

Introducing

Microsoft®

WebMatrix™



Laurence Moroney

Microsoft

Introducing Microsoft® WebMatrix™

Introducing Microsoft® WebMatrix™

Laurence Moroney

Published with the authorization of Microsoft Corporation by:
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472

Copyright © 2011 by Laurence Moroney

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

ISBN: 978-0-7356-4970-5

1 2 3 4 5 6 7 8 9 M 6 5 4 3 2 1

Printed and bound in the United States of America.

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

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

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

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

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Kristen Borg

Editorial Production and Illustration: Online Training Solutions, Inc.

Technical Reviewer: John Grieb

Copyeditor: Kathy Krause

Indexer: Ellen Troutman Zaig

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

*This book is dedicated to my family: my wonderful wife, Rebecca;
my awesome daughter, Claudia; and my home run-slammin' son, Christopher.
I also and always want to thank the God of Abraham, Isaac, Jacob,
and Jesus for making it all possible.*

Contents at a Glance

1	Introducing WebMatrix	1
2	A Tour of WebMatrix	17
3	Programming with WebMatrix.....	51
4	Using Images in WebMatrix	67
5	Using Video in WebMatrix	87
6	Forms and Controls	103
7	Databases in WebMatrix	123
8	Exposing Your Site Through Social Networking	147
9	Adding Email to Your Site	163
10	Building a Simple Web Application: Styles, Layout, and Templates.....	173
11	Building a Simple Web Application: Using Data	191
12	WebMatrix and Facebook	213
13	WebMatrix and PayPal.....	229
14	Building Your Own Web Helpers	251
15	Deploying Your Site	267
16	WordPress, WebMatrix, and PHP.....	281
A	WebMatrix Programming Basics	305

Table of Contents

Foreword	xv
Introduction	xvii
Who Should Read This Book.....	xviii
Who Should Not Read This Book.....	xviii
Organization of This Book.....	xviii
System Requirements.....	xviii
Code Samples	xviii
Installing the Code Samples	xix
Using the Code Samples	xix
Errata and Book Support.....	xix
We Want to Hear from You.....	xx
Stay in Touch	xx
Acknowledgments	xx
1 Introducing WebMatrix	1
An Introduction to Web Stacks	1
The ASP.NET Web Pages Stack	4
The ASP.NET Stack	5
The PHP on Windows Stack.....	5
Installing WebMatrix	6
Building Your First WebMatrix Application	8
The WebMatrix Stack	11
The IIS Express Server.....	12
The SQL Server Compact Database	13
The ASP.NET Web Pages Framework.....	15
Summary.....	16

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

2 A Tour of WebMatrix	17
Launching WebMatrix	17
The Web Application Gallery	19
Creating a Site by Using the Web Application Gallery	20
Creating a Site by Using a Template	23
Understanding the WebMatrix Workbench.	25
The Site Workspace	26
The Files Workspace	37
The Databases Workspace.	40
The Reports Workspace	44
Summary.	49
3 Programming with WebMatrix	51
Server Programming	51
Your First Programmed Page	52
Making Your Page Dynamic	57
Sending Data to the Server.	60
Summary.	65
4 Using Images in WebMatrix	67
Creating a Page That Uses an Image	67
Creating Thumbnails and Links	70
Programming the Image Tag	73
Using the <i>WebImage</i> Helper.	76
Using Web.config to Change the Allowed Image Size	80
Resizing an Image with <i>WebImage</i>	83
Further Exercises	85
Summary.	85
5 Using Video in WebMatrix	87
Using Video in Your Site	87
Creating a Simple Video Site in WebMatrix	88
Embedding a Media Player by Using the <object> Tag	89
Using the <i>Video</i> Helper	93
Using Flash Video	95
Using Silverlight Video.	96
Using the HTML5 <video> Tag	98
Summary.	101

6 Forms and Controls	103
How Forms Work	103
A Simple Example	104
Exploring HTTP Headers with Fiddler.....	106
Exploring the Form Controls.....	109
Text Boxes.....	109
Password Boxes.....	110
Option Buttons	112
The <i>checkbox</i> Control.....	113
The <i>TextArea</i> Control	115
The <i>select</i> Control for Lists	117
Capturing Form Input.....	120
Summary.....	122
7 Databases in WebMatrix	123
Creating a Database with WebMatrix	123
Using a Database in Code	126
Adding Data to the Database	130
Editing Your Database	134
Deleting Records from the Database	140
Summary.....	145
8 Exposing Your Site Through Social Networking	147
Sharing Your Site with Others.....	147
Using Delicious	148
Using Digg	151
Using Google Reader	153
Using Facebook.....	154
Using Reddit.....	156
Using StumbleUpon	157
Using Twitter	158
Adding Twitter to Your Site.....	159
Displaying a Twitter Profile.....	159
Displaying Twitter Search Results.....	160
Rendering Xbox Gamercards	161
Summary.....	162

9 Adding Email to Your Site	163
Using Simple Mail Transfer Protocol (SMTP).....	163
Using the <i>WebMail Helper</i>	164
Building a Simple Email Application	167
Summary.....	172
10 Building a Simple Web Application: Styles, Layout, and Templates.....	173
Creating and Styling Your Site	173
Getting Your Page Ready for CSS.....	176
Adding Some Style with CSS.....	178
Using CSS Files.....	184
Using Layout Pages and Templates	187
Using <i>RenderBody()</i>	187
Summary.....	190
11 Building a Simple Web Application: Using Data	191
Creating the Database	191
Creating a Data Retrieval Page.....	192
Creating an Add Data Page.....	197
Handling Submitted Data from an Add Form	199
Adding Data to the Database	200
Creating an Edit Page.....	202
Handling Submitted Data from an Edit Form	202
Updating the Database	206
Creating a Delete Data Page.....	207
Summary.....	212
12 WebMatrix and Facebook	213
Accessing ASP.NET Web Pages Administration	213
Installing the Facebook Helpers from NuGet	217
Getting Started with the Facebook Helpers.....	218
Configuring and Initializing Facebook	219
Using a Facebook Comments Box	220
Using the Facebook Activity Feed	223
Using Facebook Recommendations.....	224
Using the Facepile Feed	225
Using the Live Stream Feed.....	226
Summary.....	227

13 WebMatrix and PayPal.....	229
Signing Up for PayPal.....	229
Creating a PayPal Sandbox	231
Using PayPal with WebMatrix.....	235
Initializing the PayPal Helper	236
Creating a Shopping Cart	237
Running the PayPal-Enabled Bakery	238
Exploring the PayPalOrder.cshtml Page	241
Setting Up Other Types of Payment	242
Going Further	248
Going Live	248
Summary.....	249
14 Building Your Own Web Helpers	251
Using the Microsoft Translator Widget.....	251
Creating a Helper for the Widget	255
Creating a Helper by Using the Translator API	257
Getting an API Key	257
Using the Translator API.....	258
Creating the Helper	261
Using the Helper	264
Summary.....	265
15 Deploying Your Site	267
Finding Web Hosting	267
Using the Publish Settings Dialog Box	272
Creating a WordPress-Based Site.....	277
Summary.....	279
16 WordPress, WebMatrix, and PHP.....	281
Creating a WordPress Site	281
Configuring Your WordPress Site.....	291
Posts and Pages.....	291
Configuring the Site Theme	294
Using the Code Editor	296
Using WebMatrix to Edit WordPress	299
Creating a Facebook Application	299
Editing Your Code with WebMatrix.....	302
Summary.....	304

A WebMatrix Programming Basics	305
Getting Started with WebMatrix Programming	305
Variables and Data Types.....	305
Common Programming Concepts.....	307
Summary.....	312
Index	313

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Foreword

It's a really exciting time to be a web developer. The array of options that are available to you to be able to put your presence out there on the Internet is better than ever before. With the coming emergence of cloud computing, giving you global availability and infinite scalability, it feels like we are on the cusp of a new era in web computing.

The book in your hands is designed to get you started on this road. It will teach you about a new product that we at Microsoft are very proud of: WebMatrix.

With WebMatrix, you can easily get introduced to the wonderful world of open source applications. It makes it easy for you to acquire, install, and configure websites using applications such as WordPress, DotNetNuke, or Orchard. Traditionally, developers choosing to do this had to deal with the intricacies of installing and configuring these apps, making sure that they worked on their server, or making sure that databases and database connectivity were properly configured on the client *and* on the server. WebMatrix is designed to abstract that plumbing away from you and make it easy for you to have a File→New WordPress or File→New DotNetNuke experience.

Of course, if you don't want to use someone else's open source code but want to build for yourself, WebMatrix also includes the brand-new, very exciting, Microsoft ASP.NET Web Pages Framework. This is a very simple, very light, but extremely powerful framework that allows you to build fully functional, data-driven websites, more quickly and easily than ever before. And when you want to take the next step and build massively scalable websites, the syntax (nicknamed *Razor*) is part of the ASP.NET MVC 3 release, and you can reuse your code and skills there.

WebMatrix is just the beginning. With this book, you'll learn how to use it, and we hope to see you building the next generation of terrific websites and open source web applications with it.

Scott Guthrie

Introduction

Microsoft WebMatrix is a new tool from Microsoft that is aimed at making web development easy. As the web has evolved, it's become apparent that web developers fall into three main categories:

- **Developers who prefer to use existing, open source web applications that they can then customize to their site's needs** These developers don't want to focus on much of the "plumbing" required to build a site (such as authentication and membership, database construction, and so on) and instead want to focus on having a modern, powerful website. The explosive growth of websites built on WordPress, Drupal, Joomla, DotNetNuke, Umbraco, and Orchard has been fueled by this preference.
- **Developers who want to build sites for themselves but who want an easy-to-use, easy-to-learn framework** These developers are generally willing to trade off ease of use and ease of learning against scalability. For this category, inline programming methodologies such as PHP or classic ASP are desirable.
- **Developers who understand the needs of scalability and who understand design patterns and the need for separation of tiers to bring about such scalability** These web developers are willing to work with tools that have a longer learning curve to get more raw power.

WebMatrix is designed to make life easier for the first two types of developer. For those who want to use open source, WebMatrix provides a complete, coherent stack on which an open source application can run, *regardless of the technology on which it's built*. So, for example, even though the popular WordPress application uses PHP and MySQL, WebMatrix makes it easy for a developer to acquire, download, and install WordPress, including all the dependencies needed to make it run on Windows.

For those who want to build sites themselves, WebMatrix comes with the Microsoft ASP.NET Web Pages Framework, which makes building webpages and websites very straightforward and uses a simple but powerful inline syntax. With this syntax (nicknamed *Razor*), you can create HTML templates and then activate them with code that is compact, fluid, and easy to read. Your investment in skills with ASP.NET Web Pages will pay off when you are ready to scale up, because it is fully compatible with Microsoft ASP.NET, including ASP.NET Web MVC and ASP.NET Web Forms.

Who Should Read This Book

If you are interested in developing websites, this book is for you. If you are a first time web developer, or someone who wants to learn how to use open source or how to build active webpages, this book gives you a great entry into that world!

Who Should Not Read This Book

Although this book is aimed at anybody who is interested in web development, if you are looking for information about how to build the next huge website for billions of users, this book (and WebMatrix) probably aren't for you.

Organization of This Book

The goal of this book is to take you step by step through several pragmatic approaches to website development. You can pretty much drop in on any chapter and gain something.

System Requirements

You will need the following hardware and software to complete the practice exercises in this book:

- Supported operating systems are Windows 7, Windows Vista, Windows Vista SP1, Windows XP SP2+, Windows Server 2003 SP1+, Windows Server 2008, and Windows Server 2008 R2.
- You must have a live Internet connection to install WebMatrix via the Web Platform Installer.
- You must have administrator privileges on your computer to run the Web Platform Installer.

Code Samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All the sample projects are available for download from the book's page on the website for Microsoft's publishing partner, O'Reilly Media:

<http://go.microsoft.com/fwlink/?LinkId=217894>

Click the Examples link on that page. When a list of files appears, locate and download the WebMatrix.zip file.

Installing the Code Samples

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

1. Unzip the WebMatrix.zip file that you downloaded from the book's website.
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.



Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the WebMatrix.zip file.

Using the Code Samples

After you unzip the downloaded file, the samples will be in subdirectories by chapter. Any chapters dealing with open source applications, such as WordPress, will require you to step through a separate process to download and install the open source application as instructed.

Errata and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site at oreilly.com:

1. Go to <http://microsoftpress.oreilly.com>.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results. On your book's catalog page, under the cover image, you'll see a list of links.
4. Click View/Submit Errata.

You'll find additional information and services for your book on its catalog page. If you need additional support, please email Microsoft Press Book Support at mspininput@microsoft.com.

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

We Want to Hear from You

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

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

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

Stay in Touch

Let's keep the conversation going! We're on Twitter: *<http://twitter.com/MicrosoftPress>*.

Acknowledgments

I'd like to thank John Grieb, my tech reviewer, for keeping me honest; Russell Jones for being patient; and all the cast and crew at Microsoft Press and O'Reilly for the tireless work they've put into getting this book in your hands!

Chapter 1

Introducing WebMatrix

In this chapter, you will:

- Discover the purpose and goals of WebMatrix.
- Install and configure WebMatrix.
- Build a WebMatrix application.
- Explore the software layers that comprise the WebMatrix stack.

Microsoft WebMatrix is a free tool from Microsoft that developers can use to create, customize, and publish websites to the Internet. WebMatrix supports many different ways to build sites. This book explores how you can use WebMatrix to build your own sites.

WebMatrix uses the concept of *templates*, each of which is a fully functional website-in-a-box. These templates are written using HTML5 and JavaScript and powered by server-side technologies such as Microsoft SQL Server Compact edition and Microsoft ASP.NET Web Pages. In this book, you'll look at how to build your own sites by using these technologies, and in this chapter you'll take a tour of one of the templates provided by WebMatrix.

If you prefer to use open source web applications rather than creating your own, WebMatrix also makes it easy for you to get up and running with the most popular open source web applications very quickly. If you are familiar with the Microsoft web platform, some of these, such as WordPress or Drupal, might come as a surprise, because they're commonly associated with the Linux web platform. The truth is that these are PHP-based applications—and PHP runs on Windows quite well, so you can use these applications just as easily on Windows as you can on Linux. In Chapter 11, "Building a Simple Web Application: Using Data" and beyond, you'll also see how to download, install, and use the most popular open source web applications with WebMatrix.

An Introduction to Web Stacks

WebMatrix gives you the ability to do all this with a single in-the-box solution that contains the entire stack that web applications need on Windows. If you're not familiar with the term *stack*, don't worry—you soon will be. A web stack, in its simplest sense, is the collection of components that a website needs in order to run. These components include the *operating system*, the *web server*, the *database*, and the *runtime and programming framework* that underpins your application.

You can see this in Figure 1-1.

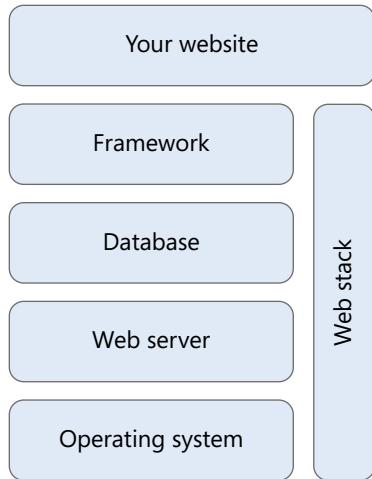


FIGURE 1-1 A typical web stack.

At the base of the stack is the Windows operating system, which is mandatory, but above that, WebMatrix gives you the option to choose the specific technologies you prefer, such as:

- **Programming framework** ASP.NET Web Pages, ASP.NET, or PHP
- **Database** SQL Server Compact, SQL Server, or MySQL
- **Web server** IIS (Internet Information Services) or IIS Express

You can see these options more clearly in Figure 1-2.

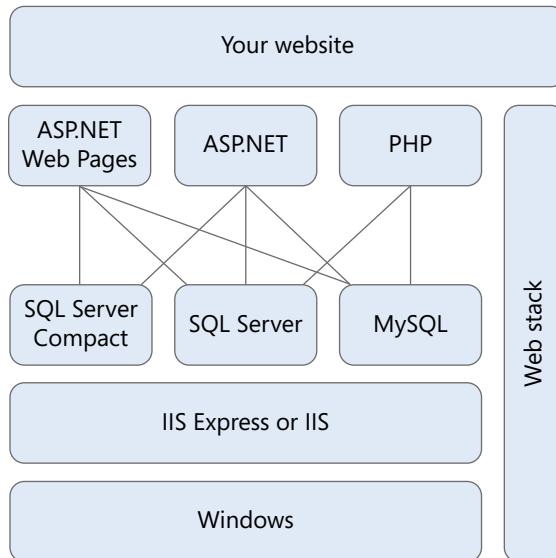


FIGURE 1-2 The WebMatrix web stack options.

At this point, all the options might look a little confusing. But don't worry; as you use WebMatrix, the options will become a little more intuitive.

You might notice that ASP.NET Web Pages and ASP.NET are shown as different elements in Figure 1-2. Although ASP.NET Web Pages is a part of the ASP.NET framework, I've listed them separately here because in WebMatrix 1.0, you effectively use them differently. You'll primarily use the ASP.NET Web Pages framework when creating new applications from a template, and you'll use ASP.NET for open source applications that have already been written using ASP.NET—specifically, the ASP.NET Web Forms or ASP.NET MVC technologies. Even though they're listed separately, the code and skills you need—and that you will explore in this book—are the same, and there's an easy migration path from ASP.NET Web Pages to ASP.NET.

Another thing to note is the connectors between the different frameworks and databases. Not every framework supports every database (in fact, ASP.NET—and of course ASP.NET Web Pages—is the only technology that supports all three), so bear that in mind when you are building applications with WebMatrix.

Note that the web server tier is represented as IIS Express or IIS. IIS stands for Internet Information Services and is the name of Microsoft's full-featured web server. In contrast, IIS Express is a simple, lightweight web server that you can use on your development machine. You'll see more details on this in the IIS Express section later in this chapter.

In general, you'll use three combinations of the stack when building applications:

- **The ASP.NET Web Pages stack** You'll use this when you build a site from a template.
- **The ASP.NET stack** You'll use this in most cases when building a site from an existing open source ASP.NET web application such as BlogEngine.NET or Umbraco.
- **The PHP on Windows stack** You'll use this in most cases when building a site from an existing open source PHP web application such as WordPress.

The ASP.NET Web Pages Stack

Figure 1-3 shows the WebMatrix stack with the ASP.NET Web Pages elements.

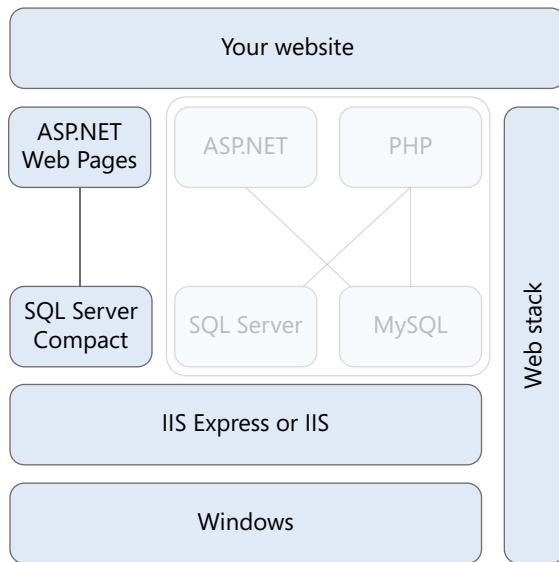


FIGURE 1-3 The ASP.NET Web Pages stack.

