `AjaxOptions` callback properties correlates to an Ajax event supported by the jQuery library. I h jQuery events in Table 23-4 for those readers who have used jQuery before. You can get details on each of these eve parameters that will be passed to your functions at http://api.jquery.com/jQuery.ajax or in n *jQuery 2.0*, also published by Apress.

In Listing 23-16, you can see how I have used a `script` element to define some basic JavaScript functions tha on the progress of the Ajax request and use the properties shown in Table 23-4 to specify the functions as handlers f events.

Listing 23-16. Using the Ajax Callbacks in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
```

```
            UpdateTargetId = "tableBody",
            Url = Url.Action("GetPeopleData"),
            LoadingElementId = "loading",
            LoadingElementDuration = 1000,
            Confirm = "Do you wish to request new data?"
        };
    }

    <script type="text/javascript">
        function OnBegin() {
            alert("This is the OnBegin Callback");
        }

        function OnSuccess(data) {
            alert("This is the OnSuccessCallback: " + data);
        }

        function OnFailure(request, error) {
            alert("This is the OnFailure Callback:" + error);
        }

        function OnComplete(request, status) {
            alert("This is the OnComplete Callback: " + status);
        }
    </script>

    <h2>Get People</h2>

    <div id="loading" class="load" style="display:none">
        <p>Loading Data...</p>
    </div>

    <table>
        <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
        <tbody id="tableBody">
            @Html.Action("GetPeopleData", new {selectedRole = Model })
        </tbody>
    </table>

    @using (Ajax.BeginForm(ajaxOpts)) {
        <div>
            @Html.DropDownList("selectedRole", new SelectList(
                new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
            <button type="submit">Submit</button>
        </div>
    }

    <div>
        @foreach (string role in Enum.GetNames(typeof(Role))) {
            <div class="ajaxLink">
                @Ajax.ActionLink(role, "GetPeople",
                    new {selectedRole = role},
                    new AjaxOptions {
                        UpdateTargetId = "tableBody",
```

eBooks-IT.org

```
                                    OnBegin = "OnBegin",
                                    OnFailure = "OnFailure",
                                    OnSuccess = "OnSuccess",
                                    OnComplete = "OnComplete"
                                })
                        </div>
                    }
</div>
```

I have defined four functions, one for each of the callbacks. For this example, I have kept things simple and simpl message to the user in each of the functions. With these changes, clicking one of the links will display a sequence of report on the progress of the Ajax request, as shown in Figure 23-8.

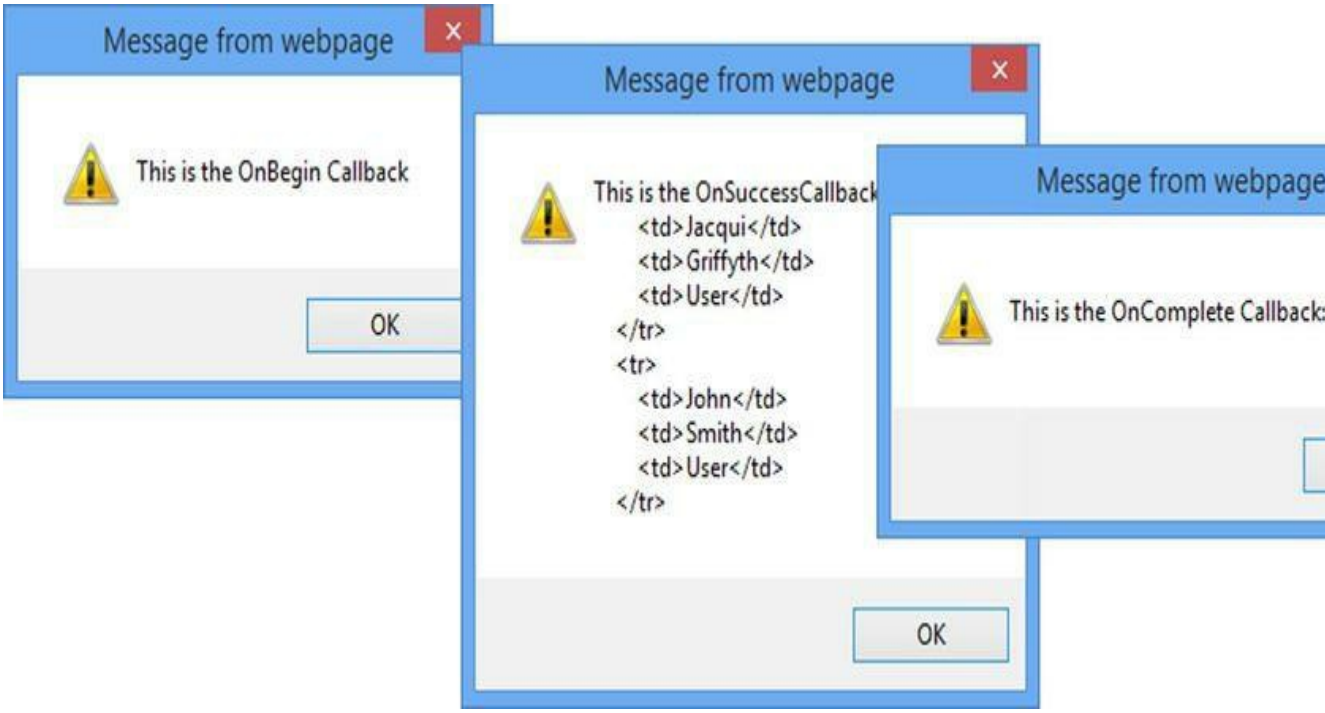


Figure 23-8. The series of dialog boxes shown in response to the Ajax callbacks

Displaying dialog boxes to the user for each callback isn't the most useful thing to do with the Ajax callbacks, but demonstrates the sequence in which they are called. These JavaScript functions can be used for any purpose: manipu HTML DOM, trigger additional requests, and so forth. One of the most useful things to do with the callbacks is hand which I describe in the next section.

Working with JSON

In the Ajax examples so far, the server has rendered fragments of HTML and sent them to the browser. This is a per acceptable technique, but it is verbose (because the server is sending the HTML elements along with the data) and it can be done with the data at the browser.

One way to address both of these issues is to use the *JavaScript Object Notation* (JSON) format, which is a langu independent way of expressing data. It emerged from the JavaScript language, but has long since taken on a life of it widely used. In this section, I'll show you how to create an action method that returns JSON data, as well as how to data in the browser.

Tip In Chapter 27, I describe the Web API feature, which is an alternative approach for creating Web services.

Adding JSON Support to the Controller

eBooks-IT.org

have ac

action method to the People controller in Listing 23-17.

Listing 23-17. An Action Method That Returns JSON Data in the PeopleController.cs File

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

namespace HelperMethods.Controllers {
    public class PeopleController : Controller {
        private Person[] personData = {
             new Person {FirstName = "Adam", LastName = "Freeman"
Role.Admin},
             new Person {FirstName = "Jacqui", LastName = "Griffyt
= Role.User},
              new Person {FirstName = "John", LastName = "Smith",
Role.User},
              new Person {FirstName = "Anne", LastName = "Jones",
Role.Guest}
        };

        public ActionResult Index() {
            return View();
        }

        private IEnumerable<Person> GetData(string selectedRole) {
            IEnumerable<Person> data = personData;
            if (selectedRole != "All") {
                        Role selected = (Role)Enum.Parse(typeo
selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            return data;
        }

        public JsonResult GetPeopleDataJson(string selectedRole =
            IEnumerable<Person> data = GetData(selectedRole);
            return Json(data, JsonRequestBehavior.AllowGet);
        }

            public PartialViewResult GetPeopleData(string selecte
"All") {
            return PartialView(GetData(selectedRole));
        }

        public ActionResult GetPeople(string selectedRole = "All")
            return View((object)selectedRole);
        }
    }
}
```

eBooks-IT.org

is a common (and private) GetData method that is responsible for performing the filtering.

I have added a new action method called GetPeopleDataJson, which returns a JsonResult object. special kind of ActionResult that tells the view engine that I want to return JSON data to the client, rather th (You can learn more about the ActionResult class and the role it plays in the MVC Framework in Chapter 17

I create a JsonResult by calling the Json method in the action method, passing in the data that I want con JSON format, like this:

```
...
return Json(data, JsonRequestBehavior.AllowGet);
...
```

In this case, I have also passed in the AllowGet value from the JsonRequestBehavior enumeration JSON data will only be sent in response to POST requests, but by passing this value as a parameter to the Json m the MVC Framework to respond to GET requests as well.

Caution You should only use JsonRequestBehavior.AllowGet if the data you are returning is n Due to a security issue in many Web browsers, it's possible for third-party sites to intercept JSON data that you retu response to a GET request, which is why JsonResult will not respond to GET requests by default. In most ca be able to use POST requests to retrieve the JSON data instead, avoiding the problem. For more information, see http://haacked.com/archive/2009/06/25/json-hijacking.aspx.

Processing JSON in the Browser

To process the JSON I retrieve from the MVC Framework application server, I specify a JavaScript function using t OnSuccess callback property in the AjaxOptions class. In Listing 23-18, you can see how I have updated GetPeople.cshtml view file to remove the handler functions I defined in the last section and use the OnSu callback to process the JSON data.

Listing 23-18. Working with JSON Data in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}

<script type="text/javascript">
    function processData(data) {
        var target = $("#tableBody");
        target.empty();
        for (var i = 0; i < data.length; i++) {
            var person = data[i];
            target.append("<tr><td>" + person.FirstName + "</td><t
                + person.LastName + "</td><td>" + person.Role
</tr>");
        }
```

eBooks-IT.org

```
</script>

<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}

<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new {selectedRole = role},
                new AjaxOptions {
                                Url = Url.Action("GetPeopleDataJso
{selectedRole = role}),
                    OnSuccess = "processData"
                })
        </div>
    }
</div>
```

I have defined a new function called `processData`, which contains some basic jQuery code that processes t objects and uses them to create the `tr` and `td` elements needed to populate the `table`.

Tip I don't go into jQuery in this book because it is a topic in and of itself. I love jQuery, though, and if you wan more about it, then I have written *Pro jQuery* 2.0 (Apress, 2013).

Notice that I have removed the value for the `UpdateTargetId` property from the `AjaxOptions` objec for the links. If you forget to do this, the unobtrusive Ajax feature will try and treat the JSON data it retrieves from t HTML. You can usually tell this is happening because the contents of the target element will be removed but not repl new data.

You can see the result of the switch to JSON by starting the application, navigating to the `/People/GetPe` and clicking one of the links. As Figure 23-9 shows, I don't get quite the right result. In particular, the information d `Role` column of the table isn't correct. I will explain why this happens and show you how to make it right in the n

eBooks-IT.org

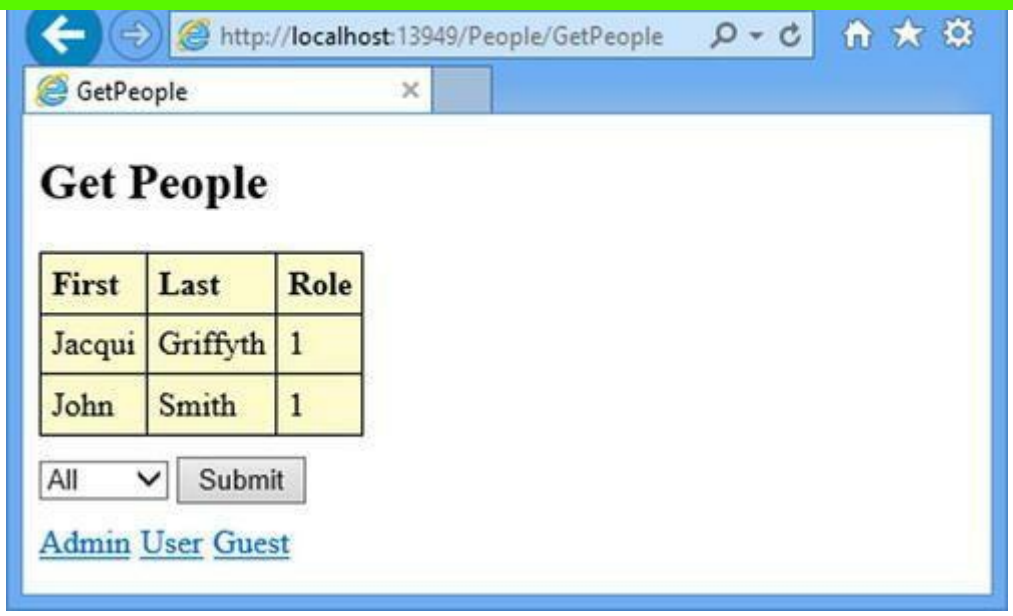


Figure 23-9. Working with JSON data instead of HTML fragments

Preparing Data for Encoding

When I called the `Json` method from within the `GetPeopleDataJson` action method, I left the MVC Fram figure out how to encode `People` objects in the JSON format. The MVC Framework doesn't have any special ins model types in an application, and so it makes a best-effort guess about what it needs to do. Here is how the MVC F expresses a single `Person` object in JSON:

```
...
{"PersonId":0,"FirstName":"Adam","LastName":"Freeman",
  "BirthDate":"\/Date(62135596800000)\/","HomeAddress":null,"IsApp
...
```

It looks like a bit of a mess, but the result is actually pretty clever—it just isn't quite what I need. First, all the prop by the `Person` class are represented in the JSON, even though I did not assign values to some of them in the `Pe` controller. In some cases, the default value for the type has been used (`false` is used for `IsApproved`, for ex in others `null` has been used (such as for `HomeAddress`). Some values are converted into a form that can be interpreted by JavaScript, such as the `BirthDate` property, but others are not handled as well, such as using `0` `Role` property rather than `Admin`.

VIEWING JSON DATA

It can be useful to see what JSON data your action methods return and the easiest way to do this is to enter a U targets the action in the browser, like this:

http://localhost:13949/People/GetPeopleDataJson?selectedRole=Gu

You can do this in pretty much any browser, but most will force you to save and open a text file before you car JSON content. I like to use the Google Chrome browser for this because it helpfully displays the JSON data in t browser window, which makes the process quicker and means you don't end up with dozens of open text file v also recommend Fiddler (www.fiddler2.com) , which is an excellent Web debugging proxy that allows right into the details of the data sent between the browser and the server.

The MVC Framework has made a good attempt, but I end up sending properties to the browser that I don't subse and the `Role` value isn't expressed in a useful way. This is a typical situation when relying on the default JSON enc some preparation of the data you want to send the client is usually required. In Listing 23-19, you can see how I hav `GetPeopleDataJson` action method in the `People` controller to prepare the data I pass to the `Json` me

```
...
public JsonResult GetPeopleDataJson(string selectedRole = "All") {
    var data = GetData(selectedRole).Select(p => new {
        FirstName = p.FirstName,
        LastName = p.LastName,
        Role = Enum.GetName(typeof(Role), p.Role)
    });
    return Json(data, JsonRequestBehavior.AllowGet);
}
...
```

I have used LINQ to create a sequence of new objects that contain just the `FirstName` and `LastName` pro the `Person` objects, along with the string representation of the `Role` value. The effect of this change is that I ge that contains just the properties I want, expressed in a way that is more useful to the jQuery code, like this:

```
...
{"FirstName":"Adam","LastName":"Freeman","Role":"Admin"}
...
```

Figure 23-10 shows the change in the output displayed in the browser. You can't tell the unused properties are not course, but you can see that the `Role` column contains the right values.

Figure 23-10. The effect of preparing the data objects for JSON encoding

Tip You may have to clear your browser history to see the changes for this example.

Detecting Ajax Requests in the Action Method

The `People` controller presently contains two action methods so that I can support requests for HTML and JSON usually how I build controllers, because I like lots of short and simple actions, but you don't have to work this way. Framework provides a simple way of detecting Ajax requests, which means that you can create a single action metho handles multiple data formats. In Listing 23-20, you can see how I have refactored the `Person` controller to conta action that handles both JSON and HTML.

Listing 23-20. Creating a Single Action Method in the PersonController.cs File

eBooks-IT.org

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

namespace HelperMethods.Controllers {
    public class PeopleController : Controller {
        private Person[] personData = {
             new Person {FirstName = "Adam", LastName = "Freeman"
Role.Admin},
             new Person {FirstName = "Jacqui", LastName = "Griffyt
= Role.User},
              new Person {FirstName = "John", LastName = "Smith",
Role.User},
              new Person {FirstName = "Anne", LastName = "Jones",
Role.Guest}
        };

        public ActionResult Index() {
            return View();
        }

        public ActionResult GetPeopleData(string selectedRole = "A
            IEnumerable<Person> data = personData;
            if (selectedRole != "All") {
                        Role selected = (Role)Enum.Parse(typeo
selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            if (Request.IsAjaxRequest()) {
                var formattedData = data.Select(p => new {
                    FirstName = p.FirstName,
                    LastName = p.LastName,
                    Role = Enum.GetName(typeof(Role), p.Role)
                });
                return Json(formattedData, JsonRequestBehavior.All
            } else {
                return PartialView(data);
            }
        }

        public ActionResult GetPeople(string selectedRole = "All")
            return View((object)selectedRole);
        }
    }
}
```

I used the `Request.IsAjaxRequest` method to detect Ajax requests and deliver the JSON format if the `true`. There are a couple of limitations that you should be aware of before you follow this approach. First, the `IsAjaxRequest` methods returns `true` if the browser has included the `X-Requested-With` header i and set the value to `XMLHttpRequest`. This is a widely used convention, but it isn't universal and so you shou whether your users are likely to make requests that require JSON data without setting this header.

The second limitation is that it assumes that all Ajax requests require JSON data. Your application may be better se

eBooks-IT.org

reason I tend to define separate action methods for data formats.

I also need to make two changes to the `GetPeople.cshtml` view to support the single action method, as s Listing 23-21.

Listing 23-21. Supporting a Single Data Action Method in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        OnSuccess = "processData"
    };
}

<script type="text/javascript">
    function processData(data) {
        var target = $("#tableBody");
        target.empty();
        for (var i = 0; i < data.length; i++) {
            var person = data[i];
            target.append("<tr><td>" + person.FirstName + "</td><t
                + person.LastName + "</td><td>" + person.Role
                + "</td></tr>");
        }
    }
</script>

<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thea
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}

<div>
```

eBooks-IT.org

```
        <div class= ajaxLink >
            @Ajax.ActionLink(role, "GetPeople",
                new {selectedRole = role},
                new AjaxOptions {
                     Url = Url.Action("GetPeopleData", new {select
role}),
                    OnSuccess = "processData"
                })
        </div>
    }
</div>
```

The first change is to the `AjaxOptions` object I use for the Ajax-enabled `form`. Because I am no longer ab an HTML fragment via an Ajax request, I had to use the same `processData` function to handle the JSON serve that I created for the Ajax-enabled links. The second change is to the value of the `Url` property for the `AjaxOpt` objects created for the links. The `GetPeopleDataJson` action no longer exists and I target the `GetPeopl` action instead.

Summary

In this chapter, I looked at the MVC Framework's unobtrusive Ajax feature, taking advantage of the functionality of library in a simple and elegant way and without needing to add lots of code to views. If you are able to work with HT fragments, then you don't need to add any JavaScript code to your views at all. But I like working with JSON, whic tend to need small JavaScript functions that use jQuery to process the data and generate the HTML elements I requir chapter, I look at one of the most interesting and useful aspects of the MVC Framework: model binding.

eBooks-IT.org

Model Binding

Model binding is the process of creating .NET objects using the data sent by the browser in an HTTP request. I hav on the model binding process each time I have defined an action method that takes a parameter. The parameter objec through model binding from the data in the request. In this chapter, I'll show you how the model binding system wo demonstrate the techniques required to customize it for advanced use. Table 24-1 provides the summary for this cha

Table 24-1. *Chapter Summary*

Problem	Solution
Bind to a simple type or a collection	Add a parameter to an action method.
Provide a fallback value for model binding	Use a nullable type for the action method parameter or use a default value.
Bind to a complex type	Ensure that the HTML generated by your views contains nested property values.
Override the default approach to locating nested complex types	Use the `Prefix` property `Bind` attribute applied to the action method parameter.
Selectively bind properties	Use the `Include` or `Exclude` properties of the `Bind` attribute, applied either to the ac method parameter or to the model class.
Manually invoke model binding	Call the `UpdateModel` or `TryUpdateModel` methods.
Create a custom value provider	Implement the `IValueProvider` interface.
Create a custom model binder	Implement the `IModelBinder` interface.

Preparing the Example Project

I have created a new Visual Studio project called `MvcModels` using the `Empty` template option and checking th include the core MVC folders and references. I will be using the same model class that you have seen in previous ch create a new class file called `Person.cs` in the `Models` folder and ensure that the contents match Listing 24-

Listing 24-1. The Contents of the Person.cs File

```
using System;

namespace MvcModels.Models {

    public class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }
```

eBooks-IT.org

```
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }

    public enum Role {
        Admin,
        User,
        Guest
    }
}
```

I have also defined a `Home` controller, as shown in Listing 24-2. This controller defines a collection of sample `Pe` objects and defines the `Index` action, which allows me to select a single `Person` by the value of the `Person`

Listing 24-2. The Contents of the HomeController.cs File

```
using System.Linq;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private Person[] personData = {
                new Person {PersonId = 1, FirstName = "Adam", La
"Freeman",
                Role = Role.Admin},
               new Person {PersonId = 2, FirstName = "Jacqui", La
"Griffyth",
                Role = Role.User},
                new Person {PersonId = 3, FirstName = "John", La
"Smith",
                Role = Role.User},
                new Person {PersonId = 4, FirstName = "Anne", La
"Jones",
                Role = Role.Guest}
        };

        public ActionResult Index(int id) {
                   Person dataItem = personData.Where(p => p.Pers
id).First();
            return View(dataItem);
        }
    }
}
```

I have created a view file called `/Views/Home/Index.cshtml` to support the action method. You can contents of this file in Listing 24-3. I have used the templated display helper to show some of the property values of view model.

Listing 24-3. The Contents of the /Views/Home/Index.cshtml File

eBooks-IT.org

```
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Person</h2>
<div><label>ID:</label>@Html.DisplayFor(m => m.PersonId)</div>
<div><label>First Name:</label>@Html.DisplayFor(m => m.FirstName)<
<div><label>Last Name:</label>@Html.DisplayFor(m => m.LastName)</d
<div><label>Role:</label>@Html.DisplayFor(m => m.Role)</div>
```

Finally, I created the `Views/Shared` folder and added a layout called `_Layout.cshtml` to it, the cont can be seen in Listing 24-4.

Listing 24-4. The Contents of the _Layout.cshtml File

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style>
          label { display: inline-block; width: 100px; font-weigh
margin: 5px; }
        form label { float: left; }
        input.text-box { float: left; margin: 5px; }
        button[type=submit] { margin-top: 5px; float: left; clear:
        form div { clear: both; }
    </style>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Understanding Model Binding

Model binding is an elegant bridge between the HTTP request and the C# methods that define actions. Most MVC Fr applications rely on model binding to some extent, including the simple example application that I created in the previ To see model binding at work, start the application and navigate to `/Home/Index/1`. The result is illustrated in

eBooks-IT.org

Figure 24-1. A simple demonstration of model binding

The URL contained the value of the `PersonId` property of the `Person` object I wanted to view, like this:

```
/Home/Index/1
```

The MVC Framework translated that part of the URL and used it as the argument when it called on the `Index` m `Home` controller class to service the request:

```
...
public ActionResult Index(int id) {
...
```

The process by which the URL segment was converted into the `int` method argument is an example of model bi sections that follow, I show the process that this simple demonstration initiated, and then move on to explain some o complex model binding features. The process that leads to model binding begins when the request is received and pr the routing engine. I have not changed the routing configuration for the example application, and the default route tha Studio adds to the `/App_Start/RouteConfig.cs` file was used to process the request. As a reminder, y the default route in Listing 24-5.

Listing 24-5. The Contents of the RouteConfig.cs File

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

namespace MvcModels {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes)
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Ind
                    id = UrlParameter.Optional }
```

eBooks-IT.org

```
        }
    }
}
```

I described how routes are defined and how they work in detail in Chapters 15 and 16, so I am not going to repeat information here. For the model binding process, the important part is the `id` optional segment variable. When I nav `/Home/Index/1` URL, the last segment of the URL, which specifies the `Person` object I am interested in, is the `id` routing variable.

The action invoker, which I introduced in Chapter 17, used the routing information to figure out that the `Index` method was required to service the request, but it couldn't call the `Index` method until it had some useful values f argument.

The default action invoker, `ControllerActionInvoker`, (introduced in Chapter 17), relies on *model b* generate the data objects that are required to invoke the action. Model binders are defined by the `IModelBinde` which is shown in Listing 24-6. I'll come back to this interface later in the chapter when I show you how to create a model binder.

Listing 24-6. The IModelBinder Interface from the MVC Framework

```
namespace System.Web.Mvc {

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

There can be multiple model binders in an MVC application, and each binder can be responsible for binding one or types. When the action invoker needs to call an action method, it looks at the parameters that the method defines and responsible model binder for the type of each one.

For the example in this section, the action invoker would examine the `Index` method and find that it has one `in` It would then locate the binder responsible for `int` values and call its `BindModel` method.

The model binder is responsible for providing an `int` value that can be used to call the `Index` method. This us transforming some element of the request data (such as form or query string values), but the MVC Framework does limits on how the data is obtained.

I will show you some examples of custom binders later in this chapter. I will also show you some of the features `ModelBindingContext` class, which is passed to the `IModelBinder.BindModel` method.

Using the Default Model Binder

Although an application can define custom model binders, most just rely on the built-in binder class, `DefaultModelBinder`. This is the binder that is used by the action invoker when it cannot find a custom bi the type. By default, this model binder searches four locations, shown in Table 24-2, for data matching the name of t being bound.

Table 24-2. *The Order in Which the DefaultModelBinder Class Looks for Parameter Data*

Source	Description
`Request.Form`	Values provided by the user in HTML `form` elements
`RouteData.Values`	The values obtained using the application routes
`Request.QueryString`	Data included in the query string portion of the request URL
`Request.Files`	Files that have been uploaded as part of the request (see Chapter 12 for a demonstration of uploading files)

The locations are searched in order. For example, in my simple example, the `DefaultModelBinder` looks

1. `Request.Form["id"]`
2. `RouteData.Values["id"]`
3. `Request.QueryString["id"]`
4. `Request.Files["id"]`

The search is stopped as soon as a value is found. In the example, the form data is searched without success, but variable is found with the right name. This means that the query string and the names of uploaded files will not be se

Tip When replying on the default model binder, it is important that the parameters for your action method match property you are looking for. My example application works because the name of the action method parameter corre name of a routing variable. If I had named the action method parameter `personId`, for example, the default mod would not have been able to locate a matching data value and my request would have failed.

Binding to Simple Types

When dealing with simple parameter types, the `DefaultModelBinder` tries to convert the string value, whic obtained from the request data into the parameter type using the `System.ComponentModel.TypeDesc` class. If the value cannot be converted (for example, if I have supplied a value of `apple` for a parameter that requi value), then the `DefaultModelBinder` won't be able to bind to the model. You can see the problem that thi starting the example application and navigating to the URL `/Home/Index/apple`. Figure 24-2 illustrates the the server.

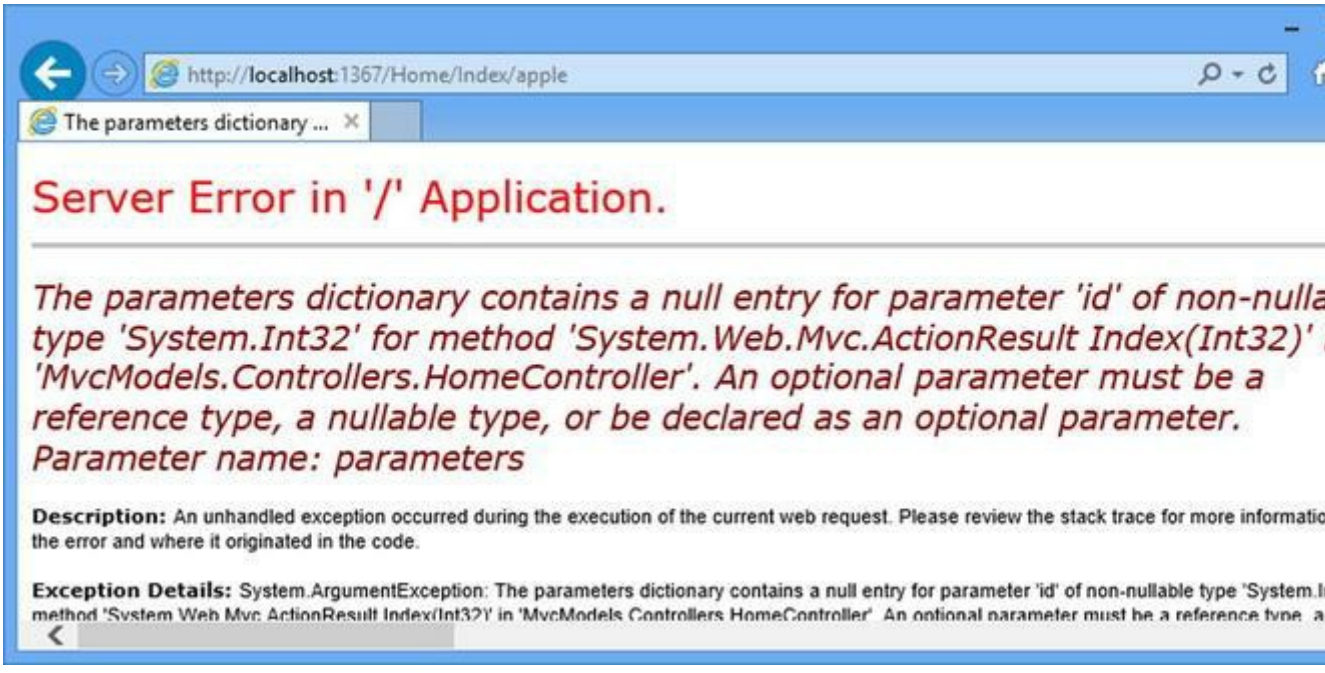


Figure 24-2. An error processing a model property

The default model binder is a little dogged. It sees that an `int` value is required and it tries to convert the value I the URL, `apple`, into an `int`, which causes the error shown by the figure. I can make things easier for the mode using a `nullable` type, which provides a fallback position. Instead of requiring a numeric value, a nullable `int` gives the model binder the choice of setting the action method argument to `null` when invoking the action. You ca have applied a nullable type to the `Index` action in Listing 24-7.

Listing 24-7. Using a Nullable Type for an Action Method Parameter in the HomeController.cs File

```
...
```

eBooks-IT.org

```
    Person dataItem = personData.Where(p => p.PersonId == id).Firs
    return View(dataItem);
}
...
```

If you run the application and navigate to `/Home/Index/apple`, you can see that I only changed, rather th the problem, as shown by Figure 24-3.

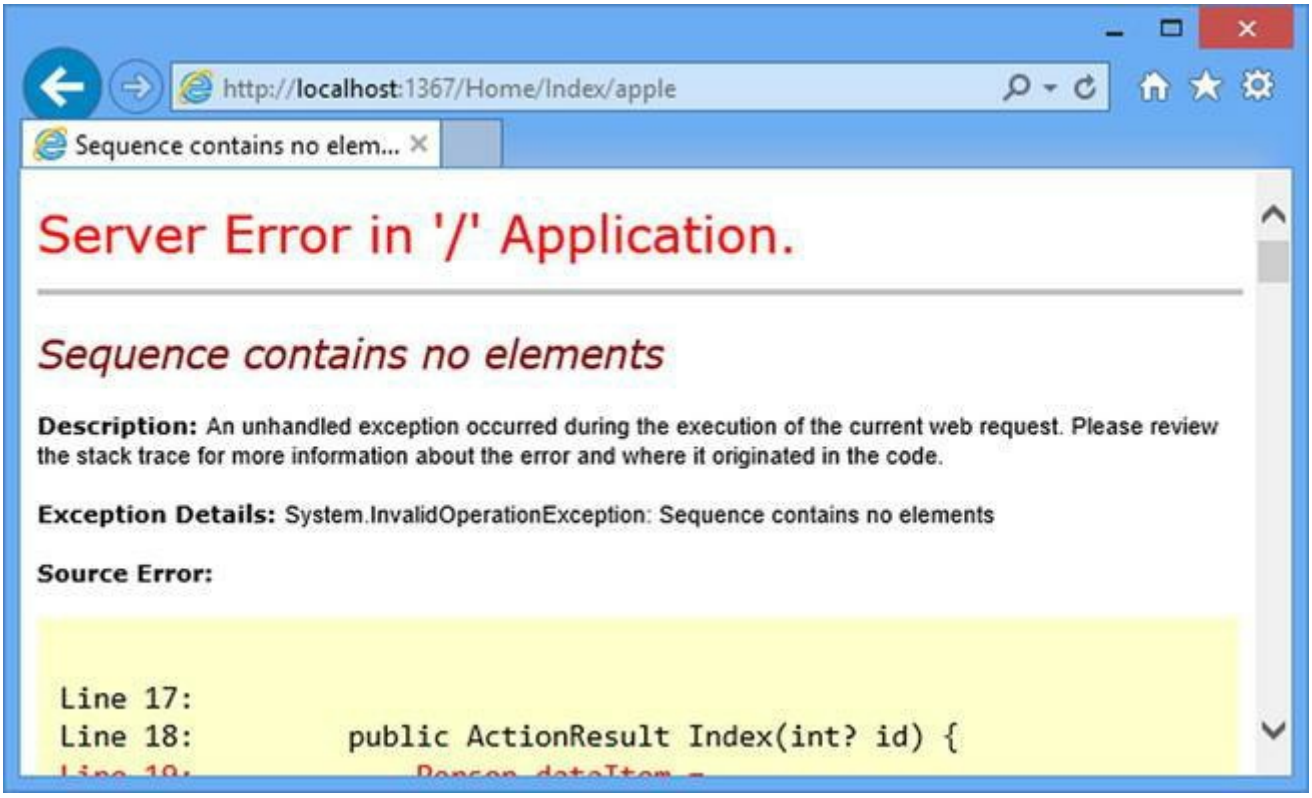


Figure 24-3. A request for a null value

The model binder is able to use `null` as the value for the `id` argument to the `Index` method, but the code ins method doesn't check for `null` values. I could fix that by explicitly checking for `null` values, but I can also set value for the parameter that will be used instead of `null`. You can see how I have applied a default parameter value `Index` action method in Listing 24-8.

Listing 24-8. Applying a Default Parameter Value in the HomeController.cs File

```
...
public ActionResult Index(int id = 1) {
    Person dataItem = personData.Where(p => p.PersonId == id).Firs
    return View(dataItem);
}
...
```

Whenever the model binder is unable to find a value for the `id` parameter, the default value of `1` will be used inste has the effect of selecting the `Person` object whose `PersonId` property has a value of 1, as shown in Figure

eBooks-IT.org

Figure 24-4. The effect of using a default parameter value in an action method

Tip Bear in mind that I have solved the problem of non-numeric values for the model binder, but that I can still g values for which there are no valid `Person` objects defined by the `Home` controller. For example, the model bind convert the final segment of the URLs `/Home/Index/-1` and `/Home/Index/500` to `int` values. This action method to call the `Index` method with a real value, but will still result in an error because I don't perform a checks in the controller. I recommend you pay attention to the range of parameter values your action method may re test accordingly.

CULTURE-SENSITIVE PARSING

The `DefaultModelBinder` class uses culture-specific settings to perform type conversions from diffe the request data. The values that are obtained from URLs (the routing and query string data) are converted using insensitive parsing, but values obtained from form data are converted taking culture into account.

The most common problem that this causes relates to `DateTime` values. Culture-insensitive dates are expect the universal format `yyyy-mm-dd`. Form date values are expected to be in the format specified by the serve that a server set to the UK culture will expect dates to be in the form `dd-mm-yyyy`, whereas a server set to culture will expect the format `mm-dd-yyyy`, though in either case `yyyy-mm-dd` is acceptable, too.

A date value won't be converted if it isn't in the right format. This means that you must make sure that all dates the URL are expressed in the universal format. You must also be careful when processing date values that users default binder assumes that the user will express dates in the format of the server culture, something that is unli happen in an MVC application that has international users.

Binding to Complex Types

When the action method parameter is a complex type (i.e., any type which cannot be converted using the `TypeCo` class), then the `DefaultModelBinder` class uses reflection to obtain the set of public properties and then b of them in turn. To demonstrate how this works, I have added two new action methods to the `Home` controller, as Listing 24-9.

Listing 24-9. Adding New Action Methods to the HomeController.cs File

```
using System.Linq;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
    public class HomeController : Controller {
```

eBooks-IT.org

```
                new Person {PersonId = 1, FirstName = "Adam", La
"Freeman",
                 Role = Role.Admin},
               new Person {PersonId = 2, FirstName = "Jacqui", La
"Griffyth",
                 Role = Role.User},
                new Person {PersonId = 3, FirstName = "John", La
"Smith",
                 Role = Role.User},
                new Person {PersonId = 4, FirstName = "Anne", La
"Jones",
                 Role = Role.Guest}
        };

        public ActionResult Index(int? id = 1) {
                   Person dataItem = personData.Where(p => p.Pers
id).First();
            return View(dataItem);
        }

        public ActionResult CreatePerson() {
            return View(new Person());
        }

        [HttpPost]
        public ActionResult CreatePerson(Person model) {
            return View("Index", model);
        }
    }
}
```

The `CreatePerson` overload without any parameters creates a new `Person` object and passes it to the vi which has the effect of rendering the `/Views/Home/CreatePerson.cshtml` view, which I created t action method and the contents of which you can see in Listing 24-10.

Listing 24-10. The Contents of the CreatePerson.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Create Person</h2>
@using (Html.BeginForm()) {
     <div>@Html.LabelFor(m => m.PersonId)@Html.EditorFor(m => m.P
</div>
    <div>@Html.LabelFor(m => m.FirstName)@Html.EditorFor(m => m.Fi
</div>
     <div>@Html.LabelFor(m => m.LastName)@Html.EditorFor(m => m.L
</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m => m.Role)</
    <button type="submit">Submit</button>
}
```

eBooks-IT.org

form e

posts the editor data back to the CreatePerson action method (the version decorated with the HttpPost a
action method uses the /Views/Home/Index.cshtml view to display the data that the form contained. Y
how the new action methods work by starting the application and navigating to /Home/CreatePerson, as s
Figure 24-5.

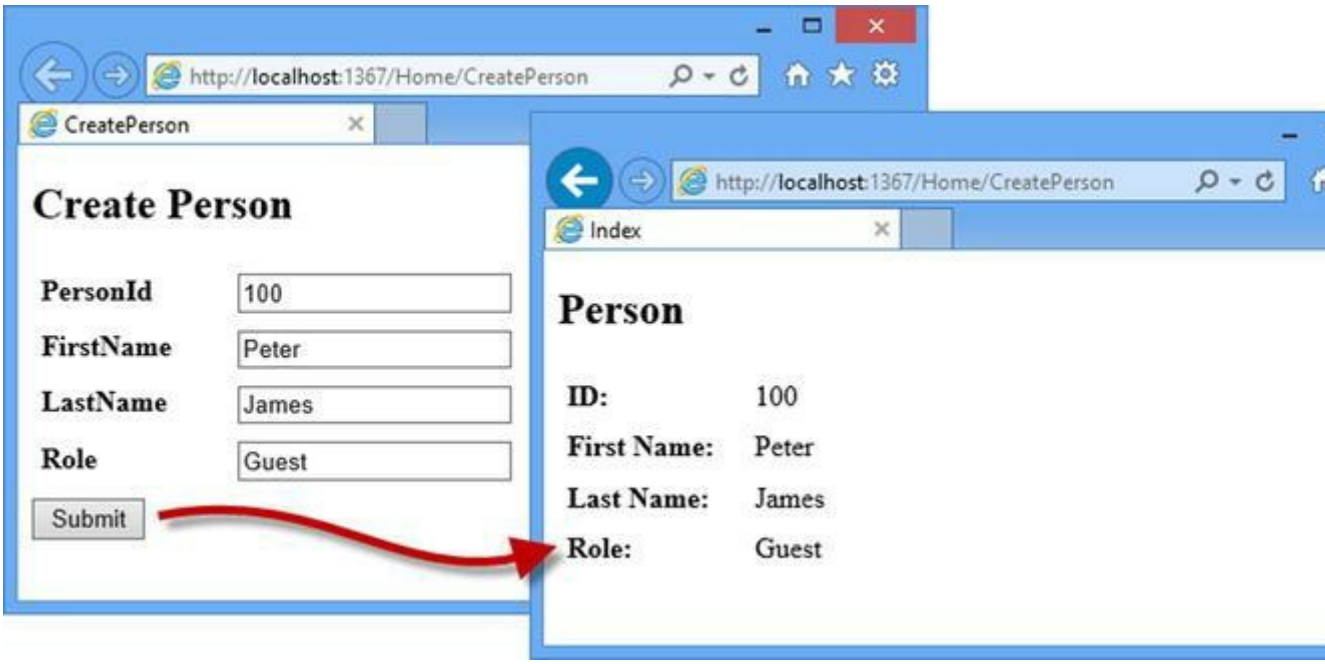


Figure 24-5. Using the CreatePerson action methods

I create a different kind of model binding situation when I post the form back to the CreatePerson method
model binder discovers that the action method requires a Person object and process each of the properties in turn
simple type property, the binder tries to locate a request value, just as it did in the previous example. So, for example,
encounters the PersonId property, the binder will look for a PersonId data value, which it finds in the form
request.

If a property requires another complex type, then the process is repeated for the new type. The set of public prope
obtained and the binder tries to find values for all of them. The difference is that the property names are nested. For
HomeAddress property of the Person class is of the Address type, which is shown in Listing 24-11.

Listing 24-11. A Nested Model Class in the Person.cs File

```
...
public class Address {
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
}
...
```

When looking for a value for the Line1 property, the model binder looks for a value for HomeAddress.L
the name of the property in the model object combined with the name of the property in the property type.

Creating Easily-Bound HTML

The use of prefixes means that I have to design views that take them into account, although the helper methods mak
do. In Listing 24-12, you can see how I have updated the CreatePerson.cshtml view file so that I captur

Listing 24-12. Updating the CreatePerson.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Create Person</h2>
@using(Html.BeginForm()) {
       <div>@Html.LabelFor(m  =>  m.PersonId)@Html.EditorFor(m=>m.P
</div>
      <div>@Html.LabelFor(m  =>  m.FirstName)@Html.EditorFor(m=>m.Fi
</div>
       <div>@Html.LabelFor(m  =>  m.LastName)@Html.EditorFor(m=>m.L
</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m=>m.Role)</di
    <div>
        @Html.LabelFor(m => m.HomeAddress.City)
        @Html.EditorFor(m=> m.HomeAddress.City)
    </div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.Country)
        @Html.EditorFor(m=> m.HomeAddress.Country)
    </div>
    <button type="submit">Submit</button>
}
```

I have used the strongly typed `EditorFor` helper method, and specified the properties I want to edit from the `HomeAddress` property. The helper automatically sets the `name` attributes of the `input` elements to match t the default model binder uses, as follows:

```
...
<input      class="text-box      single-line"      id="HomeAddress_
name="HomeAddress.Country"
    type="text" value="" />
...
```

As a consequence of this feature, I don't have to take any special action to ensure that the model binder can create `Address` object for the `HomeAddress` property. I can demonstrate this by editing the `/Views/Home/Index.cshtml` view to display the `HomeAddress` properties when they are submitte form, as shown in Listing 24-13.

Listing 24-13. Displaying the HomeAddress.City and HomeAddress.Country Properties in the Index.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Person</h2>
<div><label>ID:</label>@Html.DisplayFor(m => m.PersonId)</div>
<div><label>First Name:</label>@Html.DisplayFor(m => m.FirstName)<
<div><label>Last Name:</label>@Html.DisplayFor(m => m.LastName)</d
```

eBooks-IT.org

```
<div><label>City:</label>@Html.DisplayFor(m => m.HomeAddress.City)
<div><label>Country:</label>@Html.DisplayFor(m => m.HomeAddress.
</div>
```

If you start the application and navigate to the `/Home/CreatePerson` URL, you can enter values for the C
`Country` properties, and check that they are being bound to the model object by submitting the form, as shown i

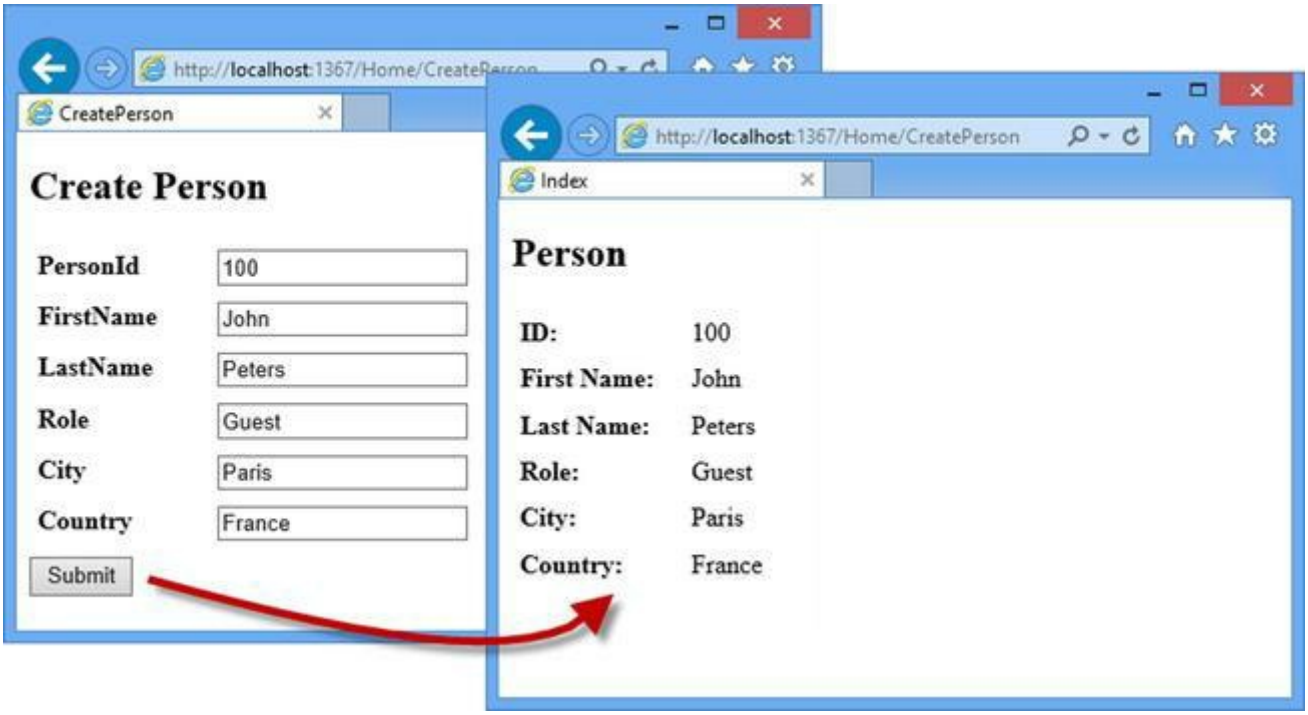


Figure 24-6. Binding to properties in complex objects

Specifying Custom Prefixes

There are occasions when the HTML you generate relates to one type of object, but you want to bind it to another. T
that the prefixes containing the view won't correspond to the structure that the model binder is expecting and your d
properly processed. To demonstrate this situation, I have created a new class file called `AddressSummary.c`
`Models` folder. You can see the contents of this file in Listing 24-14.

Listing 24-14. The Contents of the AddressSummary.cs File

```
namespace MvcModels.Models {
    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

I have added a new action method in the `Home` controller that uses the `AddressSummary` class, as shown
15.

Listing 24-15. Adding a New Action Method in the HomeController.cs File

```
using System.Linq;
using System.Web.Mvc;
using MvcModels.Models;
```

eBooks-IT.org

```
    public class HomeController : Controller {

        // ... other methods and statements omitted for brevity ..

        public ActionResult DisplaySummary(AddressSummary summary)
            return View(summary);
        }
    }
}
```

The new action method is called `DisplaySummary`. It has an `AddressSummary` parameter, which it
`View` method so that it can be displayed by the default view. I created the `DisplaySummary.cshtml` file
`/Views/Home` folder and you can see the contents in Listing 24-16.

Listing 24-16. The Contents of the DisplaySummary.cshtml File

```
@model MvcModels.Models.AddressSummary
@{
    ViewBag.Title = "DisplaySummary";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Address Summary</h2>
<div><label>City:</label>@Html.DisplayFor(m => m.City)</div>
<div><label>Country:</label>@Html.DisplayFor(m => m.Country)</div>
```

This view displays the values of the two properties defined by the `AddressSummary` class. To demonstrate
with prefixes when binding to different model types, I will change the call to the `BeginForm` helper method in th
`/Views/Home/CreatePerson.cshtml` file so that the form is submitted back to the new `Display`
action method, as shown in Listing 24-17.

Listing 24-17. Changing the Target of the Form in the CreatePerson.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Create Person</h2>
@using(Html.BeginForm("DisplaySummary", "Home")) {
       <div>@Html.LabelFor(m  =>  m.PersonId)@Html.EditorFor(m=>m.P
</div>
      <div>@Html.LabelFor(m  =>  m.FirstName)@Html.EditorFor(m=>m.Fi
</div>
       <div>@Html.LabelFor(m  =>  m.LastName)@Html.EditorFor(m=>m.L
</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m=>m.Role)</di
    <div>
        @Html.LabelFor(m => m.HomeAddress.City)
        @Html.EditorFor(m=> m.HomeAddress.City)
    </div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.Country)
        @Html.EditorFor(m=> m.HomeAddress.Country)
    </div>
```

eBooks-IT.org

}

You can see the problem if you start the application and navigate to the /Home/CreatePerson URL. Whe the form, the values that you entered for the City and Country properties are not displayed in the HTML gener DisplaySummary view.

The problem is that the name attributes in the form have the HomeAddress prefix, which is not what the m looking for when it tries to bind the AddressSummary type. I can fix this by applying the Bind attribute to t method parameter, which tells the binder which prefix to look for, as shown in Listing 24-18.

Listing 24-18. Applying the Bind Attribute in the HomeController.cs File

```
...
public ActionResult DisplaySummary([Bind(Prefix="HomeAd
AddressSummary summary) {
    return View(summary);
}
...
```

The syntax is a bit nasty, but the effect is useful. When populating the properties of the AddressSummary c model binder will look for HomeAddress.City and HomeAddress.Country data values in the reque example, I displayed editors for properties of the Person object, but used the model binder to create an instance o AddressSummary class when the form data was posted, as shown in Figure 24-7. This may seem like a long simple problem, but the need to bind to a different kind of object is surprisingly common and you are likely to need t in your projects.

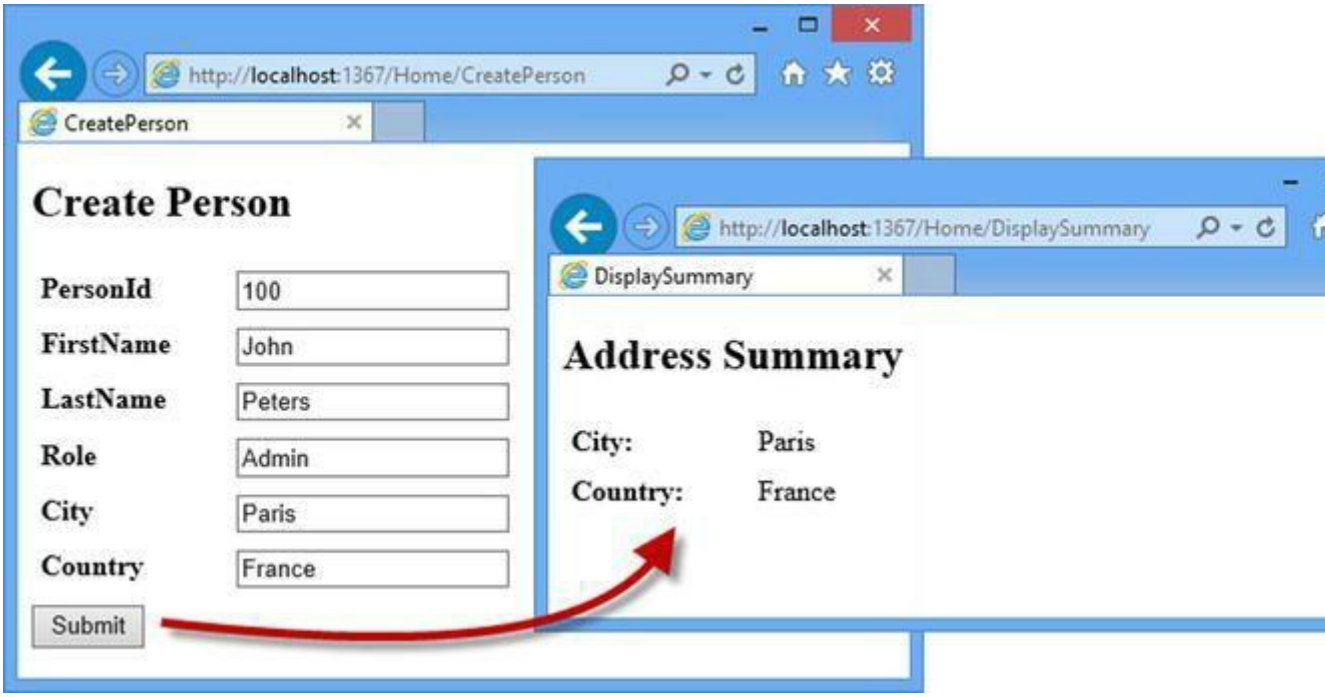


Figure 24-7. Binding to the properties of a different object type

Selectively Binding Properties

Imagine that the Country property of the AddressSummary class is especially sensitive and that I don't w be able to specify values for it. The first thing I can do is prevent the user from seeing the property or even prevent t from being included in the HTML sent to the browser, using the attributes I showed you in Chapter 22, or simply by editors for that property to the view.

However, a nefarious user could simply edit the form data sent to the server when submitting the form data and pi

eBooks-IT.org

lue for
Country property from the request, which I can do by using the Bind attribute on the action method parameter
24-19, you can see how I have used the attribute to prevent the user from providing a value for Country property
DisplaySummary action method in the Home controller.

Listing 24-19. Excluding a Property from Model Binding in the HomeController.cs File

```
...
public ActionResult DisplaySummary(
    [Bind(Prefix="HomeAddress", Exclude="Country")]AddressSummary
{
    return View(summary);
}
...
```

The Exclude property of the Bind attribute allows you to exclude properties from the model binding process
the effect by navigating to the /Home/CreatePerson URL, entering some data and submitting the form. Yo
there is no data displayed for the Country property. (As an alternative, you can use the Include property to s
those properties that should be bound in the model; all other properties will be ignored.)

When the Bind attribute is applied to an action method parameter, it only affects instances of that class that are b
action method; all other action methods will continue to try and bind all the properties defined by the parameter type.
to create a more widespread effect, then you can apply the Bind attribute to the model class itself, as shown in Lis
where I have applied the Bind method to the AddressSummary class so that only the City property is inc
bind process.

Listing 24-20. Applying the Bind Attribute in the AddressSummary.cs File

```
using System.Web.Mvc;

namespace MvcModels.Models {
    [Bind(Include="City")]
    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

Tip When the Bind attribute is applied to the model class *and* to an action method parameter, a property will be
the bind process only if neither application of the attribute excludes it. This means that the policy applied to the mode
be overridden by applying a less restrictive policy to the action method parameter.

Binding to Arrays and Collections

The default model binder includes some nice support for binding request data to arrays and collections. I demonstrat
features in the following sections, before moving on to show you how to customize the model binding process.

Binding to Arrays

One elegant feature of the default model binder is how it supports action method parameters that are arrays. To demo
have added a new method to the Home controller called Names, which you can see in Listing 24-21.

Listing 24-21. Adding the Names Action Method in the HomeController.cs File

```
using System.Linq;
```

```
using MvcModels.Models;

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ... other methods and statements omitted for brevity ..

        public ActionResult Names(string[] names) {
            names = names ?? new string[0];
            return View(names);
        }
    }
}
```

The `Names` action method takes a `string` array parameter called `names`. The model binder will look for an that is called `names` and create an array that contains those values.

Tip Notice that I have to check to see if the parameter is `null` in the action method for this example. You can constant or literal values as defaults for parameters.

In Listing 24-22, you can see the `/Views/Home/Names.cshtml` view file, which I created to show ar

Listing 24-22. The Contents of the Names.cshtml File

```
@model string[]
@{
    ViewBag.Title = "Names";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Names</h2>
@if (Model.Length == 0) {
    using(Html.BeginForm()) {
        for (int i = 0; i < 3; i++) {
            <div><label>@(i + 1):</label>@Html.TextBox("names")</d
        }
        <button type="submit">Submit</button>
    }
} else {
    foreach (string str in Model) {
        <p>@str</p>
    }
    @Html.ActionLink("Back", "Names");
}
```

This view displays different content based on the number of items there are in the view model. If there are no item display a form that contains three identical `input` elements, like this:

```
...
<form action="/Home/Names" method="post">
        <div><label>1:</label><input  id="names" name="names"  typ
value="" /></div>
        <div><label>2:</label><input  id="names" name="names"  typ
value="" /></div>
        <div><label>3:</label><input  id="names" name="names"  typ
value="" /></div>
```

eBooks-IT.org

```
</form>
...
```

When I submit the form, the default model binder sees that the action method requires a string array and looks for that have the same name as the parameter. For this example, this means the contents of all of the `input` elements together to populate an array. You can see how the action method and view operate in Figure 24-8.

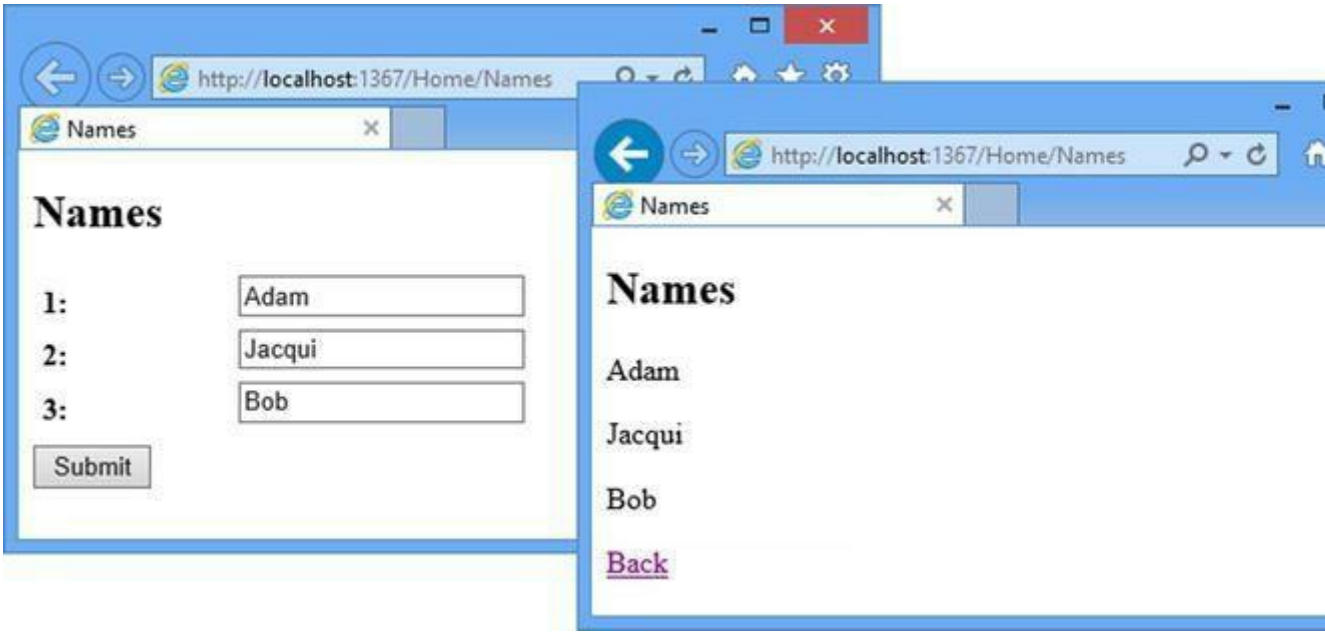


Figure 24-8. Model binding for arrays

Binding to Collections

It isn't just arrays that I can bind to. I can also use the .NET collection classes. In Listing 24-23, you can see how I the type of the `Names` action method parameter to be a strongly typed `List`.

Listing 24-23. Using a Strongly Typed Collection in the HomeController.cs File

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

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ... other methods and statements omitted for brevity ..

        public ActionResult Names(IList<string> names) {
            names = names ?? new List<string>();
            return View(names);
        }
    }
}
```

Notice that I have used the `IList` interface. I didn't need to specify a concrete implementation class (although I preferred). In Listing 24-24, you can see how I have modified the `Names.cshtml` view file to use the new m

```
@model IList<string>
@{
    ViewBag.Title = "Names";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Names</h2>
@if (Model. Count == 0) {
    using(Html.BeginForm()) {
        for (int i = 0; i < 3; i++) {
            <div><label>@(i + 1):</label>@Html.TextBox("names")</d
        }
        <button type="submit">Submit</button>
    }
} else {
    foreach (string str in Model) {
        <p>@str</p>
    }
    @Html.ActionLink("Back", "Names");
}
```

The functionality of the Names action is unchanged, but I am now able to work with a collection class rather tha

Binding to Collections of Custom Model Types

I can also bind individual data properties to an array of custom types, such as the AddressSummary model cla 24-25, you can see that I have added a new action method to the Home controller called Address, which has a collection parameter that relies on a custom model class.

Listing 24-25. Defining an Action Method in the HomeController.cs File

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

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ... other methods and statements omitted for brevity ..

        public ActionResult Address(IList<AddressSummary> addresse
            addresses = addresses ?? new List<AddressSummary>();
            return View(addresses);
        }
    }
}
```

The view that I created for this action method is the /Views/Home/Address.cshtml file, which you Listing 24-26.

Listing 24-26. The Contents of the Address.cshtml File

```
@model IList<AddressSummary>
@{
    ViewBag.Title = "Address";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Addresses</h2>
@if (Model.Count() == 0) {
    using (Html.BeginForm()) {
        for (int i = 0; i < 3; i++) {
            <fieldset>
                <legend>Address @(i + 1)</legend>
                 <div><label>City:</label>@Html.Editor("[" + i + "
</div>
                        <div><label>Country:</label>@Html.Editor("[
"].Country")</div>
            </fieldset>
        }
        <button type="submit">Submit</button>
    }
} else {
    foreach (AddressSummary str in Model) {
        <p>@str.City, @str.Country</p>
    }
    @Html.ActionLink("Back", "Address");
}
```

This view renders a `form` element if there are no items in the model collection. The `form` consists of pairs of i elements whose `name` attributes are prefixed with an array index, like this:

```
...
<fieldset>
    <legend>Address 1</legend>
    <div>
        <label>City:</label>
          <input class="text-box single-line" name="[0].City" typ
value="" />
    </div>
    <div>
        <label>Country:</label>
        <input class="text-box single-line" name="[0].Country" typ
value="" />
    </div>
</fieldset>
<fieldset>
    <legend>Address 2</legend>
    <div>
        <label>City:</label>
          <input class="text-box single-line" name="[1].City" typ
value="" />
    </div>
    <div>
        <label>Country:</label>
        <input class="text-box single-line" name="[1].Country" typ
value="" />
```

eBooks-IT.org

```
</fieldset>
...
```

When the `form` is submitted, the default model binder realizes that it needs to create a collection of `AddressS` objects and uses the array index prefixes in the `name` attributes to obtain values for the object properties. The prope with `[0]` are used for the first `AddressSummary` object, those prefixed with `[1]` are used for the second o on.

The `Address.cshtml` view defines `input` elements for three such indexed objects and displays them wh collection contains items. Before I can demonstrate this, I need to remove the `Bind` attribute from the `Address` model class, as shown in Listing 24-27; otherwise, the model binder will ignore the `Country` property.

Listing 24-27. Removing the Bind Attribute from the AddressSummary.cs File

```
using System.Web.Mvc;

namespace MvcModels.Models {
    // This attribute has been commented out
    //[Bind(Include="City")]
    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

You can see how the binding process for custom object collections works by starting the application and navigatin `/Home/Address` URL. Enter some cities and countries, and then click the `Submit` button to post the form to The model binder will find and process the indexed data values and use them to create the collection of `AddressS` objects that are then passed back to the view and displayed to you, as shown in Figure 24-9.

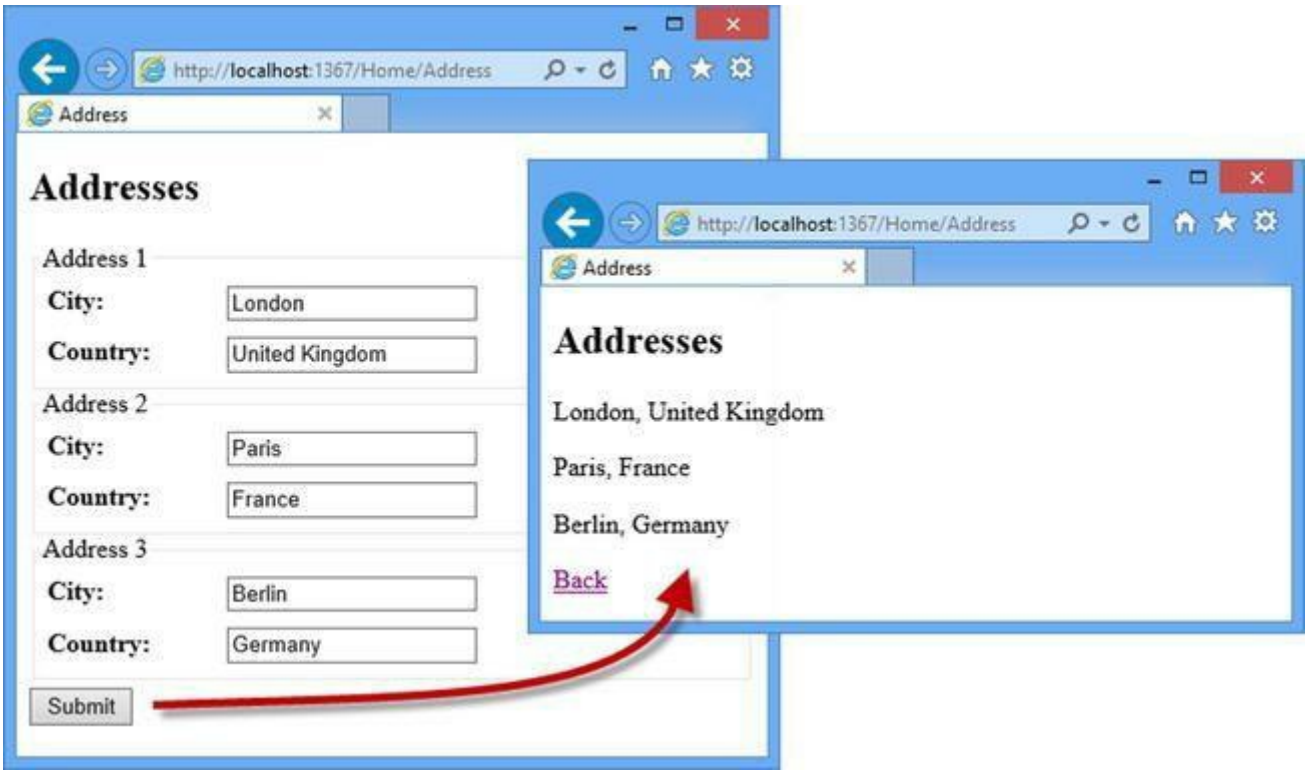


Figure 24-9. Binding collections of custom objects

eBooks-IT.org

The model binding process is performed automatically when an action method defines parameters, but I can take dir the process if I want to. This gives more explicit control over how model objects are instantiated, where data values from, and how data parsing errors are handled. Listing 24-28 demonstrates how I have changed the `Address` act the `Home` controller to manually invoke the binding process.