In this case, you build your website by using standard HTML, CSS, and JavaScript. When you need to run code on the server for dynamic or data-driven sites, you use the ASP.NET Web Pages framework. You'll be learning a lot about this in this book, so if the concepts of running code on the server or data-driven websites are foreign to you, you'll soon learn them.

Do note that ASP.NET Web Pages can work with SQL Server as well, but in most cases you'll start with SQL Server Compact. You can move up to the full version of SQL Server from there.

The ASP.NET Stack

You can see the typical ASP.NET stack in Figure 1-4.

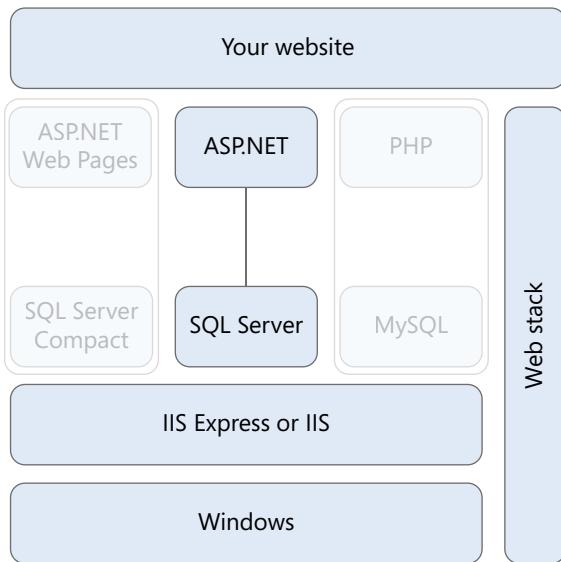


FIGURE 1-4 The typical ASP.NET web stack.

ASP.NET is flexible enough to handle any of the databases that we've discussed thus far, but typically it will use the SQL Server database, though use of SQL Server Compact is becoming more commonplace. Applications such as Orchard CMS support SQL Server Compact directly, and more applications are adding support all the time. Additionally, ASP.NET applications can use the MySQL database through a connector.

Ultimately, if you want large, scalable, secure, and reliable websites, this is the stack to use. However, if you're just getting started on the road to web development, or if you want to quickly publish a simple website, you can start with the lighter-weight ASP.NET Web Pages stack (discussed in the previous section) and easily migrate to this one as your application needs grow.

The PHP on Windows Stack

You can also use PHP to develop websites in WebMatrix. Figure 1-5 shows the PHP on Windows stack.

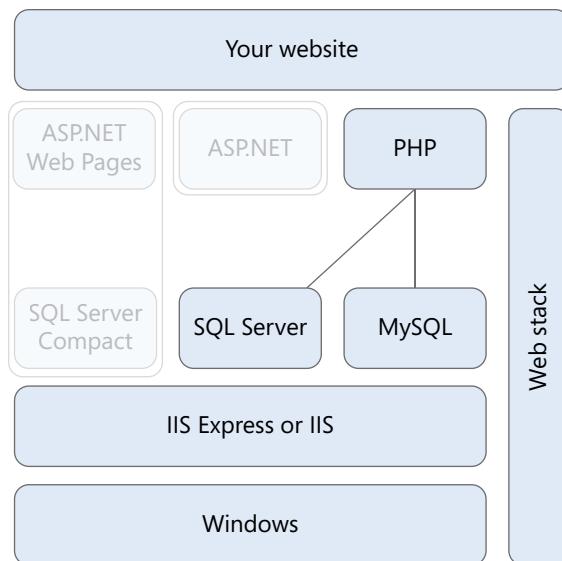


FIGURE 1-5 The PHP on Windows stack.

Applications built using the PHP stack typically use the MySQL database, but PHP can also use SQL Server as a database. However, the examples in this book (most notably WordPress) use PHP and MySQL.

But first things first; you need to get up and running with WebMatrix, and then you'll see how the individual elements of the stack work together, exploring them by building a website with WebMatrix. That's one of the strengths of WebMatrix. Although many of these technologies are disparate, WebMatrix handles the problems involved in plugging them into each other to make them work together.

Installing WebMatrix

WebMatrix is available from the Microsoft website at <http://www.microsoft.com/web>. You install it by using a tool called the Web Platform Installer (Web PI), which (in addition to WebMatrix) also offers a lot of other software for web developers. This book focuses on installing WebMatrix through the Web PI.



Note Some of the screen shots in this book might differ slightly from the screens in your version of WebMatrix—the book was written while WebMatrix was in beta.

When you visit <http://www.microsoft.com/web> and click the link to download WebMatrix, if you don't have the Web PI installed, you'll first be asked to install it. Figure 1-6 shows the first page that you'll see if you don't have Web PI installed.



FIGURE 1-6 The Microsoft Web Platform home page.

Clicking the Install WebMatrix button will launch the Web PI installation. When asked if you want to run or save the application, select Run. The Web PI program will execute and install Web PI, which will be preconfigured to install WebMatrix for you.

You can see how this looks in Figure 1-7.

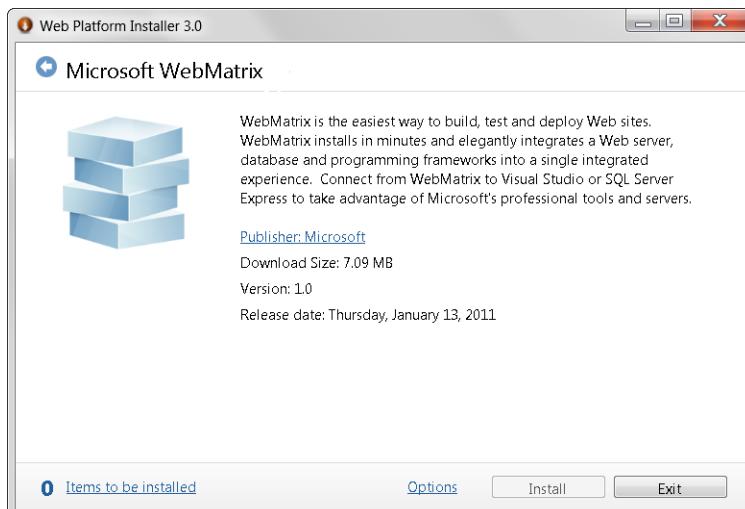


FIGURE 1-7 Installing WebMatrix.

Click the Install button, and accept the End User License Agreement (EULA). WebMatrix will install, and you'll be ready to get started.

Building Your First WebMatrix Application

Now that you have WebMatrix up and running, you'll walk through the process of using it to create your first website. You will then use that site to explore the rest of the web stack. In Chapter 2, "A Tour of WebMatrix," you'll explore the WebMatrix tool itself.

1. Launch WebMatrix. You'll get the WebMatrix welcome screen shown in Figure 1-8.



FIGURE 1-8 The WebMatrix welcome screen.

2. Choose the Site From Template option. You'll see a list of templates that come with WebMatrix. Templates are small, simple, sites-in-a-box that you can use to learn how to develop websites that use ASP.NET Web Pages. The templates shown in Figure 1-9 are included with WebMatrix. You might have a slightly different set.

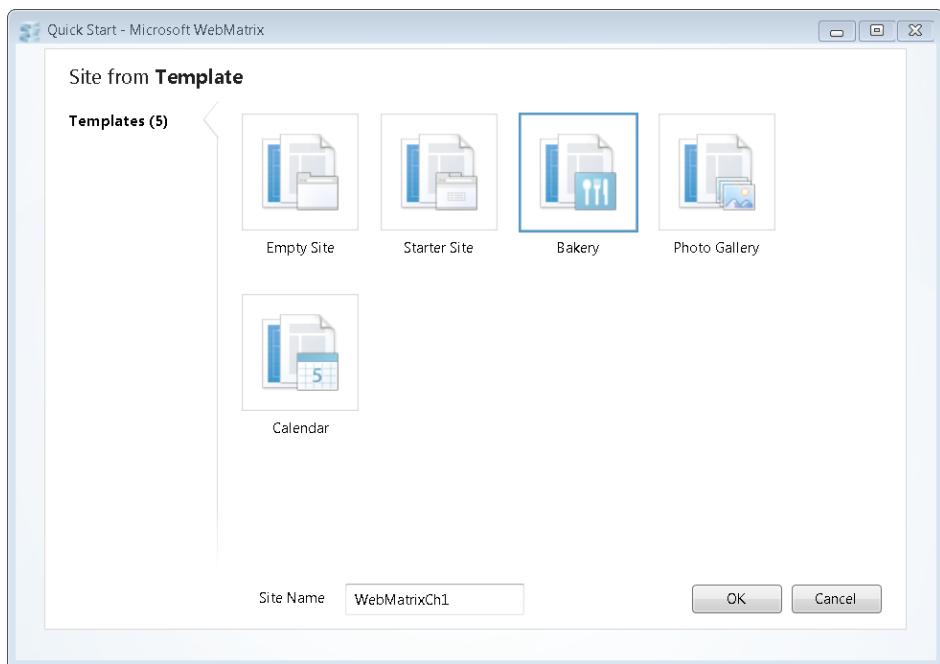


FIGURE 1-9 WebMatrix templates.

3. My favorite is the Bakery template, so click that now and name your new site **WebMatrixCh1**, as shown in Figure 1-9. Click OK when you're done.
WebMatrix will launch, and you'll see the WebMatrix workbench. You'll explore that in more detail in Chapter 2.

4. Click the Run button on the ribbon at the top of the WebMatrix window. The Bakery website will launch in your default browser and you will see the screen shown in Figure 1-10.

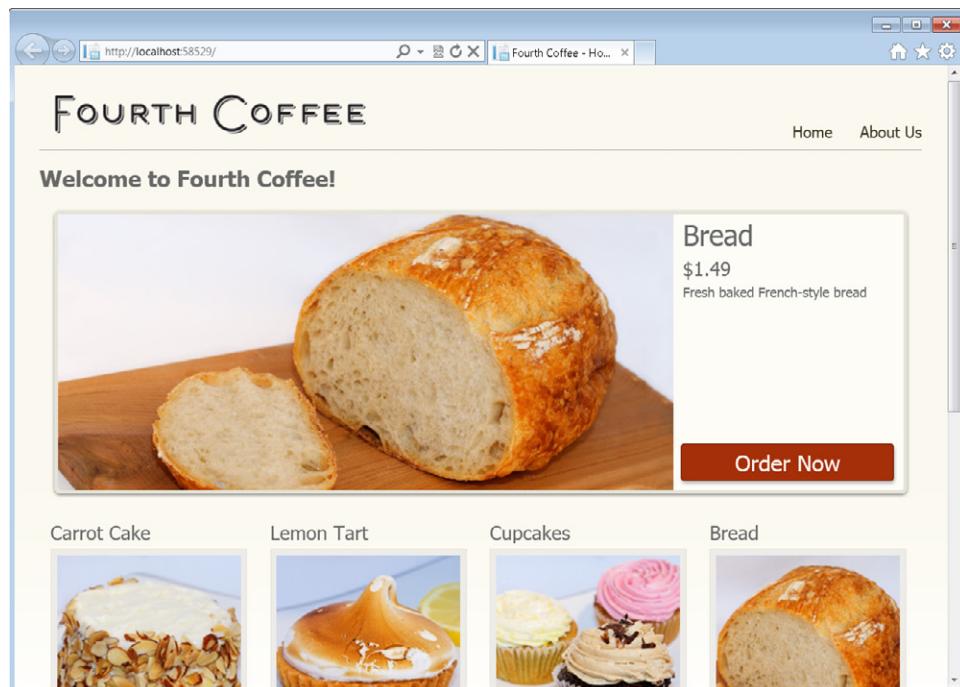


FIGURE 1-10 Running the Bakery site.

This is an example of a *dynamic* site running *server-side* code in addition to the traditional markup that you see in a webpage. This means that the details for each of the store's products are stored in a database—along with the description, price, and so on—and when a user selects a product, the dynamic site automatically generates the page content from that database content.

So, for example, if you click the Order Now button while viewing any product, you will be taken to an order detail page similar to the one shown in Figure 1-11.

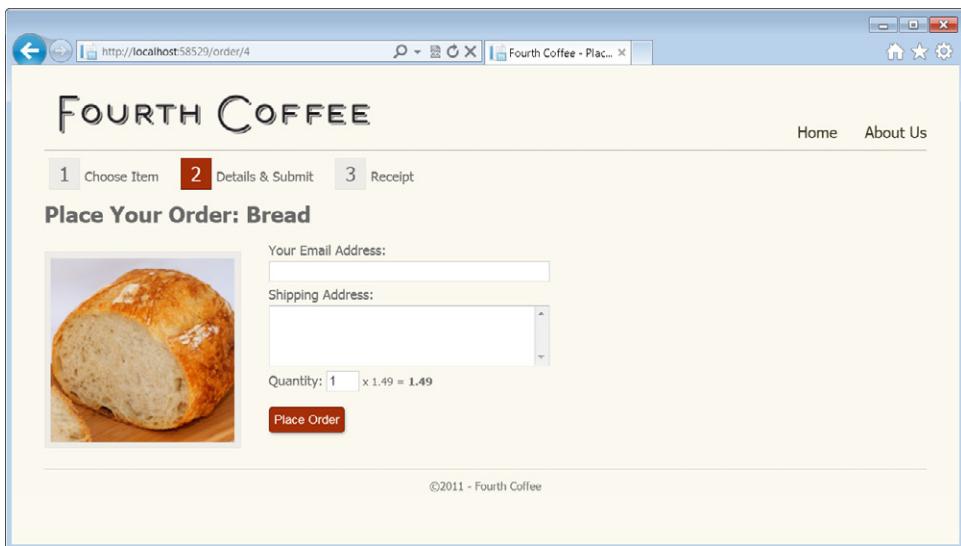


FIGURE 1-11 The order detail page for a product.

Notice how the URL shown in the browser's address bar remains the same (except for the number at the end), regardless of which product you select. That's because it is a dynamic page; the previous two figures actually show the same page, updated for every product. The number at the end of the URL tells the page which product to use. The page retrieves the selected product's picture, its description, and the price from the database. You'll learn how to do all this as you read this book!

The WebMatrix Stack

When you installed WebMatrix, the Web Installer also installed the IIS Express server, the SQL Server Compact database, and the ASP.NET Web Pages framework. This stack is shown in Figure 1-12.

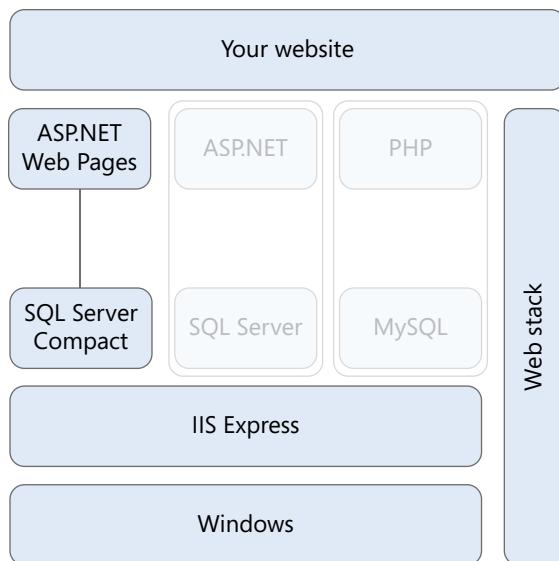


FIGURE 1-12 The ASP.NET stack in WebMatrix.

This looks very similar to the stack you saw back in Figure 1-3, except that it has only IIS Express on the web server tier.

WebMatrix comes with the entire stack you see here—IIS Express, SQL Server Compact, and ASP.NET Web Pages. It's because of this, and because it configures them to all run together, that WebMatrix can provide the simple experience that you just saw—creating a site from a template and running it. You didn't have to do any work to get the database connected to the site or to deploy the site to a server; WebMatrix just did all that for you.

With that in mind, here's a more detailed look at each of these components.

The IIS Express Server

IIS Express is a lightweight, self-contained version of IIS optimized for developers. It's designed to make it easy to use the most current version of IIS to develop and test websites. It makes your life a little easier because—unlike with the full version of IIS—you don't need administrator rights to install it, and it doesn't run as a service on your machine. Many companies don't provide users with administrator rights to install software on company machines for security reasons, so using the full IIS version on a development machine can be difficult. Additionally, the fact that IIS Express doesn't run as a service means that it's easier to use; you just launch it and go.

To see IIS in action, run your Bakery site again, and then look at the system tray on your PC, which will look something like Figure 1-13. These screen shots were taken using Windows 7, but your experience in other versions of Windows will be similar.



FIGURE 1-13 WebMatrix running IIS Express in the system tray.

You'll see the "stacked blocks" icon from WebMatrix (second from the left in Figure 1-13). If it isn't there, select the Show Hidden Icons button (click the arrow in Windows 7 to get that option), and then you'll see it.

Right-click the icon in the system tray to see a pop-up menu containing options for your IIS Express server and the applications running on it (see Figure 1-14).

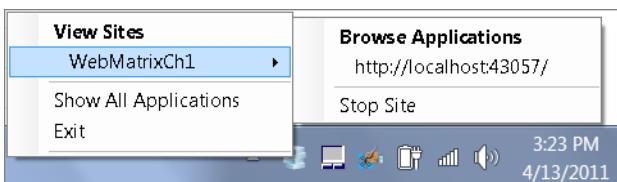


FIGURE 1-14 IIS Express options.

As shown in Figure 1-14, any currently running site is listed on the View Sites menu. You can either browse directly to that site, launching the default browser, or you can stop the running site directly from this menu. If you stop the site and it is the only active site, IIS Express will shut down as well.

The SQL Server Compact Database

Every web stack typically has a database component. In the WebMatrix stack, this is SQL Server Compact 4, the newest version of the *embedded* database from the SQL Server family. It's considered embedded because it's a file-based database that doesn't require a separate server in order to run, and because the runtime files that support the database run within the same process as your application. A common issue in website development is managing separate servers and connecting to them in order to run databases. But with SQL Server Compact, the process is as simple as deploying a file containing your data along with the rest of the files in your site—the database is included.

Click the Databases button on the left side of the WebMatrix window to take a look at the Databases workspace (you'll find out more about workspaces in Chapter 2). You'll see the bakery.sdf file that contains all your data (see Figure 1-15).

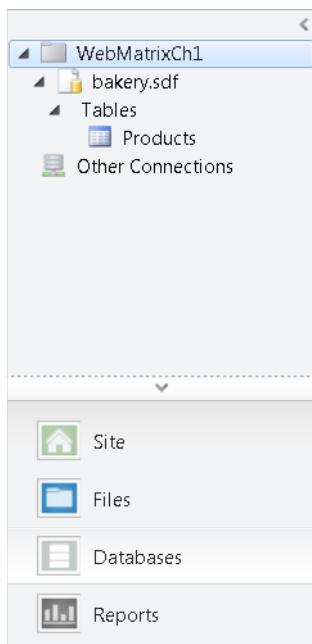


FIGURE 1-15 SQL Server Compact in WebMatrix.