Listing 24-28. Manually Invoking the Model Binding Process in the HomeController.cs File

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

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ... other methods and statements omitted for brevity ..

        public ActionResult Address() {
            IList<AddressSummary> addresses = new List<AddressSumm
            UpdateModel(addresses);
            return View(addresses);
        }
    }
}
```

The `UpdateModel` method takes a model object that I was previously defining as a parameter and tries to obt its public properties using the standard binding process.

When I manually invoke the binding process, I am able to restrict the binding process to a single source of data. B binder looks in four places: form data, route data, the query string, and any uploaded files. Listing 24-29 shows how binder to searching for data in a single location—in this case, the form data.

Listing 24-29. Restricting the Binder to the Form Data in the HomeController.cs File

```
...
public ActionResult Address() {
    IList<AddressSummary> addresses = new List<AddressSummary>();
    UpdateModel(addresses, new FormValueProvider(ControllerContext
    return View(addresses);
}
...
```

This version of the `UpdateModel` method takes an implementation of the `IValueProvider` interface, becomes the sole source of data values for the binding process. Each of the four default data locations is represented `IValueProvider` implementation, as shown in Table 24-3.

Table 24-3. *The Built-in IValueProvider Implementations*

Source	IValueProvider Implementation
`Request.Form`	`FormValueProvider`
`RouteData.Values`	`RouteDataValueProvider`
`Request.QueryString`	`QueryStringValueProvider`
`Request.Files`	`HttpFileCollectionValueProvider`

eBooks-IT.org

Each of the classes listed in Table 24-3 takes a ControllerContext constructor parameter, which I obta property called ControllerContext that is defined by the Controller class, as shown in the listing. T common way of restricting the source of data is to look only at the form values. There is a neat binding trick that I c I don't have to create an instance of FormValueProvider, as shown in Listing 24-30.

Listing 24-30. Restricting the Binder Data Source in the HomeController.cs File

```
...
public ActionResult Address(FormCollection formData) {
    IList<AddressSummary> addresses = new List<AddressSummary>();
    UpdateModel(addresses, formData);
    return View(addresses);
}
...
```

The FormCollection class implements the IValueProvider interface, and if I define the action me a parameter of this type, the model binder will provide me with an object that I can pass directly to the UpdateMo method.

Tip There are other overloaded versions of the UpdateModel method that specify a prefix to search for and properties should be included in the binding process.

Dealing with Binding Errors

Users will inevitably supply values that cannot be bound to the corresponding model properties—invalid dates or text values, for example. When I invoke model binding explicitly, I am responsible for dealing with any errors. The model expresses binding errors by throwing an InvalidOperationException. Details of the errors can be fo the ModelState feature, which I describe in Chapter 25. But when using the UpdateModel method, I mus to catch the exception and use the ModelState to express an error message to the user, as shown in Listing 24

Listing 24-31. Dealing with Model Binding Errors in the HomeController.cs File

```
...
public ActionResult Address(FormCollection formData) {
    IList<AddressSummary> addresses = new List<AddressSummary>();
    try {
        UpdateModel(addresses, formData);
    } catch (InvalidOperationException ex) {
        // provide feedback to user
    }
    return View(addresses);
}
...
```

As an alternative approach, I can use the TryUpdateModel method, which returns true if the model bind is successful and false if there are errors, as shown in Listing 24-32.

Listing 24-32. Using the TryUpdateModel Method in the HomeController.cs File

```
...
public ActionResult Address(FormCollection formData) {
    IList<AddressSummary> addresses = new List<AddressSummary>();
    if (TryUpdateModel(addresses, formData)) {
        // proceed as normal
```

eBooks-IT.org

```
        // provide feedback to user
    }
    return View(addresses);
}
...
```

The only reason to favor `TryUpdateModel` over `UpdateModel` is if you don't like catching and dealin exceptions. There is no functional difference in the model binding process.

Tip When model binding is invoked automatically, binding errors are not signaled with exceptions. Instead, you n the result through the `ModelState.IsValid` property. I explain `ModelState` in Chapter 25.

Customizing the Model Binding System

I have shown you the default model binding process. As you might expect by now, there are some different ways in binding system can be customized. I show you some examples in the following sections.

Creating a Custom Value Provider

By defining a custom value provider, I can add my own source of data to the model binding process. Value providers the `IValueProvider` interface, which is shown in Listing 24-33.

Listing 24-33. The IValueProvider Interface from the MVC Framework

```
namespace System.Web.Mvc {
    public interface IValueProvider {

        bool ContainsPrefix(string prefix);

        ValueProviderResult GetValue(string key);
    }
}
```

The `ContainsPrefix` method is called by the model binder to determine if the value provider can resolve th given prefix. The `GetValue` method returns a value for a given data key, or `null` if the provider doesn't have a data.

I have added an `Infrastructure` folder to the example project and created a new class file called `CountryValueProvider.cs`, which I will use to provide values for the `Country` property. You can s contents of this file in Listing 24-34.

Listing 24-34. The Contents of the CountryValueProvider.cs File

```
using System.Globalization;
using System.Web.Mvc;

namespace MvcModels.Infrastructure {
    public class CountryValueProvider : IValueProvider {

        public bool ContainsPrefix(string prefix) {
            return prefix.ToLower().IndexOf("country") > -1;
        }

        public ValueProviderResult GetValue(string key) {
```

eBooks-IT.org

```
                return new ValueProviderResult("USA", "USA",
                    CultureInfo.InvariantCulture);
            } else {
                return null;
            }
        }
    }
}
```

This value provider only responds to requests for values for the `Country` property and it always returns the va all other requests, I return `null`, indicating that I cannot provide data.

I have to return the data value as a `ValueProviderResult` class. This class has three constructor param first is the data item that I want to associate with the requested key. The second parameter is a version of the data va safe to display as part of an HTML page. The final parameter is the culture information that relates to the value; I hav the `InvariantCulture`.

To register the value provider with the application, I need to create a factory class that will create instances of the when they are required by the MVC Framework. The factory class must be derived from the abstract `ValueProviderFactory` class. In Listing 24-35, you can see the contents of the `CustomValueProviderFactory.cs` class file that I added to the `Infrastructure` folder.

Listing 24-35. The Contents of the CustomValueProviderFactory.cs File

```
using System.Web.Mvc;

namespace MvcModels.Infrastructure {
    public class CustomValueProviderFactory : ValueProviderFactory

        public override IValueProvider GetValueProvider(Controller
                controllerContext) {
            return new CountryValueProvider();
        }
    }
}
```

The `GetValueProvider` method is called when the model binder wants to obtain values for the binding pr implementation simply creates and returns an instance of the `CountryValueProvider` class, but you can u provided through the `ControllerContext` parameter to respond to different kinds of requests by creating d providers.

I need to register the factory class with the application, which I do in the `Application_Start` method of `Global.asax`, as shown in Listing 24-36.

Listing 24-36. Registering a Value Provider Factory in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using MvcModels.Infrastructure;

namespace MvcModels {

    public class MvcApplication : System.Web.HttpApplication {
```

eBooks-IT.org

```
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

                                ValueProviderFactories.Factories.Insert
CustomValueProviderFactory());
        }
    }
}
```

I register the factory class by adding an instance to the static `ValueProviderFactories.Factori` collection. The model binder looks at the value providers in sequence, which means I have to use the `Insert` met custom factory at the first position in the collection if I want to take precedence over the built-in providers.

If I want the custom provider to be a fallback that is used when the other providers cannot supply a data value, th the `Add` method to append the factory class to the end of the collection, like this:

```
...
ValueProviderFactories.Factories.Add(new CustomValueProviderFactor
...
```

I want the custom value provider to be used before any other provider, and so I used the `Insert` method. I nee the `Address` action method before I can test the value provider, so that the model binder doesn't just look at the f model property values. In Listing 24-37, you can see how I have removed the restriction on the source for values in `TryUpdateModel` method.

Listing 24-37. Removing the Restriction on the Sources of Model Property Values in the HomeController.cs File

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

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ... other methods and statements omitted for brevity ..

        public ActionResult Address() {
            IList<AddressSummary> addresses = new List<AddressSumm
            UpdateModel(addresses);
            return View(addresses);
        }
    }
}
```

You can see the custom value provider at work if you start the application and navigate to the `/Home/Addres` Enter city and country data, and then press the `Submit` button. You will see that the custom value provider, whic precedence over the built-in providers, has been used to generate values for the `Country` property in each of the `AddressSummary` objects that the model binder has created, as shown in Figure 24-10.

eBooks-IT.org

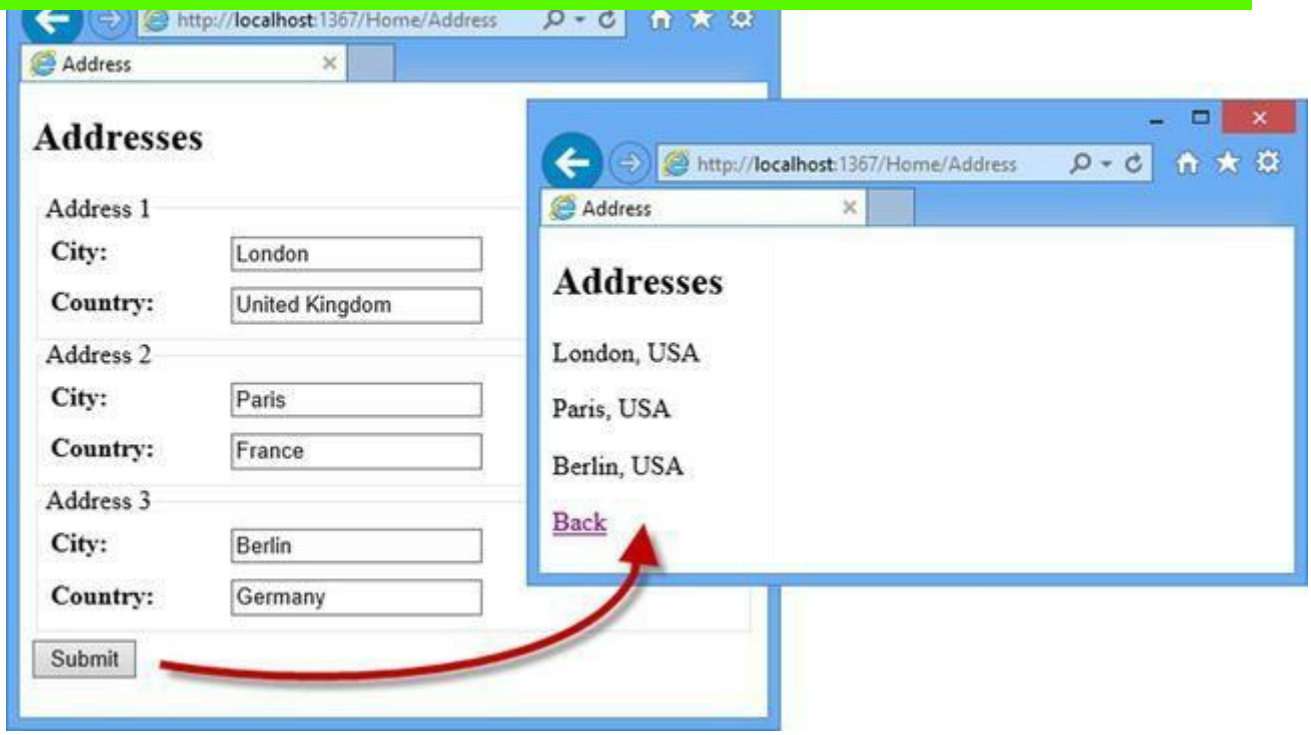


Figure 24-10. The effect of the custom value provider

Creating a Custom Model Binder

I can override the default binder's behavior by creating a custom model binder for a specific type. Custom model bin implement the `IModelBinder` interface, which I showed you earlier in the chapter. To demonstrate how to cre binder, I have added the `AddressSummaryBinder.cs` class file to the `Infrastructure` folder, th which you can see in Listing 24-38.

Listing 24-38. The Contents of the AddressSummaryBinder.cs File

```
using MvcModels.Models;
using System.Web.Mvc;

namespace MvcModels.Infrastructure {
    public class AddressSummaryBinder : IModelBinder {

        public object BindModel(ControllerContext controllerContex
                ModelBindingContext bindingContext) {

            AddressSummary model = (AddressSummary)bindingContext.
                ?? new AddressSummary();
            model.City = GetValue(bindingContext, "City");
            model.Country = GetValue(bindingContext, "Country");
            return model;
        }

         private string GetValue(ModelBindingContext context, stri
{
                name = (context.ModelName == "" ? "" : context.Mod
".") + name;

                                                ValueProviderResult    res
```

eBooks-IT.org

```
                if (result == null || result.AttemptedValue == ) {
                    return "<Not Specified>";
                } else {
                    return (string)result.AttemptedValue;
                }
            }
        }
}
```

The MVC Framework will call the `BindModel` method when it wants an instance of the model type that the b supports. I will show you how to register a model binder shortly, but the `AddressSummaryBinder` class w used to create instances of the `AddressSummary` class, which makes the code a lot simpler. (You can create binders that support multiple types, but I prefer one binder for each type.)

Tip I don't perform any input validation in this model binder, meaning that I blithely assume that the user has pro values for all of the `Person` properties. I discuss validation in Chapter 25, but for the moment, I want to focus on model binding process.

The parameters to the `BindModel` method are a `ControllerContext` object that you can use to get current request and a `ModelBindingContext` object, which provides details of the model object that is sou as access to the rest of the model binding facilities in the MVC application. In Table 24-4, I have described the most properties defined by the `ModelBindingContext` class.

***Table 24-4**. The Most Useful Properties Defined by the ModelBindingContext Class*

Property	Description
`Model`	Returns the model object passed to the `UpdateModel` method if binding has been invoked manually
`ModelName`	Returns the name of the model that is being bound
`ModelType`	Returns the type of the model that is being created
`ValueProvider`	Returns an `IValueProvider` implementation that can be used to get data values from the request

The custom model binder is simple. When the `BindModel` method is called, I check to see if the `Model` pro `ModelBindingContext` object has been set. If it has, this is the object that I will generate data value for, ar I create a new instance of the `AddressSummary` class. I get the values for the `City` and `Country` proper the `GetValue` method and return the populated `AddressSummary` object.

In the `GetValue` method, I use the `IValueProvider` implementation obtained from the `ModelBindingContext.ValueProvider` property to get values for the model object properties.

The `ModelName` property tells me if there is a prefix I need to append to the property name I am looking for. that the action method is trying to create a collection of `AddressSummary` objects, which means that the indiv elements will have `name` attribute values that are prefixed `[0]` and `[1]`. The values I am looking for in the reque `[0].City`, `[0].Country`, and so on. As a final step, I supply a default value of `<Not Specified>` a value for a property or the property is the empty string (which is what is sent to the server when the user doesn't in the `input` elements in the form).

Registering the Custom Model Binder

I have to register the custom model binder so that the MVC application knows which types it can support. I do this i `Application_Start` method of `Global.asax`, as demonstrated by Listing 24-39.

***Listing 24-39**.* Registering a Custom Model Binder

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

eBooks-IT.org

```
using System.Web.Mvc;
using System.Web.Routing;
using MvcModels.Infrastructure;
using MvcModels.Models;

namespace MvcModels {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            // This statement has been commented out
            //ValueProviderFactories.Factories.Insert(0,
            //    new CustomValueProviderFactory());

                        ModelBinders.Binders.Add(typeof(AddressSummar
AddressSummaryBinder());
        }
    }
}
```

I register the binder through the `ModelBinders.Binders.Add` method, passing in the type that the bin and an instance of the binder class. Notice that I have removed the statement that registers the custom value provide test the custom model binder by starting the application, navigating to the `/Home/Address` URL, and filling in the form elements. When you submit the form, the custom model binder will use `<Not Specified>` for all o properties for which you didn't enter a value, as shown in Figure 24-11.

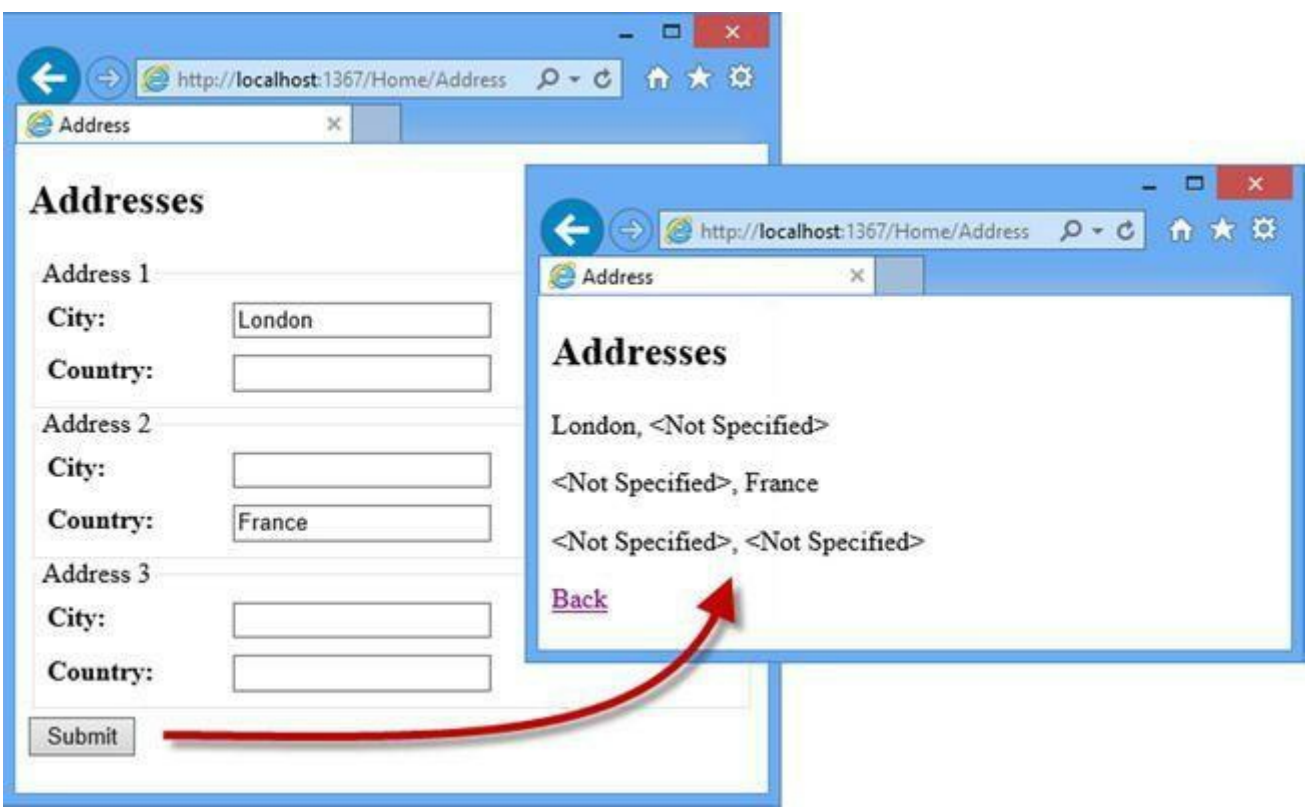


Figure 24-11. The effect of using a custom model binder

Registering a Model Binder with an Attribute

eBooks-IT.org

that you don't need to use the Global.asax file. In Listing 24-40, you can see how I have specified AddressSummaryBinder as the binder for the AddressSummary class.

Listing 24-40. Using the ModelBinder Attribute in the AddressSummary.cs File

```
using System.Web.Mvc;
using MvcModels.Infrastructure;

namespace MvcModels.Models {

    [ModelBinder(typeof(AddressSummaryBinder))]
    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

Summary

In this chapter, I introduced you to the workings of the model binding process, showing you how the default model operates and the different ways in which the process can be customized. Many MVC Framework applications will o default model binder, which works nicely to process the HTML that the helper methods generate. But for more adva applications, it can be useful to use custom binders that create model objects in a more efficient or specific way. In t chapter, I show you how to validate model objects and how to present the user with meaningful errors when invalid received.

eBooks-IT.org

Model Validation

In the previous chapter, I showed you how the MVC Framework creates model objects from HTTP requests throug binding process. Throughout that chapter, I worked on the basis that the data the user supplied was valid. The reality will often enter data that isn't valid and cannot be used, which leads me to the topic of this chapter: *model validation*

Model validation is the process of ensuring the data received by the application is suitable for binding to the model this is not the case, providing useful information to the user that will help explain the problem.

The first part of the process, checking the data received, is one of the key ways to preserve the integrity of the do Rejecting data that doesn't make sense in the context of the domain can prevent odd and unwanted states arising in t The second part, helping the user correct the problem, is equally important. Without the information and feedback th interact with the application, users become frustrated and confused. In public-facing applications, this means users v stop using the application. In corporate applications, this means the user's workflow will be hindered. Neither outcor desirable. Fortunately, the MVC Framework provides extensive support for model validation. I will show you how to features and then demonstrate some advanced techniques to fine-tune the validation process. Table 25-1 provides the this chapter.

***Table 25-1**. Chapter Summary*

Problem	Solution
Explicitly validate a model	Use the `ModelState` object to record validation errors.
Generate a summary of validation errors	Use the `Html.ValidationSummary` helper method.
Display property-level validation errors	Use the `Html.ValidationMessageFor` helper method.
Define validation rules within the model class	Apply attributes to the properties of the model class.
Create a custom validation attribute	Derive from the `ValidationAttribute` class.
Define a self-validating model	Implement the `IValidatableObject` interface.
Use client-side validation	Add the Microsoft unobtrusive validation package to the application.
Perform remote validation	Define an action method that returns a `JsonResult` and add the `Remote` attribute to the mode property to be validated.

Preparing the Example Project

For this chapter, I created a new project called `ModelValidation` using the Visual Studio `Empty` template, option to add the core MVC folders and references. Having created the project, I added a new class file called `Appointment.cs`, which you can see in Listing 25-1, to the `Models` folder.

***Listing 25-1**.* The Contents of the Appointment.cs File

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {
```

```
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        public DateTime Date { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

The Appointment model class defines three properties and I have used the DataType attribute to indicat Date property should be expressed as a date without a time component. I also created a Home controller for the e project and defined action methods that operate on the Appointment model class, as shown in Listing 25-2.

Listing 25-2. The Contents of the HomeController.cs File

```
using System;
using System.Web.Mvc;
using ModelValidation.Models;

namespace ModelValidation.Controllers {
    public class HomeController : Controller {

        public ViewResult MakeBooking() {
            return View(new Appointment { Date = DateTime.Now });
        }

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt) {

            // statements to store new Appointment in a
            // repository would go here in a real project

            return View("Completed", appt);
        }
    }
}
```

I have defined two versions of the MakeBooking action method. The one that is of most interest is the versio the HttpPost attribute has been applied, since this is the version where model binding will be used to construct t Appointment parameter object.

Notice that I added a comment to indicate where, in a real application, statements would be placed to store the det Appointment object that the model binder will create. I am not going to create a repository because I want to f model binding and validation process. But it is important to bear in mind that the main reason to validate a model is to or meaningless data from being placed in the repository and causing problems (either when trying to store the data or to process the data later).

Creating the Layout

I will need a simple layout for some of the examples in this chapter. I created the Views/Shared folder and ad _Layout.cshtml file to it, the contents of which you can see in Listing 25-3.

Listing 25-3. The Contents of the _Layout.cshtml File

eBooks-IT.org

```
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style type="text/css">
        .field-validation-error { color: #f00;}
        .validation-summary-errors { color: #f00; font-weight: bol
           .input-validation-error { border: 2px solid #f00; bac
color: #fee; }
        input[type="checkbox"].input-validation-error { outline: 2
#f00; }
    </style>
</head>
<body>
    @RenderBody()
</body>
</html>
```

I also created a view start file so that the layout is applied to views automatically. I added a new view file called `_ViewStart.cshtml` to the `Views` folder and you can see the contents of the new file in Listing 25-4. (I view start files in Chapter 5 as part of the overview of Razor functionality.)

Listing 25-4. The Contents of the _ViewStart.cshtml File

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Creating the Views

To complete the preparation, I created two views to support the action methods, both of which are located in the `/Views/Home` folder. In Listing 25-5, you can see the contents of the `MakeBooking.cshtml` file, whic form that allows the user to create a new appointment.

Listing 25-5. The Contents of the MakeBooking.cshtml File

```
@model ModelValidation.Models.Appointment
@{
    ViewBag.Title = "Make A Booking";
}
<h4>Book an Appointment</h4>

@using (Html.BeginForm()) {
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
       <p>@Html.EditorFor(m  =>  m.TermsAccepted)  I  accept  the
conditions</p>
    <input type="submit" value="Make Booking" />
}
```

When the form is posted back to the application, the `MakeBooking` action method displays the details of the a that the user has created using the `Completed.cshtml` view, which you can see in Listing 25-6.

```
@model ModelValidation.Models.Appointment
@{
    ViewBag.Title = "Completed";
}
<h4>Your appointment is confirmed</h4>
<p>Your name is: <b>@Html.DisplayFor(m => m.ClientName)</b></p>
<p>The date of your appointment is: <b>@Html.DisplayFor(m => m.D
</p>
```

As you may have gathered, the example for this chapter is based around creating appointments. You can see how starting the application and navigating to the `/Home/MakeBooking` URL. Entering details into the form and c `Submit` button will send the data to the server, which performs the model-binding process to create an `Appoin` object, the details of which are then rendered using the `Completed.cshtml` view, as shown in Figure 25-1.

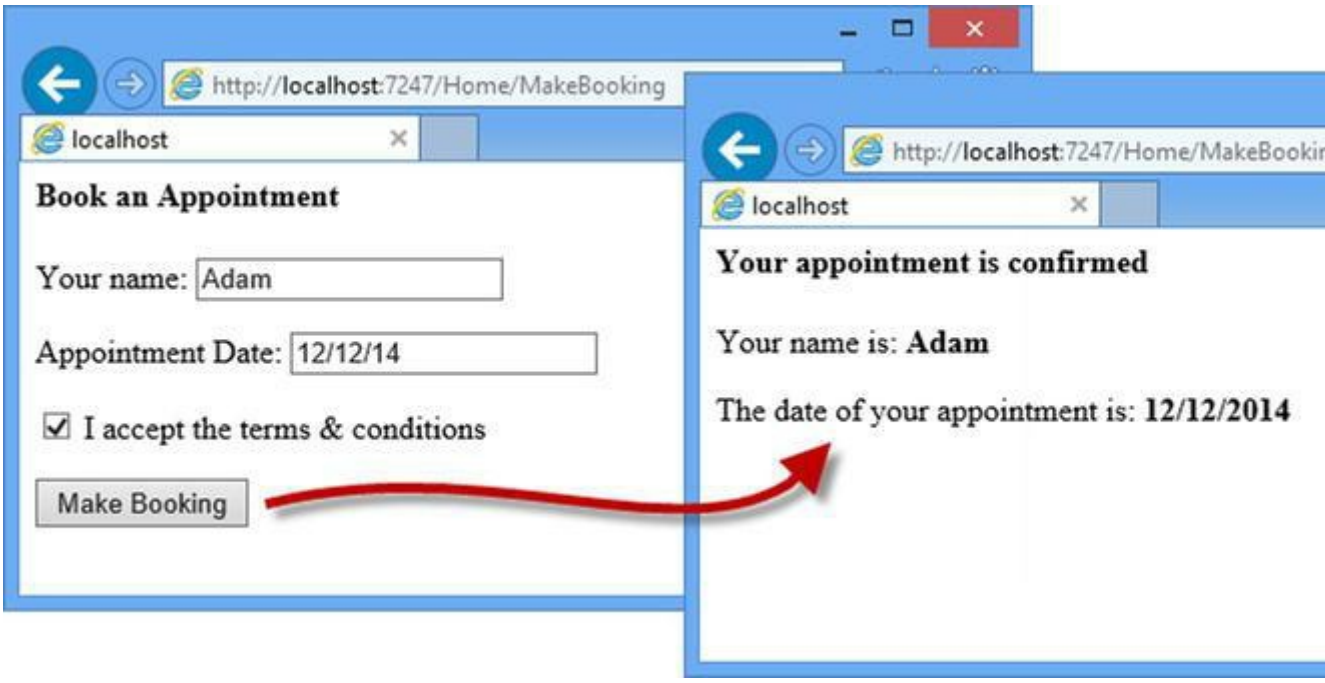


Figure 25-1. Using the example application

As it stands, the application will accept any data the user submits, but to preserve the integrity of the application ar model, I need three things to be true before I know that the user has provided an acceptable `Appointment` obje

- The user must provide a name.
- The user must provide a date (in the `mm/dd/yyyy` format) that is in the future.
- The user must have checked the check box to accept the terms and conditions.

Model validation is the process of enforcing these requirements. In the following sections, I will show you the diff techniques available for checking the data that the user has provided and to give the user feedback when the applicati the data they have submitted.

Explicitly Validating a Model

The most direct way of validating a model is to do so in the action method. Listing 25-7 shows how I have added ex for each property defined by the `Appointment` class in the `HttpPost` version of the `MakeBooking` ac

```
using System;
using System.Web.Mvc;
using ModelValidation.Models;

namespace ModelValidation.Controllers {
    public class HomeController : Controller {

        public ViewResult MakeBooking() {
            return View(new Appointment { Date = DateTime.Now });
        }

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt) {

            if (string.IsNullOrEmpty(appt.ClientName)) {
                 ModelState.AddModelError("ClientName", "Please en
name");
            }

                  if  (ModelState.IsValidField("Date")   && DateTim
appt.Date) {
                 ModelState.AddModelError("Date", "Please enter a
the future");
            }

            if (!appt.TermsAccepted) {
                ModelState.AddModelError("TermsAccepted", "You mus
the terms");
            }

            if (ModelState.IsValid) {
                // statements to store new Appointment in a
                // repository would go here in a real project
                return View("Completed", appt);
            } else {
                return View();
            }
        }
    }
}
```

I check the values that the model binder has assigned to the properties of the parameter object and register any err with the `ModelState` property, which the controller inherits from its base class. As an example, consider how I `ClientName` property:

```
...
if (string.IsNullOrEmpty(appt.ClientName)) {
    ModelState.AddModelError("ClientName", "Please enter your name
}
...
```

I want a value from the user for this property, so I use the static `string.IsNullOrEmpty` method to ch property. If I have not received a value, I use the `ModelState.AddModelError` method to specify the n property for which there is a problem (`ClientName`) and a message that should be displayed to the user to help

eBooks-IT.org

I can check to see whether the model binder was able to assign a value to a property by using the `ModelState.IsValidField` property. I do this for the `Date` property to make sure that the model bind to parse the value the user submitted; there is no point performing additional checks or reporting additional errors if r be parsed from the request data.

After I have validated all the properties in the model object, I read the `ModelState.IsValid` property to there were errors. This method returns `true` if I called the `Model.State.AddModelError` method du checks or if the model binder had any problems creating the `Appointment` object:

```
...
if (ModelState.IsValid) {
    // statements to store new Appointment in a
    // repository would go here in a real project
    return View("Completed", appt);
} else {
    return View();
}
...
```

I know I have a valid `Appointment` object if there are no problems reported by the `IsValid` property and the `Completed.cshtml` view (and, in a real project, store the `Appointment` object in the repository). I `IsValue` property returns `false`, then I know that I have a problem, which I deal with by calling the `View` m render the default view.

Displaying Validation Errors to the User

It may seem odd to deal with a validation error by calling the `View` method, but the templated view helpers that I us generate `input` elements in the `MakeBooking.cshtml` view check the view model for validation errors.

The helpers add a CSS class called `input-validation-error` to the `input` elements if an error has reported for the corresponding properties, which is why I added these CSS styles to the layout when I created the ex project:

```
...
.input-validation-error { border: 2px solid #f00; background-colo
}
input[type="checkbox"].input-validation-error { outline: 2px solid
...
```

The first style creates the effect of setting a red border and a pink background on any element for which there is a second style applies a red border to checkbox elements. These elements are hard to style and usually require special a can test the explicit validation approach by starting the application, navigating to the `/Home/MakeBooking` U clicking the `Make Booking` button without entering any data in the form. The result is shown in Figure 25-2.

eBooks-IT.org

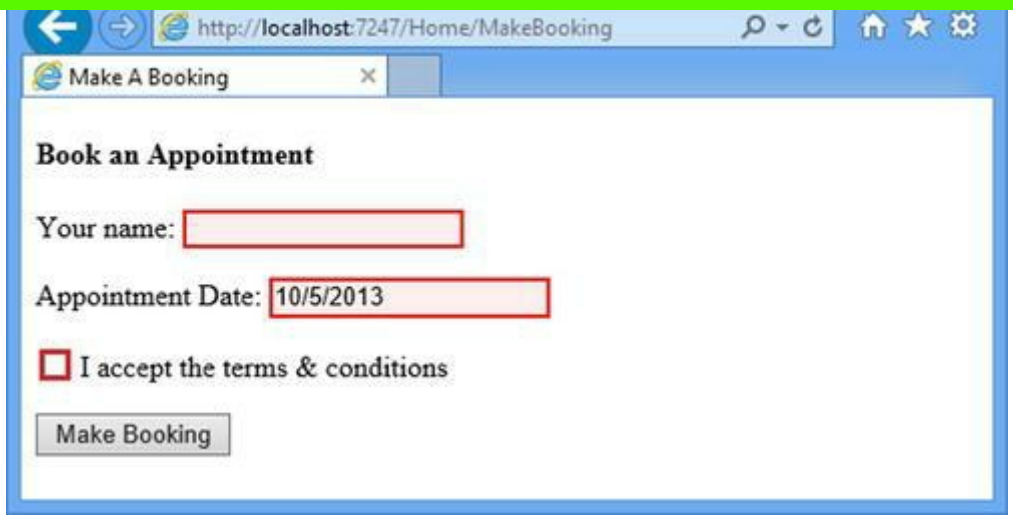


Figure 25-2. Errors result in highlighted elements

STYLING CHECK BOXES

Styling checkboxes can be difficult, especially with older browsers. An alternative to the CSS styles I defined in `_Layout.cshtml` is to replace the Boolean editor template with a custom template (as the `~/Views/Shared/EditorTemplates/Boolean.cshtml` file) and to wrap the check box element that can be more easily styled. Here is the sort of template that I use, which you can tailor to your own

```
@model bool?
@if (ViewData.ModelMetadata.IsNullableValueType) {
  @Html.DropDownListFor(m => m, new SelectList(new[] { "Not Set"
"True", "False" }, Model))
} else {
  ModelState state =
ViewData.ModelState[ViewData.ModelMetadata.PropertyName];
  bool value = Model ?? false;
  if (state != null && state.Errors.Count > 0) {
    <span class="input-validation-error" style="padding: 0; marg
1px">
      @Html.CheckBox("", value)
    </span>
  } else {
    @Html.CheckBox("", value)
  }
}
```

This template will wrap a check box in a `span` element to which the `input-validation-error` st applied if there are any model errors associated with the property that the template has been applied to. You can about replacing editor templates in Chapter 22.

When you submit the form without any data, errors are highlighted for the `ClientName` and `TermsAccep` properties because values are not provided. The default value that displayed for the `Date` property is a valid date, b the future and so is also flagged as a validation error.

The user will not be shown the `Completed.cshtml` view until the form is submitted with data that can be model browser and which passes the explicit validation checks in the `MakeBooking` action method. Until that ha submitting the form will cause the `MakeBooking.cshtml` view to be rendered with the current validation er

Displaying Validation Messages

eBooks-IT.org

not tell the user what the problem is. Fortunately, there are some convenient helper methods that assist in doing this. n a field

shows one of these helper methods, which I have applied to the `MakeBooking.cshtml` view (since this is w validation errors are shown to the user).

Listing 25-8. Using the Validation Summary Helper Method in the MakeBooking.cshtml File

```
@model ModelValidation.Models.Appointment
@{
    ViewBag.Title = "Make A Booking";
}
<h4>Book an Appointment</h4>

@using (Html.BeginForm()) {
    @Html.ValidationSummary()
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
       <p>@Html.EditorFor(m  =>  m.TermsAccepted)  I  accept  the
conditions</p>
    <input type="submit" value="Make Booking" />
}
```

The `Html.ValidationSummary` helper adds a summary of the validation errors to the user. If there are then the helper doesn't generate any HTML. Figure 25-3 demonstrates the validation summary in use. I produced this clearing the data fields and submitting the form.

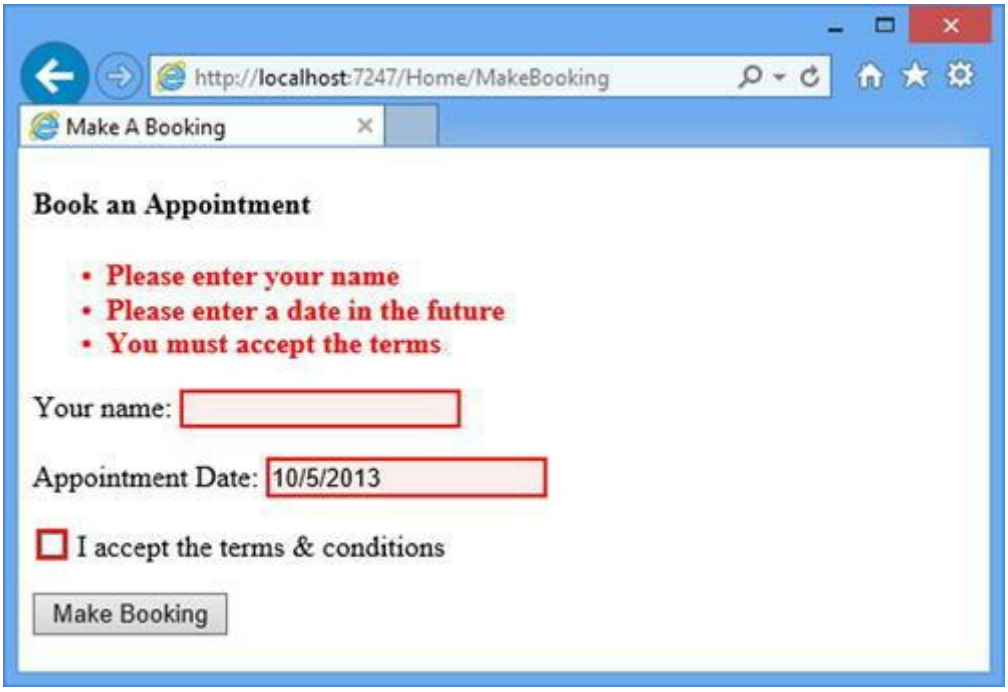


Figure 25-3. Displaying a validation summary

Note The values that I have shown for the `Date` property in this chapter follow the US date format of month/d you are in a different locale, then you can either enter valid dates in your local format (such as day/month/year, whic widely in Europe) or add `<globalization culture="en-US" uiCulture="en-US"/>` to `system.web` element in the `Web.config` file for the example project to force the MVC application to use U formats.