Double-click the Products table shown in Figure 1-15. The WebMatrix database designer will open, and you can use it to view and edit your data, as shown in Figure 1-16.

A screenshot of the Microsoft WebMatrix database designer. The top menu bar shows 'Table' and the title 'Bakery1 - Microsoft WebMatrix'. The toolbar includes buttons for New Table, Definition, Data, New Column, Delete Column, New Relationships, View, Delete, Refresh, and Delete Row. The left sidebar shows the database structure: 'Bakery1', 'bakery.sdf', 'Tables' (selected), 'Products' (selected), and 'Other Connections'. The main pane displays the 'Products' table data. The table has columns: Id, Name, Description, Price, and ImageName. The data rows are: 1. Carrot Cake, A scrumptious mini-carrot cake encrusted..., 8.99, carrot_cake....; 2. Lemon Tart, A delicious lemon tart with fresh meringu..., 9.99, lemon_tart....; 3. Cupcakes, Delectable vanilla and chocolate cupcakes, 5.99, cupcakes.jpg; 4. Bread, Fresh baked French-style bread, 1.49, bread.jpg; 6. Pear Tart, A glazed pear tart topped with sliced alm..., 5.99, pear_tart.jpg; 7. Chocolate C..., Rich chocolate frosting cover this chocolat..., 8.99, chocolate_c...*. Below the table, there's a note with an asterisk (*).

FIGURE 1-16 Editing a SQL Server Compact database in WebMatrix.

You can also use WebMatrix to amend your database by creating new tables, setting indexes, and managing relationships. You'll learn a lot more about that in Chapter 7, "Databases in WebMatrix."

The ASP.NET Web Pages Framework

The ASP.NET Web Pages framework provides the programmable layer that your website can use to create dynamic and data-driven sites. The Bakery site uses this layer extensively, for page templates, data access, and more. Think of it this way: the browser renders HTML, but the server doesn't have to send only static HTML files to the browser, it can *generate* HTML markup on the fly, and deliver that generated HTML to the browser. You can, for example, read content from a database, loop through the results, and generate HTML from it. You don't always know how many (or whether) records exist for a particular item; hence the page is *dynamic* in nature.

As an example, you can look at the Default.cshtml page to help you see how server-side code runs. Here's a snippet of code from that page:

```
<ul id="products">
    @foreach(var p in products){
        <li class="product">
            <div class="productInfo">
                <h3>@p.Name</h3>
                
                <p class="description">@p.Description</p>
            </div>
            <div class="action">
                <p class="price">$@string.Format("{0:f}", p.Price)</p>
                <a class="order-button" href="@Href("~/order", p.Id)"
                   title="Order @p.Name">Order Now</a>
            </div>
        </li>
    }
</ul>
```

If you are familiar with HTML, you'll know what the ``, ``, and `<div>` tags do—they define how the page is rendered by the browser. The bold parts of that code, beginning with the at sign (@), are ASP.NET Web Pages-specific code (in this case, Microsoft Visual C#), using a syntax nicknamed *Razor*.

This means that when the browser calls the page, the server looks at the markup and runs the code. In this case, you can see that the code loops through a set of products. For each loop iteration, the page renders the generic HTML you can see in the code, but it also inserts information specific to a selected product into the HTML.

ASP.NET Web Pages are designed to be easy to learn and easy to use. Over the next few chapters, you'll use them to learn the basics of server programming.

Summary

This chapter introduced Microsoft WebMatrix, a simple and free web stack made up of a web server, database, and programming framework. You saw how to download and install WebMatrix, and you built your first website by using a template. In Chapter 2, you'll take a tour through the WebMatrix workbench, and the different workspaces that it provides.

Chapter 2

A Tour of WebMatrix

In this chapter, you will:

- See how to launch WebMatrix.
- Explore the Web Application Gallery.
- Learn how to create a site by using the Web Application Gallery.
- Create a site by using a template.
- Explore the WebMatrix workbench.

In Chapter 1, “Introducing WebMatrix,” you learned why Microsoft is releasing WebMatrix now. You learned about the target audience and scenarios, as well as how to obtain and set up WebMatrix. This chapter provides a more detailed tour of the WebMatrix development environment. If you were to read only one chapter in this book to get a taste of what WebMatrix is all about, this is the one!

Launching WebMatrix

If you followed the instructions from Chapter 1 to install WebMatrix, you’ll see it in the Microsoft WebMatrix folder on the Start menu. You can see this in Figure 2-1.

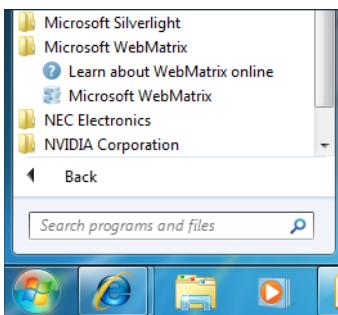


FIGURE 2-1 WebMatrix on the Start menu.

Notice the two items Learn About WebMatrix Online and Microsoft WebMatrix.

The first will take you to <http://www.microsoft.com/web/webmatrix/learn>, which provides numerous online videos and tutorials that you can follow to learn more about WebMatrix.

But in this book, you’ll dive straight into the software so you can start learning how to build websites by using WebMatrix.

When you first launch WebMatrix, you'll see the welcome screen, as shown in Figure 2-2.

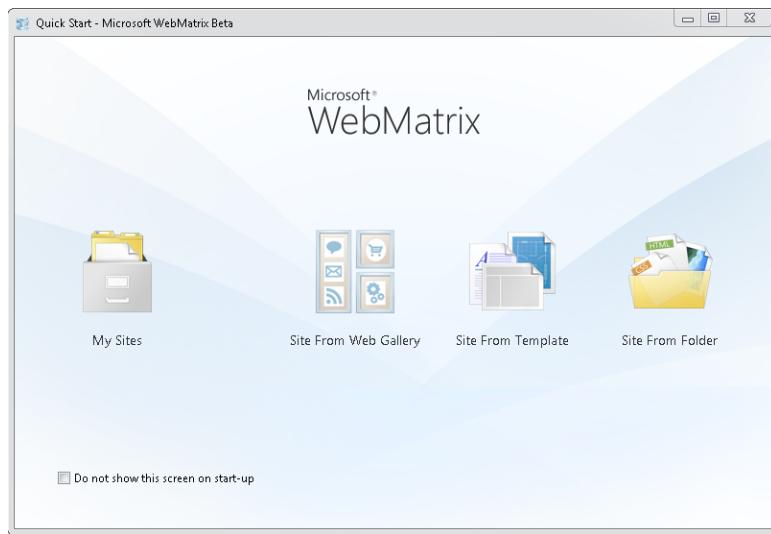


FIGURE 2-2 The Microsoft WebMatrix welcome screen.

This screen gives you a launching point for each of the main workflows you'll use when building websites using WebMatrix.

- **My Sites** Clicking this option displays a list of sites that you have already created using WebMatrix.
- **Site From Web Gallery** Clicking this displays a list of open source applications that run on the Microsoft Web Platform. From this list, you can also download and install the applications. These applications are written using a variety of languages, including PHP and Microsoft ASP.NET. When you select one of these applications, WebMatrix will detect any dependencies that you need to be able to use them. For example, many PHP applications use the MySQL database, so WebMatrix will download and install this for you if you need it.
- **Site From Template** This option allows you to create a new web application based on a template. WebMatrix comes with several templates, and it's easy to create your own.
- **Site From Folder** This allows you to turn any directory on your machine into a website. So, for example, if you've previously downloaded and installed an open source web application, you can open its root directory from here and use it within WebMatrix.

The Web Application Gallery

Select the Site From Web Gallery option on the welcome screen. You'll see the Web Application Gallery, which contains several open source applications that you can use as the basis for your website and that can simplify the task of creating a site. You can see the gallery in Figure 2-3.

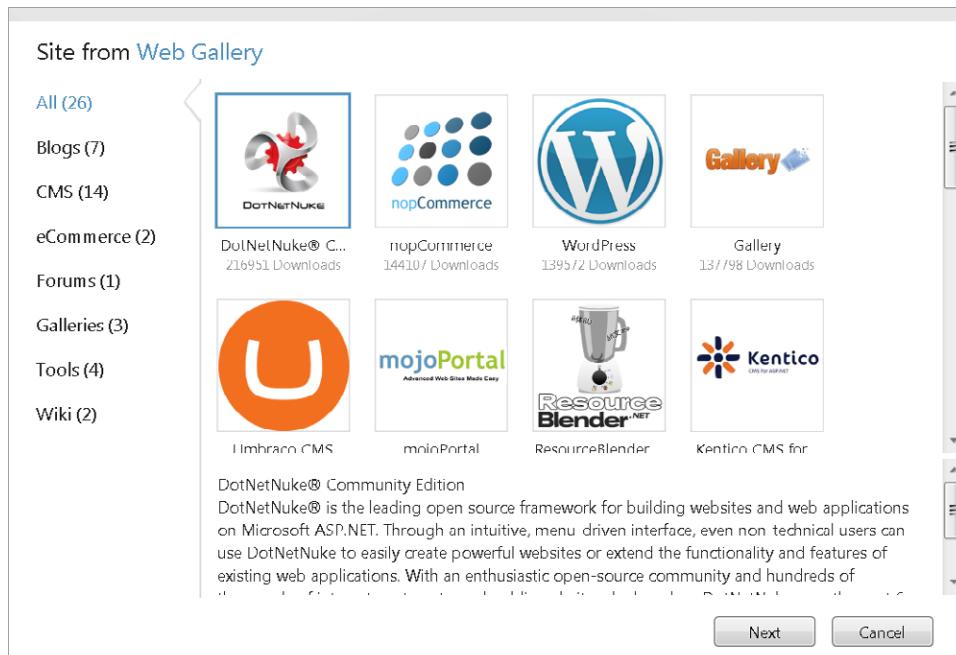


FIGURE 2-3 The Web Application Gallery.

On the left side of the screen, you'll see the categorization that the Web Application Gallery uses for different application types. The open source community has built many applications to meet many scenarios, but they typically fall into the following categories:

- **Blogs** The word *Blog* is derived from the phrase *web log* which—as its name suggests—is typically a site where users log entries in a diary-like fashion. Other users can then comment on these entries, and the entries can be syndicated using a technology called *really simple syndication* or RSS. A typical blog is used to regularly update readers on status or information. As blogging engines have grown more sophisticated, many people have found that they have enough power to serve as a complete website instead of just a diary-like log.

- **CMS (content management systems)** A content management system is a website that is built from several “blocks,” each of which can be easily created by a nontechnical user. These blocks can be arranged by a designer in a way that appeals to the end user; they’re often laid out in several columns, much like a magazine or newspaper. Site owners enter new or updated information into an administrative screen, and the CMS automatically composites the blocks and renders them as finished webpages. CMS systems can be very sophisticated, offering such features as user subscription support, bulletin boards, document libraries, blogging, and other functionality.
- **Forums** A *forum* is a piece of software that provides bulletin board capabilities where users can post messages on various topics, reply to other messages, and interact in other ways, such as via private messages. Many CMS applications include software for forums, but you might want to implement a forum by itself, without the overhead of a CMS. Forum applications provide this functionality.
- **Galleries** Galleries are websites that collect pictures or other media. A gallery gives the end user an easy-to-navigate experience in which to view the media, as well as providing features such as a user registration system and the ability for registered users to upload and manage their media, sharing the results with users of the gallery.
- **Wiki** A wiki is a type of website where users can edit the pages, allowing sites to provide powerful collaboration features. A wiki has some similarities to a CMS but generally provides a much simpler interface that focuses more on providing editing capabilities for users. Wikipedia is probably the best known wiki, where thousands of users have collaborated to create the world’s largest online encyclopedia by writing, editing, and linking content freely.
- **eCommerce** An eCommerce application brings many of the above together, with a focus on giving you the functionality to create an online store. Thus an eCommerce site will have basic CMS and gallery functionality as well as a “shopping cart” mechanism that allows people to purchase goods from your store.

The Web Application Gallery is updated constantly, so you’ll see more and more applications added over time. In the next section, you’ll step through the process of installing the popular BlogEngine.NET blogging engine. For more application types, including using PHP-based applications, take a look at Chapter 16, “WordPress, WebMatrix, and PHP.”

Creating a Site by Using the Web Application Gallery

BlogEngine.NET is an open source ASP.NET blogging project that was designed for simplicity and extensibility. You can find it in the Blogs section of the Web Application Gallery (see Figure 2-4).

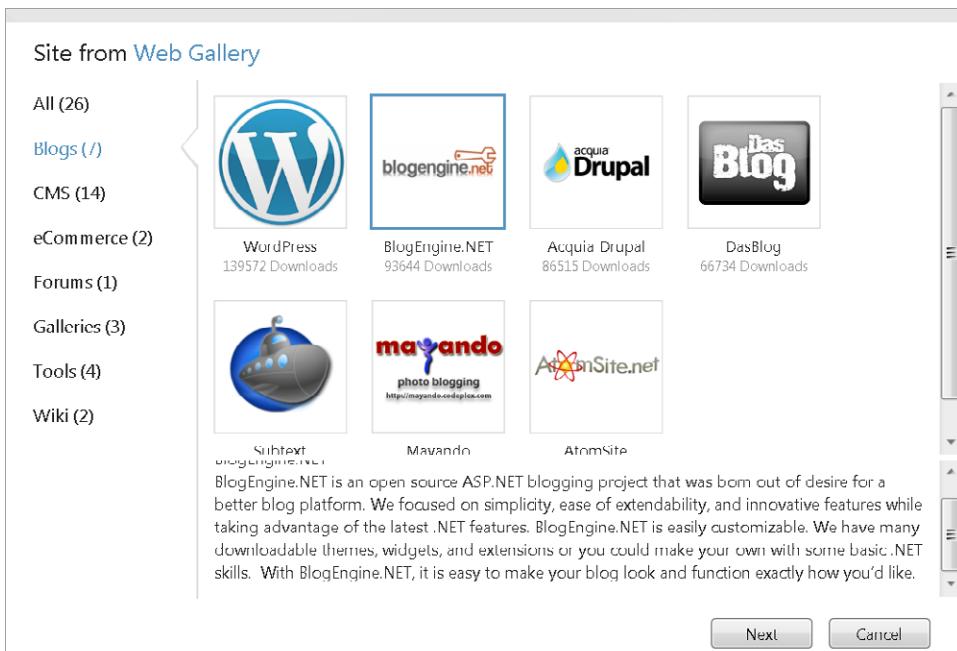


FIGURE 2-4 Installing BlogEngine.NET from the Web Application Gallery.

To install BlogEngine.NET, simply select it in the Web Application Gallery and click the Next button. As shown in Figure 2-5, you'll see the End User License Agreement (EULA) for the software, along with the download location and size.

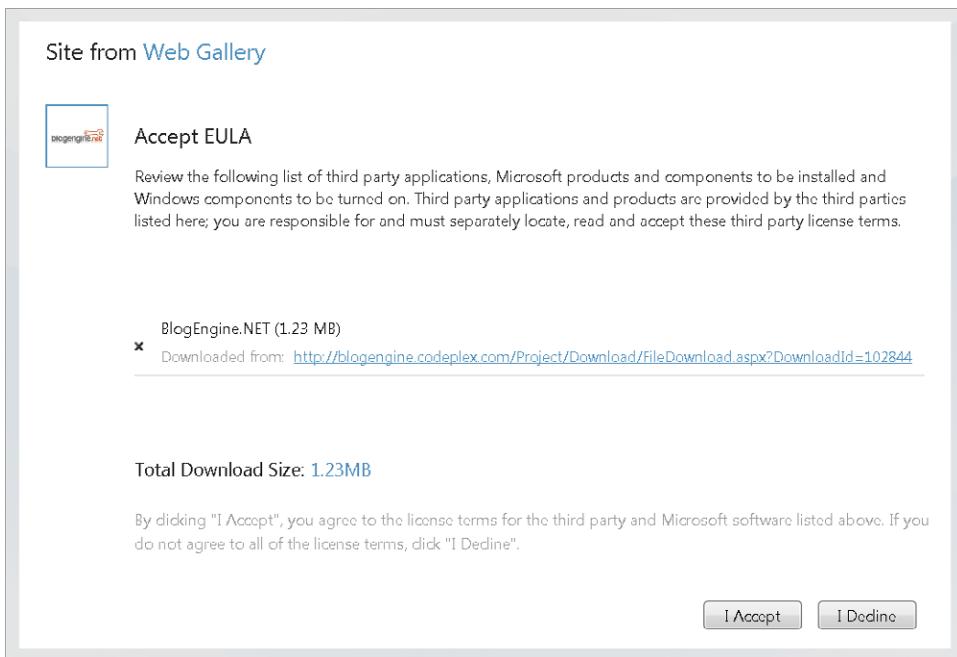


FIGURE 2-5 The BlogEngine.NET EULA screen.

BlogEngine.NET is a very small application at 1.23 MB, so it will download and install very quickly. When the installation is complete, you'll see the Success screen shown in Figure 2-6.

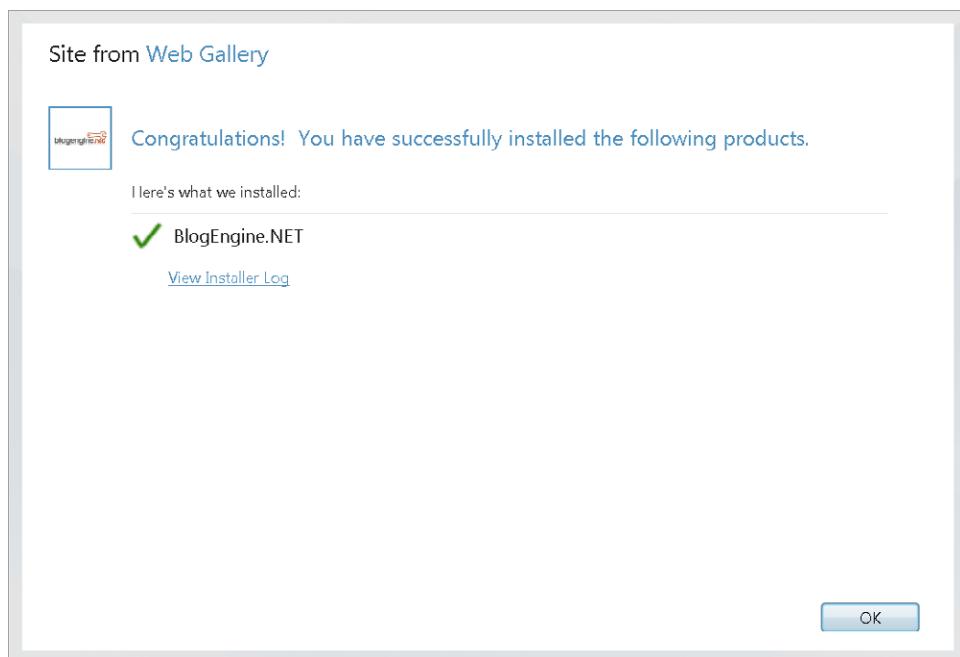


FIGURE 2-6 The BlogEngine.NET Success screen.