The validation summary displays the error messages that I registered with the `ModelState` in the `MakeBoo` action method. Here is the HTML that the helper method generates:

eBooks-IT.org

```
<div class="validation-summary-errors" data-valmsg-summary="true">
    <ul>
        <li>Please enter your name</li>
        <li>Please enter a date in the future</li>
        <li>You must accept the terms</li>
    </ul>
</div>
...
```

The errors are expressed as a list contained in a `div` element, to which the `validation-summary-er` is applied. This class corresponds to one of the styles that I defined in the `_Layout.cshtml` file when I creat at the start of the chapter:

```
...
.validation-summary-errors { color: #f00; font-weight: bold;}
...
```

There are a number of overloaded versions of the `ValidationSummary` method and Table 25-2 shows th Some of the overloads of the `ValidationSummary` helper method allow me to specify that only *model-level* be displayed. The errors that I have registered with `ModelState` so far have been *property-level errors*, meanin problem with the value supplied for a given property and changing that value can address the problem.

***Table 25-2**. Useful Overloads of the ValidationSummary Helper Method*

Overloaded Method	Description
`Html.ValidationSummary()`	Generates a summary for all validation errors
`Html.ValidationSummary(bool)`	If the `bool` parameter is true, then only model-level errors are displayed (see the explana table). If the parameter is false, then all errors are shown.
`Html.ValidationSummary(string)`	Displays a message (contained in the `string` parameter) before a summary of all the va
`Html.ValidationSummary(bool, string)`	Displays a message before the validation errors. If the `bool` parameter is `true`, only mo will be shown.

By contrast, model-level errors can be used when there is some problem arising from an interaction between two property values. As a simple example, let's imagine that customers named Joe cannot make appointments on Monday 9 shows how I can enforce this rule with an explicit validation check in the `MakeBooking` action method and re problems as model-level validation errors.

***Listing 25-9**.* A Model-Level Validation Error in the HomeController.cs File

```
...
[HttpPost]
public ViewResult MakeBooking(Appointment appt) {

    if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError("ClientName", "Please enter your
    }

    if (ModelState.IsValidField("Date") && DateTime.Now > appt.Dat
            ModelState.AddModelError("Date", "Please enter a date
future");
    }

    if (!appt.TermsAccepted) {
            ModelState.AddModelError("TermsAccepted", "You must ac
terms");
```

```
                    if     (ModelState.IsValidField("ClientName")
ModelState.IsValidField("Date")
                && appt.ClientName  ==  "Joe"   && appt.Date.DayOf
DayOfWeek.Monday) {
            ModelState.AddModelError("", "Joe cannot book appointm
Mondays");
    }

    if (ModelState.IsValid) {
        // statements to store new Appointment in a
        // repository would go here in a real project
        return View("Completed", appt);
    } else {
        return View();
    }
}
...
```

Before I check to see whether Joe is trying to book on a Monday, I use the `ModelState.IsValidFiel` ensure that I have valid `ClientName` and `Date` values to work with. This means I will not generate a model-le unless the previous checks on the properties have been successful. I register a model-level error by passing the empt as the first parameter to the `ModelState.AddModelError` method, like this:

```
...
ModelState.AddModelError("" , "Joe cannot book appointments on Mon
...
```

I can then update the `MakeBooking.cshtml` view file to use the version of the `ValidationSumma` method that takes a `bool` parameter to display only the model-level errors, as shown in Listing 25-10.

Listing 25-10. Display Only Model-Level Errors in the MakeBooking.cshtml File

```
@model ModelValidation.Models.Appointment
@{
    ViewBag.Title = "Make A Booking";
}
<h4>Book an Appointment</h4>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
       <p>@Html.EditorFor(m  =>  m.TermsAccepted)  I  accept  the
conditions</p>
    <input type="submit" value="Make Booking" />
}
```

You can see the result of these changes in Figure 25-4, where I have entered the name `Joe` and specified a date, Monday.

eBooks-IT.org

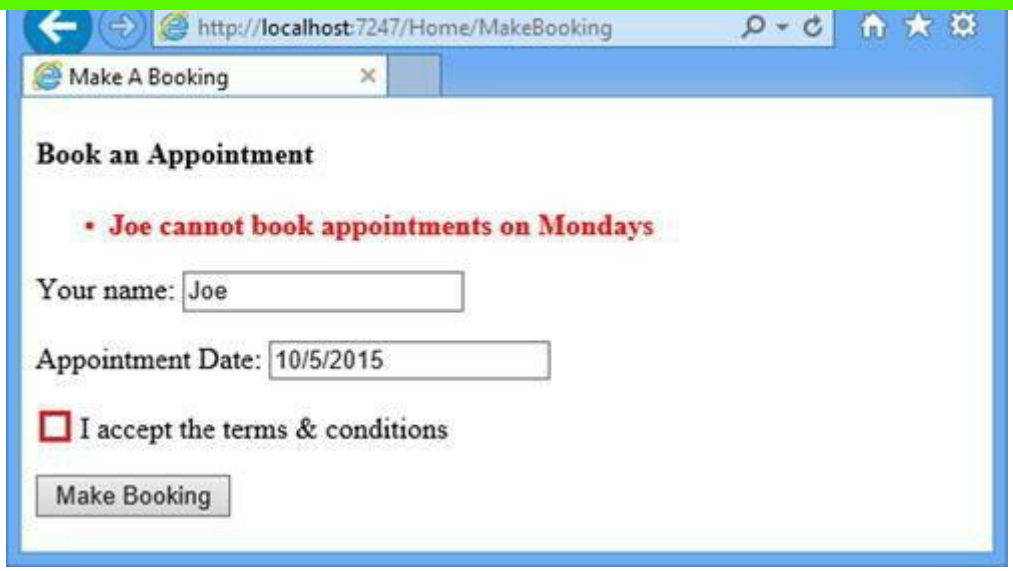


Figure 25-4. Displaying validation summary information for model-level errors

You can see from the figure that there are two validation errors. The first is the model-level error that arises from get a Monday appointment. The second is that the terms and conditions check box is unchecked. Since I am displayi model-level errors in the validation summary, the user will not see any information about the second problem in the s something I will address in the next section.

Displaying Property-Level Validation Messages

The reason you might want to restrict the validation summary to model-level errors is to display property-level errors fields themselves, in which case you will not want to duplicate the property-specific messages. Listing 25-11 shows updated the `MakeBooking.cshtml` view to display model-level errors in the summary and to display propert alongside the corresponding input field.

Listing 25-11. Using Property-Specific Validation Error Messages in the MakeBooking.cshtml File

```
@model ModelValidation.Models.Appointment
@{
    ViewBag.Title = "Make A Booking";
}
<h4>Book an Appointment</h4>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>@Html.ValidationMessageFor(m => m.Date)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.ValidationMessageFor(m => m.TermsAccepted)</p>
        <p>@Html.EditorFor(m => m.TermsAccepted) I accept the
conditions</p>
    <input type="submit" value="Make Booking" />
}
```

The `Html.ValidationMessageFor` helper displays validation errors for a single model property. You effect it has on the `MakeBooking` view in Figure 25-5.

eBooks-IT.org

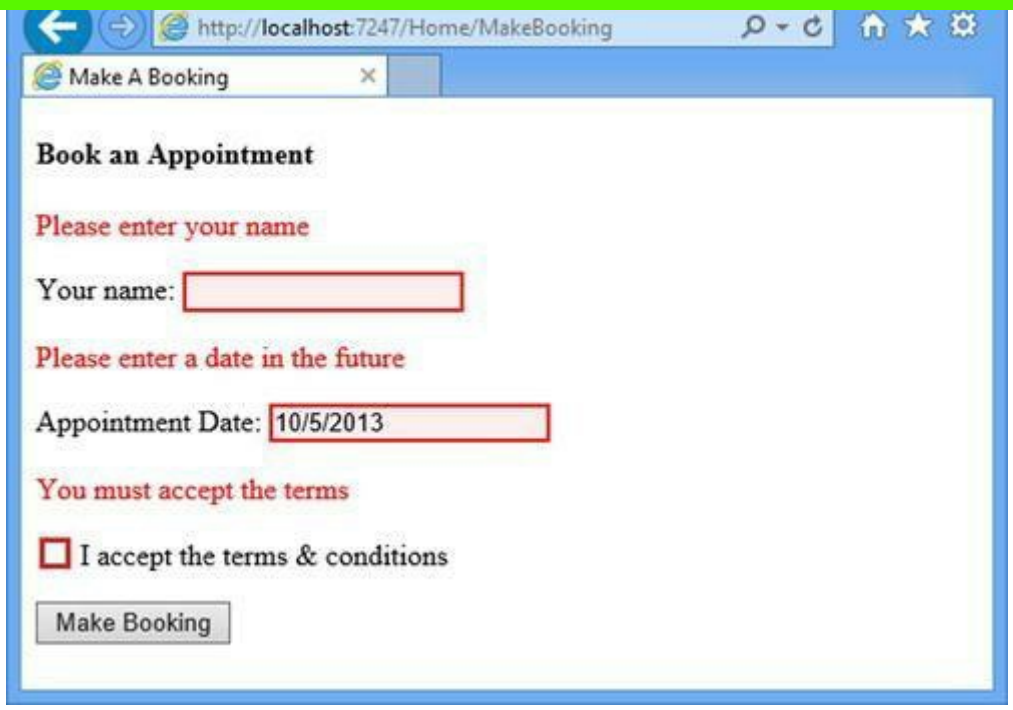


Figure 25-5. Using the per-property validation message helper

The helper only inserts HTML into the response if there is a validation error for the property it is applied to and ge elements like this:

```
...
<p>
    <span class="field-validation-error" data-valmsg-for="ClientNa
            data-valmsg-replace="true">
        Please enter your name
    </span>
</p>
...
```

The class that the elements are assigned to corresponds to the one of the styles that I defined in the `_Layout.` file:

```
...
.field-validation-error { color: #f00;}
...
```

Using Alternative Validation Techniques

Performing model validation in the action method is only one of the validation techniques available in the MVC Frame following sections, I show different approaches.

Performing Validation in the Model Binder

The default model binder performs validation as part of the binding process. As an example, Figure 25-6 shows wha clear the `Date` field and submit the form.

eBooks-IT.org

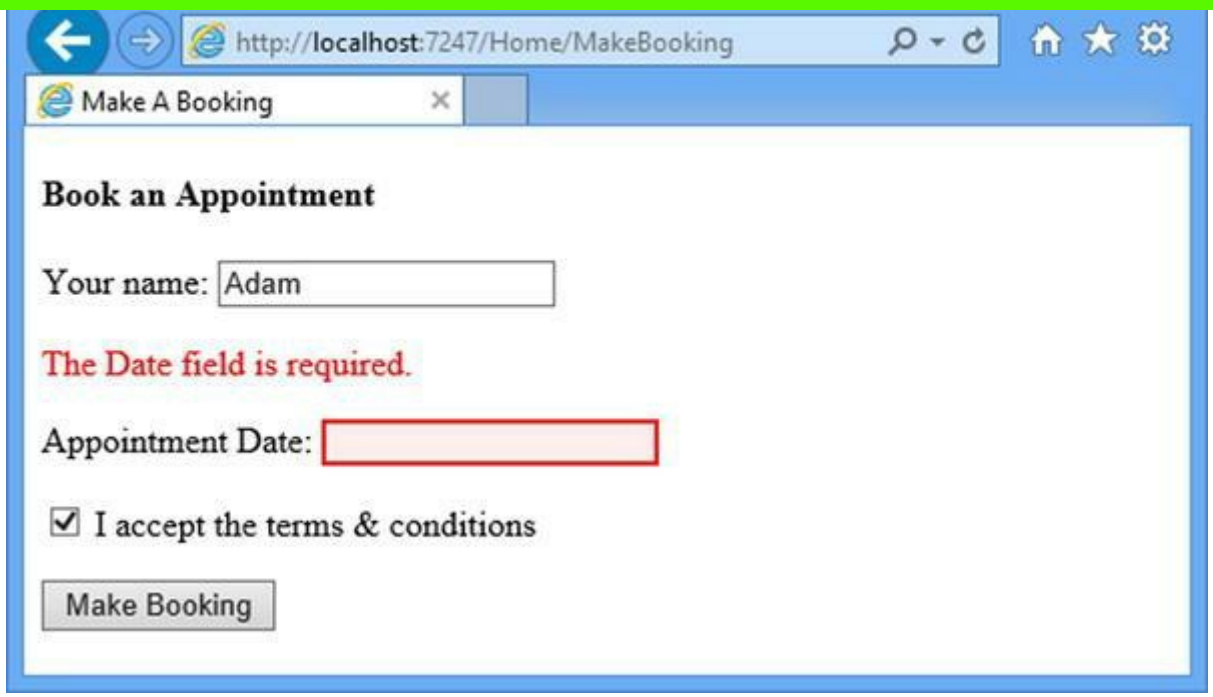


Figure 25-6. A validation message from the model binder

The error displayed for the `Date` field has been added by the model binder because it wasn't able to create a `Da` object from the empty field posted in the form. The model binder performs basic validation for each of the properties object. If a value has not been supplied, the message shown in Figure 25-6 will be displayed. If I supply a value that parsed into the model property type, then a different message is displayed, as shown in Figure 25-7.

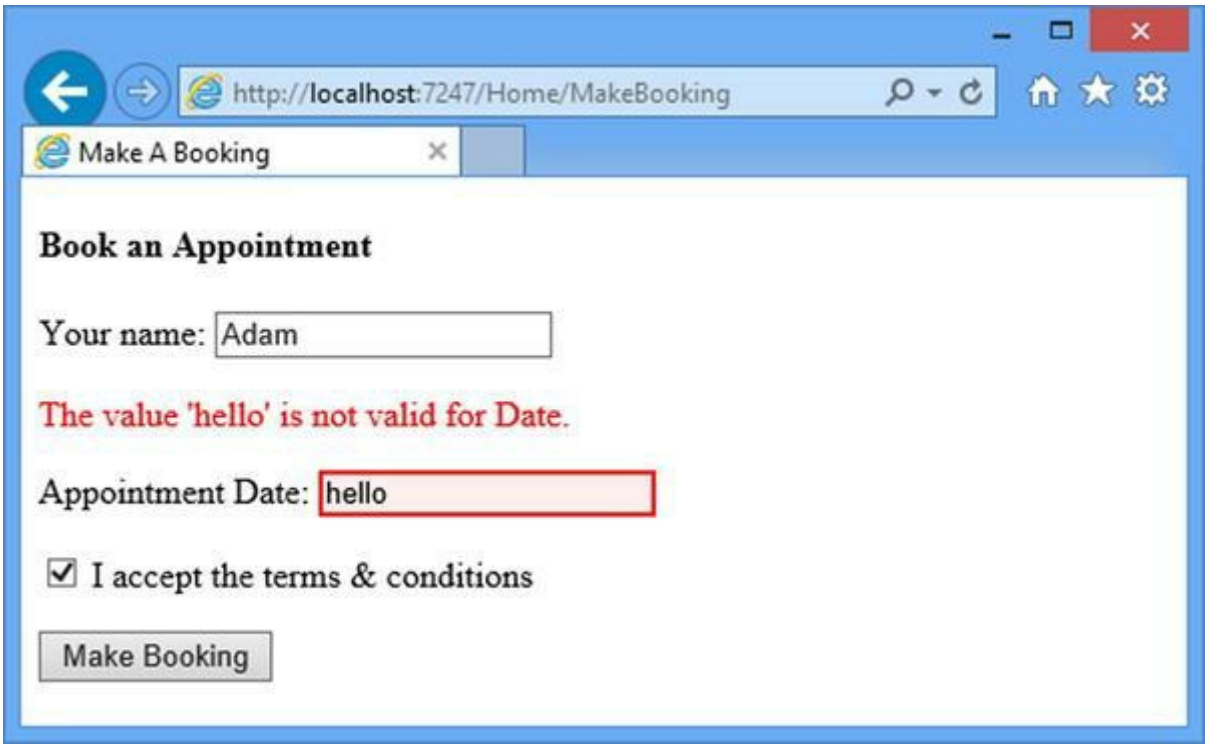


Figure 25-7. A format validation error displayed by the model binder

The built-in default model binder class, `DefaultModelBinder`, provides some useful methods that can be to add validation to a binder. Table 25-3 describes these methods.

***Table 25-3**. DefaultModelBinder Methods for Adding Validation to the Model Binding Process*

eBooks-IT.org

OmModelUpdated	Called when the binder has tried to assign values to all of the properties in the model object	Applies the validation rules defined by the model metadata and registers any errors w ModelState. I describe the use of metadata for validation later in this chapter.
SetProperty	Called when the binder wants to apply a value to a specific property	If the property cannot hold a null value and there was no value to apply, then t <name> field is required error is registered with ModelState. If there cannot be parsed, then the The value <value> is not valid for <na registered.

I can override the methods shown in Table 25-3 to push validation logic into the binding process when creating a c binder, which I demonstrated in Chapter 24. This is not a technique I like, however, because it feels like the wrong p MVC pattern to put the validation logic—although, as with so much in an MVC application, it is a matter of personal preference. I prefer to handle validation using metadata applied to the model class, which I demonstrate in the next se

Specifying Validation Rules Using Metadata

The MVC Framework supports the use of metadata to express model validation rules. The advantage of using metada validation rules are enforced anywhere that the binding process is applied throughout the application, not just in a sing method. The validation attributes are detected and enforced by the built-in default model binder class, DefaultModelBinder, which I described in Chapter 24. In Listing 25-12, you can see how I have applied s validation attributes to the Appointment model class.

Listing 25-12. Validation Rules in the Appointment.cs File

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {
    public class Appointment {

        [Required]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        [Required(ErrorMessage="Please enter a date")]
        public DateTime Date { get; set; }

           [Range(typeof(bool), "true", "true", ErrorMessage = "Y
accept the terms")]
        public bool TermsAccepted { get; set; }
    }
}
```

I used two validation attributes in the listing: Required and Range. The Required attribute specifies tha validation error if the user doesn't submit a value for a property. The Range attribute specifies a subset of accepta Table 25-4 shows the set of built-in validation attributes available in an MVC application.

Table 25-4. *The Built-in Validation Attributes*

Attribute	Example	Description
Compare	[Compare("MyOtherProperty")]	Two properties must have the same value. This is useful when y to provide the same information twice, such as an e-mail addres password.
Range	[Range(10, 20)]	A numeric value (or any property type that implement ICo must not lie beyond the specified minimum and maximum valu boundary on only one side, use a MinValue or MaxValue co

eBooks-IT.org

RegularExpression	[RegularExpression("pattern")]	A string value must match the specified regular expression patt the pattern has to match the *entire* user-supplied value, not within it. By default, it matches case sensitively, but you can m insensitive by applying the (?i) modifier—that is, [RegularExpression("(?i)mypattern")].
Required	[Required]	The value must not be empty or be a string consisting only of s want to treat whitespace as valid, use [Required(AllowEmptyStrings = true)].
StringLength	[StringLength(10)]	A string value must not be longer than the specified maximum also specify a minimum length: [StringLength(10, MinimumLength=2)].

All of the validation attributes support specifying a custom error message by setting a value for the ErrorMess property, like this:

```
...
[Required(ErrorMessage="Please enter a date")]
...
```

If there is no custom error message, then the default messages will be used, such as the ones I showed you earlier chapter. The built-in validation attributes are basic, and they only do property-level validation. Even so, some sleight required to get things working consistently. As an example, consider the validation attribute I applied to the TermsA property:

```
...
[Range(typeof(bool), "true", "true", ErrorMessage="You must acc
terms")]
...
```

I want to make sure that the user checks the box to accept the terms. I cannot use the Required attribute, bec templated helper for bool values generates a hidden HTML element to ensure that I get a value even when the box checked. To work around this, I use a feature of the Range attribute that lets me provide a Type and specify the lower bounds as string values. By setting both bounds to true, I create the equivalent of the Required attribut properties that are edited using check boxes.

Tip The DataType attribute cannot be used to validate user input, only to provide hints for rendering values u templated helpers (described in Chapter 22). So, for example, do not expect the DataType(DataType.EmailAddress) attribute to enforce a specific format.

Creating a Custom Property Validation Attribute

The trick of using the Range attribute to re-create the behavior of the Required attribute is a little awkward. F validation isn't limited to just the built-in attributes; I can also create my own by deriving from the ValidationAttribute class and implementing custom validation logic. This is a lot more useful and to de how this works, I have added an Infrastructure folder to the example project and created a class file called MustBeTrueAttribute.cs within it. Listing 25-13 shows the contents of the new class file.

Listing 25-13. A Custom Property Validation Attribute in the MustBeTrueAttribute.cs File

```
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Infrastructure {
    public class MustBeTrueAttribute : ValidationAttribute {

        public override bool IsValid(object value) {
            return value is bool && (bool)value;
```

```
    }
}
```

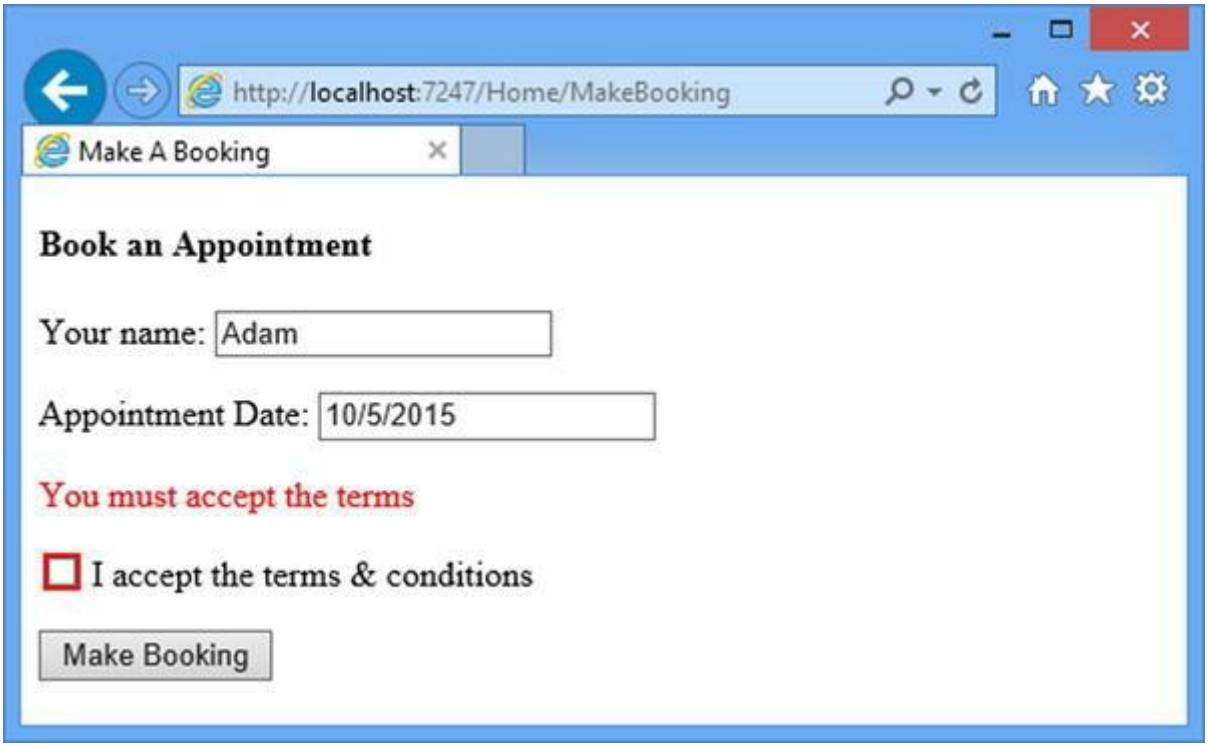
This class defines a new attribute that I have called `MustBeTrueAttribute` and which overrides the `Is` method of the base class. This is the method that the model binder will call to validate properties to which the attribu passing in the value that the user has provided as the parameter.

The validation logic is simple; a value is valid if it is a `bool` that has a value of `true`. I indicate that a value is v returning `true` from the `IsValid` method. In Listing 25-14, you can see how I have replaced the `Range` attr custom `MustBeTrue` attribute in the `Appointment` class.

Listing 25-14. Applying a Custom Validation Attribute in the Appointment.cs File

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;

namespace ModelValidation.Models {
    public class Appointment {

        [Required]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        [Required(ErrorMessage="Please enter a date")]
        public DateTime Date { get; set; }

        [MustBeTrue(ErrorMessage="You must accept the terms")]
        public bool TermsAccepted { get; set; }
    }
}
```

This is neater and easier to make sense of than abusing the `Range` attribute. You can see the effect of the custo validation attribute in Figure 25-8.



Deriving from the Built-In Validation Attributes

In the previous example, I built a validation attribute from scratch, but I can also derive new classes from the built-in which gives me the ability to extend their behavior. In Listing 25-15, you can see the contents of a new class file call `FutureDateAttribute.cs` that I added to the `Infrastructure` folder.

Listing 25-15. The Contents of the FutureDateAttribute.cs Class File

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Infrastructure {
    public class FutureDateAttribute : RequiredAttribute {

        public override bool IsValid(object value) {
                        return  base.IsValid(value)  &&  ((DateTime)v
DateTime.Now;
        }
    }
}
```

I have derived the new `FutureDataAttribute` class from `RequiredAttribute` and overridden `IsValid` method to validate that the date is in the future. Since I have called the base implementation of the `IsV` method, the custom attribute will perform all of the basic validation steps contained in the `Required` attribute. Yo how I have applied the new attribute to the `Appointment` model class in Listing 25-16.

Listing 25-16. Applying a Custom Model Validation Attribute in the Appointment.cs File

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;
using System.Web.Mvc;

namespace ModelValidation.Models {
    public class Appointment {

        [Required]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        [FutureDate(ErrorMessage="Please enter a date in the futur
        public DateTime Date { get; set; }

        [MustBeTrue(ErrorMessage="You must accept the terms")]
        public bool TermsAccepted { get; set; }
    }
}
```

Creating a Model Validation Attribute

The custom validation attributes I have created so far are applied to individual model properties and this means they a to raise property-level validation errors. I can use attributes to validate the entire model as well, which allows me to r

eBooks-IT.org

the Infrastructure folder. The contents of the new file are shown in Listing 25-17.

Listing 25-17. The Contents of the NoJoeOnMondaysAttribute.cs File

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Models;

namespace ModelValidation.Infrastructure {
    public class NoJoeOnMondaysAttribute : ValidationAttribute {

        public NoJoeOnMondaysAttribute() {
            ErrorMessage = "Joe cannot book appointments on Monday
        }

        public override bool IsValid(object value) {
            Appointment app = value as Appointment;
            if (app == null || string.IsNullOrEmpty(app.ClientName
                    app.Date == null) {
                // I don't have a model of the right type to vali
I don't have
                  // the values for the ClientName and Date prop
require
                return true;
            } else {
                return !(app.ClientName == "Joe" &&
                    app.Date.DayOfWeek == DayOfWeek.Monday);
            }
        }
    }
}
```

When I apply a validation attribute to the model class, as opposed to a single property, the `object` parameter th binder will pass to the `IsValid` method will be the model object—an `Appointment` in this example. The val attribute checks to make sure that I really do have an `Appointment` object and, if so, that I have values for the `ClientName` and `Date` properties that I can work with. If I have the data I need, then I make sure that Joe isn get a booking on a Monday. In Listing 25-18, you can see how I applied the custom attribute to the `Appointmen`

Listing 25-18. Applying a Model-Level Custom Validation Attribute in the Appointment.cs File

```
using System;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;

namespace ModelValidation.Models {
    [NoJoeOnMondays]
    public class Appointment {

        [Required]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        [FutureDate(ErrorMessage="Please enter a date in the futur
        public DateTime Date { get; set; }
```

eBooks-IT.org

```
        [MustBeTrue(ErrorMessage = "You must accept the terms")]
        public bool TermsAccepted { get; set; }
    }
}
```

At this point, I am performing the same kinds of validation in the action method and the validation attributes, whicl the user will see two similar error messages for the same validation problem. To resolve this, I have removed the exp checks from the `MakeBooking` action method in the `Home` controller, as shown in Listing 25-19, which has th making the validation attributes solely responsible for performing the custom validation checks.

Listing 25-19. Removing the Explicit Validation Checks from the HomeController.cs File

```
using System;
using System.Web.Mvc;
using ModelValidation.Models;

namespace ModelValidation.Controllers {
    public class HomeController : Controller {

        public ViewResult MakeBooking() {
            return View(new Appointment { Date = DateTime.Now });
        }

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt) {
            if (ModelState.IsValid) {
                // statements to store new Appointment in a
                // repository would go here in a real project
                return View("Completed", appt);
            } else {
                return View();
            }
        }
    }
}
```

An important point to note is that model-level validation attributes will not be used when a property-level problem i see how this works, start the application and navigate to the `/Home/MakeBooking` URL. Enter `Joe` as the r `10/5/2020` as the date, and leave the check box unchecked. When you submit the form, you will see only the w the check box. Check the box and submit again. Only now will you see the model-level error, as illustrated in Figure

eBooks-IT.org

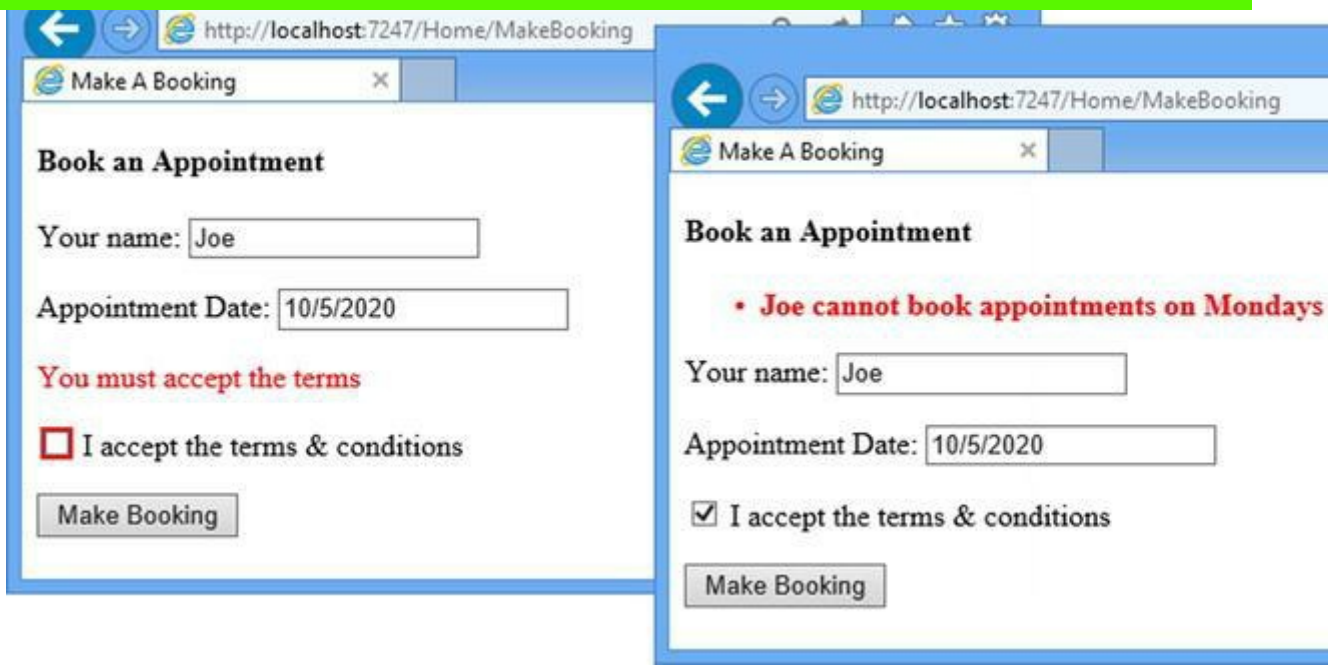


Figure 25-9. Property-level errors being displayed before model-level errors

The problem from the user's perspective is that I have implicitly accepted the name and data values by not flaggin for them in the first panel. This may seem like a minor issue, but it is worth paying careful attention to any situation t frustrate users.

Defining Self-Validating Models

Another validation technique is to create *self-validating models*, where the validation logic is part of the model class. validating model implements the `IValidatableObject` interface, as shown in Listing 25-20.

Listing 25-20. Adding Self-Validation to the Appointment.cs File

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using ModelValidation.Infrastructure;

namespace ModelValidation.Models {
    public class Appointment : IValidatableObject {

        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        public DateTime Date { get; set; }

        public bool TermsAccepted { get; set; }

        public IEnumerable<ValidationResult> Validate(ValidationCo
                validationContext) {

            List<ValidationResult> errors = new List<ValidationRes

            if (string.IsNullOrEmpty(ClientName)) {
                    errors.Add(new ValidationResult("Please ent
```

eBooks-IT.org

```
            }

            if (DateTime.Now > Date) {
                  errors.Add(new ValidationResult("Please enter a
the future"));
            }

            if (errors.Count == 0 && ClientName == "Joe"
                && Date.DayOfWeek == DayOfWeek.Monday) {

                errors.Add(
                     new ValidationResult("Joe cannot book appoint
Mondays"));
            }

            if (!TermsAccepted) {
                     errors.Add(new ValidationResult("You must ac
terms"));
            }

            return errors;
        }
    }
}
```

The `IValidatableObject` interface defines one method, `Validate`. This method takes a `ValidationContext` parameter, although this type isn't MVC-specific and isn't a great deal of use. The res `Validate` method is an enumeration of `ValidationResult` objects, each of which represents a validatio

If the model class implements the `IValidatableObject` interface, then the `Validate` method will b the model binder has assigned values to each of the model properties. This approach has the benefit of combining the putting the validation logic in the action method, but with the consistency of being applied any time the model binding creates an instance of the model type.

One benefit of this approach is that the model- and property-level validation is combined in one place, which mean the errors are displayed together, as shown in Figure 25-10. Some programmers don't like putting the validation logic class, but I think it sits nicely in the MVC design pattern—and I like the flexibility and consistency, of course.

eBooks-IT.org

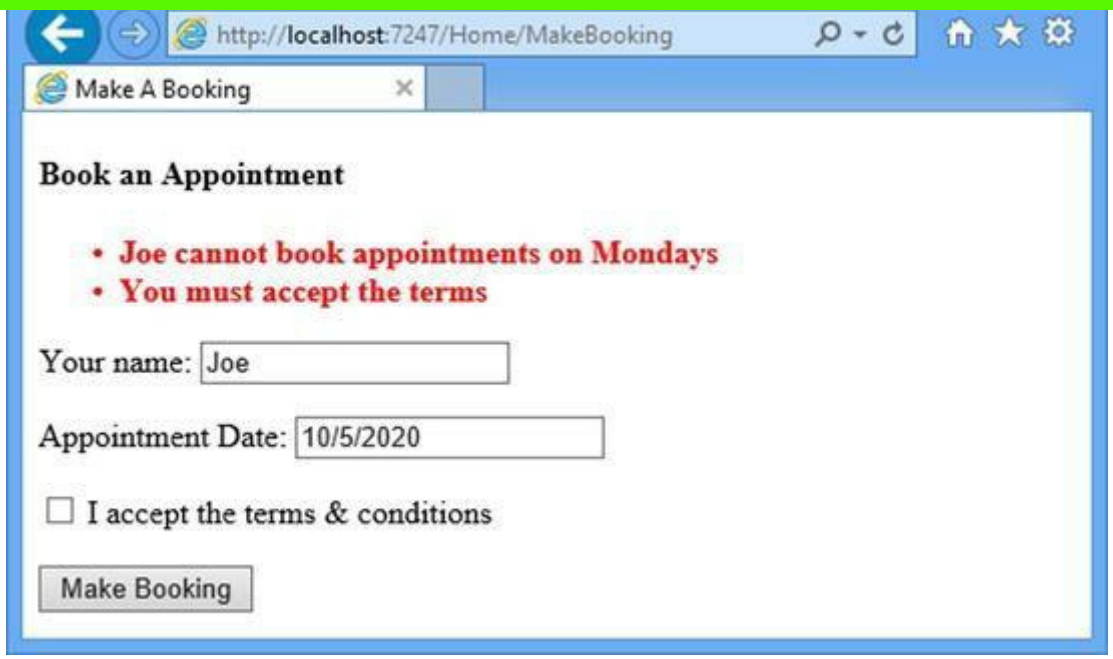


Figure 25-10. The effect of a self-validating model class

Performing Client-Side Validation

The validation techniques I have demonstrated so far have all been examples of server-side validation. This means the their data to the server, and the server validates the data and sends back the results of the validation (either success i the data or a list of errors that need to be corrected).

In Web applications, users typically expect immediate validation feedback—without having to submit anything to t This is known as *client-side validation* and is implemented using JavaScript. The data that the user has entered is va being sent to the server, providing the user with immediate feedback and an opportunity to correct any problems.

The MVC Framework supports unobtrusive client-side validation. The term *unobtrusive* means that validation rules expressed using attributes added to the HTML elements that views generate. These attributes are interpreted by a Jav that is included as part of the MVC Framework that, in turn, configures the jQuery Validation library, which does the validation work. In the following sections, I will show you how the built-in validation support works and demonstrat extend the functionality to provide custom client-side validation.

Tip Client-side validation is focused on validating individual properties. In fact, it is hard to set up model-level clie validation using the built-in support that comes with the MVC Framework. To that end, most MVC applications use c validation for property-level issues and rely on server-side validation for the overall model.

Enabling Client-Side Validation

Client-side validation is controlled by two settings in the `Web.config` file, as shown in Listing 25-21.

Listing 25-21. Controlling Client-Side Validation in the Web.config File

```
...
<appSettings>
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
...
```

eBooks-IT.org

Studio created these entries and set them to true.

Tip You can also configure client-side validation on a per-view basis by setting the `HtmlHelper.ClientValidationEnabled` and `HtmlHelper.UnobtrusiveJavaScriptEnabled` in block.

Adding the NuGet Packages

Ensuring that the MVC Framework will generate the attributes required for validation is only part of the setup proces to add the JavaScript packages that process those attributes and check the data that the user has entered into the form required packages are available through NuGet, so select `Package Manager Console` from the `Visua Tools` ➤ `Library Package Manager` menu and enter the following commands:

```
Install-Package jQuery -version 1.10.2
Install-Package jQuery.Validation -version 1.11.1
Install-Package Microsoft.jQuery.Unobtrusive.Validation -version 3
```

These packages add files to the `Scripts` folder, which I then need to add to the layout with `script` elemen in Listing 25-22.

Listing 25-22. Adding Script Elements for the Validation Libraries to the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style type="text/css">
        .field-validation-error { color: #f00;}
        .validation-summary-errors { color: #f00; font-weight: bol
            .input-validation-error { border: 2px solid #f00; bac
color: #fee; }
        input[type="checkbox"].input-validation-error { outline: 2
#f00; }
    </style>
    <script src="~/Scripts/jquery-1.10.2.js"></script>
    <script src="~/Scripts/jquery.validate.js"></script>
    <script src="~/Scripts/jquery.validate.unobtrusive.js"></scrip
</head>
<body>
    @RenderBody()
</body>
</html>
```

Tip In Chapter 26, I'll show you the bundles feature, which makes it easier to manage JavaScript and CSS files i

The order in which the script elements are added to the layout is important. You must add the jQuery library fi by the jQuery Validation library and only then can you add the Microsoft unobtrusive validation library.

Using Client-Side Validation

Once I have enabled client-side validation and ensured that the JavaScript libraries are referenced in the layout, I can perform client-side validation. The simplest way of doing this is to apply the metadata attributes that I previously use

eBooks-IT.org

class with these annotations applied. (I have removed the implementation of the `IValidatableObject` inter
has no effect on client-side validation.)

Listing 25-23. Validation Attributes Applied in the Appointment.cs File

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models {
    public class Appointment {

        [Required]
        [StringLength(10, MinimumLength = 3)]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        public DateTime Date { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

That's all I have to do to get the basic client-side validation working. I have applied a slightly different mix of the b
validation attributes so that I can demonstrate some of the client-side validation features. But once you have the JavaS
included in the HTML that is sent to the client, everything just starts to work. You can see the effect of the client-sid
by starting the application, navigating to the `/Home/MakeBooking` URL and entering the letter `X` into the nam
the tab key or click one of the other input elements and you will immediately see a validation message produced by th
running in the browser, as shown in Figure 25-11.

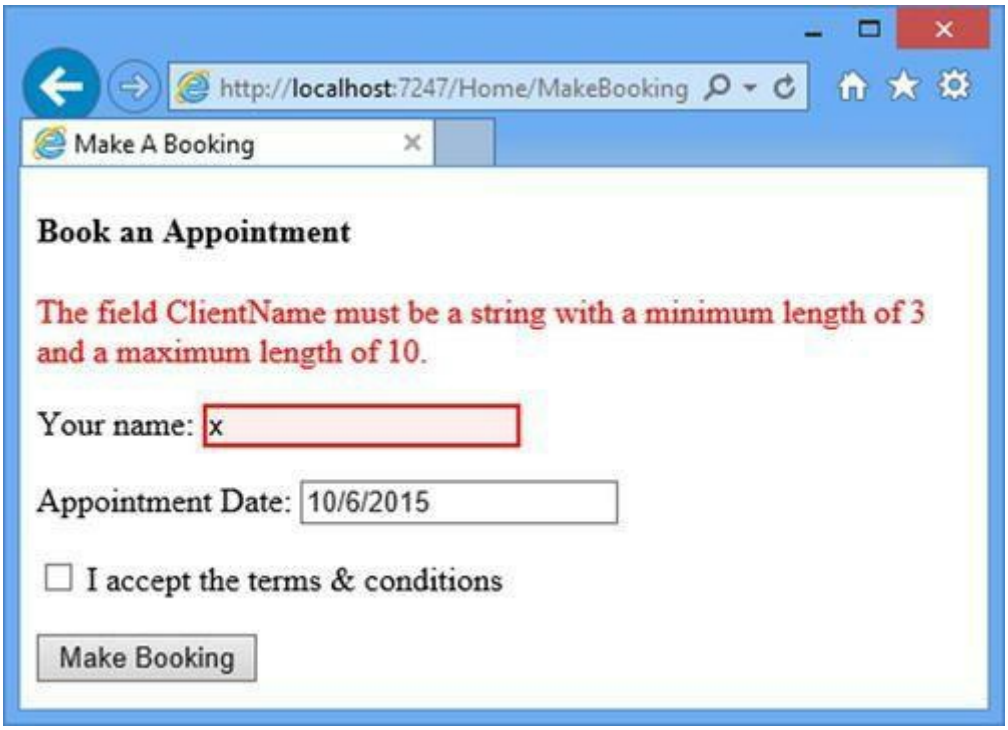


Figure 25-11. Immediate feedback from the client-side validation feature

I applied the `StringLength` validation attribute to the `Appointment` class in Listing 25-23 and it is the
message from that attribute that you can see in the figure. The feedback presented in the browser is immediate and n
been made to the server. In fact, the JavaScript code that is performing the validation will prevent the form from bein
until there are no outstanding validation errors.

eBooks-IT.org

yping, t
error will be removed when the name you have entered is three or more characters long. But if you keep typing until
eleventh character, you will see the error reappear. This is because I specified a minimum length of three characters
maximum length of ten for the `ClientName` property with the `StringLength` attribute. This kind of dyna
helps the user to provide the data that the application requires without having to submit the form to the server and wa
response.

Understanding How Client-Side Validation Works

One of the benefits of using the MVC Framework client-side validation feature is that I do not have to write any Java
Instead, the validation rules are expressed using HTML attributes. Here is the HTML that is rendered by the `Html.EditorFor` helper for the `ClientName` property when client-side validation is disabled:

```
...
<input  class="text-box  single-line"  id="ClientName"  name="Cli
type="text"
    value="" />
...
```

And here is the HTML rendered for the same property when client-side validation is switched on:

```
...
<input class="text-box single-line" data-val="true"
    data-val-length="The field ClientName must be a string with a
length of 3 and
        a maximum length of 10." data-val-length-max="10" data-val
min="3"
    data-val-required="The ClientName field is required." id="Clie
    name="ClientName" type="text" value="" />
...
```

The MVC client-side validation support doesn't generate any JavaScript or JSON data to direct the validation proc
of the rest of the MVC Framework, validation relies on convention. The first attribute that was added is `data-va`
jQuery Validation library identifies those fields that require validation by looking for this attribute.

Individual validation rules are specified using an attribute in the form `data-val-` <*name*>, where `name` is th
applied. So, for example, the `Required` attribute I applied to the model class has resulted in a `data-val-re`
attribute in the HTML. The value associated with the attribute is the error message associated with the rule. Some ru
additional attributes. You can see this with the length rule, which has `data-val-length-min` and `data-length-max` attributes to let me specify the minimum and maximum string lengths that are allowed. The interpr
required and length validation rules is provided by the jQuery Validation library, on which the MVC client validation f
built.

AVOIDING CONFLICTS WITH BROWSER VALIDATION

Some of the current generation of HTML5 browsers support simple client side validation based on the attributes
`input` elements. The general idea is that, say, an `input` element to which the `required` attribute has b
for example, will cause the browser to display a validation error when the user tries to submit the form without
value.

If you are generating form elements from models, as I have been doing in this chapter, then you won't have any
with browser validation because the MVC Framework generates and uses data attributes to denote validation rul
for example, an `input` element that must have a value is denoted with the `data-val-required` attrib
browsers do not recognize).

However, you may run into problems if you are unable to completely control the markup in your application, so
often happens when you are passing on content generated elsewhere. The result is that the jQuery validation and
validation can both operate on the form, which is just confusing to the user. To avoid this problem, you can add
`novalidate` attribute to the `form` element.

eBooks-IT.org

dation r
applied at the client and at the server. This means that data from browsers that do not support JavaScript are subject
validation as those that do, without requiring any additional effort.

MVC CLIENT VALIDATION VERSUS JQUERY VALIDATION

The MVC client-validation features are built on top of the jQuery Validation library. If you prefer, you can use th library directly and ignore the MVC features. The Validation library is flexible and feature-rich. It is well worth e only to understand how to customize the MVC features to take best advantage of the available validation options jQuery Validation library in depth in my *Pro jQuery* 2.0 book, also published by Apress.

Performing Remote Validation

The last validation feature I will look at in this chapter is remote validation. This is a client-side validation technique th action method on the server to perform validation.

A common example of remote validation is to check whether a username is available in applications when such nar unique; the user submits the data, and the client-side validation is performed. As part of this process, an Ajax request the server to validate the username that has been requested. If the username has been taken, a validation error is displ the user can enter another value.

This may seem like regular server-side validation, but there are some benefits to this approach. First, only some pr be remotely validated; the client-side validation benefits still apply to all the other data values that the user has entered request is relatively lightweight and is focused on validation, rather than processing an entire model object.

The third difference is that the remote validation is performed in the background. The user doesn't have to click th button and then wait for a new view to be rendered and returned. It makes for a more responsive user experience, es there is a slow network between the browser and the server.

That said, remote validation is a compromise. It strikes a balance between client-side and server-side validation, bu require requests to the application server, and it is not as quick to validate as normal client-side validation.

The first step toward using remote validation is to create an action method that can validate one of the model prop going to validate the `Date` property of the `Appointment` model to ensure that the requested appointment is in (This is one of the original validation rules I used at the start of the chapter, but which isn't possible to validate using client-side validation features.) In Listing 25-24, you can see the `ValidateDate` action method that I added to controller.

Listing 25-24. Adding a Validation Action Method to the HomeController.cs File

```
using System;
using System.Web.Mvc;
using ModelValidation.Models;

namespace ModelValidation.Controllers {
    public class HomeController : Controller {

        public ViewResult MakeBooking() {
            return View(new Appointment { Date = DateTime.Now });
        }

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt) {
            if (ModelState.IsValid) {
                // statements to store new Appointment in a
                // repository would go here in a real project
                return View("Completed", appt);
            } else {
                return View();
            }
```

```
        public JsonResult ValidateDate(string Date) {
            DateTime parsedDate;

            if (!DateTime.TryParse(Date, out parsedDate)) {
                return Json("Please enter a valid date (mm/dd/yyyy
                    JsonRequestBehavior.AllowGet);
            } else if (DateTime.Now > parsedDate) {
                return Json("Please enter a date in the future",
                    JsonRequestBehavior.AllowGet);
            } else {
                return Json(true, JsonRequestBehavior.AllowGet);
            }
        }
    }
}
```

Actions methods that support remote validation must return the `JsonResult` type, which tells the MVC Fram am working with JSON data, as explained in Chapter 23. In addition to the result, validation action methods must def parameter that has the same name as the data field being validated: this is `Date` for the example. I make sure that I `DateTime` object from the value that the user has submitted and, if I can, check to see that the date is in the futu

Tip I could have taken advantage of model binding so that the parameter to my action method would be a `Date` object, but doing so would mean that the validation method wouldn't be called if the user entered a nonsense value lik for example. This is because the model binder wouldn't have been able to create a `DateTime` object from `appl` an exception when it tries. The remote validation feature doesn't have a way to express that exception and so it is qu discarded. This has the unfortunate effect of *not* highlighting the data field and so creating the impression that the va user has entered is valid. As a general rule, the best approach to remote validation is to accept a `string` parameter method and perform any type conversion, parsing, or model binding explicitly.

I express validation results using the `Json` method, which creates a JSON-formatted result that the client-side re validation script can parse and process. If the value that I am processing meets my requirements, then I pass `true` parameter to the `Json` method, like this:

```
...
return Json(true , JsonRequestBehavior.AllowGet);
...
```

If I am unhappy with the value, I pass the validation error message that the user should see as the parameter, like t

```
...
return Json("Please enter a date in the futu
JsonRequestBehavior.AllowGet);
...
```

In both cases, I must also pass the `JsonRequestBehavior.AllowGet` value as a parameter. This is MVC Framework disallows `GET` requests that produce JSON by default, and I have to override this behavior to han validation request. Without this additional parameter, the validation request will quietly fail, and no validation errors w displayed to the client.

To use the remote validation method, I apply the `Remote` attribute to the property I want to validate in the mode Listing 25-25, you can see how I have applied the attribute to the `Date` property.

Listing 25-25. Using the Remote Attribute in the Appointment.cs File

```
using System;
```

eBooks-IT.org

```
using System.Web.Mvc;

namespace ModelValidation.Models {
    public class Appointment {

        [Required]
        [StringLength(10, MinimumLength = 3)]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        [Remote("ValidateDate", "Home")]
        public DateTime Date { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

The arguments for the attribute are the name of the action and the controller that should be used to generate the U JavaScript validation library will call to perform the validation—in this case, the `ValidateDate` action on the H controller. The actual URL used will be created according to the application routing configuration.

You can see how the remote validation works by starting the application, navigating to the `/Home/MakeBool` and entering a date that is in the past. As you type, you will see the validation message appear, as shown in Figure 25

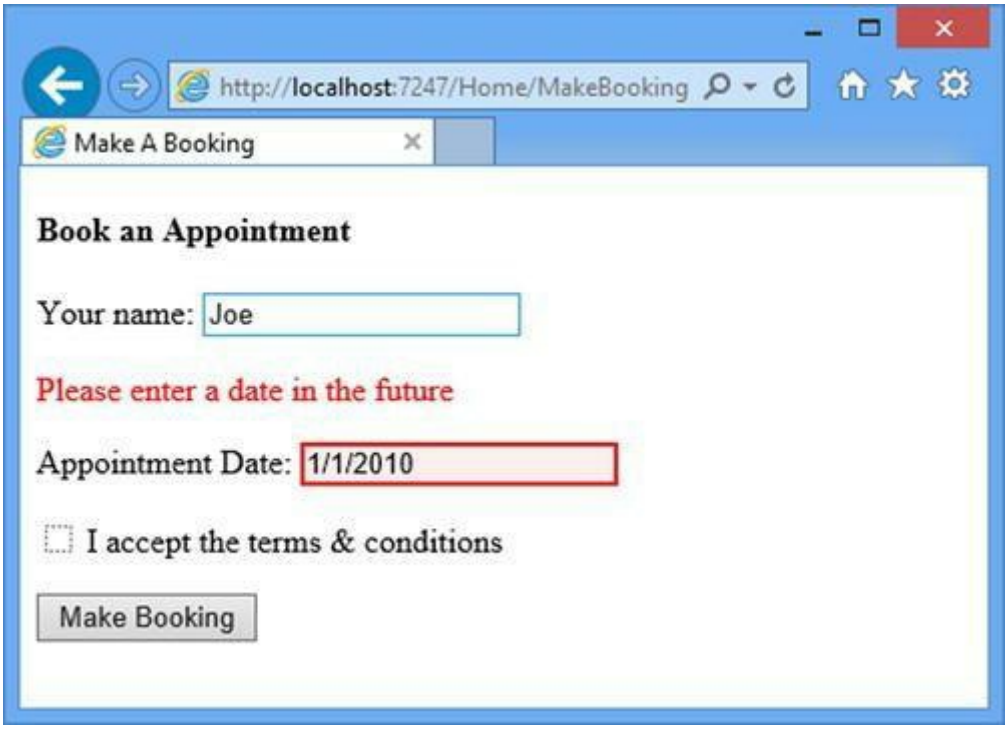


Figure 25-12. Performing remote validation

Caution The validation action method will be called when the user first submits the form and then again each tin edits the data. In essence, every keystroke will lead to a call to the server. For some applications, this can be a signif of requests and must be taken into account when specifying the server capacity and bandwidth that an application re production. Also, you might choose *not* to use remote validation for properties that are expensive to validate (for exa have to query a slow Web service to determine whether a username is unique).

Summary

In this chapter, I examined the wide range of techniques available to perform model validation, ensuring that the data

eBooks-IT.org

Model validation is an important topic, and getting the right validation in place for an application is essential to ensu users have a good and frustration-free experience. Equally important is the fact that the integrity of the domain mode In the next chapter I show you the bundles feature, which is used to manage JavaScript and CSS files.

eBooks-IT.org

Bundles

In this chapter, I am going to look at the *bundles* feature, which the MVC Framework provides to organize and optin
and JavaScript files that views and layouts cause the browser to request from the server. Table 26-1 provides the su
this chapter.

Table 26-1. *Chapter Summary*

Problem	Solution
Define bundles	Create instances of the `StyleBundle` and `ScriptBundle` classes and add them to the bun table.
Prepare an application for bundles	Ensure that the `Views/web.config` file includes a reference for the `System.Web.Optimization` namespace.
Add a bundle to a view or layout	Use the `Styles.Render` and `Scripts.Render` helpers.
Enable concatenation and minification of bundles	Set the `debug` attribute of the `compilation` attribute in the `Web.config` file to `false`

Preparing the Example Application

For this chapter, I have created a new MVC project called `ClientFeatures` using the `Empty` template opti
the option to add the core MVC folders and references.

Adding the NuGet Packages

The bundles feature that I describe in this chapter makes it easier to manage JavaScript and CSS files. To that end, I
install a number of NuGet packages that are commonly used for client-side development. Select `Package Mana`
`Console` from the `Visual Studio Tools` ➤ `Library Package Manager` menu and enter
commands:

```
Install-Package jQuery -version 1.10.2
Install-Package jQuery.Validation -version 1.11.1
Install-Package Microsoft.jQuery.Unobtrusive.Validation -version 3
Install-Package Bootstrap -version 3.0.0
Install-Package Microsoft.jQuery.Unobtrusive.Ajax -version 3.0.0
```

Creating the Model and Controller

I am going to create a variation on the application that I used in the previous chapter, so I started by creating a new c
called `Appointment.cs` in the `Models` folder. You can see the contents of this file in Listing 26-1.

Listing 26-1. The Contents of the Appointment.cs File

```
using System;
using System.ComponentModel.DataAnnotations;
```

eBooks-IT.org

```
    public class Appointment {

        [Required]
        public string ClientName { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

I created a Home controller that operates on the Appointment model class, as shown in Listing 26-2.

Listing 26-2. The Contents of the HomeController.cs File

```
using System;
using System.Web.Mvc;
using ClientFeatures.Models;

namespace ClientFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult MakeBooking() {
            return View(new Appointment {
                ClientName = "Adam",
                TermsAccepted = true
            });
        }

        [HttpPost]
        public JsonResult MakeBooking(Appointment appt) {
            // statements to store new Appointment in a
            // repository would go here in a real project
            return Json(appt, JsonRequestBehavior.AllowGet);
        }
    }
}
```

There are two version of the MakeBooking method in this controller. The version with no parameters create Appointment object and passes it to the View method to render the default view. The HttpPost version MakeBooking method relies on the model binder to create an Appointment object and uses the Json m encode the Appointment and send it back to the client in the JSON format.

I am focused on an MVC Framework feature that support client-side development in this chapter, so I have taken shortcuts in the controller that wouldn't be sensible or useful in a real project. Most importantly, I do not perform an validation when I receive a POST request and just send the details of the object created by the model binder back to as JSON (with no support for HTML responses).

Creating the Layout and View

I created the Views/Shared folder and added a view file called _Layout.cshtml to it, the content of w see in Listing 26-3. The main purpose of this the layout is to import the JavaScript and CSS files that I added via Nu they can be used in views.

Listing 26-3. The Contents of the _Layout.cshtml File

```
@{
```

```
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style>
        .field-validation-error { color: #f00; }
        .validation-summary-errors { color: #f00; font-weight: bol
           .input-validation-error { border: 2px solid #f00; bac
color: #fee; }
        input[type="checkbox"].input-validation-error { outline: 2
#f00; }
        div.hidden { display: none; }
        div.visible { display: block; }
    </style>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <script src="~/Scripts/jquery-1.10.2.js"></script>
    <script src="~/Scripts/jquery.validate.js"></script>
    <script src="~/Scripts/jquery.validate.unobtrusive.js"></scrip
    <script src="~/Scripts/jquery.unobtrusive-ajax.js"></script>
    @RenderSection("Scripts", false)
</head>
<body>
    @RenderSection("Body")
</body>
</html>
```

I defined two Razor sections in the view. The `Scripts` section will allow views to add JavaScript code to the l element section of the HTML response to the server, and the `Body` section allows the view to add content to the b (I explained how Razor sections worked in Chapter 20.) I added a view file called `MakeBooking.cshtml` to `Home/Views` folder, as shown in Listing 26-4.

Listing 26-4. The Contents of the MakeBooking.cshtml File

```
@model ClientFeatures.Models.Appointment

@{
    ViewBag.Title = "Make Appointment";
    Layout = "~/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        OnSuccess = "processResponse"
    };
}

@section Scripts {

<script type="text/javascript">

    function switchViews() {
        $(".hidden, .visible").toggleClass("hidden visible");
```

```
    function processResponse(appt) {
        $('#successClientName').text(appt.ClientName);
        switchViews();
    }

    $(document).ready(function () {
        $('#backButton').click(function (e) {
            switchViews();
        });
    });
</script>

}

@section Body {

<div id="formDiv" class="visible well">
    @using (Ajax.BeginForm(ajaxOpts)) {
        @Html.ValidationSummary(true)
        <div class="form-group">
            <label for="ClientName">Your name:</label>
            <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
               @Html.TextBoxFor(m => m.ClientName, new  {@class =
control" })
        </div>
        <div class="checkbox">
            <label>
                @Html.CheckBoxFor(m => m.TermsAccepted)
                I accept the terms & conditions
            </label>
        </div>
             <input type="submit" value="Make Booking" class="b
primary"/>
    }
</div>

<div id="successDiv" class="hidden well">
    <h4 class="lead">Your appointment is confirmed</h4>
    <p>Your name is: <b id="successClientName"></b></p>
    <button id="backButton" class="btn btn-primary">Back</button>
</div>

}
```

My goal in this view is just to use all of the JavaScript and CSS files that I defined in the layout. To that end, I hav Ajax form that uses the unobtrusive Ajax library (described in Chapter 23) and that relies on the unobtrusive client-si library (described in Chapter 25). Both of these libraries depend on jQuery and I have used Bootstrap CSS classes to content.

I have taken advantage of the `Scripts` section I defined in the layout to include some JavaScript code that resp JSON response from the controller and manipulates the markup to display the results using some simple jQuery. This with a single view for the example.

I want to create a typical scenario for a complex view file without needing to create a complex application, which added lots of JavaScript and CSS files for such a simple example. The key idea is that there are lots of files to be ma you are writing real applications, you will be struck by just how many script and style files you have to deal with in y

eBooks-IT.org

URL. The form is pre-populated with data so that you can just click the `Make Booking` button to submit the fo
the server using Ajax. When the response is received, you will see a summary of the `Appointment` object that
by the model binder from the form data, along with a `button` element that will return you to the form, as illustrate
26-1.

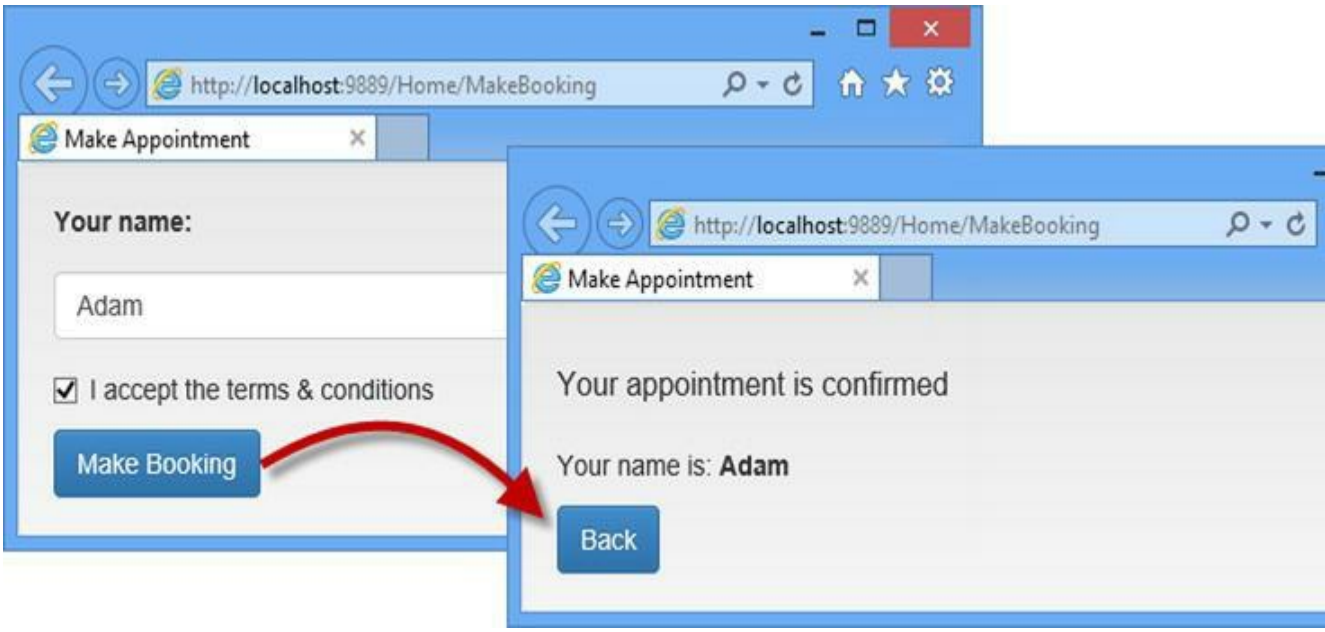


Figure 26-1. Using the example application

The project contains a number of JavaScript and CSS files, which are combined with inline JavaScript code and C
generate HTML for the browser. This is a typical mix that you will encounter in most MVC Framework projects.

Developers tend to write view files just as they would write HTML pages, which is fine but isn't the most effectiv
As I will show you in the sections that follow, there are some hidden problems in the `MakeBooking.cshtml`
I show you a number of improvements in the way that scripts and style sheets are managed.

Profiling Script and Style Sheet Loading

When considering any kind of optimization in any kind of project, you need to start by taking some measurements. I
efficient and optimized applications, but my experience is that people rush to optimize problems that don't have much
in doing so, make design decisions that cause problems later.

For the problems that I am going to look at in this chapter, I am going to take the measurements using the Internet
tools (so called because you access them by pressing the F12 key).

I want to focus just on the HTTP requests that are made in the normal execution of the application, and that mean
Visual Studio Browser Link feature, which works by adding JavaScript code to the HTML sent to the browser, leadi
additional HTTP requests.

Click on the small down arrow next to the Browser Link button on the Visual Studio toolbar and uncheck the `Ena`
`Browser Link` menu item, as shown in Figure 26-2.

eBooks-IT.org

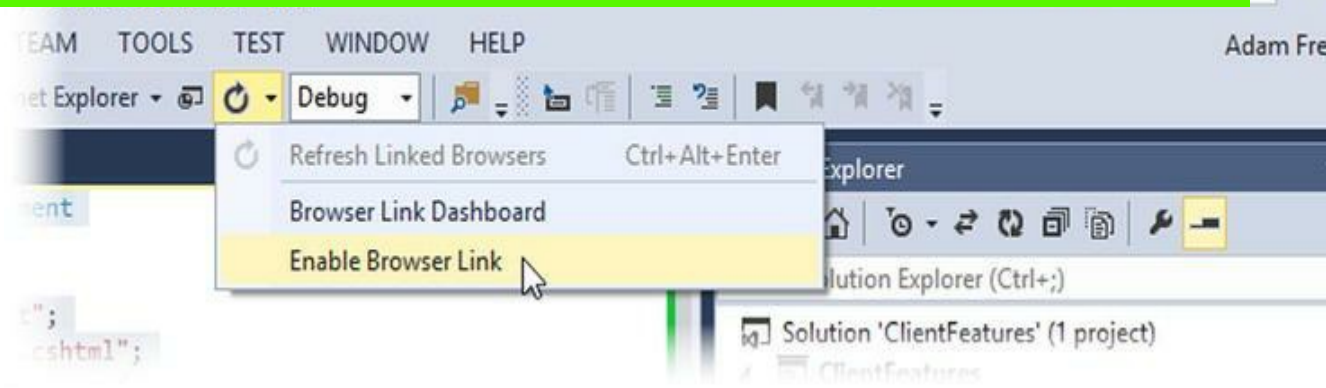


Figure 26-2. Disabling the browser link feature in Visual Studio

Load the application, navigate to the `/Home/MakeBooking` URL, and then press the F12 key. When the too opens, navigate to the `Network` tab and click the green arrow button to start capturing the HTTP requests that th makes. Then click the `Clear Browser Cache` button, which will ensure that the browser requests the con the JavaScript and CSS files that are referenced in the layout. Reload the contents of the browser tab (right-click in t window and select `Refresh`), and you will see the results shown in Figure 26-3.

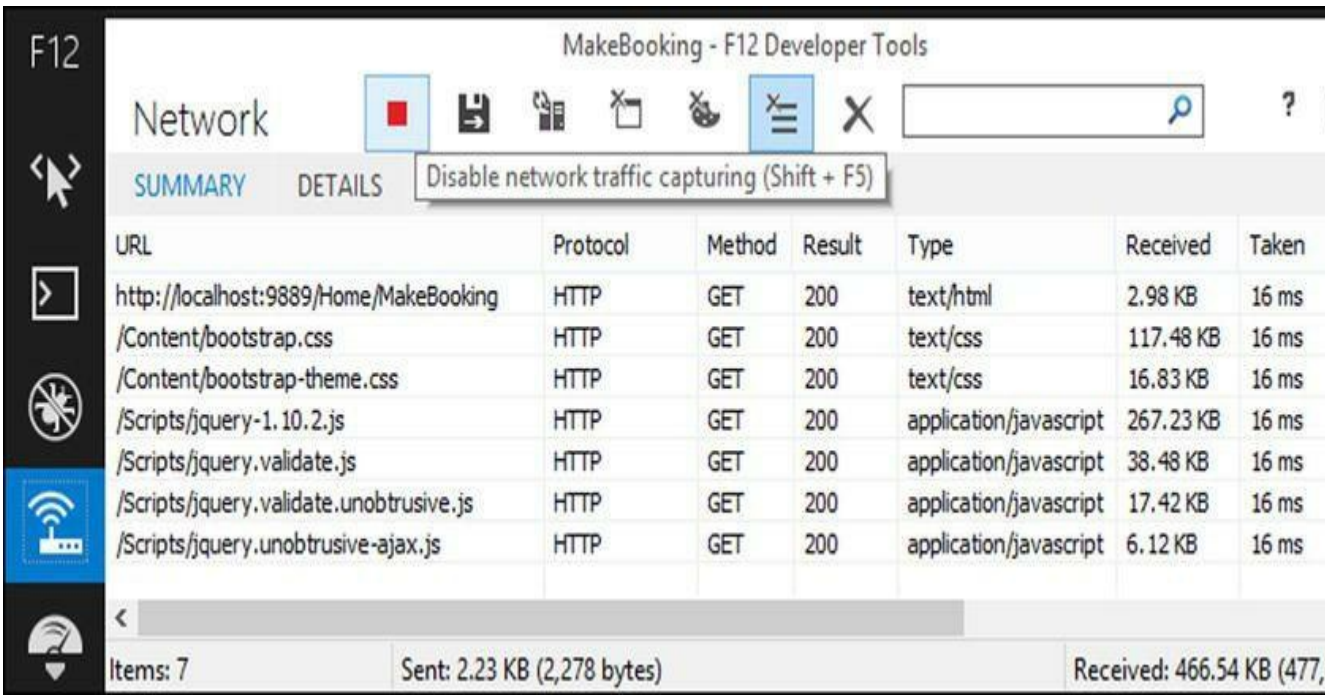


Figure 26-3. Profiling script and style sheet loading for the example application

The F12 tools allow you to profile the network requests that your application makes. (All of the mainstream brows similar developer tools and there are other alternatives. My favorite is Fiddler, which you can get from `www.fiddler2.com`).

So that I can assess the optimizations that I make in this chapter, I will use the data shown in Figure 26-3 as the b are the key figures:

- The browser made seven requests for the `/Home/MakeBooking` URL.
- There were two requests for CSS files.
- There were four requests for JavaScript files.
- A total of 2,278 bytes were sent from the browser to the server.
- A total of 477,733 bytes were sent from the server to the browser

eBooks-IT.org

have done this because it allows me to easily create a measurable starting point, even though I know that real world u
improved by the browser caching files from previous requests.

If I reload the `/Home/MakeBooking` URL without clearing the cache, then I get the following results:

- The browser made seven requests for the `/Home/MakeBooking` URL.
- There were two requests for CSS files.
- There were four requests for JavaScript files.
- A total of 2,086 bytes were sent from the browser to the server.
- A total of 5,214 bytes were sent from the server to the browser.

This is the best-case scenario, where all the requests for CSS and JavaScript files were able to be serviced using p
cached files.

Note In a real project, I would stop at this point and ask myself if I have a problem to solve or if the current stat
application is acceptable. It may seem that 473K is a lot of bandwidth for a simple Web page, but context is everythi
developing an application for Intranet use where bandwidth is cheap and plentiful, and optimizations of any sort are c
the cost of the developer, who could be working on more important projects. Equally, I could be writing an applicati
operates over the Internet with high-value customers in countries with low-speed connections, in which case it is wo
the time to optimize every aspect of the application. The point is that you shouldn't automatically assume that you ha
every optimization into every application. There will often be better things you could be doing. (This is *always* the ca
sneakily optimizing your application without telling anyone. Stealth optimization is a bad idea and will catch up with y
eventually.)

Using Script and Style Bundles

My goal is to turn the JavaScript and CSS files into *bundles*, which allows me to treat several related files as a single
through the steps required to set up and apply bundles in the sections that follow.

Adding the NuGet Package

The bundles feature requires a NuGet package that isn't included in the Visual Studio `Empty` template. Select `Pac`
`Manager Console` from the `Visual Studio Tools` ➤ `Library Package Manager` n
the following command:

```
Install-Package Microsoft.AspNet.Web.Optimization -version 1.1.1
```

Defining the Bundles

The convention is to define bundles in a file called `BundleConfig.cs`, which is placed in the `App_Start`
Listing 26-5, you can see the contents of the `BundleConfig.cs` file that I added to the example project. (You
to create this file yourself if you are using some of the other Visual Studio project templates because it will be added
automatically.)

Listing 26-5. The Contents of the BundleConfig.cs File

```
using System.Web.Optimization;

namespace ClientFeatures {

    public class BundleConfig {

        public static void RegisterBundles(BundleCollection bundle
```

eBooks-IT.org

```
                bundles.Add(new StyleBundle("~/Content/css").Include(
                    "~/Content/*.css"));

                                                                bundles
ScriptBundle("~/bundles/clientfeaturesscripts")
                .Include("~/Scripts/jquery-{version}.js",
                    "~/Scripts/jquery.validate.js",
                    "~/Scripts/jquery.validate.unobtrusive.js",
                    "~/Scripts/jquery.unobtrusive-ajax.js"));
        }
    }
}
```

Tip Notice that I have changed the namespace in which the class is defined in this file. The convention is that the defined in the files in the `App_Start` folder are defined in the top-level namespace for the application, which is `ClientFeatures` for this project.

The static `RegisterBundles` method is called from the `Application_Start` method in `Globa` which I'll set up in the next section—when the MVC Framework application first starts. The `RegisterBundl` takes a `BundleCollection` object, which I use to register new bundles of files through the `Add` method.

Tip The classes that are used for creating bundles are contained in the `System.Web.Optimization` n and, as I write this, the MSDN API documentation for this namespace isn't easy to find. You can navigate directly to http://msdn.microsoft.com/en-us/library/system.web.optimization.as want to learn more about the classes in this namespace.

I can create bundles for script files and for style sheets and it is important that I keep these types of files separate MVC Framework optimizes the files differently. Styles are represented by the `StyleBundle` class and scripts a by the `ScriptBundle` class.

When you create a new bundle, you create an instance of either `StyleBundle` or `ScriptBundle`, both take a single constructor argument that is the path that the bundle will be referenced by. The path is used as a URL f browser to request the contents of the bundle, so it is important to use a scheme for your paths that won't conflict w your application supports. The safest way to do this is to start your paths with `~/bundles` or `~/Content`. (importance of this will become apparent as I explain how bundles work.)

Once you have created the `StyleBundle` or `ScriptBundle` objects, you use the `Include` method of the style sheets or script files that the bundle will contain. There are some nice features available for making your flexible.

I started by creating a `StyleBundle` with the `~/Content/css` path. I want this bundle to include all th the application, so I passed the value `~/Content/*.css` as the argument to the `Include` method. The aste character is a wild card, which means that the bundle refers to all of the CSS files in the `/Content` folder of the is an excellent way of ensuring that files in a directory are automatically included in a bundle and where the order in are loaded isn't important.

Tip The order in which the browser loads the Bootstrap CSS files isn't important, so using a wildcard is just fine are relying on the CSS style precedence rules, then you need to list the files individually to ensure a specific order, w did for the JavaScript files.

The other bundle in the `BundleConfig.cs` file is a `ScriptBundle` whose path I set to `~/bundles/clientfeaturesscripts`. You will see the paths for both bundles again when I apply th application shortly. I used the `Include` method for this bundle to specify individual JavaScript files, separated by could have used another wildcard, but the order in which JavaScript files are processed usually matters and so I have individual files. Notice how I specified the jQuery library file:

```
...
~/Scripts/jquery- {version} .js
...
```

eBooks-IT.org

d and i

configuration of the application to select either the regular or minified version of the file (which I'll explain shortly). T
installed of the jQuery library is 1.10.2, which means that the bundle will include the `/Scripts/jquery-1.`
file

The benefit of using `{version}` is that you can update the libraries you use to new versions without having to
your bundles. The drawback is that the `{version}` token isn't able to differentiate between two versions of the
in the same directory. So, for example, if I were to add the `jquery-2.0.2.js` file to the `Scripts` folder,
up with both the 1.10.2 and 2.0.2 files being shipped to the client. Since this would undermine the goal of the optimiz
ensure that only one version of the library is in the `/Scripts` folder.

Note The jQuery team has done something unusual with their version numbering and is maintaining two differen
branches. As of jQuery 1.9, the jQuery 1.x and 2.x branches have the same API, but the jQuery 2.x release doesn't s
Microsoft browsers. You should use the 1.x release in your projects unless you are sure that none of your users are
Internet Explorer versions 6, 7 or 8. For more details see my *Pro jQuery 2.0* book, published by Apress.

Since I used the `Empty` template to create the example project, I need to add a statement to the `Global.asa`
the `RegisterBundles` method in the `BundleConfig` class, as shown in Listing 26-6.

Listing 26-6. Setting Up the Bundles in the Global.asax File

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

namespace ClientFeatures {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);,bv
        }
    }
}
```

Applying Bundles

The first thing I need to do, before I can apply the bundles, is make sure that the namespace that contains the bundle
classes is available for use within my view. To do this, I added an entry to the `pages/namespaces` element i
`Views/web.config` file, as shown in Listing 26-7.

Listing 26-7. Adding the Bundles Namespace to the Web.config File

```
...
<pages pageBaseType="System.Web.Mvc.WebViewPage">
  <namespaces>
    <add namespace="System.Web.Mvc" />
    <add namespace="System.Web.Mvc.Ajax" />
    <add namespace="System.Web.Mvc.Html" />
    <add namespace="System.Web.Routing" />
    <add namespace="System.Web.Optimization"/>
```

```
    </namespaces>
</pages>
...
```

You won't need to do this if you are using one of the more complex Visual Studio project templates, but Visual Stu set this up automatically when the `Empty` template is used.

The next step is to apply the bundles to the layout. You can see the changes I made to the `_Layout.cshtml` Listing 26-8.

Listing 26-8 Applying Bundles to the _Layout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style>
        .field-validation-error { color: #f00; }
        .validation-summary-errors { color: #f00; font-weight: bol
           .input-validation-error { border: 2px solid #f00; bac
color: #fee; }
        input[type="checkbox"].input-validation-error { outline: 2
#f00; }
        div.hidden { display: none; }
        div.visible { display: block; }
    </style>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/clientfeaturesscripts")
    @RenderSection("Scripts", false)
</head>
<body>
    @RenderSection("Body")
</body>
</html>
```

Bundles are added using the `@Scripts.Render` and `@Styles.Render` helper methods and you can have used these helpers to replace the `link` and `script` elements with the combined bundles of files.

Tip Notice that I have left the `Scripts` section in the layout so that the view can define its inline code. You c match bundles and regular `script` and `link` elements freely, although you should consider moving inline code a external files to maximize the optimization that the MVC Framework can perform, which I describe shortly.

You can see the HTML that these helper methods generate by starting the application, navigating to the `/Home/MakeBooking` URL, and viewing the page source. Here is the output produced by the `Styles.Re` method for the `~/Content/css` bundle:

```
...
<link href="/Content/bootstrap-theme.css" rel="stylesheet"/>
<link href="/Content/bootstrap.css" rel="stylesheet"/>
...
```

```
...
<script src="/Scripts/jquery-1.10.2.js"></script>
<script src="/Scripts/jquery.unobtrusive-ajax.js"></script>
<script src="/Scripts/jquery.validate.js"></script>
<script src="/Scripts/jquery.validate.unobtrusive.js"></script>
...
```

Optimizing the JavaScript and CSS Files

Organizing JavaScript and CSS files into related groups is a useful way to make sure that you don't forget to include your layouts include whatever version of a file that is included in the project. But the real magic of bundles is that the to optimize the delivery of JavaScript and CSS content to the browser.

The key to this is contained in the `Web.config` file (the one in the root folder this time) and the `debug` attri `compilation` element. Open the `Web.config` file and set the attribute value to `false`, as shown in List

Listing 26-9. Disabling Debug Mode in the Web.config File

```
...
<system.web>
    <compilation debug="false" targetFramework="4.5.1" />
    <httpRuntime targetFramework="4.5.1" />
</system.web>
...
```

When the `debug` attribute is set to `true`, the HTML sent to the browser contains `link` and `script` elem individual files. When the attribute is `false`, the minified versions of the files are selected and concatenated togeth can be delivered to the client as a blob.

Note Minification processes a JavaScript or CSS file to remove whitespace and, in the case of JavaScript files, s variable and function names so that the files require less bandwidth to transfer. Most libraries provide both debug (i.e readable) and minified versions of their files, which is why the `Scripts` folder contains, for example, the `jque validate.js` and `jquery-validate.min.js` file. The extra `.min` in the file name denotes the mi The selection between the files is automatic and, for the most part, minification is a simple and successful process. S advanced libraries (such as one of my favorites, AngularJS), however, require a special minification process. Caution recommended.

Select `Start Without Debugging` from the Visual Studio `Debug` menu. (You can't run the debugg debug attribute is set to `false`.) Navigate to the `/Home/MakeBooking` URL and press the F12 key to brin developer tools. Switch to the network tab, clear the browser cache and then click the green arrow button to start re requests made by the browser. Reload the page to see the effect of setting the `debug` attribute to `false`, which illustrates.

eBooks-IT.org

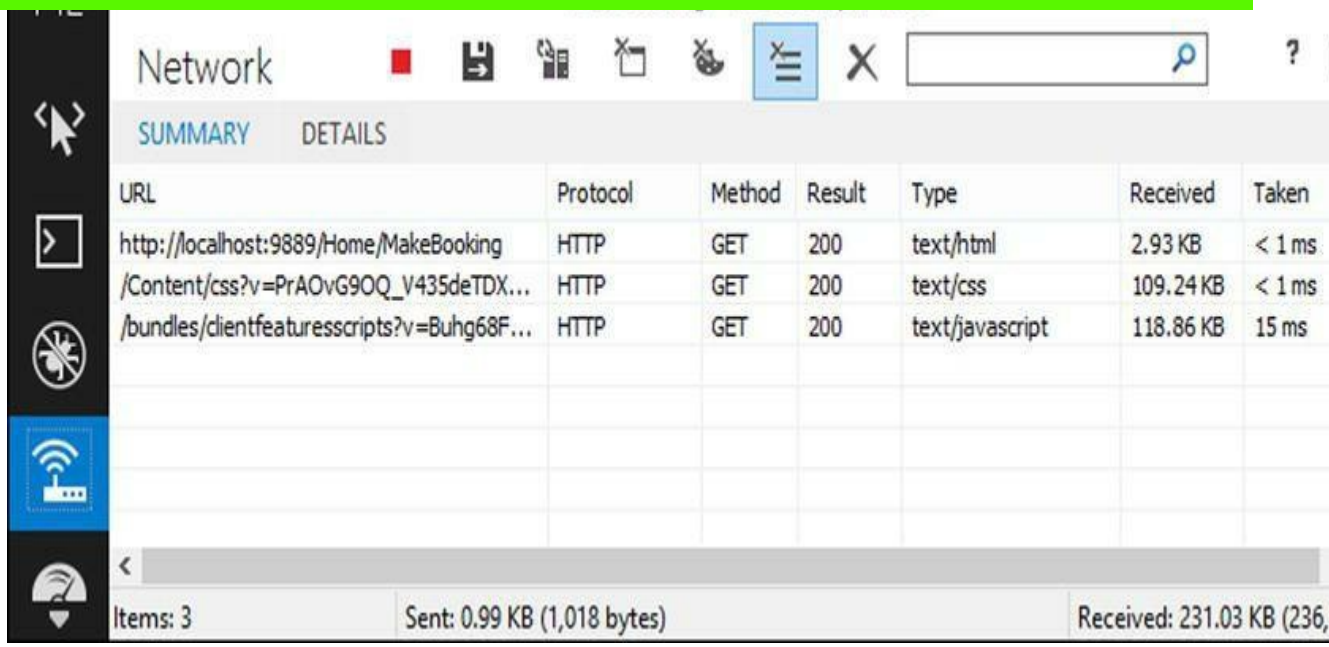


Figure 26-4. Profiling with bundles

Here is the summary of the profile information:

- The browser made three requests for the `/Home/MakeBooking` URL.
- There was one request for a CSS file.
- There was one request for a JavaScript file.
- A total of 1,018 bytes were sent from the browser to the server.
- A total of 236,578 bytes were sent from the server to the browser.

That's not bad. I have shaved about 50 percent off the amount of data that is sent to the browser just by letting AS the MVC Framework optimize my JavaScript and CSS files. You can see how this works if you look at the HTML th application renders. Here is the HTML that the `Styles.Render` method has produced:

```
...
<link href="/Content/css?v=PrAOvG9OQ_V435deTDX5p8RzKE4Gs8_LEeYxl29
    rel="stylesheet"/>
...
```

And here is the HTML produced by the `Scripts.Render` method:

```
...
<script
                                        src="/bundles/clientfeatures
v=Buhg68FkCPk3xjXtPsE87M94MTb7DCZx3zKAYD0xRIA1">
</script>
...
```

These long URLs are used to request the contents of a bundle in a single blob of data. The MVC Framework minif differently from JavaScript files, which is why I have to keep style sheets and scripts in different bundles.

The impact of the optimizations is significant. I have far fewer requests from the browser (which reduces the amo sent to the client) and I have less data sent in return—all of which helps keep down the cost of running the web appl

This is the point where I usually stop optimizing the requests. I could go further: moving the inline scripts into sep that they can be minified, for example. But I don't like to optimize too heavily unless I have a tangible problem to sol optimization makes the application harder to debug and harder to maintain.

eBooks-IT.org

In this chapter, I showed you the bundles feature, which can help manage the JavaScript and CSS files in an applicat optimize their delivery to the client. In the next chapter, I will show you the Web API (which makes it easy to create that clients can consume), which is the foundation for *single page applications*.

eBooks-IT.org

Web API and Single-page Applications

In this chapter, I describe the Web API feature, which is a relatively new addition to the ASP.NET platform that allo quickly and easily create Web services that provide an API to HTTP clients, known as *Web APIs*.

The Web API feature is based on the same foundation as the MVC Framework applications, but is not part of the Framework. Instead, Microsoft has taken some key classes and characteristics that are associated with the `System.Web.Mvc` namespace and duplicated them in the `System.Web.Http` namespace. The idea is th part of the core ASP.NET platform and can be used in other types of Web applications or used as a stand-alone Web engine. I have included Web API in this book because one of the main uses for it is to create *single-page application* combining the Web API with MVC Framework features you have seen in previous chapters. I'll explain what SPAs a they work later in the chapter.

That is not to take away from the way that Web API simplifies creating Web services. It is a huge improvement o Microsoft Web service technologies that have been appearing over the last decade or so. I like the Web API and you for your projects, not least because it is simple and built on the same design that the MVC Framework uses.

I start this chapter by creating a regular MVC Framework application and then using the Web API to transform it page application. This is a surprisingly simple example, so I have treated the process like an extended example and ap the relevant techniques from earlier chapters because you can never have enough examples. Table 27-1 provides the this chapter.

Table 27-1. *Chapter Summary*

Problem	Solution
Create a RESTful web service	Add a Web API controller to an MVC Framework application.
Map between HTTP methods and action names in a Web API controller	Apply attributes such as `HttpPut` and `HttpPost` to the methods.
Create a single-page application	Use Knockout and jQuery to obtain data via Ajax and bind it to HTML elements.

Understanding Single-page Applications

The term *single-page application* (SPA) is a broadly applied term. The most consistently-used definition is a web app whose initial content is delivered as a combination of HTML and JavaScript and whose subsequent operations are pe a RESTful web service that delivers data via JSON in response to Ajax requests.

This differs from the kind of application I have been building in most of the chapters of this book, where operatio by the user result in new HTML documents being generated in response to synchronous HTTP requests, which I wi round-trip applications (RTAs).

The advantages of a SPA are that less bandwidth is required and that the user receives a smoother experience. The disadvantages are that the smoother experience can be hard to achieve and that the complexity of the JavaScript code a SPA demands careful design and testing.

Most applications mix and match SPA and RTA techniques, where each major functional area of the application is SPA, and navigation between functional areas is managed using standard HTTP requests that create a new HTML do

Preparing the Example Application

For this chapter, I created a new ASP.NET project called `WebServices` using the Empty template. I checked th

eBooks-IT.org

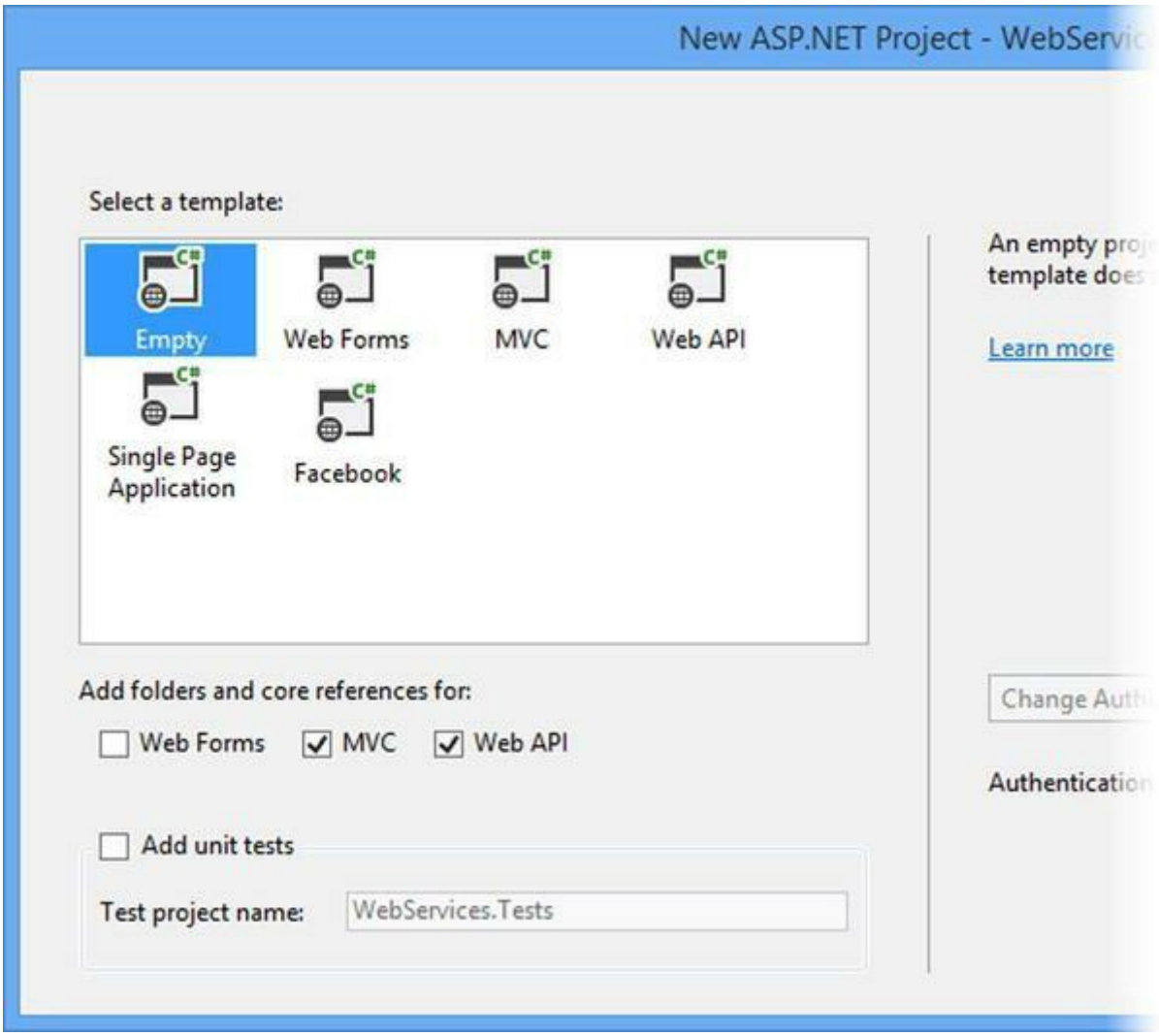


Figure 27-1. Creating the project with the MVC and Web API references

I will use this project to create a regular MVC Framework application and then use the Web API to create a web s the web service is complete, I'll return to the MVC Framework application and make it into a single-page application.

Creating the Model

This application will create and maintain a series of reservations. I want to keep the application simple so that I can f mechanics of the features I describe, and so these reservations will consist of just a name and a location. I added a c `Reservation.cs` to the `Models` folder, the contents of which are shown in Listing 27-1.

Listing 27-1. The Contents of the Reservation.cs File

```
namespace WebServices.Models {
    public class Reservation {
        public int ReservationId { get; set; }
        public string ClientName { get; set; }
        public string Location { get; set; }
    }
}
```

I am going to create a simple in-memory collection of `Reservation` objects to act as the model repository. I to go to the trouble of setting up a database, but I do need to be able to perform CRUD operations on a collection of so that I can demonstrate some important aspects of the Web API. I added a class file called

Listing 27-2. The Contents of the ReservationRespository.cs File

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

namespace WebServices.Models {
    public class ReservationRespository {
                    private   static   ReservationRespository   repo
ReservationRespository();

        public static ReservationRespository Current {
            get {
                return repo;
            }
        }

        private List<Reservation> data = new List<Reservation> {
            new Reservation {
                 ReservationId = 1, ClientName = "Adam", Location
Room"},
            new Reservation {
                    ReservationId = 2, ClientName = "Jacqui", Lo
"Lecture Hall"},
            new Reservation {
                   ReservationId = 3, ClientName = "Russell", Lo
"Meeting Room 1"},
        };

        public IEnumerable<Reservation> GetAll() {
            return data;
        }

        public Reservation Get(int id) {
                             return   data.Where(r   =>   r.Reservati
id).FirstOrDefault();
        }

        public Reservation Add(Reservation item) {
            item.ReservationId = data.Count + 1;
            data.Add(item);
            return item;
        }

        public void Remove(int id) {
            Reservation item = Get(id);
            if (item != null) {
                data.Remove(item);
            }
        }

        public bool Update(Reservation item) {
            Reservation storedItem = Get(item.ReservationId);
            if (storedItem != null) {
```

eBooks-IT.org

```
                storedItem.Location = item.Location;
                return true;
            } else {
                return false;
            }
        }
    }
}
```

Tip In a real project, I would be concerned about tight coupling between classes and introduce interfaces and de injection into the application. My focus in this chapter is just on the Web API and SPA applications, so I am going to shortcuts when it comes to other standard techniques.

The repository class has an initial list of three `Reservation` objects and defines methods that allow me to vie delete and update the collection. Since there is no persistent storage, any changes that are made to the repository will the application is stopped or restarted, but this example is all about the way in which content can be delivered and no stored by the server. To ensure that there is some persistence between requests, I have created a static instance of th `ReservationRespository` class, which is accessible through the `Current` property.

Adding the NuGet Packages

I am going to rely on three NuGet packages in this chapter: jQuery, Bootstrap and Knockout. I have already describe jQuery and Bootstrap in earlier chapters. Knockout is the library that Microsoft has adopted for single-page applicatio created by Steve Sanderson, whom I worked with on an earlier edition of this book and who works for the Microsof team. Even though Steve works for Microsoft, the Knockout package is open source and widely used and you can le about it at http://knockoutjs.com. I'll explain how Knockout works later in the chapter, but for the mo need to install the NuGet packages. Select `Package Manager Console` from the Visual Studio `Tools` `Library Package Manager` menu and enter the following commands:

```
Install-Package jquery -version 1.10.2
Install-Package bootstrap -version 3.0.0
Install-Package knockoutjs -version 3.0.0
```

Adding the Controller

I added a controller called `Home` to the example project, the definition of which you can see in Listing 27-3.

Listing 27-3. The Contents of the HomeController.cs File

```
using System.Web.Mvc;
using WebServices.Models;

namespace WebServices.Controllers {

    public class HomeController : Controller {
                                private    ReservationRespository    r
ReservationRespository.Current;

        public ViewResult Index() {
            return View(repo.GetAll());
        }

        public ActionResult Add(Reservation item) {
            if (ModelState.IsValid) {
                repo.Add(item);
```

```
            } else {
                return View("Index");
            }
        }

        public ActionResult Remove(int id) {
            repo.Remove(id);
            return RedirectToAction("Index");
        }

        public ActionResult Update(Reservation item) {
            if (ModelState.IsValid && repo.Update(item)) {
                return RedirectToAction("Index");
            } else {
                return View("Index");
            }
        }
    }
}
```

This is a fairly typical controller for such a simple application. Each of the action methods corresponds directly to methods in the repository and the only value that the controller adds is to perform model validation, to select views, a redirections. In a real project, there would be more business domain logic, of course, but because the example applic using is so basic, the controller ends up being little more than a wrapper around the repository.

Adding the Layout and Views

To generate the content for the application, I started by creating the `Views/Shared` folder and adding a view f `_Layout.cshtml` to it, the contents of which are shown by Listing 27-4.

Listing 27-4. The Contents of the _Layout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    @RenderSection("Scripts")
</head>
<body>
    @RenderSection("Body")
</body>
</html>
```

This is a basic layout that has `link` elements for the Bootstrap CSS files. I have defined two layout sections, `Sc` `Body`, that I will use to insert content into the layout. My next step was to create the top-level view for the applica Although I am going through the process of creating a regular MVC Framework application, I know that I am going with a single-page application and the transformation will be made easier if I create a single view that contains all the

Views/Home folder, the contents of which you can see in Listing 27-5.

Listing 27-5. The Contents of the Index.cshtml File

```
@using WebServices.Models

@model IEnumerable<Reservation>
@{
    ViewBag.Title = "Reservations";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Scripts {

}

@section Body {
    <div id="summary" class="section panel panel-primary">
        @Html.Partial("Summary", Model)
    </div>

    <div id="editor" class="section panel panel-primary">
        @Html.Partial("Editor", new Reservation())
    </div>
}
```

The view model for this view is an enumeration of `Reservation` objects and I rely on two partial views to p functional building blocks that the user will see. The first partial view file is called `Summary.cshtml`. I created `Views/Home` folder and you can see the contents of the file in Listing 27-6.

Listing 27-6. The Contents of the Summary.cshtml File

```
@model IEnumerable<WebServices.Models.Reservation>

<div class="panel-heading">Reservation Summary</div>
<div class="panel-body">
    <table class="table table-striped table-condensed">
        <thead>
            <tr><th>ID</th><th>Name</th><th>Location</th><th></th>
        </thead>
        <tbody>
            @foreach (var item in Model) {
                <tr>
                    <td>@item.ReservationId</td>
                    <td>@item.ClientName</td>
                    <td>@item.Location</td>
                    <td>
                        @Html.ActionLink("Remove", "Remove",
                                new { id = item.ReservationId },
                                new { @class = "btn btn-xs btn-pri
                    </td>
                </tr>
            }
        </tbody>
```

eBooks-IT.org

```
</div>
```

The view model for the partial view is the same enumeration of `Reservation` object and I use it to generate styled `table` element that displays the object property values. I use the `Html.ActionLink` helper method t link that will invoke the `Remove` action on the `Home` controller and use Bootstrap to style it as a button.

The other partial view is called `Editor.cshtml` and I put this in the `Views/Home` folder as well. Listing the contents of this file. This partial view contains a form that can be used to create new reservations. Submitting the invokes the `Add` action on the `Home` controller.

Listing 27-7. The Contents of the Editor.cshtml File

```
@model WebServices.Models.Reservation

<div class="panel-heading">
    Create Reservation
</div>
<div class="panel-body">
   @using(Html.BeginForm("Add", "Home")) {
        <div class="form-group">
            <label>Client Name</label>
                @Html.TextBoxFor(m => m.ClientName, new {@class =
control"})
        </div>
        <div class="form-group">
            <label>Location</label>
                @Html.TextBoxFor(m => m.Location, new { @class =
control" })
        </div>
        <button type="submit" class="btn btn-primary">Save</button
   }
</div>
```

Setting the Start Location and Testing the Example Application

The last preparatory step is to set the location that Visual Studio will navigate to when the application is started. Selec `WebServices Properties` from the Visual Studio `Project` menu, switch to the `Web` tab and check `Specific Page` option in the `Start Action` section. You don't have to provide a value. Just checking enough. To test the application in its classic MVC Framework form, select `Start Debugging` from the Visu `Debug` menu. You will see the (slightly odd) all-in-one layout that provides the user with a list of the current reserv ability to create and delete items, as shown in Figure 27-2.

eBooks-IT.org

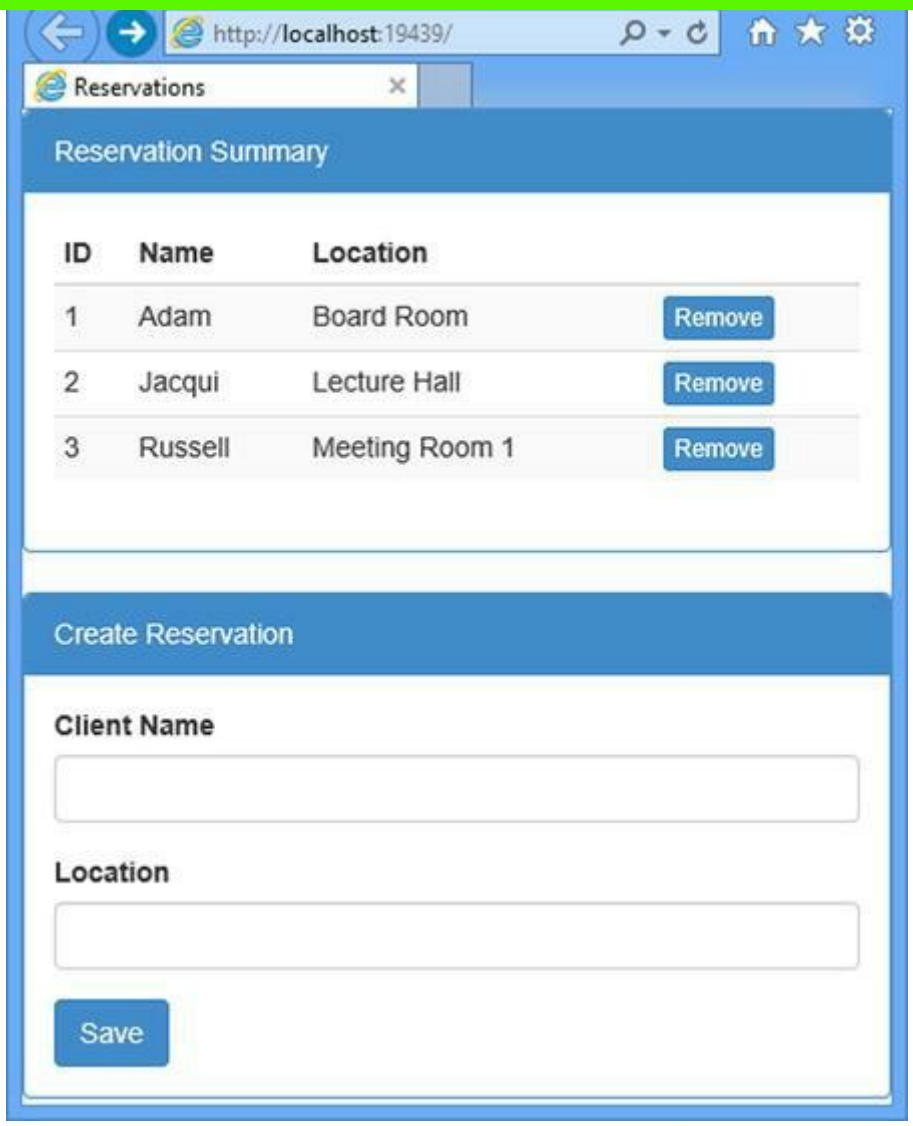


Figure 27-2. Testing the example application

Using Web API

The Web API feature is based on adding a special kind of controller to an MVC Framework application. This kind of
called an *API Controller*, has two distinctive characteristics:

1. Action methods return model, rather than `ActionResult`, objects.
2. Action methods are selected based on the HTTP method used in the request.

The model objects that are returned from an API controller action method are encoded as JSON and sent to the cl
controllers are designed to deliver Web data services, so they do not support views, layouts, or any of the other featu
used to generate HTML in the example application.

Tip The inability of an API controller to generate HTML from views is the reason that single-page applications co
standard MVC Framework techniques with the Web API. The MVC Framework performs the steps required to deliv
content to the user (including authentication, authorization, and selecting and rendering a view). Once the HTML is d
browser, the Ajax requests generated by the JavaScript it contains are handled by the Web API controller.

As I demonstrated in Chapter 23, you can create action methods in regular controllers that return JSON data to su
but the API controller offers an alternative approach that separates the data-related actions in your application from th

Creating the Web API Controller

Adding Web API to an application is incredibly simple. In part this is because I am creating a basic web service, but the integration with the underpinnings of the MVC Framework means that little work is required. I created the `WebController.cs` class file in the `Controllers` folder of the project and used it to define the contro Listing 27-8.

Listing 27-8. The Contents of the WebController.cs File

```
using System.Collections.Generic;
using System.Web.Http;
using WebServices.Models;

namespace WebServices.Controllers {

    public class WebController : ApiController {
                                private    ReservationRespository    r
ReservationRespository.Current;

        public IEnumerable<Reservation> GetAllReservations() {
            return repo.GetAll();
        }

        public Reservation GetReservation(int id) {
            return repo.Get(id);
        }

        public Reservation PostReservation(Reservation item) {
            return repo.Add(item);
        }

        public bool PutReservation(Reservation item) {
            return repo.Update(item);
        }

        public void DeleteReservation(int id) {
            repo.Remove(id);
        }
    }
}
```

That is all that is required to create a Web API. The API controller has a set of five action methods that map to the of the repository and provide web service access to the `Reservation` objects.

Testing the API Controller

I will explain how the API controller works shortly, but first I am going to perform a simple test. Start the applicatio browser loads the root URL for the project, navigate to the `/api/web` URL. The result that you see will depend o browser that you are using. If you are using Internet Explorer, then you will be prompted to save or open a file that c following JSON data:

```
[{"ReservationId":1,"ClientName":"Adam","Location":"Board Room"},
 {"ReservationId":2,"ClientName":"Jacqui","Location":"Lecture Hall
 {"ReservationId":3,"ClientName":"Russell","Location":"Meeting Roo
```

eBooks-IT.org

XML data.