When you see the Success screen, click the OK button. The WebMatrix workbench will open. Over the next few sections, you'll work through the workbench, looking at each of its individual features, but for now, just click the Run button in the Site group on the Home tab of the ribbon, as shown in Figure 2-7.

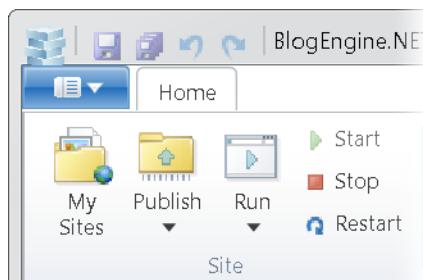


FIGURE 2-7 The Site group on the ribbon in WebMatrix.

This will open your default browser and launch the BlogEngine.NET site that you just created. You can see the default site in Figure 2-8.



FIGURE 2-8 Your first website, built using BlogEngine.NET and WebMatrix.

This brief overview should show you how simple it is to build a basic website using an existing open source application. The site is a long way from being finished, of course, but consider how complex it would have been to install a web server, a database, and a coding runtime; download and install the source code for the open source application; and then finally make it all work together. All these steps have already been done for you behind the scenes. That's a brief introduction to the power of WebMatrix. As you work through this book, you'll see many other examples of the time-saving features built into WebMatrix, but for now it's worth continuing this exploration of WebMatrix and what it can offer you as a developer.

Creating a Site by Using a Template

On the WebMatrix welcome screen, all the way back in Figure 2-2, another option was to create a new site by using a template. WebMatrix ships with several templates, including a Starter Site, a simple eCommerce site (the Bakery template), a simple gallery site (the Photo Gallery template) and a simple link directory site (the Link Directory template). It's easy to create new templates and add them to WebMatrix. Templates are being added all the time, so your screen might look slightly different than the one shown here. The Site From Template dialog box is shown in Figure 2-9.

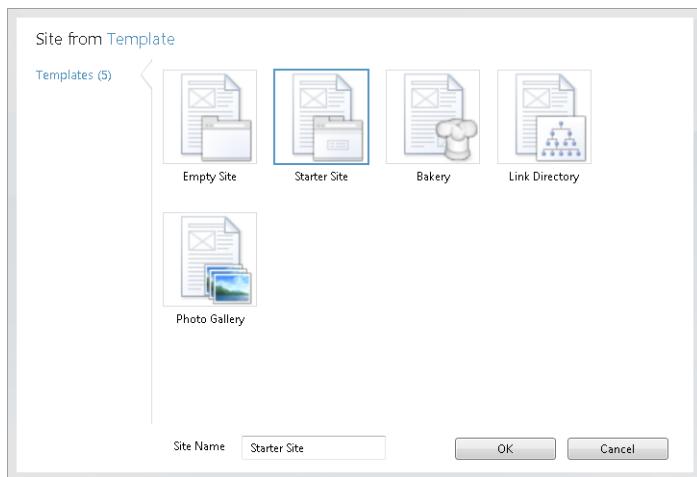


FIGURE 2-9 Creating a site from a template.

Creating a site from a template is as easy as picking the template you want and clicking OK. WebMatrix will create the site for you and load it into the workbench. Figure 2-10 shows the workbench immediately after a site has been created based on the Starter Site template.

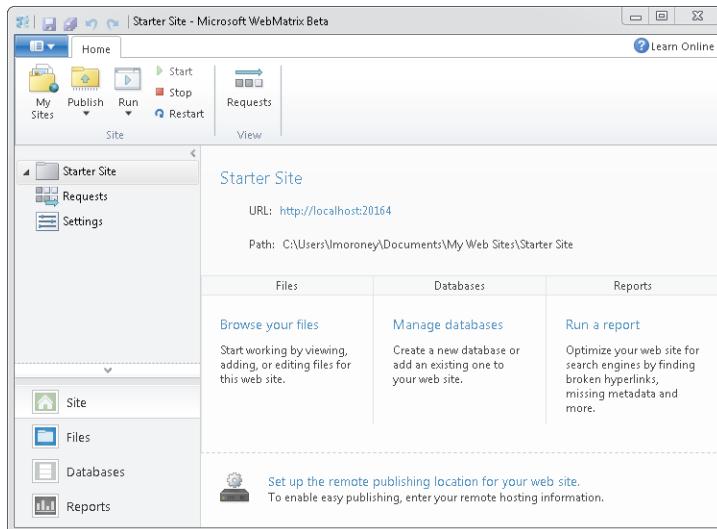


FIGURE 2-10 The WebMatrix workbench with the starter site loaded.

If you haven't done so already, fire up WebMatrix and create a starter site. You'll be using that template site in the rest of this chapter to tour through the various workspaces that the workbench provides. Click the Run button on the ribbon to launch the starter site. You can see this website in Figure 2-11.

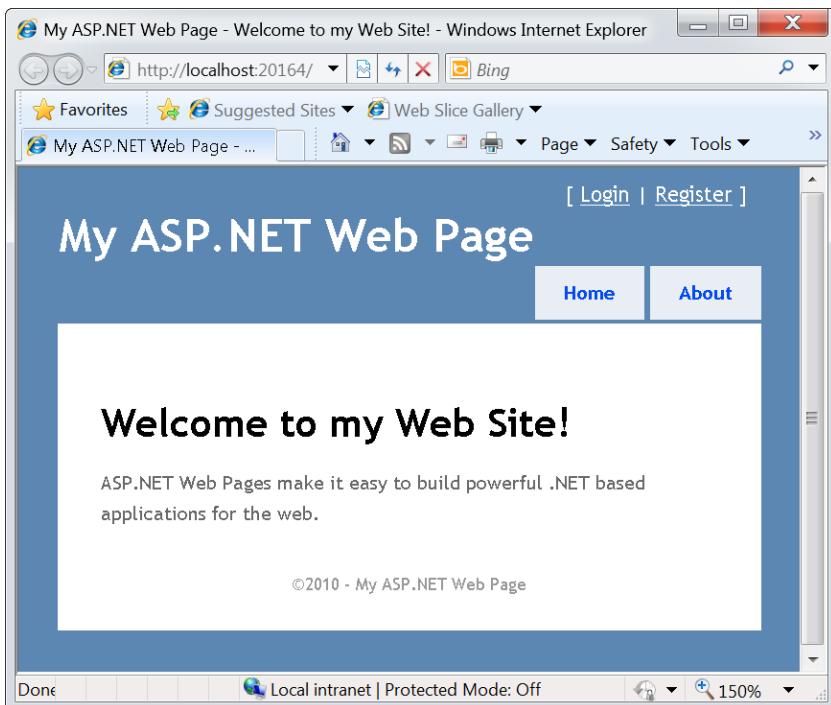


FIGURE 2-11 A site created based on the Starter Site template.

Although this site doesn't look like it does much, behind the scenes it actually implements a full role-based authentication system that lets users register for your site and, optionally, lets a site administrator assign those users to different roles. What the roles are and how they will be used is up to you. For example, you could use this as the basis for an application that manages school grades. In such an application, some roles might be Teacher, Parent, and Student. Teachers would be able to assign and change grades, but Parents and Students would only be able to view the assigned grades, not change them.

As you work through this book and learn how to program using WebMatrix, you'll see how easy it is to build applications like that!

Understanding the WebMatrix Workbench

WebMatrix includes four integrated workspaces designed to help you focus on different areas of your website. Each workspace is accessible via the tabs on the left side of the screen. Note that as you select the tabs, the ribbon also changes to support what you are currently doing.

- The Site workspace, as its name suggests, gives you the tools to manage your site itself, including the URL at which the site runs, the location of your site on the computer's hard disk, and much more.

- The Files workspace gives you access to the source files and databases that your site uses. It includes a full code editor with syntax coloring for many different file types, including ASP.NET, HTML, CSS (cascading style sheets), and PHP, to help you write the code that runs your site.
- The Databases workspace provides tools for creating, editing, and managing the databases that your site uses.
- The Reports workspace gives you access to the search engine optimization (SEO) reports that WebMatrix can run on your site.

Over the next few sections, you'll tour these four workspaces in more detail. The new website you created with WebMatrix from the Starter Site template includes some databases that you'll inspect with the Databases workspace.

The Site Workspace

When you select the Site workspace, you'll see a screen that looks something like the one shown in Figure 2-12. You'll explore the Site workspace in detail in this section.

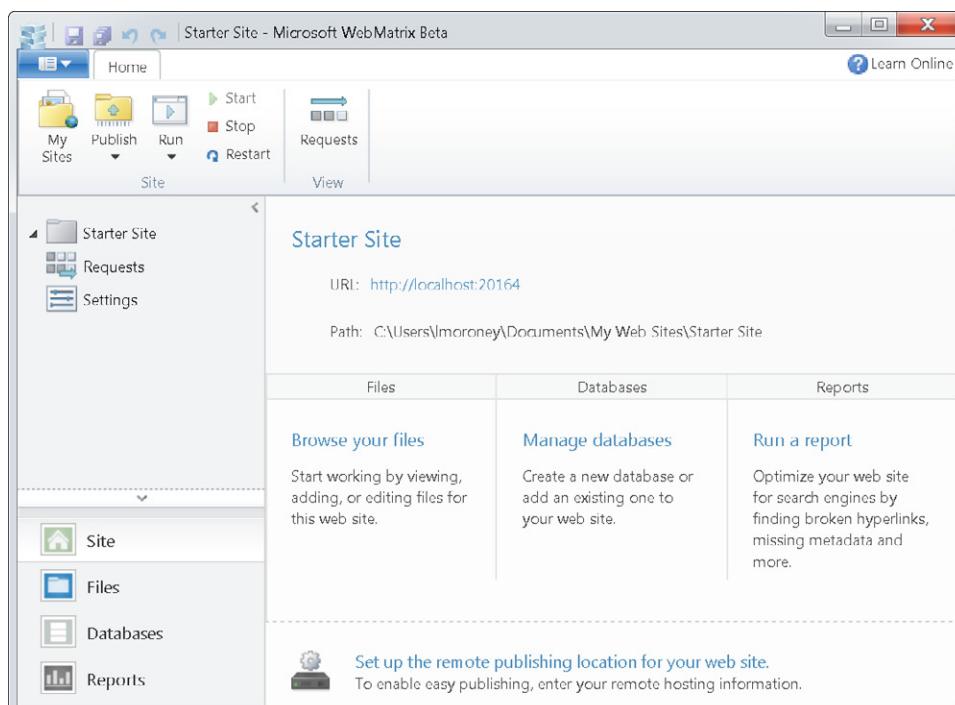


FIGURE 2-12 The Site workspace.

The Site workspace screen is broken up into three main areas. At the top is the ribbon, which gives you quick access to the various commands available from this screen. On the left you'll see both the tabs that allow you to open the other workspaces and a list containing the different *sections* of the current workspace. In Figure 2-12, for the Site workspace, these are Requests and Settings. The rest of the screen is taken up by the main work area for whatever section you've selected.

Exploring the Ribbon

The first thing to look at is the ribbon across the top of the window. The ribbon provides tools that are relevant to managing your site. You'll explore these one by one. For convenience, the ribbon is shown in Figure 2-13.

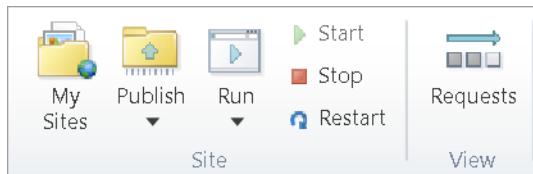


FIGURE 2-13 The ribbon in the Site workspace.

First is the My Sites button, which provides the same functionality as the My Sites button on the welcome screen that you saw at the beginning of this chapter. My Sites gives you a list of the sites that you have already created using WebMatrix—sorted into sites that you've worked on recently followed by all your sites. Although you haven't worked on many sites yet, you can get a feel for how this looks in Figure 2-14. As you can see, it's pretty straightforward to use. Simply pick the site that you want to open and click OK. Click Cancel to return to the current site.

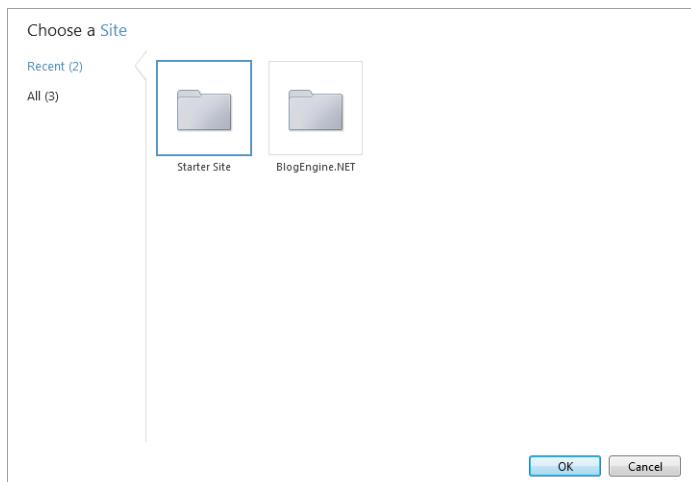


FIGURE 2-14 The My Sites dialog box.

The next button on the ribbon is the Publish button. If you look closely, you'll see that it has an arrow under it. The arrow is there because this is a combined button and menu control, which is quickly becoming a standard with Microsoft Office and Windows user interfaces. Clicking the button portion of the control performs the button's basic action (in this case, Publish), whereas clicking the arrow gives you access to supplementary features (in this case, Settings and other features, as shown in Figure 2-15).

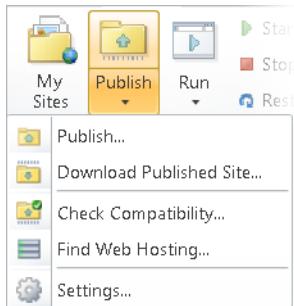


FIGURE 2-15 The Publish button.

This is part of the design philosophy of WebMatrix. It's designed to be an all-in-one, end-to-end solution, from acquiring code through publishing your site to the Internet. Thus, as part of the Publish workflow, you might need to configure your settings, or you might need to find a site on the Internet that will host your application. WebMatrix has been designed to bring all this functionality into one place. You'll learn more about finding a host and configuring your site for publication in Chapter 15, "Deploying Your Site."

The Run button on the ribbon also includes an embedded menu, which lets you select a browser in which to run your site. Figure 2-16 shows what the options would look like for a system on which both the Windows Internet Explorer and Opera browsers are installed.

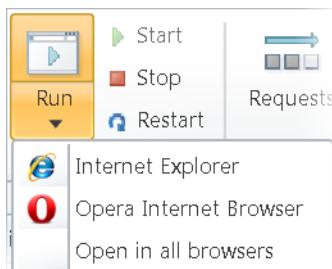


FIGURE 2-16 The Run button.

As you can see in the figure, the Run button also provides the option of running your application in all browsers installed on your system, so you can test your site in multiple browsers at the same time with one click. Clicking the Run button without using the menu launches the site in the default browser installed on your machine.

Figure 2-17 shows the starter site running in Opera.

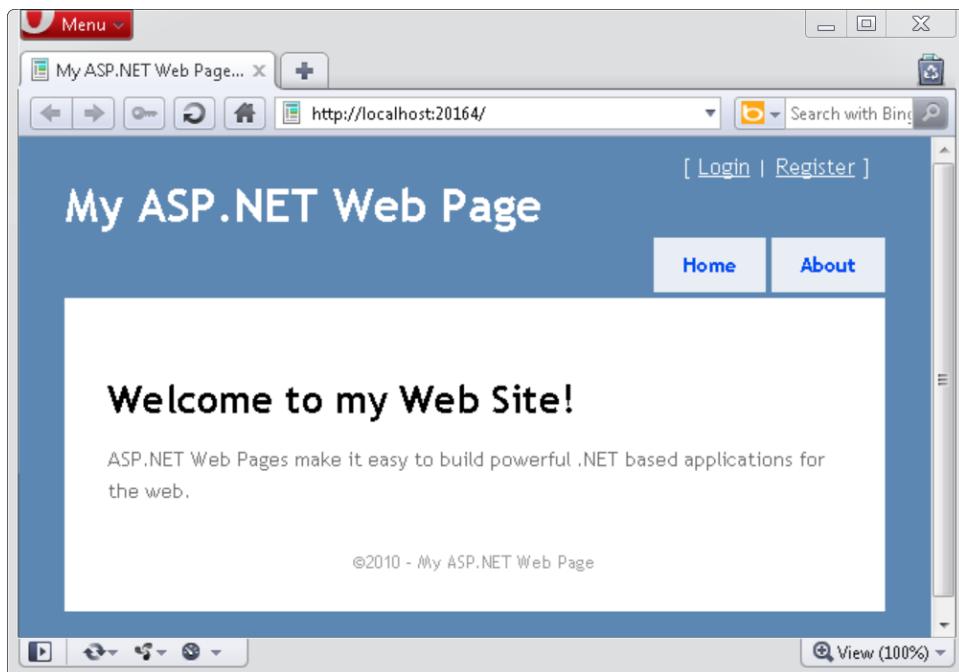


FIGURE 2-17 The starter site running in Opera.

To the right of the Run button, you'll see three smaller buttons: Start, Stop, and Restart. These control the built-in IIS Express web server that was discussed in Chapter 1. Running a site automatically starts the server if it's stopped.

When you start the server manually, you'll notice that a couple of things happen. First, the WebMatrix user interface updates to display the server status, using a notification update at the bottom of the window, as shown in Figure 2-18.



FIGURE 2-18 Server status notification.

Additionally, the system tray will show an icon signifying that IIS Express is running. You can right-click this icon to manage your site instances. You can see the IIS Express icon in Figure 2-19.

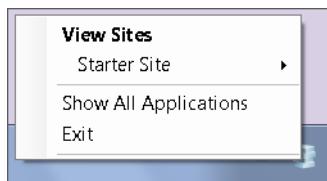


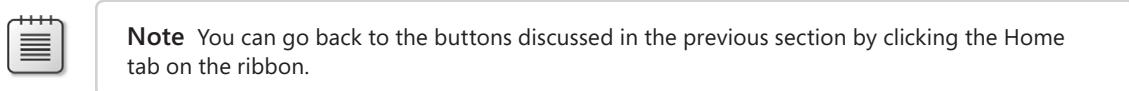
FIGURE 2-19 The IIS Express server icon.

The final button on the ribbon is Requests. Clicking Requests opens the Requests management tools in your workspace.

Managing Site Requests

You can access the Requests management part of the Sites workspace either by clicking the Requests section button on the left side of the window or by clicking the Requests button on the ribbon. Requests are calls from the browser to your site to get content. Whenever you launch the browser to test the site, you'll generate a request to the default page, which in turn will generate requests to various other files in your site, such as images or style files.

You'll also see a new tab named *Requests* on the ribbon.



You can see all this in Figure 2-20.