```
<ArrayOfReservation>
    <Reservation>
        <ClientName>Adam</ClientName>
        <Location>Board Room</Location>
        <ReservationId>1</ReservationId>
    </Reservation>
    <Reservation>
        <ClientName>Jacqui</ClientName>
        <Location>Lecture Hall</Location>
        <ReservationId>2</ReservationId>
    </Reservation>
    <Reservation>
        <ClientName>Russell</ClientName>
        <Location>Meeting Room 1</Location>
        <ReservationId>3</ReservationId>
    </Reservation>
</ArrayOfReservation>
```

There are a couple of interesting things to note here. The first is that the request for the `/api/web` URL has pr of all of the model objects and their properties, from which we can infer that the `GetAllReservations` act the `Reservation` controller was called.

The second point to note is that different browsers received different data formats. You might get different results this yourself because later versions of the browsers may change the way they make requests, but you can see that o has produced JSON and the other has produced XML. (You can also see why JSON has largely replaced XML for W XML is more verbose and harder to process, especially when you are using JavaScript.)

The different data formats are used because the Web API uses the HTTP `Accept` header contained in the reque out what data type the client would prefer to work with. Internet Explorer got JSON because this is the `Accept` h

```
...
Accept: text/html, application/xhtml+xml, */*
...
```

The browser specified that it would like `text/html` content most of all, and then `application/xhtm` The final part of the `Accept` header is `*/*`, which means the browser will accept any data type if the first two a available.

The Web API supports JSON and XML, but it gives preference to JSON, which is what it used to respond to the the IE `Accept` header. Here is the `Accept` header that Google Chrome sent:

```
...
Accept:text/html,application/xhtml+xml, application/xml;q=0.9,*/*;
...
```

I have highlighted the important part: Chrome has said that it prefers to receive `application/xml` data in p the `*/*` catchall. The Web API controller honored this preference and delivered the XML data. I mention this becau problem with Web API is getting an undesired data format. This happens because the `Accept` header gives unexp preference to a format, or it is missing from the request entirely.

Understanding How the API Controller Works

You will understand a lot more about how the API controller works by navigating to the `/api/web/3` URL. You following JSON (or the equivalent XML if you are using another browser):

```
{"ReservationId":3,"ClientName":"Russell","Location":"Meeting Room
```

eBooks-IT.org

corresponds to the last segment of the URL I requested. The format of the URL and the use of the URL segment sho you of Chapter 15, where I explained how MVC Framework routes work.

API controllers have their own routing configuration, which is completely separate from the rest of the application the default configuration that Visual Studio creates for new projects by looking at the `/App_Start/WebApiConfig.cs` file, which I have shown in Listing 27-9. This is one of the files that V adds to the project when you check the Web API box during project creation.

Listing 27-9. The Contents of the WebApiConfig.cs File

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

namespace WebServices {
    public static class WebApiConfig {
        public static void Register(HttpConfiguration config) {

            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

The `WebApiConfig.cs` file contains the routes used for API Controllers, but uses different classes from th MVC routes defined in the `RouteConfig.cs` file. The Web API feature is implemented as a stand-alone ASP.N and it can be used outside of the MVC Framework, which means that Microsoft has duplicated some key MVC Fran functionality in the `System.Web.Http` namespace to keep MVC and Web API features separate. (This seems duplicative when writing an MVC Framework application but makes sense since Microsoft is trying to target non-MV with Web API, too.) Visual Studio also adds a call from the `Application_Start` method in the `Global.` so that the Web API routes are added to the application configuration, as shown in Listing 27-10.

Listing 27-10. The Contents of the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.Security;
using System.Web.SessionState;
using System.Web.Http;

namespace WebServices {
    public class Global : HttpApplication {
        void Application_Start(object sender, EventArgs e) {
            AreaRegistration.RegisterAllAreas();
            GlobalConfiguration.Configure(WebApiConfig.Register);
```

eBooks-IT.org

```
            }
        }
}
```

The result is that the application has two sets of routes: those used for MVC Framework controllers and those use API controllers.

Understanding API Controller Action Selection

The default Web API route, which you can see in Listing 27-9, has a static `api` segment, and `controller` an variables, the latter being optional. They key difference from a regular MVC route is that there is no `action` segm and this is where the behavior of API controllers takes shape.

When a request comes in to the application that matches a Web API route, the action is determined from the HTT used to make the request. When I tested the API controller by requesting `/api/reservation` using the brov browser specified the `GET` method.

The `ApiController` class, which is the base for API controllers, knows which controller it needs to target route and uses the HTTP method to look for suitable action methods.

The convention when naming API controller action methods is to prefix the name with the action method that it su include some reference to the model type that it operates on. But this is just a convention because Web API will matc method whose name contains the HTTP method used to make the request.

For the example, that means that a GET request results in a choice between the `GetAllReservations` a `GetReservation`, but method names like `DoGetReservation` or just `ThisIsTheGetAction` be matched.

To decide between the two action methods, the controller looks at the arguments that the contenders accept and u routing variables to make the best match. When requesting the `/api/reservation` API, there were no routi except for `controller`, and so the `GetAllReservations` method was selected because it has no argu requesting the `/api/reservation/3` URL, I supplied a value for the optional `id` segment variable, which `GetReservation` the better match because it accepts an `id` argument.

The other actions in the Web API controller are targeted using other HTTP methods: POST, DELETE, and PUT. foundation for the *Representation State Transfer* (REST) style of Web API, known more commonly as a *RESTful* se an operation is specified by the combination of a URL and the HTTP method used to request it.

Note REST is a style of API rather than a well-defined specification, and there is disagreement about what exact Web service RESTful. One point of contention is that purists do not consider Web services that return JSON as bein Like any disagreement about an architectural pattern, the reasons for the disagreement are arbitrary and dull. I try to about how patterns are applied, and JSON services are RESTful as far as I am concerned.

Mapping HTTP Methods to Action Methods

I explained that the `ApiController` base class uses the HTTP method to work out which action methods to t nice approach, but it does mean that you end up with some unnatural method names that are inconsistent with conve might be using elsewhere. For example, the `PutReservation` method might be more naturally called `UpdateReservation`. Not only would `UpdateReservation` make the purpose of the method more it may allow for a more direct mapping between the actions in your controller and the methods in your repository.

Tip You might be tempted to derive your repository class from `ApiController` and expose the repository directly as a Web API. I recommend against that and strongly suggest you create a separate controller, even as simp created in the example. At some point, the methods you want to offer via your API and the capabilities of your repos diverge, and having a separate API controller class will make that easier to manage.

The `System.Web.Http` namespace contains a set of attributes that you can use to specify which HTTP m action should be used for. You can see how I have applied two of these attributes in Listing 27-11 to create a more n method names.

Listing 27-11. Applying Attributes in the WebController.cs File

eBooks-IT.org

```
using System.Web.Http;
using WebServices.Models;

namespace WebServices.Controllers {

    public class WebController : ApiController {
                                private    ReservationRespository    r
ReservationRespository.Current;

        public IEnumerable<Reservation> GetAllReservations() {
            return repo.GetAll();
        }

        public Reservation GetReservation(int id) {
            return repo.Get(id);
        }

        [HttpPost]
        public Reservation CreateReservation(Reservation item) {
            return repo.Add(item);
        }

        [HttpPut]
        public bool UpdateReservation(Reservation item) {
            return repo.Update(item);
        }

        public void DeleteReservation(int id) {
            repo.Remove(id);
        }
    }
}
```

You can see the duplication with the MVC Framework features and the Web API. The `HttpPost` and `HttpP` that I used in Listing 27-11 have the exact same purpose as the attributes with the same name that I used in Chapter are defined in the `System.Web.Http` namespace and not `System.Web.Mvc`. Duplication aside, the attr the same way and I have ended up with more useful method names that will still work for the `POST` and `PUT` HTT (There are, of course, attributes for all of the HTTP methods, including `GET`, `DELETE` and so on.)

Using Knockout for Single-page Applications

The purpose of creating the Web API web service is to refactor the example application so that operations on the app can be performed using Ajax requests whose JSON results will be used to update the HTML in the browser. The ove functionality of the application will be the same, but I won't be generating complete HTML documents for each inter between the client and the server.

The transition to a single-page application puts more of a burden on the browser because I need to preserve applic the client. I need a data model that I can update, a series of logic operations that I can perform to transform the data UI elements that allows the user to trigger those operations. In short, I need to recreate a miniature version of the MV the browser.

The library that Microsoft has adopted for single-page applications is Knockout, which creates a JavaScript implem the MVC pattern (or, more accurately, the MVVM pattern which I described in Chapter 3 and is sufficiently close to pattern that I am going to treat them as the same thing). In the sections that follow, I am going to return to the MVC side of the example application and apply the Knockout library to create a simple SPA.

eBooks-IT.org

rary in
to see what it is capable of. You can learn more at http://knockoutjs.com or from my *Pro JavaScript*
book, which is published by Apress. I like Knockout, but for more complex applications I prefer AngularJS. It has a
learning curve, but the investment is worthwhile. You can learn more at http://angularjs.org or read
AngularJS book, which—as you might have guessed by now—is also published by Apress.

Adding the JavaScript Libraries to the Layout

The first step is to add the Knockout and jQuery files to the layout so that they are available in the view. You can see
`script` element I added in Listing 27-12.

Listing 27-12. Adding the Knockout JavaScript File to the _Layout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    <script src="~/Scripts/jquery-1.10.2.js"></script>
    <script src="~/Scripts/knockout-3.0.0.js"></script>
    @RenderSection("Scripts")
</head>
<body>
    @RenderSection("Body")
</body>
</html>
```

I will be using Knockout to create the MVC implementation and jQuery to handle the Ajax requests.

Implementing the Summary

The first major change I will make is to replace the `Summary` partial view with some inline Knockout and jQuery.
have to use Knockout in a single file, but I want to leave the partial views intact so you can see the difference betwe
standard MVC Framework way of working and the SPA techniques. In Listing 27-13, you can see the changes that
`Index.cstml` view file.

Listing 27-13. Using Knockout and jQuery to Implement the Summary in the Index.cshtml File

```
@using WebServices.Models
@{
    ViewBag.Title = "Reservations";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Scripts {
    <script>
        var model = {
            reservations: ko.observableArray()
```

```
        function sendAjaxRequest(httpMethod, callback, url) {
            $.ajax("/api/web" + (url ? "/" + url : ""), {
                type: httpMethod, success: callback
            });
        }

        function getAllItems() {
            sendAjaxRequest("GET", function(data) {
                model.reservations.removeAll();
                for (var i = 0; i < data.length; i++) {
                    model.reservations.push(data[i]);
                }
            });
        }

        function removeItem(item) {
            sendAjaxRequest("DELETE", function () {
                getAllItems();
            }, item.ReservationId);
        }

        $(document).ready(function () {
            getAllItems();
            ko.applyBindings(model);
        });
    </script>
}

@section Body {
    <div id="summary" class="section panel panel-primary">
        <div class="panel-heading">Reservation Summary</div>
        <div class="panel-body">
            <table class="table table-striped table-condensed">
                <thead>
                    <tr><th>ID</th><th>Name</th><th>Location</th><
</tr>
                </thead>
                <tbody data-bind="foreach: model.reservations">
                    <tr>
                        <td data-bind="text: ReservationId"></td>
                        <td data-bind="text: ClientName"></td>
                        <td data-bind="text: Location"></td>
                        <td>
                            <button class="btn btn-xs btn-primary"
                                                    data-bind
removeItem">Remove</button>
                        </td>
                    </tr>
                </tbody>
            </table>
        </div>
```

eBooks-IT.org

```
    <div id="editor" class="section panel panel-primary">
        @Html.Partial("Editor", new Reservation())
    </div>
}
```

There is a lot going on here, so I am going to break down the changes and explain each of them in turn. The overa the changes, however, is that the browser uses the Web API controller to get details of the current reservations.

Note Notice that I have removed the `@model` expression from the `Index` view in Listing 27-13. I am not us model objects to generate the content in the view, which means that I don't need a view model. The controller is still `Reservation` objects from the repository and passing them to the view, but I'll fix this later in the chapter.

Defining the Ajax Functions

jQuery has excellent support for making Ajax requests. To that end, I have defined a function called `sendAjaxR` that I will use to target methods on the Web API controller, as follows:

```
...
function sendAjaxRequest(httpMethod, callback, url) {
    $.ajax("/api/web" + (url ? "/" + url : ""), {
        type: httpMethod, success: callback
    });
}
...
```

The `$.ajax` function provides access to the jQuery Ajax functionality. The arguments are the URL you want to an object that contains configuration parameters. The `sendAjaxRequest` function is a wrapper around the jQ functionality and its arguments are the HTTP method that should be used for the request (which affects the action m selected by the controller), a callback function that will be invoked when the Ajax request has succeeded and an opti suffix. Using the `sendAjaxRequest` function as a foundation, I defined functions to get all of the data availab delete a reservation, like this:

```
...
function getAllItems() {
    sendAjaxRequest("GET", function(data) {
        model.reservations.removeAll();
        for (var i = 0; i < data.length; i++) {
            model.reservations.push(data[i]);
        }
    });
}

function removeItem(item) {
    sendAjaxRequest("DELETE", function () {
        getAllItems();
    }, item.ReservationId);
}
...
```

The `getAllItems` function targets the `GetAllReservations` controller action method and retrieve reservations from the server. The `removeItem` function targets the `DeleteReservation` action metho `getAllItems` function to refresh the data after a deletion.

Defining the Model

```
...
var model = {
    reservations: ko.observableArray()
};
...
```

Knockout works by creating *observable objects* that it monitors for changes and uses to update the HTML display browser. My model is simple. It consists of an observable array, which is just like a regular JavaScript array, but is that any changes I made are detected by Knockout. You can see how I use the model in the Ajax functions, like this:

```
...
function getAllItems() {
    sendAjaxRequest("GET", function(data) {
        model.reservations.removeAll();
        for (var i = 0; i < data.length; i++) {
            model.reservations.push(data[i]);
        }
    });
}
...
```

The two statements I highlighted are how I get the data from the server into the model. I call the `removeAll` remove any existing data from the observable array and then iterate through the results I get from the server with the method to populate the array with the new data.

Defining the Bindings

Knockout applies changes in the data model to HTML elements via *data bindings*. Here are the most important data b `Index` view:

```
...
<tbody data-bind="foreach: model.reservations" >
    <tr>
        <td data-bind="text: ReservationId" ></td>
        <td data-bind="text: ClientName" ></td>
        <td data-bind="text: Location" ></td>
        <td>
            <button class="btn btn-xs btn-primary"
                    data-bind="click: removeItem" >Remove</button>
        </td>
    </tr>
</tbody>
...
```

Knockout is expressed using the `data-bind` attribute and there is a wide range of bindings available, three of used in the view. The basic format for a data-bind attribute is:

```
data-bind="type: expression"
```

The types of the three bindings in the listing are `foreach`, `text` and `click`, and I picked these three becau represent the different ways in which Knockout can be used.

The first two, the `foreach` and `text` bindings, generate HTML elements and content from the data model. W `foreach` binding is applied to an element, Knockout generates the child elements for each object in the expression Razor `@foreach` that I was using in the partial view.

eBooks-IT.org

eans th

this binding, for example:

```
...
<td data-bind="text: ClientName"></td>
...
```

Knockout will insert the value of the `ClientName` property of the current object being processed by the `for` binding, which has the same effect as the Razor `@Model.ClientName` expression I used previously.

The click directive is different: it sets up an even handler for the `click` event on the element to which it has bee You don't have to use Knockout to set up events, of course, but the `click` binding is integrated with the other bin function you specify to call when the event is triggered is passed the data object that was being processed by the `fo` binding when the binding was applied. This is why the `removeItem` function is able to define an argument that r `Reservation` object (or its JavaScript representation, anyway).

Processing the Bindings

Knockout bindings are not processed automatically, which is why I included this code in the `script` element:

```
...
$(document).ready(function () {
    getAllItems();
    ko.applyBindings(model);
});
...
```

The `$(document).ready` call is a standard jQuery technique to defer execution of a function until all of t elements in the document have been loaded and processed by the browser. Once that happens, I call the `getAllI` function to load the data from the server and then the `ko.applyBindings` function to use the data model to `data-bind` attributes. This final call is the one that connects the data objects to the HTML elements, generates t require, and sets up the event handlers.

Testing the Summary Bindings

You might be wondering why I have gone to all this trouble, given that I have essentially replaced Razor expressions equivalent Knockout bindings. There are three important differences and to demonstrate them fully, I am going to us F12 tools.

The first difference is that the model data is no longer included in the HTML that is sent to the browser. Instead, o HTML has been processed, the browser makes an Ajax request to the Web API controller and gets the list of reserva expressed as JSON. You can see this by starting the application and using the F12 tools to monitor the requests that makes (as described in Chapter 26). Figure 27-3 shows the results.

eBooks-IT.org

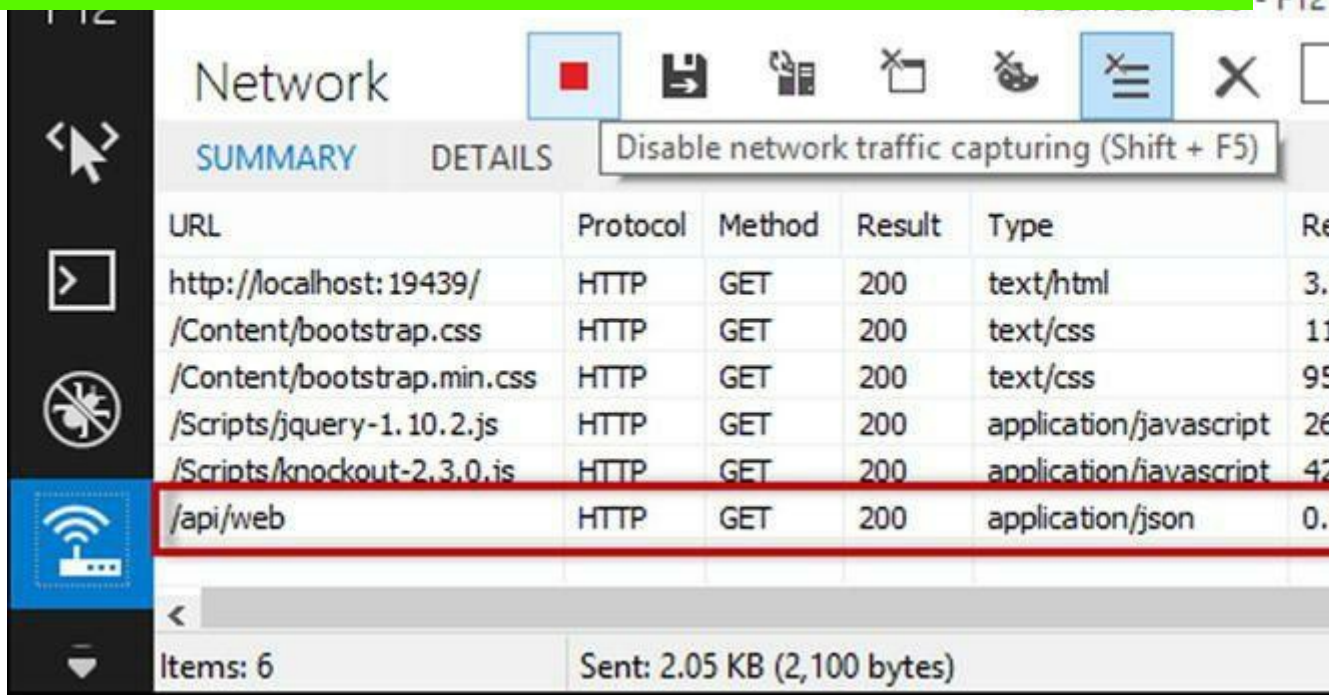


Figure 27-3. Monitoring the connections made by the browser

The second difference is that the data is processed by the browser, rather than at the server, as the view is render this, you can edit the `getAllItems` function so that it doesn't make the Ajax request or process the data it rece

```
...
function getAllItems() {
    return;
    sendAjaxRequest("GET", function(data) {
        model.reservations.removeAll();
        for (var i = 0; i < data.length; i++) {
            model.reservations.push(data[i]);
        }
    });
}
...
```

The function will return before the Ajax request is made and you can see the effect by restarting the application, as Figure 27-4. This may seem obvious, but it is an important characteristic of SPAs that the browser does a lot more including processing data and generating HTML content.

eBooks-IT.org

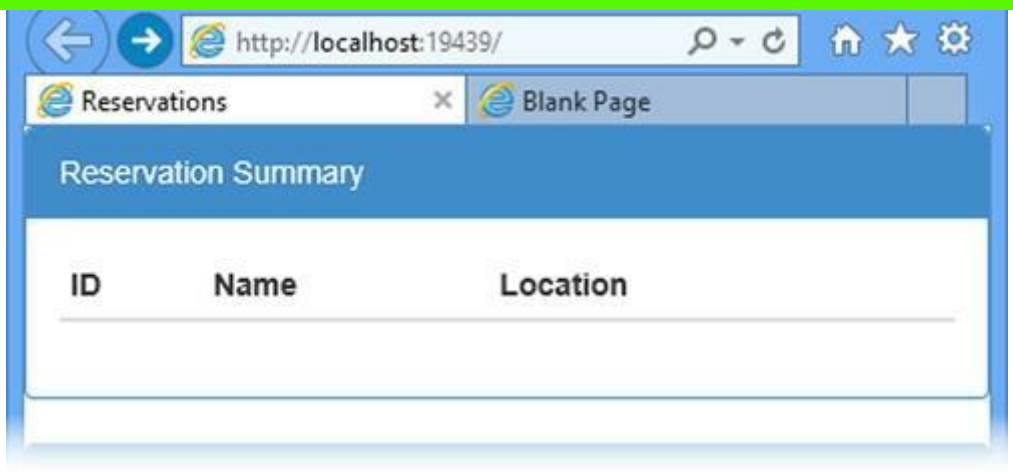


Figure 27-4. Demonstrating that the data is retrieved and processed by the browser

The final difference is that the data bindings are *live*, meaning that changes in the data model are reflected in the co `foreach` and `text` bindings generate. To test this, ensure that you return the `getAllItems` function to its state and reload the application. Once the browser has requested, received and processed the data, open the F12 tools to the `Console` section. Enter the following command into the console and hit Enter:

```
model.reservations.pop()
```

This expression removes the last item from the array of data objects in the model and as soon as you enter the com layout of the HTML page will reflect the change, as shown in Figure 27-5. The overall effect is that I have shifted so complexity of generating the HTML from the server to the client.

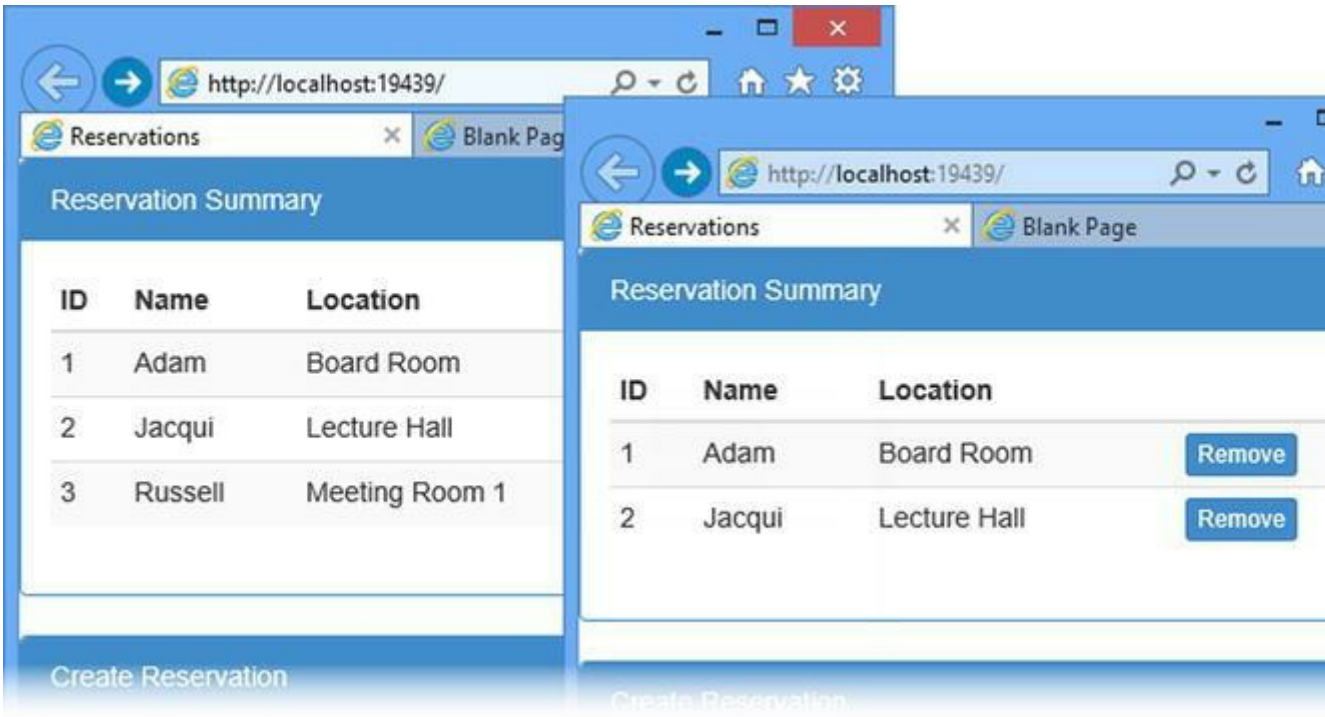


Figure 27-5. Manipulating the model via the JavaScript console

Improving the Delete Feature

Now that you have seen how applying Knockout has changed the nature of the client, I am going to quickly loop bac a shortcut that I took when I defined the Ajax methods for the application. The `removeItem` function is badly w

eBooks-IT.org

```
function removeItem(item) {
    sendAjaxRequest("DELETE", function () {
        getAllItems();
    }, item.ReservationId);
}
...
```

I have highlighted the problem: the function makes two Ajax requests to the server—the first to perform the deleti
second to request the contents of the repository to update the data model. Now that I have demonstrated that the clie
its own model and that live bindings reflect model changes in the HTML, I can improve upon this function, as shown
27-14.

Listing 27-14. Improving the RemoveItem Function in the Index.cshtml File

```
...
function removeItem(item) {
    sendAjaxRequest("DELETE", function () {
        for (var i = 0; i < model.reservations().length; i++) {
                                if  (model.reservations()[i].Reservati
item.ReservationId) {
                model.reservations.remove(model.reservations()[i])
                break;
            }
        }
    }, item.ReservationId);
}
...
```

When the request to the server succeeds, I remove the corresponding data object from the model, which means th
Ajax request is no longer required.

GETTING USED TO THE KNOCKOUT SYNTAX

There are some syntax quirks when working with Knockout observable arrays and two of them can be seen in
To get an item from the array, I have to treat `model.reservations` like a function, as follows:

```
...
model.reservations() [i].ReservationId
...
```

And when it comes to removing items from the array, I use a function that is not standard JavaScript:

```
...
model.reservations. remove (model.reservations()[i]);
...
```

Knockout tries to maintain the standard JavaScript syntax, but there are some compromises required to track ch
objects, such as these quirks. They can be confusing when you first start working with Knockout, but you soo
them. And you learn that when you don't get the effect you require, the likely cause is a mismatch between the
JavaScript syntax and that required for a Knockout observable object or array. You can get further information
Knockout API at http://knockoutjs.com.

Implementing the Create Feature

The next step is to use Knockout to replace the `Editor` partial view. Once again, I could have updated the partial
contain the Knockout functionality, but I have chosen to include everything in the `Index.cshtml` file, as show

Listing 27-15. Implementing the Create Feature in the Index.cshtml File

```
@using WebServices.Models
@{
    ViewBag.Title = "Reservations";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Scripts {
    <script>
        var model = {
            reservations: ko.observableArray(),
            editor: {
                name: ko.observable(""),
                location: ko.observable("")
            }
        };

        function sendAjaxRequest(httpMethod, callback, url, reqDat
            $.ajax("/api/web" + (url ? "/" + url : ""), {
                type: httpMethod,
                success: callback,
                data: reqData
            });
        }

        // ... other functions omitted for brevity ...

        function handleEditorClick() {
            sendAjaxRequest("POST", function (newItem) {
                model.reservations.push(newItem);
            }, null, {
                ClientName: model.editor.name,
                Location: model.editor.location
            });
        }

        $(document).ready(function () {
            getAllItems();
            ko.applyBindings(model);
        });
    </script>
}

@section Body {
    <div id="summary" class="section panel panel-primary">
        <!-- elements omitted for brevity -->
    </div>
    <div id="editor" class="section panel panel-primary">
        <div class="panel-heading">
            Create Reservation
        </div>
```

eBooks-IT.org

```
            <div class="form-group">
                <label>Client Name</label>
                            <input  class="form-control"  data-bind
model.editor.name" />
            </div>
            <div class="form-group">
                <label>Location</label>
                            <input  class="form-control"  data-bind
model.editor.location" />
            </div>
            <button class="btn btn-primary"
                data-bind="click: handleEditorClick">Save</button>
        </div>
    </div>
}
```

To create the editor, I have used Knockout in a different way, as I'll explain step-by-step in the sections that follo

Extending the Model

I need to collect two pieces of information from the user in order to create a new `Reservation` in the reposito of the client (corresponding to the `ClientName` property) and the location (corresponding to the `Location` first step is to extend the model so that I define variables that I can use to capture these values, like this:

```
...
var model = {
    reservations: ko.observableArray(),
    editor: {
        name: ko.observable(""),
        location: ko.observable("")
    }
};
...
```

The `ko.observable` function creates an observable value, which I will rely on later in the chapter. Changes values will be reflected in any bindings that use the `name` and `location` properties.

Implement the Input Elements

The next step is to create the `input` elements through which the user will supply values for my new model proper used the Knockout `value` binding, which sets the `value` attribute on an element, as follows:

```
...
<input class="form-control"data-bind="value: model.editor.name" />
...
```

The `value` bindings ensure that the values entered by the user into the `input` elements will be used to set the properties.

Tip Notice that I don't need a `form` element anymore. I will be using an Ajax request to send the data values t response to a button click, none of which relies on the standard browser support for forms.

Creating the Event Handler

eBooks-IT.org

put e

follows:

```
...
<button class="btn btn-primary" data-bind
handleEditorClick">Save</button>
...
```

The binding specifies that the `handleEditorClick` function should be called when the button is clicked a this function in the `script` element, as follows:

```
...
function handleEditorClick() {
    sendAjaxRequest("POST", function (newItem) {
            model.reservations.push(newItem);
        }, null, {
            ClientName: model.editor.name,
            Location: model.editor.location
        });
}
...
```

The event handler function calls the `sendAjaxRequest` function. The callback adds the newly created data back from the server to the model. I send an object containing the new model properties to the `sendAjaxRequ` function, which I have extended so that it will send them to the server as part of the Ajax request, using the data `op` property.

Testing the Create Feature

You can see how the Knockout implementation of the create feature works by starting the application, entering a nan location into the `input` elements and clicking the `Save` button, as illustrated by Figure 27-6.

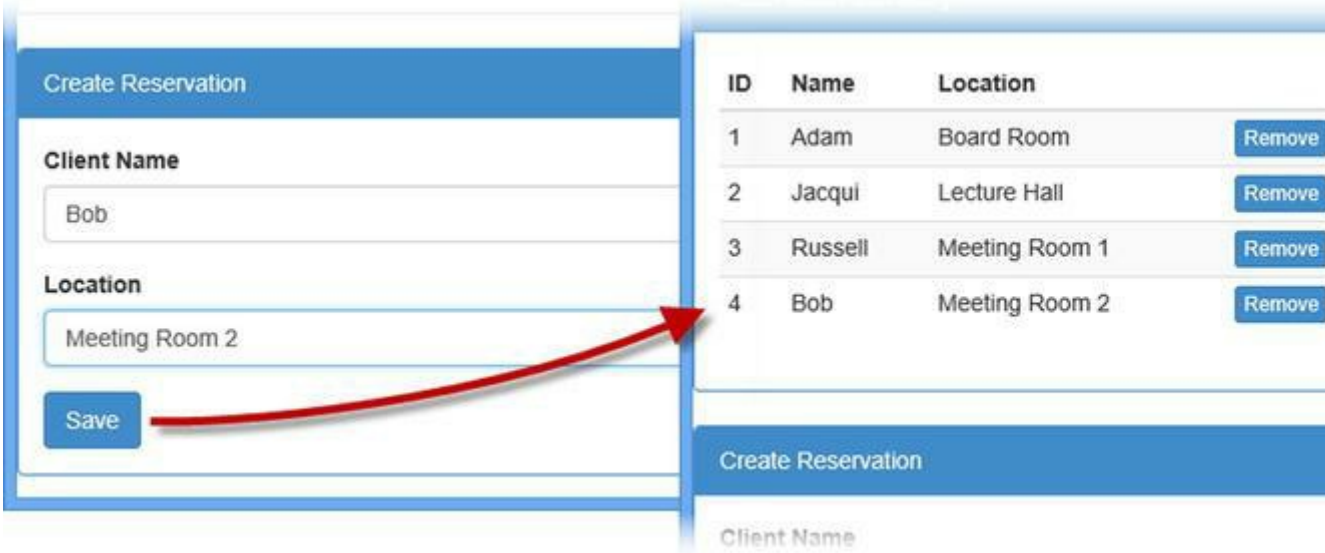


Figure 27-6. Creating a new reservation

Completing the Application

Now that you have seen how I can apply Knockout and the Web API to create a single-page application, I am going chapter by completing the application to add some missing features and remove some of the quirks.

The Home controller is still set up with action method to manipulate the repository to retrieve and manage Reser objects, even though all of the data displayed by the client is being requested via Ajax to the Web API controller. In L you can see how I have updated the controller to remove the action methods that the Web API controller has replace updated the Index action method so that it no longer passes a view model object.

Listing 27-16. Removing the Data Selection from the HomeController.cs File

```
using System.Web.Mvc;
using WebServices.Models;

namespace WebServices.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            return View();
        }
    }
}
```

Manage Content Visibility

The final change I am going to make is to manage the visibility of the elements in the HTML document so that only t or the editor is visible. You can see how I have done this in Listing 27-17.

Listing 27-17. Managing Element Visibility in the Index.cshtml File

```
@using WebServices.Models
@{
    ViewBag.Title = "Reservations";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Scripts {
    <script>
        var model = {
            reservations: ko.observableArray(),
            editor: {
                name: ko.observable(""),
                location: ko.observable("")
            },
            displaySummary: ko.observable(true)
        };

        function sendAjaxRequest(httpMethod, callback, url, reqDat
            $.ajax("/api/web" + (url ? "/" + url : ""), {
                type: httpMethod,
                success: callback,
                data: reqData
            });
        }

        function getAllItems() {
```

eBooks-IT.org

```
                model.reservations.removeAll();
                for (var i = 0; i < data.length; i++) {
                    model.reservations.push(data[i]);
                }
            });
        }

        function removeItem(item) {
            sendAjaxRequest("DELETE", function () {
                for (var i = 0; i < model.reservations().length; i
                                if  (model.reservations()[i].Reservati
item.ReservationId) {
                                model.reservations.remove(model.reserv
[i]);
                            break;
                        }
                }
            }, item.ReservationId);
        }

        function handleCreateClick() {
            model.displaySummary(false);
        }

        function handleEditorClick() {
            sendAjaxRequest("POST", function (newItem) {
                model.reservations.push(newItem);
                model.displaySummary(true);
            }, null, {
                ClientName: model.editor.name,
                Location: model.editor.location
            });
        }

        $(document).ready(function () {
            getAllItems();
            ko.applyBindings(model);
        });
    </script>
}

@section Body {
    <div id="summary" class="section panel panel-primary"
            data-bind="if: model.displaySummary">
        <div class="panel-heading">Reservation Summary</div>
        <div class="panel-body">
            <table class="table table-striped table-condensed">
                <thead>
                    <tr><th>ID</th><th>Name</th><th>Location</th><
</tr>
                </thead>
                <tbody data-bind="foreach: model.reservations">
                    <tr>
                        <td data-bind="text: ReservationId"></td>
```

eBooks-IT.org

```
                                <td data-bind="text: Location"></td>
                                <td>
                                    <button class="btn btn-xs btn-primary"
                                                                        data-bind
removeItem">Remove</button>
                                </td>
                            </tr>
                        </tbody>
                    </table>
                    <button class="btn btn-primary"
                        data-bind="click: handleCreateClick">Create</butto
                </div>
            </div>
            <div id="editor" class="section panel panel-primary"
                data-bind="ifnot: model.displaySummary">
            <div class="panel-heading">
                Create Reservation
            </div>
            <div class="panel-body">
                <div class="form-group">
                    <label>Client Name</label>
                                <input  class="form-control"  data-bind
model.editor.name" />
                </div>
                <div class="form-group">
                    <label>Location</label>
                                <input  class="form-control"  data-bind
model.editor.location" />
                </div>
                <button class="btn btn-primary"
                    data-bind="click: handleEditorClick">Save</button>
            </div>
</div>
}
```

I have added a property to the model that specifies whether the summary should be shown. I use this property wit `ifnot` bindings, which add and remove elements to and from the DOM based on their expression. If the `displaySummary` property is `true`, then the data summary will be shown and if it is `false`, then the edit shown. The final changes I made were to add a `Create` button that calls a function that changes the `display` property and an addition to the callback function that processes new items that changes it back again. You can see th in Figure 27-7.

eBooks-IT.org

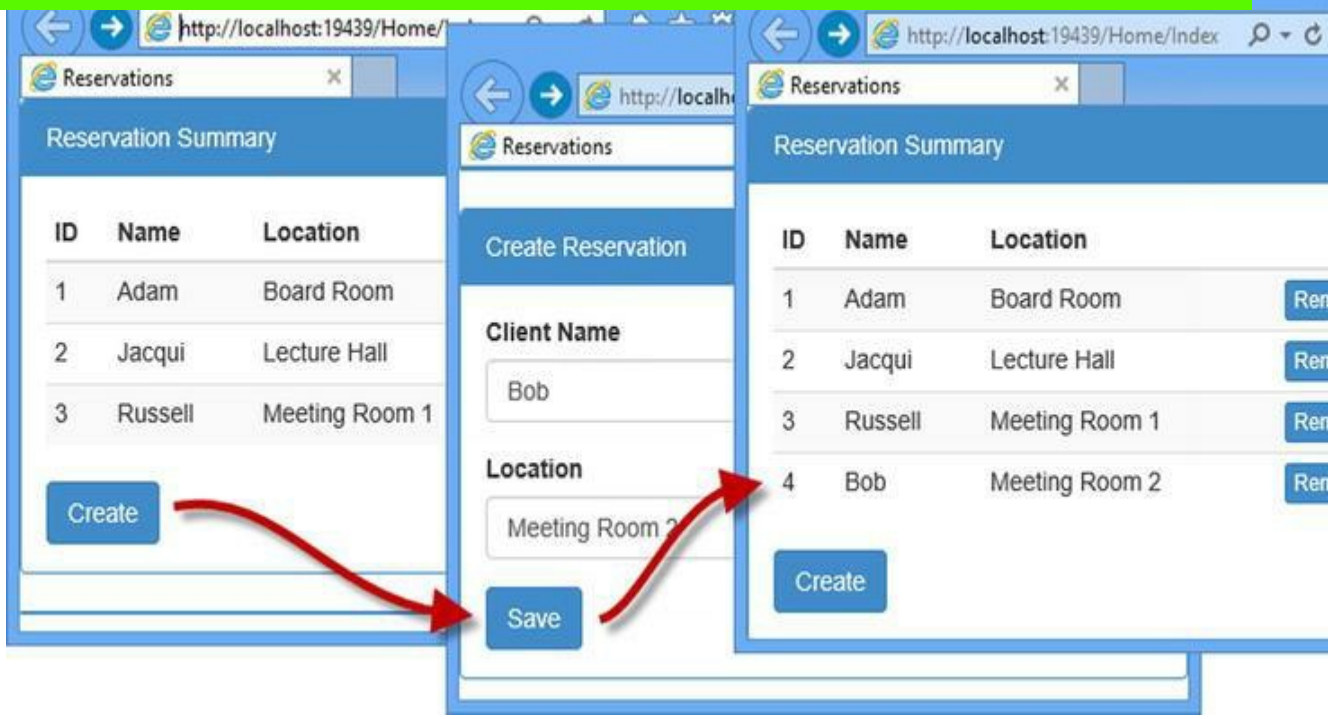


Figure 27-7. Adding a reservation using the final application

Summary

In this chapter, I showed you how to use the Web API and Knockout to create a single-page application that perform operations using RESTful Web service. While not part of the MVC Framework, the Web API is modeled so closely c and structure of MVC that it is familiar to MVC developers and, as I demonstrated, Web API controllers can be adde regular MVC controllers in an application.

And that is all I have to teach you about the MVC Framework. I started by creating a simple application, and then comprehensive tour of the different components in the framework, showing you how they can be configured, custo replaced entirely. I wish you every success in your MVC Framework projects and I can only hope that you have enj this book as much as I enjoyed writing it.

eBooks-IT.org

A

Account Controller
- contents AccountController.cs File
- LoginViewModel.cs File contents

Action filters
- ActionExecutingContext properties
- CustomActionAttribute.cs File
- effect
- HomeController.cs File
- IActionFilter Interface
- OnActionExecuted method
- OnActionExecuting method

Action method
- C# optional parameter feature
- parameters
- passing data
- redirecting
- selection
- value providers and model binders

Action results
- Built-in RedirectResult Object in DerivedController.cs File
- built-in types
- Controller Convenience Method in DerivedController.cs File
- CustomRedirectResult Class in DerivedController.cs File
- CustomRedirectResult.cs File
- ExecuteResult method

AdminController.ChangeLoginName

AdminController.cs File
- action method
- contents
- edit action method
- POST-handling edit action method

AdminLayout.cshtml File
- client-side validation
- contents
- ViewBag message

Ajax functions
- getAllItems function
- sendAjaxRequest

ArgumentNullException

Arrays and collections

ASP.NET MVC framework
- architecture
- components and facilities
- extensibility
- features
- *Glyphicons Halflings*
- HTML and HTTP
- modern API
- new features
- OSI
- routing system
- server-side event handler code
- testability
- *Visual Studio Express 2013 for Web*
- web development
  - agile methodologies
  - .NET open source
  - Node.js
  - Ruby on Rails
  - TDD
  - web standards and REST
- web forms
  - false sense, separation of concerns

- low testability
- page life cycle
- view state weight
- web forms technology stack

ASP.NET MVC Framework
Async and Await keywords
Asynchronous methods
- Async and Await keywords
- ContentLength header
- GetPageLength
- MyAsyncMethods
- MyAsyncMethods.cs.
- task continuation

Attribute routing
- combining constraints
- enabling and applying
- route constraints
- route prefix
- segment variables

Attribute values
Authentication filters
- and authorization filters
- check
  - HomeController.cs File
  - HttpUnauthorizedResult
  - OnAuthentication method
- final challenge request
- IAuthenticationFilter Interface
  - AuthenticationChallengeContext class
  - GoogleAccountController.cs File
  - GoogleAuthAttribute.cs File
  - OnAuthenticationChallenge method

Authorization filters
- and authentication
- AuthorizeCore method
- built-in
- CustomAuthAttribute.cs File
- custom authorization filter
- IAuthorizationFilter interface

B

BDD. *See* Behavior-driven development (BDD)
BeginForm method
- complex overload, BeginForm Method in CreatePerson.cs File
- CreatePerson.cshtml File
- overloads

BeginRouteForm helper method
Behavior-driven development (BDD)
Bootstrap CSS, _Layout.cshtml File
Bootstrap package
Breakpoints
- definition
- Index action method

Browser Link
- Browse With dialog window
- multiple browsers selection
- refreshing linked browsers

Built-in authorization filter
- AuthorizeAttribute properties
- protecting action method

Built-in exception filter
- custom error, Web.config File
- HandleErrorAttribute
- HandleErrorAttribute Filter, HomeController.cs File
- HandleErrorInfo Model Object, RangeError.cshtml File
- HandleErrorInfo properties

Built-in form helper methods

- form elements
  - action methods, HomeController.cs File
  - creation
  - HTML form
  - _Layout.cshtml File
  - Person.cs Model
- Bundles
  - layout and view
    - /Home/MakeBooking URL
    - _Layout.cshtml File
    - MakeBooking.cshtml File
  - model and controller creation
    - Appointment.cs File contents
    - HomeController.cs File
    - MakeBooking method
  - NuGet packages
  - script and style bundles (*see* Script and style bundles)
  - script and style sheet loading

C

- CanChangeLoginName method
- Catchall segment variable
  - in RouteConfig.cs File
  - matching URLs
- Checkout process, SportsStore
  - action method, CartController.cs File
  - Checkout.cshtml File contents
  - Checkout now button
  - DisplayName property
  - effect of, Display attribute
  - generating labels, property names
  - Index.cshtml File
  - reducing duplication, Checkout.cshtml File
  - shipping details form
  - ShippingDetails object
- Child actions
  - action methods
  - creation
  - Html.Action helper
  - List.cshtml File
- Client-side validation
  - attributes, Appointment.cs File
  - HTML attributes
  - immediate feedback
  - importing JavaScript files
  - NuGet packages
  - Web.config File
- Code-first
- Constraining routes
  - custom constraint
  - HTTP methods
  - regular expression
  - specific values
  - type and value constraints
- Controller activator
  - CustomControllerActivator.cs File
  - IControllerActivator Interface
  - in Global.asax File
  - instantiation
- Controller extensibility
  - action method
  - built-in action invoker
    - action method selection
    - custom action method selector
    - custom action name
    - handling unknown actions
  - built-in Controller Factory

dependency resolver
namespaces
custom action invoker
contents, ActionInvokerController.cs File
CustomActionInvoker.cs File
effect of
IActionInvoker Interface
Custom Controller Factory (*see* Custom Controller factory)
CustomerController.cs File
Product Controller.cs File
Result.cs File
Result.cshtml File
specialized controllers
asynchronous controller
asynchronous methods, controller
Data.cshtml File
RemoteDataController.cs File
RemoteService.cs File
sessionless controllers
session state management
start URL
Controllers
Admin controller
creating view
creation, controller class
DerivedController.cs File
MyView.cshtml File
System.Web.Mvc.Controller
creation, IController
BasicController.cs File
Execute method
HttpContextBase object
RequestContext class
System.Web.Mvc.IController Interface
CustomerController.cs File
HomeController.cs File
namespaces
additionalControllers/HomeController
disabling fallback namespaces
multiple routes, resolution control
resolution order, RouteConfig.cs File
producing output
401 Result
404 Result
action results (*see* Action results)
BasicController.cs File
DerivedController.cs File
HTML
HTTP Result Code
pass data, action method to view
ProduceOutput method
Redirections (*see* Redirections)
View Bag (*see* View Bag feature)
view model object
CRUD controller
Custom constraint
in RouteConfig.cs File
navigation
Google Chrome browser
Internet Explorer
UserAgentConstraint.cs File
Custom Controller factory
CustomControllerFactory.cs File
Fallback Controller
IControllerFactory interface
instantiating Controller Classes
ProductController class
registering
RequestContext properties

GetAll method
MVC Framework
refactoring, Home Controller
RegisterServices method
TryGet method
Custom Model Binder
AddressSummaryBinder.cs File
properties
registering
Custom segment variables
action method parameters
default value
in HomeController.cs File
URL pattern, RouteConfig.cs File
Custom view engine
Custom IView
contents, DebugDataView.cs
implementation
HomeController.cs File
IViewEngine Interface
IView Interface
registering
Render method
ViewContext properties
ViewDataDictionary properties
ViewEngineResult class

D

Data-entry application
action method
Bootstrap
CSS style sheets
default start URL
domain model
GuestResponse objects
handling forms
highlighted validation errors
Html.BeginForm helper method
HTML helper methods
HTTP POST and GET requests
input-validation-error
model binding
NuGet
RSVP form
RsvpForm view
setting
strongly typed view
styling, content
Thanks view
validation
Visual Studio IntelliSense
Visual Studio project templates
WebMail helper
Debugging MVC
application state, Debugger Windows
Call Stack window
Locals window
breakpoints (*see* Breakpoints)
controller creation
data values, code editor
Index action method
value of, variable
value, Visual Studio code editor
ViewBag.Message property
DebuggingDemo
edit and continue
bug correction

- project properties
- effect of, additions
- Stop Debugging
- unhandled exceptions
  - exception helper
  - HomeController.cs File
- ViewBag Call
- view creation
  - adding style, /Content/Site.css File
  - Index.cshtml File
- Visual Studio debugger
  - debug attribute, Web.config File
  - debug configuration
  - dialog, Web.config File
  - running

DefaultModelBinder
- binding
  - arrays
  - collections
  - properties
- complex types binding
  - action methods, HomeController.cs File
  - CreatePerson action methods
  - CreatePerson.cshtml File contents
  - nested model class, Person.cs File
- culture-sensitive parsing
- custom prefixes
  - action method, HomeController.cs File
  - AddressSummary.cs File contents
  - contents, DisplaySummary.cshtml File
  - CreatePerson.cshtml File
  - properties, AddressSummary object
- data matching
- easily-bound HTML
- simple types binding
  - default parameter value, action method
  - error processing, model property
  - nullable type, action method parameter
  - parameter value, HomeController.cs File
  - request, null value
- value, id parameter

Default values
- action and controller
- in RouteConfig.cs File
- matching URLs

Deleting products

Dependency injection container
- application
- chain resolution
- constructor parameter values
- definition
- IEmailSender interface
- object lifecycle management
- PasswordResetHelper object

Dependency injection (DI). *See also* Ninject

Deployment, Windows Azure
- Azure platform
- changes
- debug settings
- details, Azure service
- portal
- preview section, Publish Web dialog
- Products Table creation
- Publish Web dialog
- remote administration
- schema command, Server Explorer window
- schema creation
- settings
- SportsStore application

SQL statement, add data
Web.config file
web site and database creation
configuration
effect of
mvc5sportsstore_db
SportsStore application
Design-enhanced SportsStore application
DI. *See* Dependency injection (DI)
Disk files routing
application server
defining routes
route evaluation
StaticContent.html File
Domain model
Dot notation

E

Edit .cshtml File
action method
contents
updating
validation messages
EditorForModel helper method
EFProductRepository.cs File
Element visibility management
EndForm helper method
EssentialTools project
ASP.NET MVC Web Application template
Controllers, HomeController
LinqValueCalculator.cs
Product.cs
ShoppingCart.cs
view, Index
Exception filters
action, HomeController.cs File
built-in
creation
ControllerContext properties
ExceptionContext properties
IExceptionFilter Interface
OnException method
RangeExceptionAttribute.cs File
default exception handling response
error message
RangeError.cshtml view File
RangeErrorPage.html File
RangeExceptionAttribute.cs File
ViewResult object
External helper method
C# extension method
custom
child action
@Html.<helper>
Index.cshtml File
CustomHelpers.cs File
properties, ViewContext Class
TagBuilder class
TagBuilder.ToString method
ViewContext property

F

FakeMembersRepository
FastController

eBooks-IT.org

- authentication filters (*see* Authentication filters)
- authorization, action methods
- authorization filters (*see* Authorization filters)
- authorize attribute, AdminController.cs File
- contents, AccountController.cs File
- contents, HomeController.cs File
- controllers and action methods
- default authorization filter
- effect of, Authorize filter
- exception filters (*see* Exception filters)
- execution
  - multiple filters, CustomerController.cs File
  - order property
  - Order value
  - SimpleMessageAttribute.cs File
- global
  - CustomerController.cs File
  - effect
  - FilterConfig.cs File
  - Global.asax File
  - RegisterGlobalFilters
- Login.cshtml File contents
- overriding
- result filters (*see* Result filters)
- Start URL setting
- types
  - ActionFilterAttribute class
  - MVC Framework
- user credentials, Web.config File

G

GetControllerSessionBehavior
GetImage Action method
- ImageTests.cs
- ProductController.cs File
- product listing

GetPeople.cshtml file
- Ajax callbacks
- Ajax-enabled links
- contents
- degrading forms
- feedback to user
- graceful Ajax-enabled links
- JSON data
- single data action method
- unobtrusive Ajax form
- updating
- user before making an asynchronous request

GET request method
GetRouteData method
GetVirtualPath method
Glyphicons Halflings
Google Web Accelerator
Graphical user interface (GUI)
GuestResponse object
- domain class
- HTML helper method
- RsvpForm method
- Thanks view
- validation

GUI. *See* Graphical user interface (GUI)

H

Helper methods

- custom
  - external
  - inline
- CustomHelpers.cs File
- data values encoding
- Displaymessage helper method, Index.cshtml File
- encoding
  - CustomHelpers.cs File
  - HtmlHelper class
  - Razor, CustomHelpers.cs File
  - selectively encoding content
  - view engine
- HomeController.cs File
- input element, model property
  - CreatePerson.cshtml File
  - DataValue.First.Name
  - string argument
- input helpers
  - checkbox helper
  - CreatePerson.cshtml File
  - elements, basic input helper methods
  - HTML helpers
- select elements
- start URL setting
- string encoding
- String.Format method
- strongly typed input helpers

Home controller
- action method
- data selection removal

HTML helper
Html.ValidationSummary

I

IgnoreRoute Method
Image uploads
- database extension
- domain model
- product images
  - product description
  - ProductSummary.cshtml File
- saving images to database
  - GetImage Action method
  - handling image data, AdminController.cs File
  - values, EFProductRepository.cs File
- upload user interface elements
  - enctype attribute
  - support, Edit.cshtml File

Inbound URL matching
Index.cshtml file
INetworkTransport interface
Inline helper method
- changing contents, Index.cshtml File
- creation, Index.cshtml File
- ListArrayItems
- markup, changing

Integration testing
IProductRepository.cs File
IProductRespository.cs File
IValueProvider

J

JavaScript Object Notation (JSON)
- action method

eBooks-IT.org

- description
- encoding

K

- Knockout, SPA
  - Ajax functions (*see* Ajax functions)
  - and jQuery
  - bindings
    - definition
    - processing
    - testing
  - create feature
  - delete feature
  - event handler
  - extending, model
  - input elements
  - JavaScript libraries
  - model definition
  - syntax
  - testing create feature

L

- Lambda expressions
  - extension method, delegate
  - filtering expression
  - filtering extension method
  - HomeController.cs File
  - replace delegate definition
- Language features
  - anonymous types
  - assemblies
  - async methods (*see* Asynchronous methods)
  - automatically implemented properties
    - AutoProperty
    - definition
    - Name
    - Product cs.File
    - to regular property
  - automatic type inference
  - definition
  - extension methods
    - defintion
    - filtering
    - ShoppingCart class
    - to interface
    - TotalPrices
  - HomeController
  - Lambda expressions
  - LINQ (*see* Language Integrated Queries)
  - object and collection initializers
    - collections and arrays
    - construction
    - in HomeController
  - result
- Language Integrated Query (LINQ)
  - API
  - deferred
  - extension methods
  - FindProducts action method
  - OrderByDescending method
  - query data
- Layouts
  - applying
  - creation

eBooks-IT.org

View Start File
LINQ. *See* Language Integrated Query (LINQ)
LiveReload
LocalDB feature
Loose coupling

M

Mobile specific content
display modes
effect of
jQuery Mobile library
layout creation
jQuery Mobile
_Layout.Mobile.cshtml File
SportsStore application
views
data-transition attribute
effect of
FlexMenu.Mobile.html File
jQuery Mobile
List action method
ProductSummary.Mobile.cshtml File
Razor foreach expression
Mobile web development
browserstack.com
mobile-specific content
display modes
jQuery Mobile library
layout creation
views
responsive design (*see* Responsive design)
SportsStore application, tablet
Mocking tool
Model binding
ControllerActionInvoker
customization
Custom Model Binder (*see* Custom Model Binder)
custom value provider
DefaultModelBinder (*see* DefaultModelBinder)
definition
demonstration
HomeController.cs File contents
IModelBinder Interface
_Layout.cshtml File contents
manually invoking
built-in IValueProvider implementations
errors
HomeController.cs File
restricting binder data source
UpdateModel method
Person.cs File contents
RouteConfig.cs File contents
/Views/Home/Index.cshtml File contents
Model binding, SportsStore
CartController class
C# types
custom model binder
CartController.cs File
CartModelBinder Class, Global.asax.cs File
CartModelBinder.cs File contents
CartTests.cs file, SportsStore.UnitTests project
GetCart method
IModelBinder interface
Model Metadata
application
buddy class
control editing and visibility

- labels
- model class
- scaffolding

Model validation
- Appointment.cs File contents
- attribute
- built-in validation attributes
- client-side validation
  - attributes, Appointment.cs File
  - data-val-<name> attribute
  - HTML attributes
  - immediate feedback
  - NuGet packages
  - server-side validation
  - StringLength validation attribute
  - unobtrusive
  - Web.config File
- custom property validation attribute
- definition
- errors
  - input-validation-error
  - Make Booking button
  - styling checkboxes
- explicitly validating, model
  - HomeController.cs File
  - IsValid property
  - ModelState.IsValidField property
- HomeController.cs File contents
- layout creation
- messages
  - Html.ValidationSummary helper
  - MakeBooking.cshtml File
  - model-level errors
  - ModelState.IsValidField method
  - validation summary information
  - ValidationSummary method
- metadata
- model binder
- MVC client-validation *vs.* jQuery validation
- property-level validation messages
- remote validation (*see* Remote validation)
- self-validating models
- views creation
  - Appointment object
  - Completed.cshtml File contents
  - MakeBooking.cshtml file

Model-view architecture

Model-view-controller (MVC) architecture. *See* ASP.NET MVC framework

Model-view-presenter (MVP)

Model-view-view model (MVVM)

Moq
- adding, Unit Test
- adding, Visual Studio Project
- behavior, MinimumDiscountHelper Class
- creation
- defined, It class
- description
- IsInRange method
- LinqValueCalculator.cs File
- MinimumDiscountHelper class
- Object property
- Returns method
- Setup method
- ShoppingCart Class, UnitTest2.cs File
- specific values and exception

MVC application
- ASP.NET MVC project
  - controllers
  - empty project template

eBooks-IT.org

Solution Explorer
template
data-entry application (*see* Data-entry application)
visual studio
Web Pages
dynamic output
view method
MVC Framework routing systems
areas
ambiguous controller issue
creation
link generation
populating
with attributes
bypassing
customization, routing system (*see* Routing system customization)
disk files (*see* Disk files routing)
optimization package
outgoing URLs (*see* Outgoing URLs)
simplifying routes
unit test project
URL schema
clean and human-friendly
GET and POST requests
MVC pattern
AdminController.ChangeLoginName
and DI
application
ASP.NET implementation
breaking and declaring dependencies
ChangeLoginName method
controllers
dependency injection
domain model
FakeMembersRepository
imaginary application
integration testing
IUserRepository interface
models
model-view architecture
MVP
MVVM
MyEmailSender
PasswordResetHelper
smart UI application
software architecture pattern
TDD and red-green-refactor workflow
test-driven development (TDD)
test fixtures
three-tier architectures
unit testing
views
visual feedback
Web applications
MVC Projects
Browser Link (*see* Browser Link)
debugging MVC (*see* Debugging MVC)
Visual Studio MVC projects (*see* Visual Studio MVC Projects
MVC Unobtrusive Ajax
description
Synchronous Form View
Unobtrusive Ajax
MVP. *See* Model-view-presenter (MVP)
MVVM. *See* Model-view-view model (MVVM)
MyEmailSender

N

- category lists generation
  - alphabetical order
  - menu method, NavController.cs File
- category navigation menu
- creation, controller
  - menu action method, NavController.cs File
  - output, Menu action method
  - RenderAction Call, _Layout.cshtml File
- filtering product list
  - Can_Paginate test
  - Can_Send_Pagination_View_Model test
  - category filtering function
  - category support, List Action method
  - CurrentCategory
  - ProductsListView .cs File
  - query string to filter
- highlighting
  - list action method
  - menu action method
  - page count
  - selected category
  - view bag feature, NavController.cs File
- shopping cart (*see* Shopping cart)
- URL scheme
  - ASP.NET routing system
  - pagination links, List.cshtml File
  - RouteConfig.cs File
  - route summary
- view creation

Ninject
- Bind method
- calculator class, HomeController.cs File
- chains, dependency
- C# interface
- conditional binding
- constructor, LinqValueCalculator.cs File
- constructor parameter, WithConstructorArgument method
- custom dependency resolver (*see* Custom dependency resolver)
- dependency injection (DI)
- features
- functionality, Index Action Method
- Get method
- HomeController.cs File
- interface, ShoppingCart.cs File
- IValueCalculator.cs File
- kernel
- LinqValueCalculator.cs File
- request scope, NinjectDependencyResolver.cs File
- System.Diagnostics.Debug class
- tightly coupled classes
- To method
- Visual Studio Project
- WithPropertyValue method

Node.js

NUnit
- arrange/act/assert (A/A/A) pattern
- contents, MinumumDiscountHelper.cs File
- creation, project
- getTestObject method
- IDiscountHelper interface
- implementation
- reference adding, MVC project
- remaining tests, UnitTest1.cs File
- running and failing
- Static Assert Method
- testing and fixing, code
- UnitTest1.cs File

OnActionExecuted method
action filter, HomeController.cs File
context properties
performance measurement
ProFileActionAttribute.cs File
Open Source Initiative (OSI)
Optional URL segment
definition
in HomeController.cs File
in RouteConfig.cs File
matching URLs
separation of concern
Order processor
application settings, Web.config File
EmailOrderProcessor.cs File contents
EmailSettings object
IOrderProcessor.cs File contents
ninject bindings, IOrderProcessor
Ninject WithConstructorArgument method
OSI. *See* Open Source Initiative (OSI)
Outgoing URLs
fully qualified URLs, links
generation
ActionName.cshtml File
Html.ActionLink helper method
in action methods
specific route
HTML attributes
passing values
routes, RouteConfig.cs File
segment variables
routing system
ActionLink method
click on link
matching route, conditions
RouteConfig.cs File
targeting controllers
ActionName.cshtml File
different controller

P

Pagination
adding, view model
description
displaying, page links
HTML Helper Method
page links display
support, List Action Method
UnitTest1.cs file, SportsStore.UnitTests project
URLs
View Model Data
Partial views
creation
List.cshtml File
strongly typed
PasswordResetHelper class
PeopleController.cs file
action methods
contents
JSON encoding
returns JSON data
PersonController.cs file
Post/Redirect/Get pattern
POST request method
ProductController
Product.cs.File

- definition
- model metadata
- validation attributes
- verbose property definitions

Q

Query syntax

R

Razor. *See also* Razor view engine
- Action Method and View
- arrays and collections
- ASP.NET MVC Web Application
- conditional statements
- contents, Index.cshtml File
- controller
- defined, model
- insertion, data values
- layouts (*see* Layouts)
- @model Razor.Models.Product
- namespaces
- setting, attribute values

Razor view engine
- contents, HomeController.cs File
- dynamic content
  - child actions (*see* Child actions)
  - layout sections
  - optional sections
  - partial views (*see* Partial views)
  - testing, sections
- Execute method
- generated C# class
- Index action method, Home controller
- Index.cshtml File
- view search locations
  - action method, HomeController.cs File
  - custom locations
  - CustomLocationViewEngine.cs File
  - Index view
  - properties
  - registering, Global.asax File
  - /Views/Common/List.cshtml File
- Write and WriteLiteral methods

Red-green-refactor

Redirections
- errors and HTTP codes
- HTTP codes to browser
- to action method
- to literal URL
  - ExampleController.cs File
  - RedirectResult class
- to routing system URL

Regression testing

ReleaseController method

Remote validation
- action method
- action method, HomeController.cs File
- Ajax request
- Appointment.cs File
- DateTime object
- Json method
- JsonRequestBehavior.AllowGet value

Representational State Transfer (REST)

Request data

action method, information
action method parameters
and properties
default parameter values
optional and compulsory parameters
value providers and model binders
Responsive design
controller selecting view
Html.Action method
_Layout.cshtml File, Horizontal Buttons
Menu Action Method, NavController.cs File
Menu.cshtml layout
MenuHorizontal.cshtml File
revised product listing, small screen
definition
header
Bootstrap
branding
_Layout.cshtml File
mobile first *vs.* desktop first
modified SportsStore header, on an iPhone simulator
summary.cshtml File
visible-xs and hidden-xs classes
limitations
product list
div element
grid, product layout
hidden-* and visible-* classes
_Layout.cshtml File
RenderBody method
SportsStore application, smartphone
view duplication removal
conditional attributes
FlexMenu.cshtml File
local variables
menu action, NavController.cs File
null values
Razor views
ViewContext property
wrapperClasses variable
REST. *See* Representational State Transfer (REST)
Result filters
actionFilterAttribute Class
ActionFilterAttribute.cs File
effect
HomeController.cs File
IResultFilter Interface
ProFileAll filter
ProFileResultAttribute.cs File
without attributes
Route prefix
Routes. *See also* Constraining routes
Application_Start method
constraint classes
default contents
Global.asax.cs File
RouteConfig.cs File
definition
navigating to /Admin/Index
registering
Routing system customization
custom routebase implementation
custom route handler
CustomRouteHandler.cs File
IRouteHandler interface
RouteConfig.cs File
incoming URLs
outgoing URLs
RsvpForm action method

eBooks-IT.org

S

Script and style bundles
- BundleConfig.cs File
- Global.asax File
- JavaScript and CSS Files
- _Layout.cshtml File
- namespace, Web.config File
- NuGet Package
- RegisterBundles method
- Scripts.Render method
- static RegisterBundles method
- Styles.Render method

Script and style sheet loading

Security
- Account Controller (*see* Account Controller)
- authentication provider
  - Authenticate method
  - FormsAuthentication class
  - FormsAuthProvider.cs File
  - IAuthProvider.cs File contents
  - NinjectDependencyResolver.cs File
  - SetAuthCookie method
- filters
  - authorize attribute, AdminController.cs File
  - default authorization filter
  - definition
  - effect of, authorize filter
- image uploads (*see* Image uploads)
- policy
  - forms authentication, Web.config File
  - loginUrl attribute
  - username and password, Web.config File
  - Web.config file
- view
  - _AdminLayout.cshtml layout
  - AdminSecurityTests.cs
  - client-side data validation
  - FormsAuthentication.Authenticate method
  - Html.ValidationSummary
  - Login.cshtml File contents
  - login view

Selenium RC

Self-validating models
- Appointment.cs File
- effect of
- IValidatableObject interface

SessionStateBehavior Enumeration

Setter injection

Shared layouts

Shopping cart
- add to cart buttons
  - BeginForm helper method
  - Html.BeginForm helper
  - Product Summary.cshtml File View
- Cart Controller
  - AddToCart and RemoveFromCart methods
  - CartController.cs File contents
  - GetCart method
  - session state objects
- class
- contents
  - Add to cart button
  - CartIndexViewModel.cs File
  - Continue shopping button
  - Index Action Method, CartController.cs File
  - Index.cshtml File

cart and cartLine classes, Cart.cs File
CartTests.cs
contents, cart
Product
quantity, CartLine
RemoveLine method
flow
Single-page application (SPA)
advantages
controller
definition
knockout (*see* Knockout, SPA)
layout and views
Editor.cshtml
Index.cshtml File contents
_Layout.cshtml File contents
summary.cshtml File
model
Reservation.cs File contents
ReservationRespository.cs File contents
MVC and Web API references
NuGet packages
start location and
Start Debugging
testing example application
WebServices properties
Smart UI pattern
SPA. *See* Single-page application (SPA)
SportsStore. *See also*, SportsStore application
cart controller completion
CartController.cs File
Checkout method
empty cart/invalid details
ModelState property
test method
validity, shipping details
client-side validation
confirmation message
CRUD controller
deleting products
edit action method
edit and list page
editing products
edit view
list view
mobile (*see* Mobile web development)
model binding (*see* Model binding, SportsStore)
model validation
new layout
new products
POST requests
product catalog
product repository
removing items from cart
Html.HiddenFor helper method
remove button, Index.cshtml File
Remove buttons
submitting orders
checkout process
domain model
order processor
summary page
Completed.cshtml File
thank-you page
summary screen
CartController.cs File
Html.Action helper method
partial view, _Layout.cshtml File
summary.cshtml File

ErrorStyles.css File
link element, _Layout.cshtml File
validation messages
SportsStore application
adding, pagination (*see* Pagination)
basic application functionality
Bootstrap package
Bootstrap styles, layout
debugging
default route, RouteConfig.cs File
definition, Product class
design goal
development, ASP.NET MVC Framework
editing, _Layout.cshtml File
editing, List.cshtml File
error page
installing, Tool Packages
IProductsRepository.cs
NinjectDependencyResolver.cs File
NinjectWebCommon.cs File
partial view creation
preparation, database
adding datas
adding, new table
effect, SQL statement
Entity Framework
Entity Framework context
IproductRepository
LocalDB feature
MVC Framework development process
product repository
selection, data source
Server Explorer window
setting up, SportsStore database
SQL Statement, Table creation
ProductController.cs
adding, Action Method
initial contents
required project dependencies
Solution Explorer window
Views/Product/List.cshtml view
Visual Studio solution and projects
Static URL segments
aliasing
and default values, RouteConfig.cs File
mixed segment, RouteConfig.cs File
RouteConfig.cs File
String encoding, helper method
System.Web.Mvc.Controller class

T

TDD. *See* Test-driven development (TDD)
TempData message
Templated helper methods
built-in templates
complex properties
CreatePerson.cshtml File
custom editor template
Editor and EditorFor
generic template
HomeController.cs File
HTML
label and display elements
model Metadata (*see* Model Metadata)
MVC Templated HTML
Person.cs File
summary

TestingDemo
Three-tier architectures
TotalPrices
TryUpdateModel method
Type and value constraints
  in RouteConfig.cs File
  route constraint classes

U

Uniform resource locator (URL). *See also* URL and Ajax helper methods; URL routing
  default start project
  HomeController's Index method
  Html.ActionLink
Unit testing, Visual Studio. *See* NUnit
Unobtrusive Ajax Form
  AjaxOptions object
  Controller
  GetPeople.cshtml File
UpdateModel method
URL. *See* Uniform resource locator (URL)
URL and Ajax helper methods
  creating Ajax links
  CSS Styles
  graceful degradation
  JSON (*see* JavaScript Object Notation (JSON))
  links and URLs
  MVC Unobtrusive Ajax (*see* MVC Unobtrusive Ajax)
  NuGet packages
  peopleController.cs file
  summary
  Unobtrusive Ajax Form (*see* Unobtrusive Ajax Form)
  user before making
  user with feedback while making
  working with Ajax callbacks
URL routing
  attribute routing (*see* Attribute routing)
  controllers (*see* Controllers)
  custom segment variables (*see* Custom segment variables)
  default values
  optional URL segments (*see* Optional URL segments)
  patterns
    behavior
    matching URLs
    segments
    segment variables
    URL schema/scheme
  route (*see* Routes)
  setting, start URL
  static URL segments

V

Variable-length routes
View Bag feature
  ExampleController.cs File
  reading data, Index.cshtml File
  ViewResult.ViewBag property
View engines
  custom view engine (*see* Custom view engine)
  Razor view Engine (*see* Razor view engine)
  testing
    clear method, Global.asax file
    DebugData view
    requesting, unsupported view
View model object

Index.cshtml File
strongly typed views
Visual Studio MVC Projects
conventions
controller classes
layouts
views
creation
default content, empty and MVC templates
Empty project option
items summary
NuGet packages
project type, folders and assemblies selection
MVC conventions
convention over configuration
scripts folder
Visual Studio test

W, X, Y, Z

WatiN
Web API
action selection
characteristics
creation
creation
Global.asax File
mapping HTTP methods
testing
Google Chrome browser
HTTP Accept header
JSON data
WebApiConfig.cs File contents
WebMail helper method
Web standards
Worker thread pool