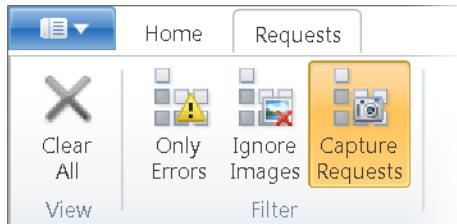
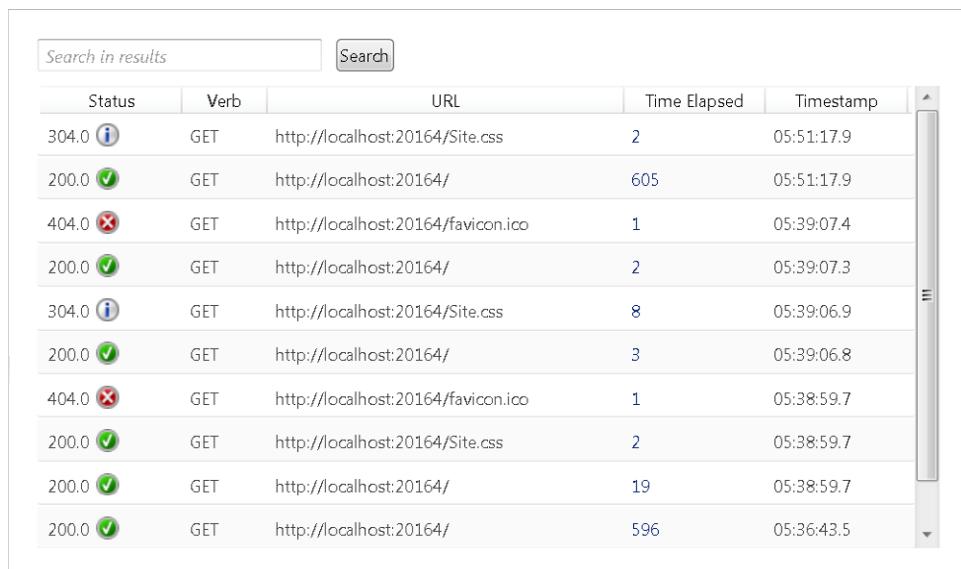


FIGURE 2-20 The Requests tab on the ribbon.

The ability to view requests is extremely useful for troubleshooting common errors in your site. For example, one classic error that just about every web developer encounters is the “missing images” error, which occurs after you’ve created a site that uses the HTML `` tag to specify an image. This tag contains an `src` attribute that specifies the path to the actual image. If the `src` path is specified incorrectly, the browser can’t render the image; it renders a default “broken” image instead. You can use the Requests tools to narrow the problem down quickly.

As shown earlier in Figure 2-20, the Requests tab includes buttons that allow you to filter the monitored requests to display only errors, so that the screen isn’t overpopulated with successful requests. Another filter option is to ignore image requests, which helps you track down problems other than missing image errors.

The requests themselves are listed in the main workspace, as shown in Figure 2-21. Each request shows you the HTTP status of the call, along with an icon representing the result of the call, the HTTP verb used to make the call (`GET` or `POST`), the URL for the requested resource, the time (in milliseconds) it took for the call to complete, and a time stamp for the call itself.



The screenshot shows the Requests tab in the WebMatrix interface. At the top, there is a search bar labeled "Search in results" and a "Search" button. Below the search bar is a table with ten rows of data, each representing an incoming request. The columns are labeled "Status", "Verb", "URL", "Time Elapsed", and "Timestamp". Each row contains an icon representing the request's status (e.g., green checkmark for success, red X for error, blue info icon for informational), the HTTP verb (GET), the URL, the time taken in milliseconds, and the timestamp. The table has a vertical scrollbar on the right side.

Status	Verb	URL	Time Elapsed	Timestamp
304.0 ⓘ	GET	http://localhost:20164/Site.css	2	05:51:17.9
200.0 ✓	GET	http://localhost:20164/	605	05:51:17.9
404.0 ✗	GET	http://localhost:20164/favicon.ico	1	05:39:07.4
200.0 ✓	GET	http://localhost:20164/	2	05:39:07.3
304.0 ⓘ	GET	http://localhost:20164/Site.css	8	05:39:06.9
200.0 ✓	GET	http://localhost:20164/	3	05:39:06.8
404.0 ✗	GET	http://localhost:20164/favicon.ico	1	05:38:59.7
200.0 ✓	GET	http://localhost:20164/Site.css	2	05:38:59.7
200.0 ✓	GET	http://localhost:20164/	19	05:38:59.7
200.0 ✓	GET	http://localhost:20164/	596	05:36:43.5

FIGURE 2-21 Viewing incoming requests.

This display shows only high-level request information, but you can select any request to get more detailed information about that request. For example, the first request in the list in Figure 2-21 has an “information” icon in the Status column. I’m curious to see what that means, so I select that line.

The right side of the workspace updates with the details for that first request, which you can see in Figure 2-22.

The screenshot shows the Microsoft WebMatrix interface. On the left, a table lists several HTTP requests. The first row, which corresponds to the selected item in the list, is highlighted with a blue background. This row contains the status code 304.0, the verb GET, and the URL http://localhost:20164/Site.css. To the right of the table, a detailed view of the selected request is displayed. The title of this view is "304.0 Content Not Modified". Below the title, the URL is shown as "http://localhost:20164/Site.css". The "Path:" field contains "C:\Users\lMoroney\Documents\My Web Sites\Starter Site\Site.css". The "Referrer:" field contains "http://localhost:20164/". A "Details" section explains that the browser used a cached copy instead of downloading it from the Web site. A "Recommendations" section states there is no action required unless the user believes the browser should not have used cached content.

Status	Verb	URL
304.0	GET	http://localhost:20164/Site.css
200.0	GET	http://localhost:20164/
404.0	GET	http://localhost:20164/favicon.ico
200.0	GET	http://localhost:20164/
304.0	GET	http://localhost:20164/Site.css
200.0	GET	http://localhost:20164/
404.0	GET	http://localhost:20164/favicon.ico
200.0	GET	http://localhost:20164/Site.css
200.0	GET	http://localhost:20164/

FIGURE 2-22 Request details.

The details view shows you considerably more detail about any individual request. For example, you can see that the status message 304.0 means that the resource for this request wasn’t downloaded from the server because it was already cached in the browser.

In addition to request details, the details view also provides recommendations about what you might be able to do to solve request problems. As an example, in Figure 2-21 you can see that there are a couple of 404.0 messages (the requests with a red X beside them), indicating that those requests failed. When the first of these requests is selected, the details shown in Figure 2-23 are shown.

The screenshot shows the 'Request Details' pane of the WebMatrix interface. On the left is a table of requests, and on the right is a detailed view of a specific request.

Status	Verb	URL
304.0	GET	http://localhost:20164/Site.css
200.0	GET	http://localhost:20164/
404.0	GET	http://localhost:20164/favicon.ico
200.0	GET	http://localhost:20164/
304.0	GET	http://localhost:20164/Site.css
200.0	GET	http://localhost:20164/
404.0	GET	http://localhost:20164/favicon.ico
200.0	GET	http://localhost:20164/Site.css
200.0	GET	http://localhost:20164/
200.0	GET	http://localhost:20164/

404.0 File Not Found

<http://localhost:20164/favicon.ico>

Path: C:\Users\lmoroney\Documents\My Web Sites\Starter Site\favicon.ico

Referrer: http://localhost:20164/

Details
The URL specified does not exist on the Web server.

Recommendations
Make sure that the resource exists and that there is not a typo in the URL.

[Edit](#) [More Information](#)

FIGURE 2-23 Inspecting the details for a failed request.

Looking at the request details pane, you can quickly see that the requested file (favicon.ico) was not found. Every web browser looks for a favicon.ico file and displays it as the “site icon” for that site, which gives each site a bit of a unique personality. You don’t need to explicitly specify this file in your pages—browsers always look for the file in the site root. For example, if you visit <http://www.microsoft.com/web>, an M icon appears in the browser’s address bar beside the URL. You can see this in Figure 2-24.



FIGURE 2-24 Viewing the favicon.ico using the browser.

You get this icon because the site administrator for Microsoft added a favicon.ico file to the root website. The browser makes the request automatically and displays the icon when the favicon.ico file exists. The starter site includes the favicon.ico file, so to reproduce this and explore the error, simply delete it before running the report.

Managing Site Settings

When you select Settings from the list on the left side of the screen, you'll see the Site Settings section (see Figure 2-25), which allows you to configure several things that your site needs when running on your computer.

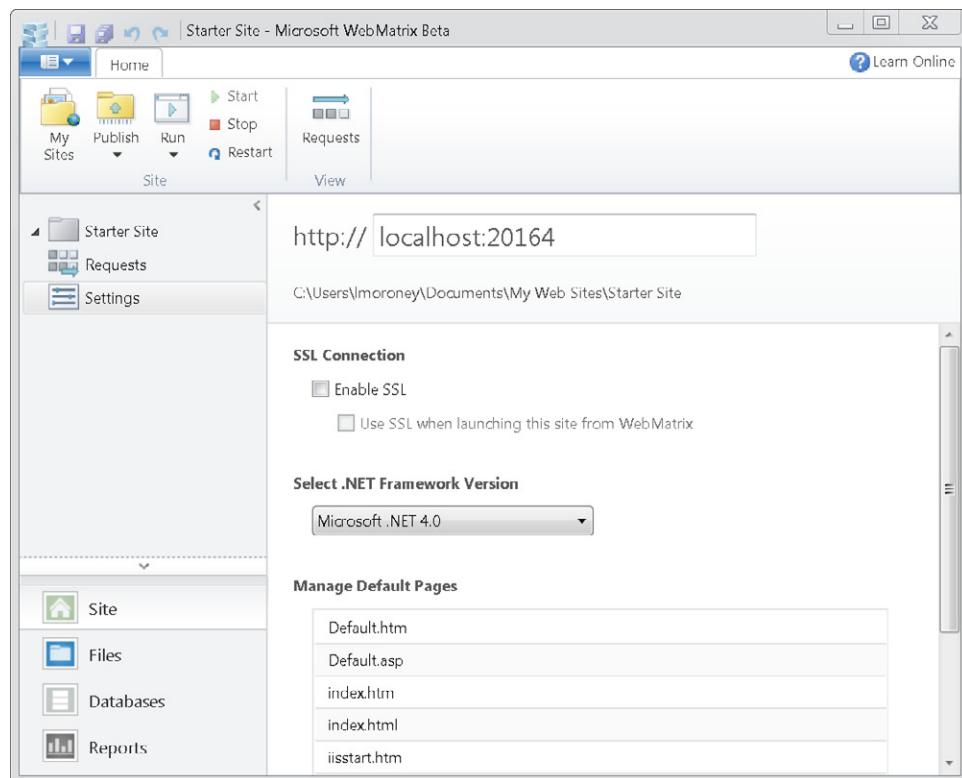


FIGURE 2-25 The Site Settings section.

At the top, you can set the URL that WebMatrix will use to run the site on your computer. This URL will typically follow the syntax `http://localhost:<NUMBER>`. In Figure 2-25 it is `http://localhost:20164`. The number is the *port* number from which the local web server serves your pages. On most web servers, this will typically be port 80, which is the standard TCP/IP port for running webpages that use the HTTP protocol. However, on your development machine, you might have many different sites, so WebMatrix gives each one a different port number to keep them all separate on the IIS Express server. If you don't want to use the automatically assigned number (20164, in this case), you can simply change it here.

Using SSL Some sites use the HTTPS (HTTP Secure) protocol when they are dealing with sensitive information such as password sign-ins for email or banking. This protocol uses security and encryption services that are built into both the browser and the server. Secure Sockets Layer (SSL) pages require that the server identify itself by using a certificate validated by an independent verification authority. It's a little bit like using a passport when you enter a new country; you need a passport issued by your country of residence to validate your stated identity.

You can see an example of the verification for a validated page in Figure 2-26.



FIGURE 2-26 A site using HTTPS and SSL.

In Figure 2-26, you can see that the protocol is HTTPS (not HTTP) and that the browser has identified that the site you are about to send sensitive information to (in this case, a password) is in fact Microsoft's site as identified by VeriSign.

This secure way to browse pages is made possible by the SSL protocol. The IIS Express server that comes with WebMatrix supports SSL, so if you need to provide SSL-based security, you can select the Enable SSL check box. To test the site in secure mode on your local machine, you can select the Use SSL When Launching This Site From WebMatrix check box. After doing this, click the Save button on the title bar to save your settings. The screen changes to provide a new port on which you can test the SSL-secured version of your site (see Figure 2-27).

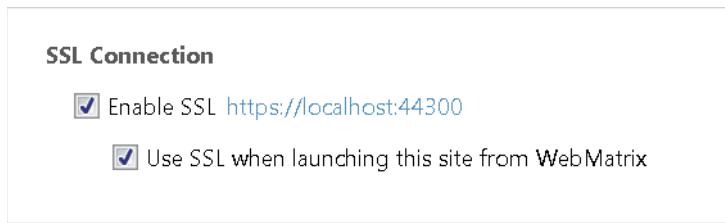


FIGURE 2-27 Configuring your site to use SSL.

Your site must have a certificate to validate its identity when running on SSL, and because you don't have a certificate yet, you will encounter some problems when running the site. Each browser handles such problems differently. Figure 2-28 shows Internet Explorer's response to attempting to browse a site that uses SSL when no certificate is present.

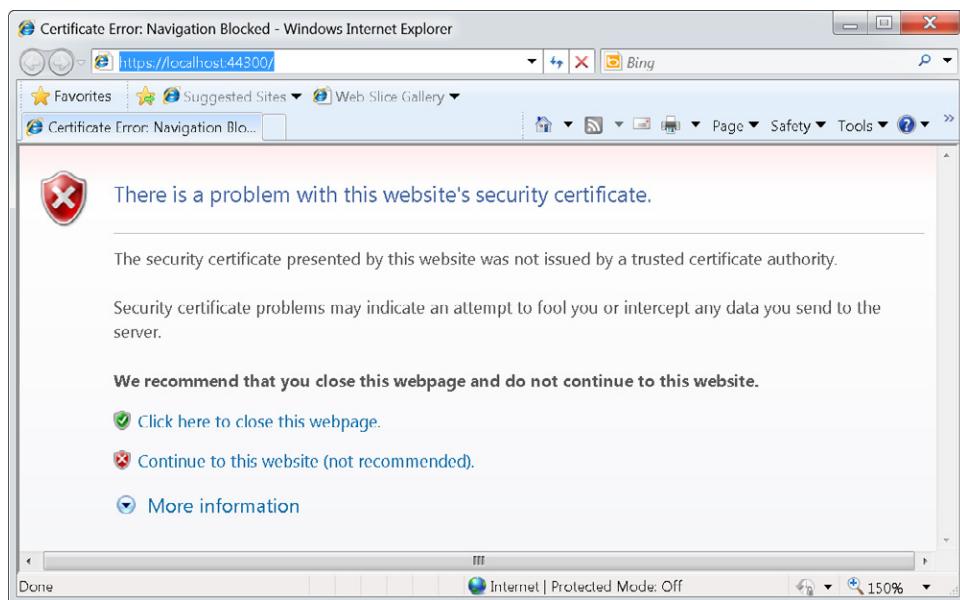


FIGURE 2-28 Navigating to a secure site where there is no certificate.

When this error occurs, users are given the choice of whether to continue to the page—with the recommendation that they should not continue. Remember, this error occurs because the browser is being told to go to a secured site, but the site has no certificate to prove its identity. If you are working on a Windows Server machine and can configure a certificate for *localhost*, you can bypass this problem. Similarly, you can also avoid it when you deploy to a hosting provider and purchase a certificate from that provider, along with the appropriate server configuration. You'll need to consider these steps if you want to use SSL on your site.

Managing default pages The final setting on the Site Settings screen lets you manage *default pages*. This is a list of pages that the server will return when no page is specified in the request URL. The server will return only the *first* page that it finds on this list. If it finds none of them, the server returns an error.

In this example, the site root is *http://localhost:20164*. When you request this, note that the request doesn't specify a specific page. The server looks in the requested directory and finds that the page Default.cshtml is a default page, so it returns that page in response to the request. You can add to or remove default pages from the site easily using this Site Settings screen. As an example, you could create a new page called foo.html and add it to the default pages list. The list has buttons that allow you to move a page up or down in the default page sequence. If you leave the foo.html file name at the top and create a page called foo.html in your site, when users visit your site without specifying a page name, they'll get the foo.html page. If the page did not exist, the server would try to return the next default page in the list—Default.htm in this case—and so on. The server will always return the *first* default page on the list that it finds.

The Files Workspace

As its name might suggest, the Files workspace is used to manage all the files in your website, including files that contain HTML markup, code, images, styles, databases, or anything else. When you select the Files workspace, you'll notice that the ribbon changes yet again, this time to provide operations that are appropriate to working with files.

You can see the Files workspace ribbon in Figure 2-29.

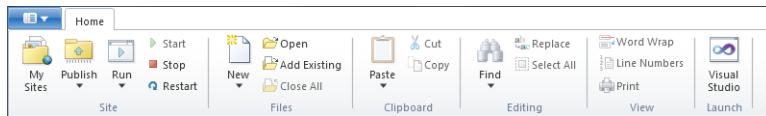


FIGURE 2-29 The Files workspace ribbon.

The Site section toward the left is the workhorse of managing your websites. By using the buttons on it, you can look at all the sites on your machine, publish your site (more on this in Chapter 15), run your site on any of your installed browsers, and start/stop/reset your IIS Express.

The rest of the ribbon is pretty self-explanatory, providing the shortcuts for cutting, copying, pasting, and other editing options. However, the Visual Studio button is noteworthy. If you have Microsoft Visual Studio installed, this button will launch it, opening it to your current project. If you don't already have Visual Studio installed, it will give you the option to download and install the free version, Microsoft Visual Web Developer 2010 Express. Although Visual Web Developer 2010 Express is free, it's full-featured, providing you with the ability to develop websites, applications, and Microsoft Silverlight-based rich Internet applications (RIAs).

Creating New Files

Clicking the arrow on the New button provides options for creating a new file or folder. If you open the menu and click New File, you'll see the dialog box in Figure 2-30.

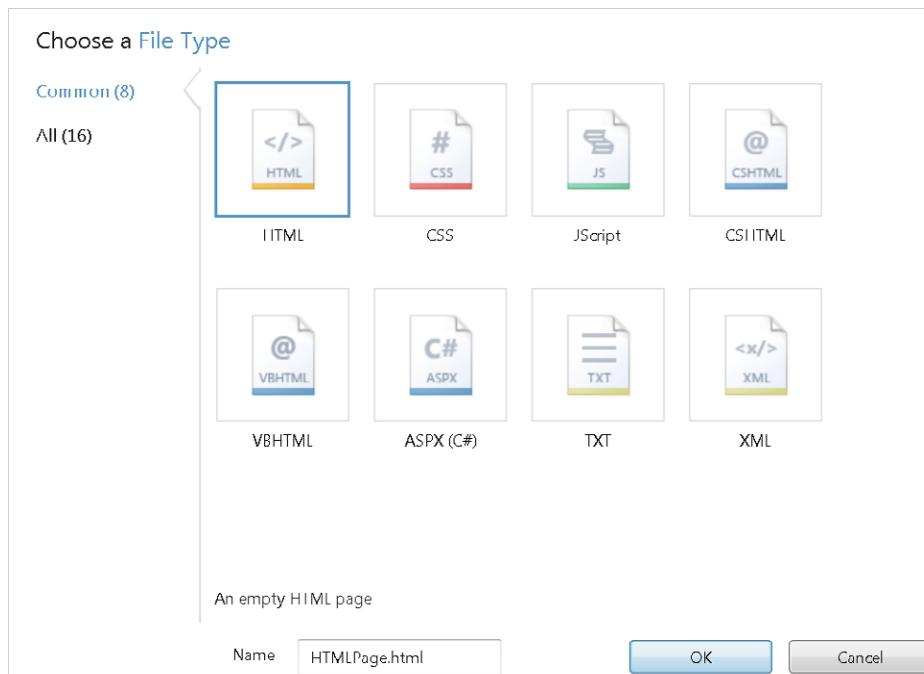


FIGURE 2-30 New file types supported in WebMatrix.

This gives you the ability to quickly create basic files in each of the indicated types. Here's a list of the common file types:

- **HTML** This stands for *Hypertext Markup Language*, which is a way of describing a webpage by breaking it down into sections, with the content in each section defined using tags contained within angle brackets. For example, the body of the page is contained within a `<body>` section, which might contain paragraphs within `<p>` tags and hyperlinks within `<a>` (the `a` stands for *anchor*) tags.
- **CSS** This stands for *cascading style sheets*, which are a way of defining styling information for HTML. HTML tags can include styling information defined directly within each tag, but creating webpages that way leads to a lot of replicated code. For example, if you want standard paragraphs (in `<p>` tags) in your site to all use the same font face and size, you could specify the font face and size every time you write a `<p>` tag—or you can specify the desired font face and size once, within a CSS file, and have every `<p>` tag use it. Moreover, if you ever need to change the paragraph style, you can change all the paragraphs in the entire site by changing just the CSS style definition.
- **JScript** This is a file that contains JavaScript code. JavaScript is a programming language that browsers can understand. The JavaScript code runs within the browser, so JavaScript gives you a way to write code that runs on the end user's machine rather than on your server.

- **CSHTML** This is a new file type that WebMatrix uses. It's a way of writing code that runs on your server but that generates HTML that gets rendered on your clients' browsers. You'll be using a lot of it in this book. CSHTML pages use the Microsoft Visual C# (pronounced "C-sharp") programming language, so that's where the name comes from; combining *CS* (which stands for C#) with *HTML* creates the acronym *CSHTML*.
- **VBHTML** This is the same as CSHTML, except that it uses the Visual Basic (VB) programming language.
- **ASPX (C#)** The ASP.NET framework supports pages that use the ASPX extension. ASPX stands for *Active Server Pages eXtended*. These pages are similar to the CSHTML and VBHTML pages but also have backward compatibility with previous versions of ASP and ASP.NET.
- **TXT** This is a simple text file.
- **XML** XML stands for *eXtensible Markup Language*, which is a markup language that uses tags to define data in a similar manner to HTML. However, XML isn't limited to using the predefined, built-in tags as HTML is—you can create a tag to define any data. For example, `<name>Laurence</name>` defines a name using XML. But you could just as easily write `<xyzzy>Laurence</xyzzy>`, and if you write your application to recognize content within an `<xyzzy>` tag as a name, that would work just as well. Of course, it's much more humanly readable to use `<name>`!

One handy WebMatrix feature is that when it creates a page, it populates it with some of the basic information that you'll need to make the page work. For example, when you create a new HTML page, you don't get a completely empty page; instead, you get a file containing the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
  </body>
</html>
```

Any file you create will open directly in the WebMatrix designer, allowing you to edit the file from within WebMatrix. You'll also get syntax coloring, which can make it easier to understand and modify the file. One exception to this is if you open an SDF (SQL Database File) file—rather than opening the file in the designer, WebMatrix launches the Databases workspace, because an SDF file is used by Microsoft SQL Server Compact edition (included with WebMatrix), which stores and manages databases. You'll look at this in the next section.

The Databases Workspace

The Databases workspace lets you create, edit, update, and delete databases used by your web applications. By default, databases you create are in the SQL Server Compact SDF format, but you aren't limited to this type. For example, if you use the MySQL database, or if you installed an application from the Web Application Gallery that uses MySQL, you'll still be able to modify it from within WebMatrix—you won't need a separate database editor.

As with the other workspaces, the Databases workspace has a custom ribbon at the top. You can see it in Figure 2-31.

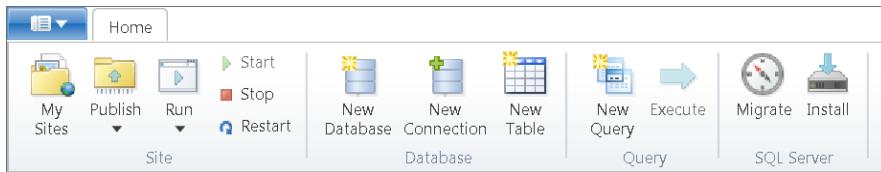


FIGURE 2-31 The Databases workspace ribbon.

On the left side, you can see the Site group, which provides the now-familiar site tools, including controls for publishing and running the site and managing your server. You'll also see three other groups: the Database group, which allows you to create or connect to databases as well as create tables; the Query group, which allows you to define and execute queries (special commands used to retrieve specific data) against your database; and the SQL Server group, which supports scaling up from the SQL Server Compact edition to a full SQL Server edition. The last group also includes a one-click Install button that allows you to upgrade to the free Microsoft SQL Server Express edition.

Creating a Database

The New Database button allows you to create a new database. When you click it, a database will be added to the workspace with the name <*Site Name*>.sdf. So, for example, if you are using the Starter Site template and have created a site called *Starter Site*, clicking the New Database button will create a database file called Starter Site.sdf. This might be a little confusing, because the template already contains a database called StarterSite.sdf, as you can see in Figure 2-32.

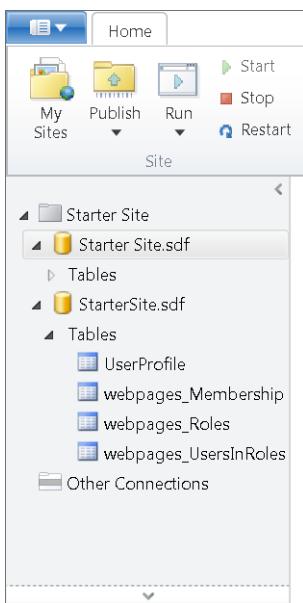


FIGURE 2-32 Viewing the database for the starter site.

Although the names might be confusing to you, don't worry! The computer understands the difference. That being said, it's not good practice to have multiple databases in your application that use similar names. Also, you might have noticed that you cannot *delete* a database using the Databases workspace. This is because the entire database is a file, and thus you should use the Files workspace if you want to get rid of it. You can, of course, delete tables, columns, and other database artifacts.

Using Tables

Databases are typically broken down into separate tables, each containing a set of related data. For example, suppose you had a website for selling books. You could have one table for all the books that you're selling, another to remember all the different transactions that have taken place, and a third to store your users' registration data. It would be really confusing to have all of these stored as individual records in the same table!

You can use the New Table button to create a new table in the current database. When you click the New Table button, you'll see a table editor interface in which you can specify the column names within the table, each column's data type, and whether or not you allow the value to be null. You can see this in Figure 2-33.

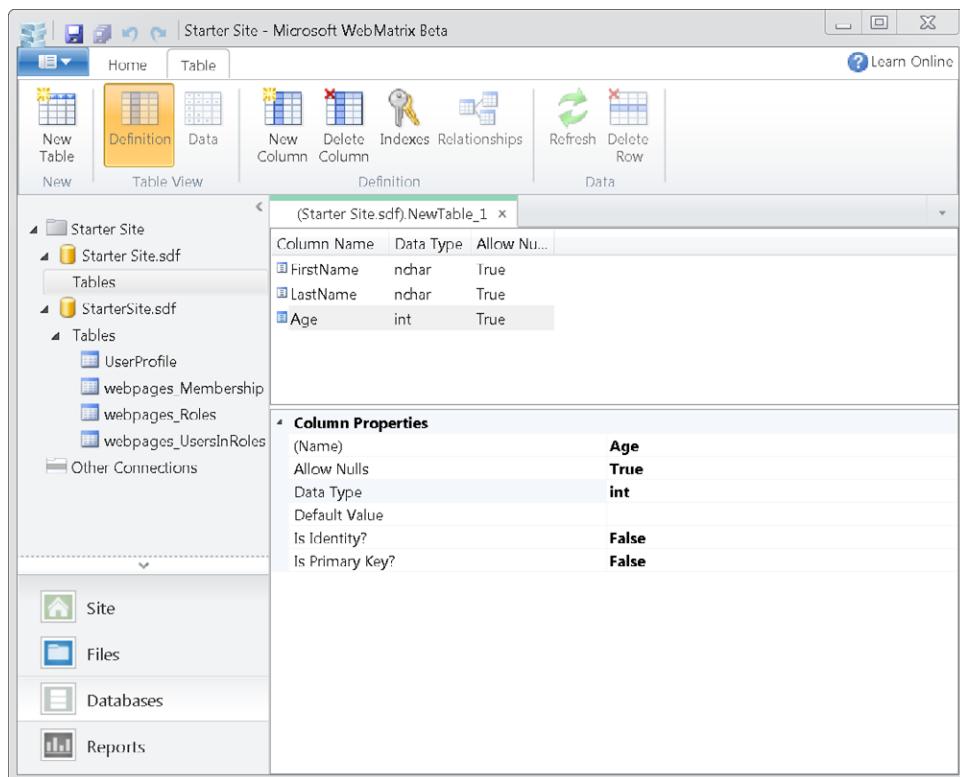


FIGURE 2-33 Creating a table in your database.

For example, a table that contains contact details would probably have columns for first name and last name, which would be string values, a column for age, which would be an integer, and so on. You'll see more about creating and using database tables in Chapter 7, "Databases in WebMatrix," including how to manage indexes and relationships, using some of the other new buttons that you can see on the ribbon.

Editing Data in a Table

If you double-click a table in the Databases workspace, WebMatrix opens the table in Data View, which provides a spreadsheet-like interface that you can use to add or modify data. The editor validates values that you type against the column type for that table. For example if you have a column that was specified to hold an *int* type and you try to type a string value—such as a name—into it, WebMatrix will raise a warning. Figure 2-34 shows what happens when you enter invalid data into a database field using the editor; it displays the Invalid Data message box, which tells you what the problem is and where it occurred, giving you a chance to fix it.

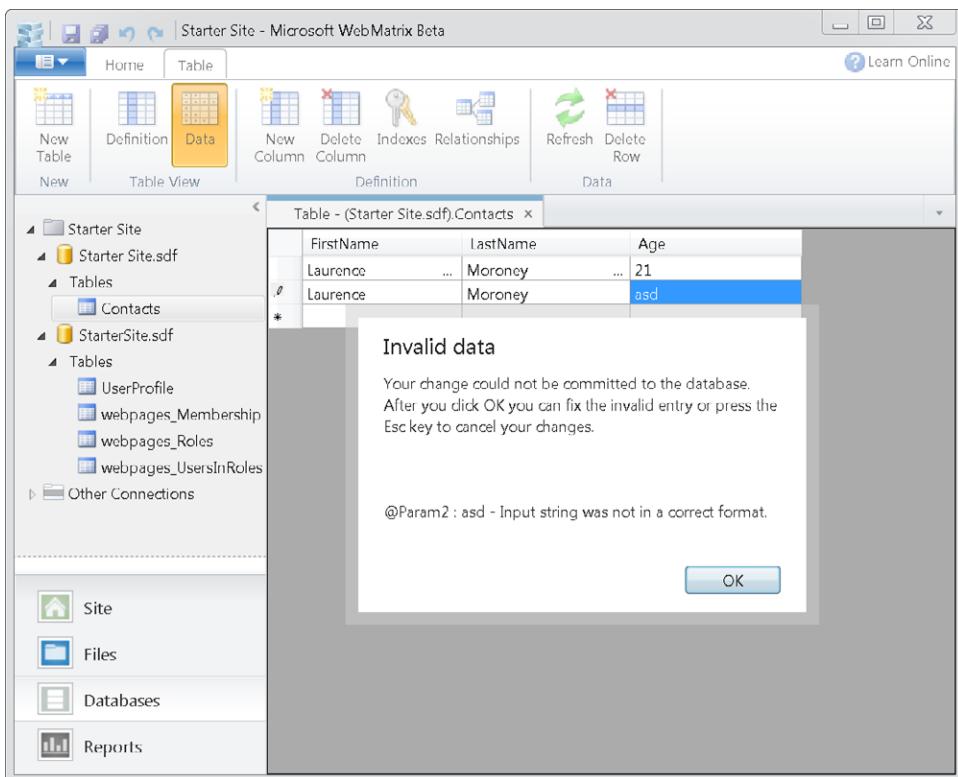


FIGURE 2-34 Using the data editor.

Querying Data

Also available in WebMatrix is the ability to edit and run queries that use Structured Query Language (SQL), a standard way of querying and managing databases. You'll learn more about SQL in Chapter 7. As a brief example now, the simple Contacts table pictured in Figure 2-34 contains two records, both for someone called *Laurence Moroney*. One of those records has the value 21 in the Age column, and the other has something else. You can use SQL to find the record for the 21-year-old contact by using a query that looks like this:

```
SELECT * FROM Contacts WHERE Age=21
```

When you execute this query, WebMatrix will pass the query to your database and display the results. You can see an example in Figure 2-35.

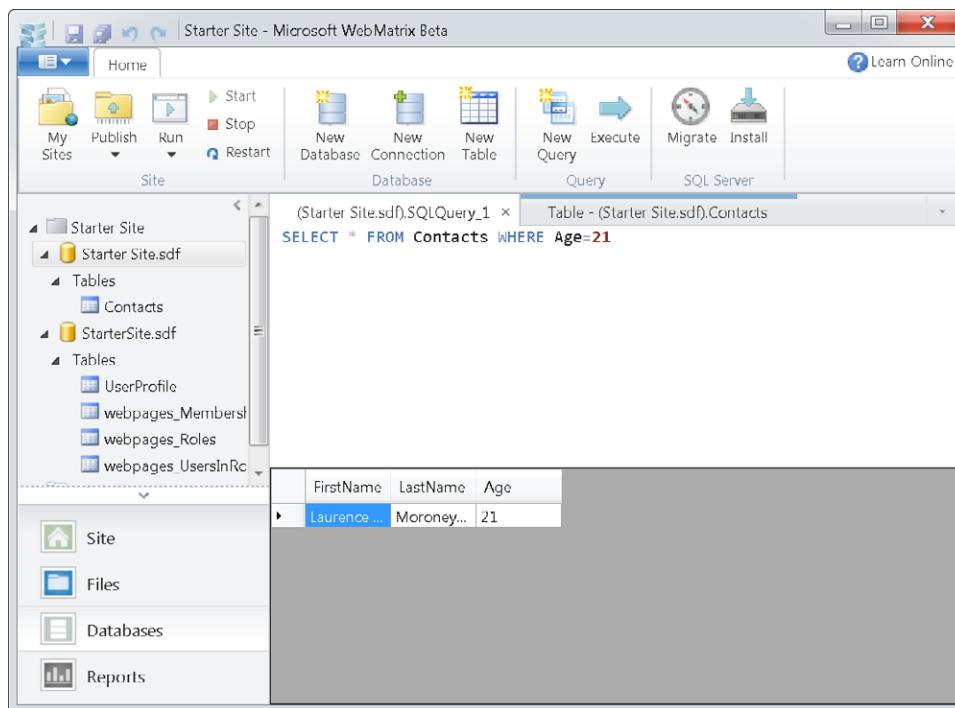


FIGURE 2-35 Creating and executing a query.

Though this example is a bit trivial, SQL is actually a powerful way of making applications run quickly. For example, if you had an online bookstore of 10,000 books divided into different categories, and your user wanted to see only science fiction books, you could write an SQL query that queries your book database for science fiction titles. Suppose this query finds 50 books. You can then send only those 50 titles to the browser, instead of all 10,000, giving your end user a lot less to download and making your application both considerably faster and much more user-friendly.

Database servers tend to be optimized for SQL, so operations of this type happen very quickly. By using SQL judiciously, you can let your database perform most data tasks, rather than having to handle large data sets in ASP.NET or PHP code, or on the client by using JavaScript.

The Reports Workspace

The Reports workspace allows you to run reports against your site. In this first version of WebMatrix, the only report type available is a Search Engine Optimization (SEO) report. This report looks at your website the same way the major search engines do.

Have you ever wondered why a bookstore called *A* might show up first in a Google or Bing search, whereas a bookstore called *B* might show up in 100th place when you perform a web search for it? It turns out that SEO efficiency is a huge factor in how high your sites appear in search hit lists. If your site breaks some of the rules that search engines expect it to follow, it will score poorly and thus appear farther down in their list.

Many companies spend thousands of dollars having experts look over their websites to improve their ranking in search engines. These experts help find and fix *violations* of the known search engine rules. Running an SEO report by using WebMatrix does the same thing—but it's free!

To run an SEO violations report, open the Reports workspace and click New on the ribbon. You can see this in Figure 2-36.

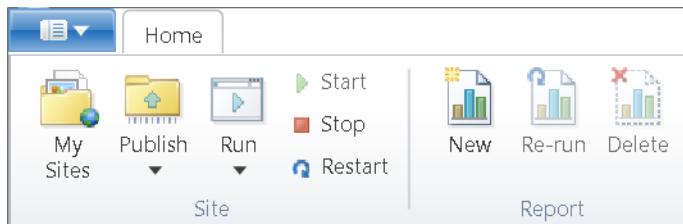


FIGURE 2-36 The Reports workspace ribbon.

When you click the New report button, you'll see a dialog box in which you can specify basic and advanced settings for the report, as shown in Figure 2-37.

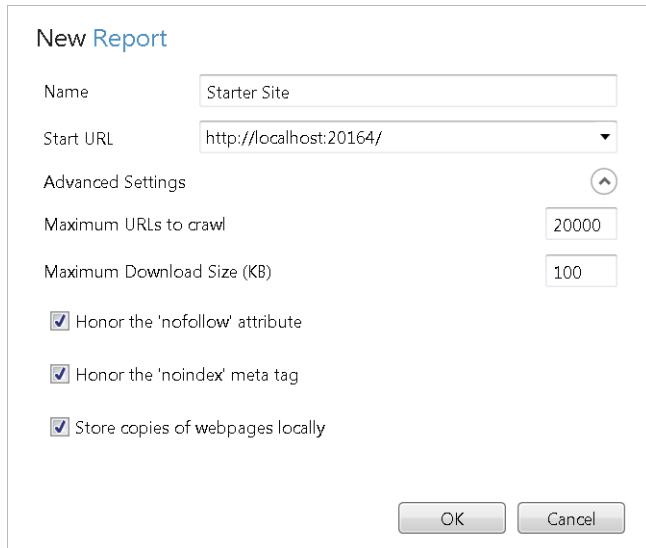


FIGURE 2-37 Creating a new SEO report.

When you click OK, WebMatrix crawls through the pages in your site, looking for SEO violations. When that process completes, WebMatrix shows you a detailed report of problems it found, which includes the URL of the offending page, the number of errors found on that page, and so on. Figure 2-38 shows an example of an SEO violations report.

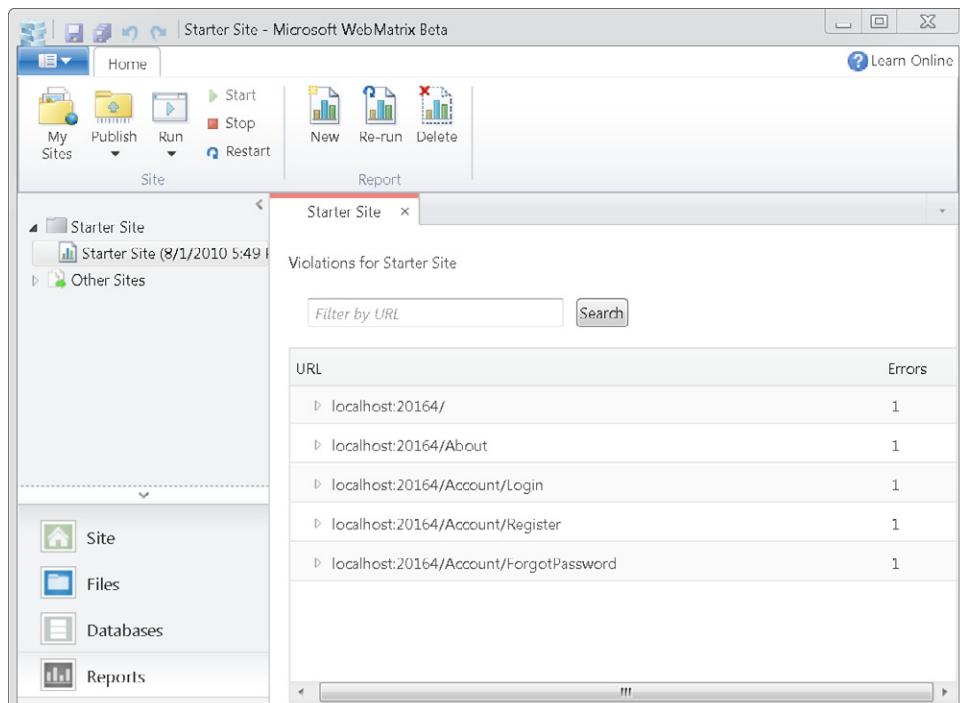


FIGURE 2-38 Viewing SEO violations.

Of course, it doesn't do you much good to know that you have a violation unless you also have a way of understanding exactly what that violation is and fixing it. As with the Requests list you saw earlier in this chapter, WebMatrix lets you click any violation entry to see details on that violation along with recommendations about how to fix it. Figure 2-39 shows an example.

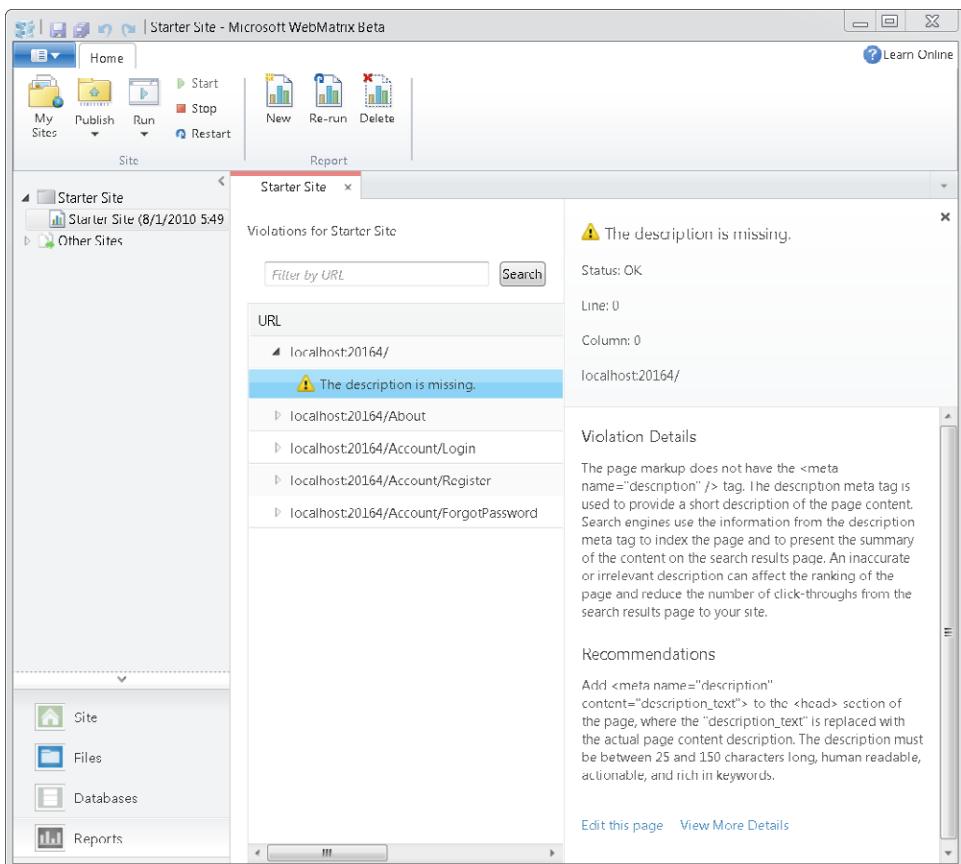


FIGURE 2-39 Exploring and fixing SEO violations.

Additionally, each violation detail screen provides a link to take you directly to the offending page, so you can edit it right away and fix the problem. You'll also find another link where you can get more information about the problem, allowing you to understand the issue in more detail. The link opens a dialog box that contains issue details, provides you with the ability to inspect the page headers, supplies a basic word analysis, and more. You can see an example of this dialog box in Figure 2-40.



FIGURE 2-40 Viewing the headers on the Headers tab of the Violation Details dialog box.

Finally, the Reports workspace automatically saves each report, allowing you to track how your site's SEO status has changed over time. You can, of course, delete any report at any time to keep old reports from building up.

Summary

This chapter took you on a tour of the WebMatrix workbench, where you learned the ins and outs of all the tools that WebMatrix puts at your fingertips.

You saw how you can use WebMatrix to create sites based on existing open source applications from the Web Application Gallery, or new sites based on the built-in templates.

You then took a tour of the WebMatrix workbench and its four main workspaces:

- The Site workspace, which lets you manage everything to do with your site
- The Files workspace, in which you manage all the files in your site and write and edit code
- The Databases workspace, which you can use to manipulate the data or data connections from your site
- The Reports workspace, which gives you a reporting engine that discovers SEO violations and instructs you how to fix them

With all these in mind, you'll switch gears over the next few chapters and start exploring how to program websites by using WebMatrix. In particular, you'll be looking at how to use the simple inline coding model (nicknamed *Razor*) using the C# programming language.

Chapter 3

Programming with WebMatrix

In this chapter, you will:

- Explore server programming.
- Create a page that uses server-side code.
- Explore sending data to the server.

The previous two chapters introduced you to Microsoft WebMatrix and took you on a tour of the WebMatrix programming environment. In this chapter, you'll roll up your sleeves and start working with the actual code that you use to build an active page using WebMatrix. You'll start by creating some very simple pages with some active content to help you get familiar with WebMatrix. After that, you'll go back and review how you implement some basic programming concepts by using WebMatrix.

Server Programming

If you are new to developing server-oriented applications, don't worry—it's a lot easier than it sounds. WebMatrix is designed to make it easy for you to build websites that use programmable servers. You might be used to *client* programming, such as building applications that run on a phone, a desktop, or even JavaScript applications that run within the browser. The important difference with server programming is that much of your application code doesn't run on the client device. Instead, the end user's actions launch a page request to the server, and if that page is an active page, the server runs code and uses that code to generate HTML markup and values that it sends down to the client browser. The browser then renders this HTML, and users see the result.

As you improve your skills, you'll find that sometimes it makes sense to mix things up a little, with some code running in your browser (typically using JavaScript or a rich Internet application (RIA) technology such as Microsoft Silverlight), and the rest running on your server. This book concentrates primarily on server code. In the next couple of sections, you'll work through some very simple examples that use server code, and then you'll get a short tour of some of the programming basics that are available to you in the Microsoft .NET Framework and WebMatrix.

Your First Programmed Page

For this exercise, you'll start by creating an empty site, add some HTML to it, and then edit the HTML slightly. This process will help you understand what the server does when it runs your page.

1. Launch WebMatrix. From the welcome screen, select Site From Template. Choose the Empty Site template, and name your site **3-1**. WebMatrix will create your site, and the workbench will look something like the one in Figure 3-1.

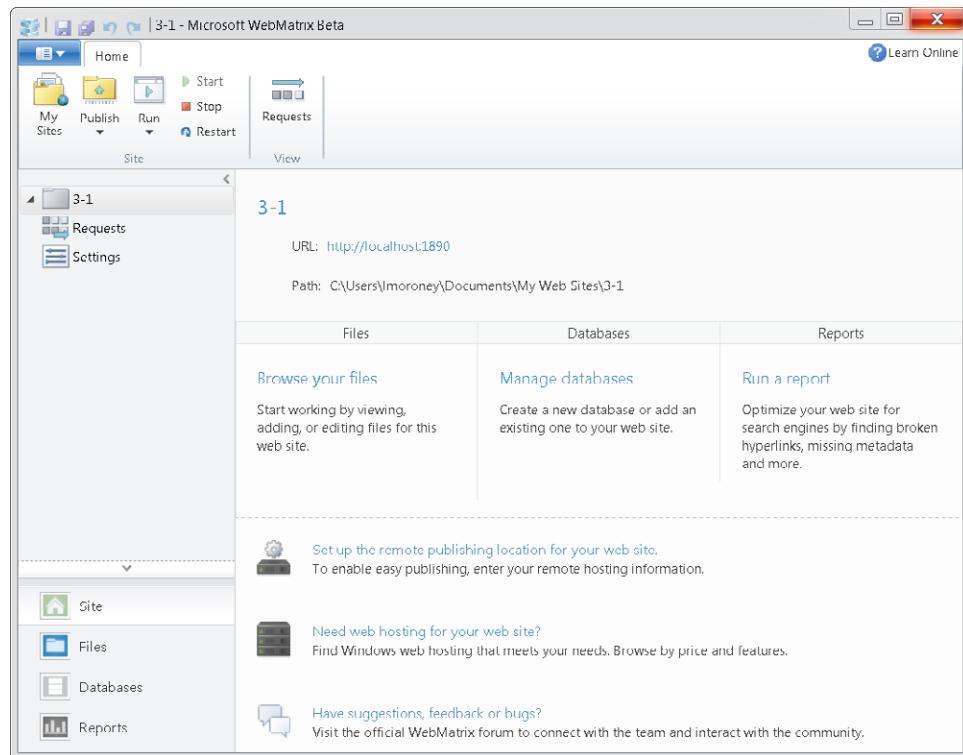


FIGURE 3-1 A website in WebMatrix.

2. Go to the Files workspace. You'll see that despite the fact that your site was based on the Empty Site template, WebMatrix has actually created a file within your site. The file is called robots.txt. You can delete this file by selecting it in the Files workspace, right-clicking it, and then selecting Delete from the pop-up menu (see Figure 3-2). You don't need to delete it—it's a very useful file, and we're only using it in this exercise to show you how the Files workspace gives you control over your files.

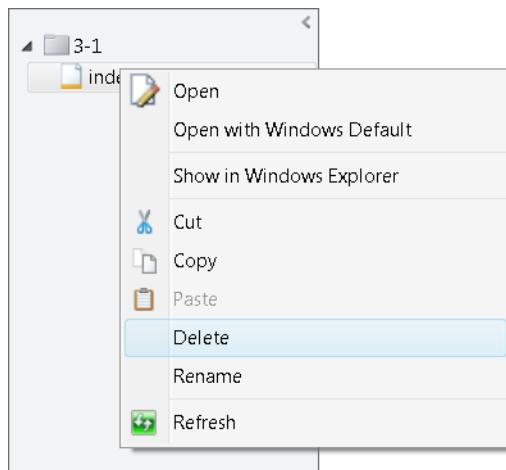


FIGURE 3-2 Deleting a file from the Files workspace.

3. Now add a new file by clicking the New button on the ribbon. You'll see a dialog box with a variety of types for the files you can add. (These types were discussed in Chapter 2, "A Tour of WebMatrix.") For this example, select the HTML file type and give the new file the name **index.html** by typing the name in the field at the bottom of the screen, as shown in Figure 3-3.

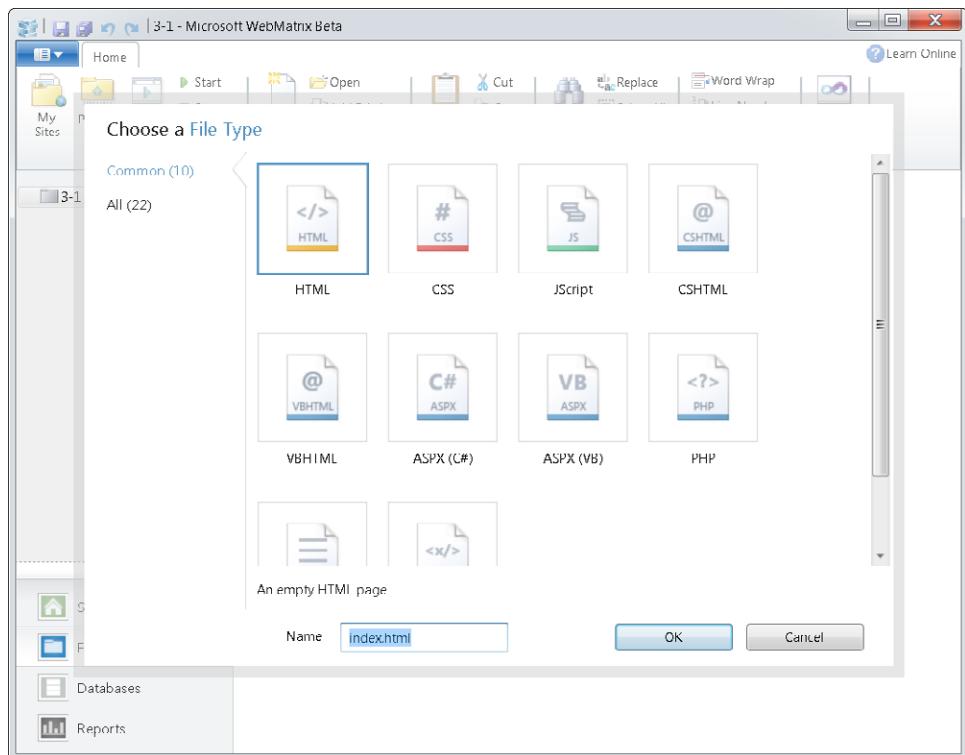


FIGURE 3-3 Creating a new HTML file.

4. Click OK. WebMatrix will now create the index.html page. It will contain the following code:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>

    </body>
</html>
```

This is HTML at its most basic. All information in HTML describes your page, its layout, and its contents. It stores this information by using tags: an HTML element is defined within an open tag consisting of one word, such as `<head>`, and a closing tag with the same word preceded by a forward slash (/), such as `</head>`. You can nest elements, embedding some elements within other elements. In the preceding example, you can see that the `<head></head>` element is embedded within the `<html></html>` tags.

Similarly, the `<title></title>` element is embedded within the `<head></head>` tags.

The `<head>` section contains information about your HTML page that the browser doesn't render on the page, such as the page title.

5. Change your code so that it looks like this:

```
<!DOCTYPE html>

<html>
    <head>
        <title>My First Page in WebMatrix</title>
    </head>
    <body>

    </body>
</html>
```

This will set the title of your page as rendered in the browser.

6. Run your application by clicking the Run button on the ribbon. You should see a page similar to the one shown in Figure 3-4.

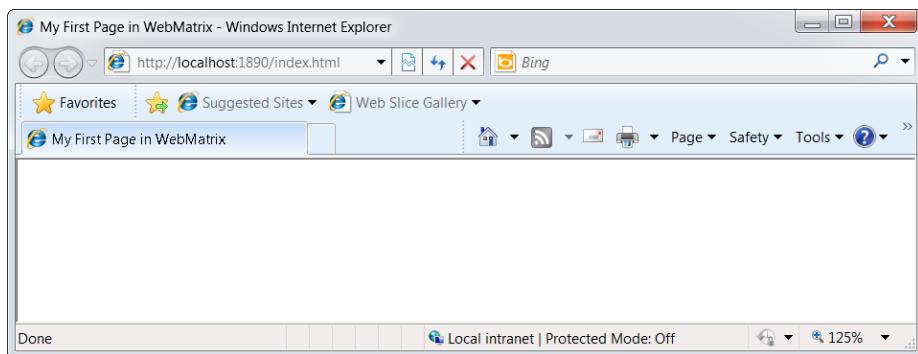


FIGURE 3-4 Using the `<title>` tags.

Notice that the page title in the browser's title bar now reads *My First Page in WebMatrix*, as does the tab for this page. That's the effect of setting the title tag value. But you'll see that the page is empty, because the `<body>` element, which contains the contents that the browser will render, is still empty.

7. You will now change that by adding some content to the `<body>` element. You can specify various heading types in HTML by using `<h>` tags. For example, you can create a top-level header by using an `<h1>` tag, a second-level header by using an `<h2>` tag, and so on. Default paragraph text for the page can be placed within `<p>` tags. Edit the page contents now so that your code looks like the code shown here. Note that all the new tags are within the `<body>` tags:

```
<!DOCTYPE html>

<html>
  <head>
    <title>My First Page in WebMatrix</title>
  </head>
  <body>
    <h1>Hello, I can write HTML</h1>
    <p>But I'm still learning, please be patient</p>
    <h2>Today's Date</h2>
    <p>Today is Thursday August 19th, 2027</p>
    <h2>Today's Time</h2>
    <p>It's 14:22 Mars Standard Time</p>
  </body>
</html>
```

8. Run the application. You'll see that the content of your page has been rendered from your edited HTML (see Figure 3-5).

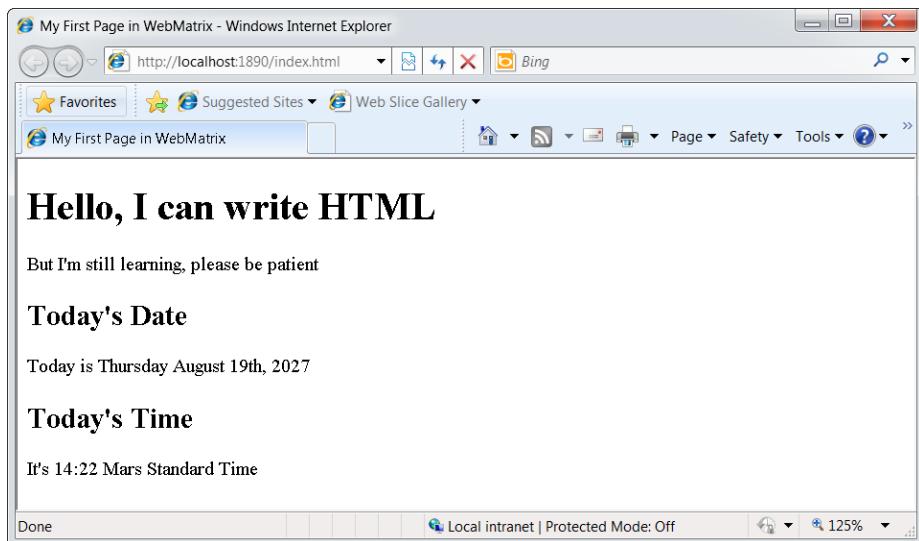


FIGURE 3-5 Setting the content of the page.

The server has simply delivered your HTML code to the browser, and the browser has rendered it. If you look at the code the browser received (the *source code*), you'll see that for HTML pages, the code that the browser receives and renders is identical to what you typed. To see the source code in Windows Internet Explorer, click the Page menu, and then select View Source. A window will open that shows the source code (see Figure 3-6).

A screenshot of the "Original Source" view in Microsoft Internet Explorer. The title bar says "http://localhost:1890/timedate.cshtml - Original Source". The menu bar includes File, Edit, and Format. The main area displays the raw HTML code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My First Page in WebMatrix</title>
  </head>
  <body>
    <h1>Hello, I can write HTML</h1>
    <p>But I'm still learning, please be patient</p>
    <h2>Today's Date</h2>
    <p>Today is Sunday, August 01, 2010</p>
    <h2>Today's Time</h2>
    <p>It's 10:05 PM Earth Standard Time</p>
  </body>
</html>
```

FIGURE 3-6 Your Source HTML as the browser sees it.

You've created your first page. Well done! But when you think about it, for a page like this that contains information that *changes*, such as the date and time, it doesn't really make sense to do it like this. Can you imagine having to change the content of the `<p>` tags by yourself every minute and every day just to render the time correctly in a user's browser? That would be a lot of work!

This is where the concept of a programmable server comes in. Instead of creating HTML that doesn't change (called *static HTML*), you can create a program that runs on the server and that generates the correct HTML at the time you request it. The following section shows how you can do this.

Making Your Page Dynamic

First, your current file (`index.html`), won't do. It's an HTML file; the server recognizes files with an `.html` extension as static HTML. When the user browses to this file, the server will just dish it up without running any code. To cause the server to run a program, you have to use a file type that the server recognizes as one that contains programming instructions that it should run. This is what a CSHTML (or VBHTML, for Microsoft Visual Basic) page is for. (All of the samples in this book use CSHTML.) In this next exercise, you'll create a CSHTML file to fill in the date and time dynamically.

1. Use the New button to create a new CSHTML file. Name it **timedate.cshtml**.

You'll see that the code created for you by WebMatrix is *identical* to the code that defined your HTML page. This is because CSHTML is designed to be as easy as possible to program with, so if you know HTML, you'll already have a great head start.

2. Edit the CSHTML page so that it contains the same markup as the `index.html` file you created in the previous example. You can simply copy and paste the code from `index.html`.
3. Make sure that you have `timedate.cshtml` open in the WebMatrix code editor, and then click the Run button. WebMatrix helps make this easy because it launches the *active* page rather than a site's default page. When you click Run, you'll browse straight to your open CSHTML page, as shown in Figure 3-7.

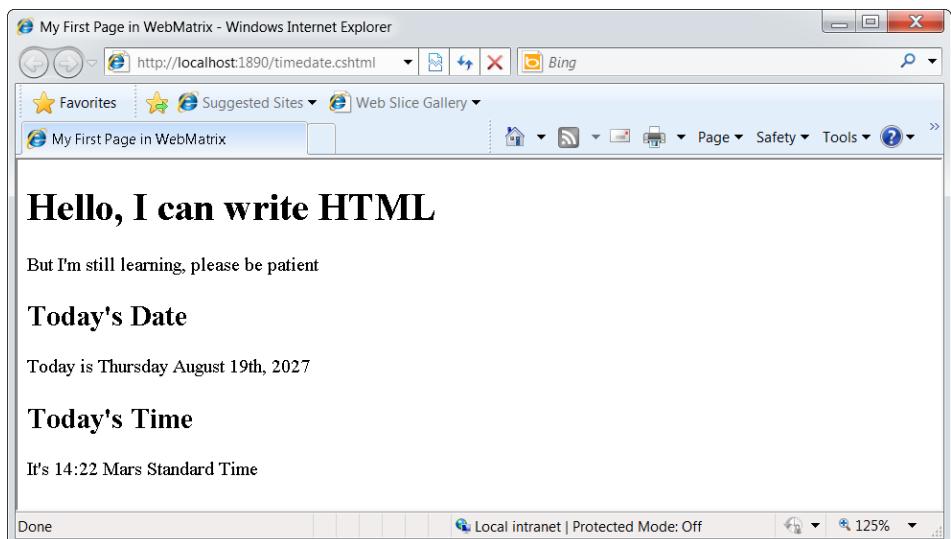


FIGURE 3-7 Running your CSHTML page.

Because the code is the same, the result will be exactly the same. But with this “template” page in place, you can write some code in the page that makes it more active.

4. The .NET Framework supports an object called *DateTime* that you use to hold dates and times. The property *DateTime.Now* returns the current date and time. There are several ways that you can format dates and times. The function *DateTime.Now.ToString()* returns the date in a long format—for example, *Thursday, August 19th, 2027*—whereas *DateTime.Now.ToShortDateString()* gives you the date in a shorter format—*8/19/2027*. These methods are ideal for this example, so you’ll use them in code within your CSHTML page.

You need to let the server know how to differentiate between static content, such as HTML, and active content, such as code. To do that, all you have to do is precede code blocks with the at sign (@), which tells the server that the following text is code. Coding in this manner uses a syntax nicknamed “Razor.” You’ll see a lot of the Razor syntax as you work through this book. Here, instead of writing out the characters *DateTime.Now.ToString()* and showing that in the browser, WebMatrix will evaluate the Razor code and output the results obtained by running that code to the browser instead.

Here’s the complete code. Copy this into your *timedate.cshtml* page. The lines in bold text are the altered lines:

```
<!DOCTYPE html>

<html>
    <head>
        <title>My First Page in WebMatrix</title>
    </head>
    <body>
        <h1>Hello, I can write HTML</h1>
        <p>But I'm still learning, please be patient</p>
        <h2>Today's Date</h2>
        <p>Today is @DateTime.Now.ToString()</p>
        <h2>Today's Time</h2>
        <p>It's @DateTime.Now.ToShortTimeString() Earth Standard Time</p>
    </body>
</html>
```

Now when you run the page, you’ll see the correct time and date! You’ve just written a dynamic server page that *generates* HTML rather than simply serves static HTML. Now, whenever a user visits your page, they’ll see something like Figure 3-8.

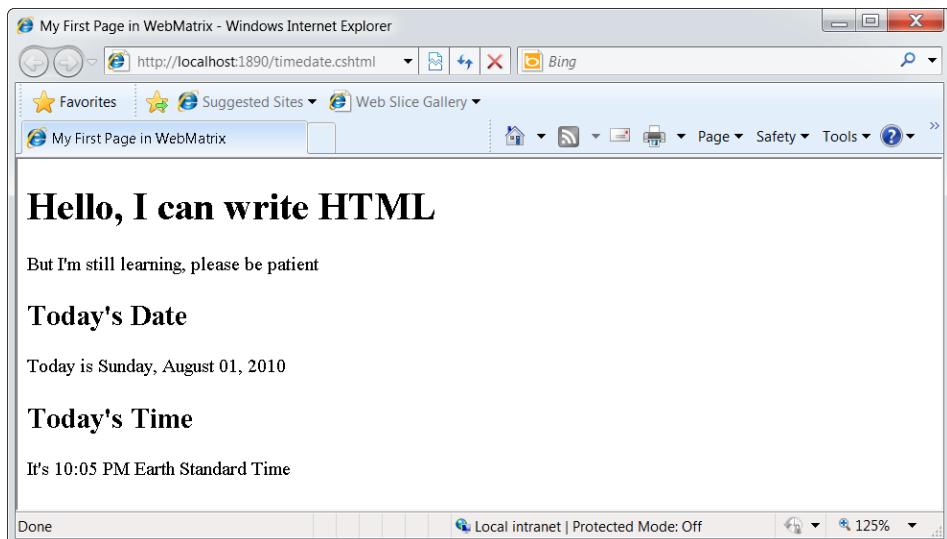


FIGURE 3-8 A page that uses code to generate HTML.

5. Take a look at the source code in the browser now (remember, to do that in Internet Explorer, click the Page menu, and then select View Source). You'll see that the `@DateTime.Now` commands are not there. That's because the code runs *on the server* and the server uses it to *generate* the appropriate HTML. The browser knows nothing about the current time and date; it just renders whatever the server sends. Figure 3-9 shows the source code from the browser's point of view.

A screenshot of a Microsoft Notepad window titled "http://localhost:1890/timedate.cshtml - Original Source". The code is:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My First Page in WebMatrix</title>
  </head>
  <body>
    <h1>Hello, I can write HTML</h1>
    <p>But I'm still learning, please be patient</p>
    <h2>Today's Date</h2>
    <p>Today is Sunday, August 01, 2010</p>
    <h2>Today's Time</h2>
    <p>It's 10:12 PM Earth Standard Time</p>
  </body>
</html>
```

FIGURE 3-9 The code generated by the server.

Before the screenshot in Figure 3-9 was taken, the browser was refreshed and the time updated by 7 minutes—from 10:05 to 10:12. In other words, refreshing the page generated a new call to the server, which re-evaluated the `DateTime.Now` commands and generated new HTML. This is a very important concept in web application development. You have to start thinking in terms of some code running on the server and generating HTML (or other content) and the browser using that content to render pages.

Sending Data to the Server

With HTTP, your browser typically makes requests to the server by using the *GET* verb from the HTTP protocol, which, as its name suggests, gets information from the server. You've done this already in this chapter but might not have realized it, because using *GET* is implicit in the way browsers request pages.

It's nice to have an application that is dynamic, as you've just seen, but the logical next step would be to ask how difficult it would be to send data to the server, have the server do something with the data, and then return the result. I'm sure you've seen hundreds of sites where you type in some information and click a Submit button to send that information to the server.

Such applications use an HTML *form*. When you click Submit, the browser sends the information in the form fields to the server by using the *POST* verb in the HTTP protocol. When sending data to the server, you'll use the *POST* verb. Again, all this happens behind the scenes; you don't need to do anything special to use *POST*, but if your code on the server side knows what kind of verb a request is using, the server can respond accordingly. You'll see how to do that in this section.

You'll start by creating a form that allows users to enter some data.

1. To start, create a new empty site and name it **3-2**.
2. As in the previous section, add a new page called **index.cshtml**. Edit the contents of that page so it looks like the following code. Note that the code adds a new *<form>* element to the *<body>* section of the page:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <form action="" method="post">
            <p>First Number:<input type="text" name="fNum" /></p>
            <p>Second Number:<input type="text" name="sNum" /></p>
            <p><input type="submit" value="Add 'em up" /></p>
        </form>
    </body>
</html>
```

Here's a quick explanation of the code you've just added.

The first is the `<form>` tag. This tells the page that content within the form tags should be used as data that is passed back to the server by using a *POST* operation whenever a user clicks the Submit button. If there's no Submit button, the form will be ignored. You use the *action* property of the form tag to specify what page will handle the posted data. If, as in this example, you don't specify an *action* property, the browser will post the page to the same URL that the page originally came from. You'll add the code to specify an *action* property in a moment.

Within the `<form>` tags, you'll see that there are three `<input>` controls. Two are of type *text*. These are controls that render as basic text boxes for data input. These text boxes each have a name (*fNum* and *sNum*), which will be used to identify the data that users type into the text box when they click the Submit button. The third `<input>` control is the Submit button itself, which has a *type* property of *submit*. Note that the text that will appear on the button reads *Add 'em up*, not *Submit*. You can put whatever text you like on a Submit button—changing the text doesn't change the way it acts.

3. Run the page. It should look similar to the page shown in Figure 3-10.

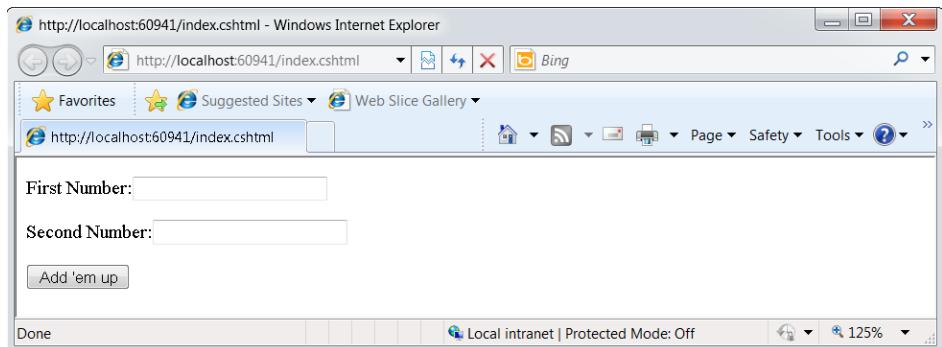


FIGURE 3-10 Your first HTML form.

4. As a little experiment, click the Add 'Em Up button. You'll see that the progress bar moves but the page doesn't change. When you think about this a little, you'll see why. The first time you run the page, by opening the browser and navigating to it, you are issuing an HTTP *GET* instruction to retrieve `index.cshtml`. When you click the Submit button, your browser issues a *POST* request—but because the form action wasn't set, the *POST* goes to the original page. Because there's no code in the page that handles the *POST*, the server simply reruns the code and sends the page back to the browser, and as such, you don't see a change.

5. So, next, let's handle the *POST* by adding the two numbers and returning the result. Edit your code so it looks like the following complete code shown here:

```
@{  
    var sum = 0;  
    var sumText = "";  
    var num1="";  
    var num2="";  
    if(IsPost) {  
        num1 = Request["fNum"];  
        num2 = Request["sNum"];  
        sum = num1.ToInt() + num2.ToInt();  
        sumText = "The answer is : " + sum;  
    }  
}  
  
<!DOCTYPE html>  
<html>  
    <head>  
        <title></title>  
    </head>  
    <body>  
        <form action="" method="post">  
            <p>First Number:<input type="text" name="fNum" /></p>  
            <p>Second Number:<input type="text" name="sNum" /></p>  
            <p><input type="submit" value="Add 'em up" /></p>  
        </form>  
        <p>@sumText</p>  
    </body>  
</html>
```

At the top of the page above the HTML, there's a new code section that begins with the @ symbol. You used this symbol earlier to embed a simple line of code within HTML, but, because this example uses several lines of code, you wrap the lines within curly braces ({}) to tell the server that there's an entire block of code here.

The code is pretty straightforward. First, it creates several variables (the lines that begin with the *var* keyword). The first variable will hold the sum of the two posted numbers. The second will hold the text that displays as the answer. The next two will hold the values of the two posted numbers.

The interesting part is the next line, which reads:

```
if(IsPost)
```

Earlier you saw that the first time you called the page, by typing its address in your browser (or clicking Run to launch the file), your browser requested the pages by using the HTTP *GET* verb to get the page. Later, when you clicked the Submit button, because the *action* property was blank, the HTTP *POST* operation just called the same page.

Well—this is where you handle the *POST* verb. The .NET Framework lets you check the verb without getting into the complexity of breaking up the HTTP headers to inspect what type of message you’re getting; you simply use the *if(IsPost)* check, which, when *true*, means that the user has used a form to post the information.

When you post a form to a server with HTML, the form values (those entered or selected by the user) are stored in the *Request* message. With the .NET Framework, to get them, you simply use the *Request* variable, which is an array of values. When you created the text boxes, you gave them the names *fNum* and *sNum*. Therefore, the values that users enter are transmitted to the server in the *POST* request and are available from your code as *Request["fNum"]* and *Request["sNum"]*.

These values are sent to the server as *text*, not *numbers*, because you used text boxes for the form input. So the next step is to convert these text values to numbers and add them up:

```
var num1 = Request["fNum"];
var num2 = Request["sNum"];
sum = num1.ToInt() + num2.ToInt();
sumText = "The answer is : " + sum;
```

The preceding code assigns the posted values to the *num1* and *num2* variables, and then the third (bold) line converts these values to integers and adds them up, storing the result in the *sum* variable. You can load the result into a text string that says “*The answer is:*” followed by your sum result. The last line places all this text into a variable called *sumText*. You can output the value of any variable very simply on your page by prefacing the variable with the @ symbol (for example, @*sumText*). You can see the HTML markup to do this at the bottom of the index.cshtml page.

6. Now run your application. You’ll first see the results of the *GET* request—the empty form with no answers.
7. Type in a couple of numbers and click the Submit button. This will perform a *POST* request. The page recognizes that this request is a *POST*, so it retrieves the user input, adds the two numbers, generates the text string containing the answer, and embeds it on the page. You can see this in Figure 3-11.

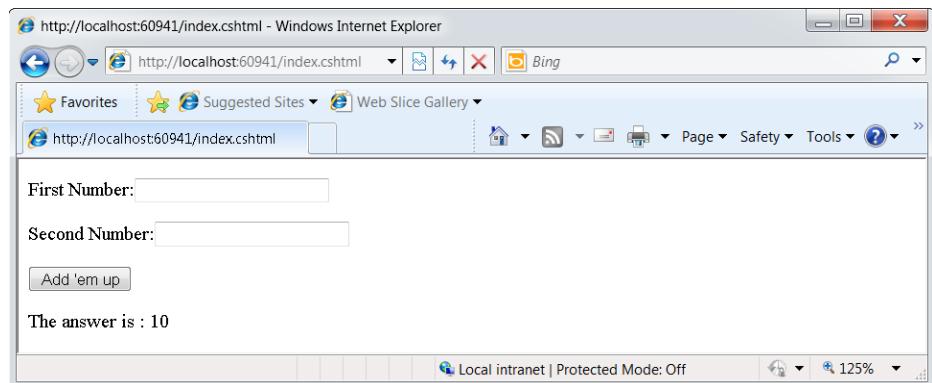


FIGURE 3-11 The result of adding two numbers entered in a form.

8. The problem now is that although you have the answer, the numbers have disappeared from the text boxes. So what do you do about that? The answer is simple. You can use the *value* attribute of the *input* tags to write back the value of the *num1* and *num2* variables, so that when the *POST* request completes and the page is generated, those values are written into the text boxes. Right now, the HTML doesn't specify any value for the input fields, thus the browser leaves them blank. Edit your code so that it includes *value=@num1* and *value=@num2* attributes within the *<input>* tags, as shown in bold in the complete page code below:

```
@{  
    var sum = 0;  
    var sumText = "";  
    var num1="";  
    var num2="";  
    if(IsPost) {  
        num1 = Request["fNum"];  
        num2 = Request["sNum"];  
        sum = num1.ToInt() + num2.ToInt();  
        sumText = "The answer is : " + sum;  
    }  
}  
  
<!DOCTYPE html>  
<html>  
    <head>  
        <title></title>  
    </head>  
    <body>  
        <form action="" method="post">  
            <p>First Number:<input type="text" name="fNum" value="@num1" /></p>  
            <p>Second Number:<input type="text" name="sNum" value="@num2" /></p>  
            <p><input type="submit" value="Add 'em up" /></p>  
        </form>  
        <p>@sumText</p>  
    </body>  
</html>
```

9. Re-run the application and you'll see a screen similar to Figure 3-12.

Note that when you first run the application (the *GET* request), the values of *num1* and *num2* are blank, so you'll see empty text boxes. When you submit the form, the values you enter are set to whatever you typed in, so when the server returns the results of the *POST*, the values will be loaded into the text boxes and the page maintains the same values you posted (see Figure 3-12).

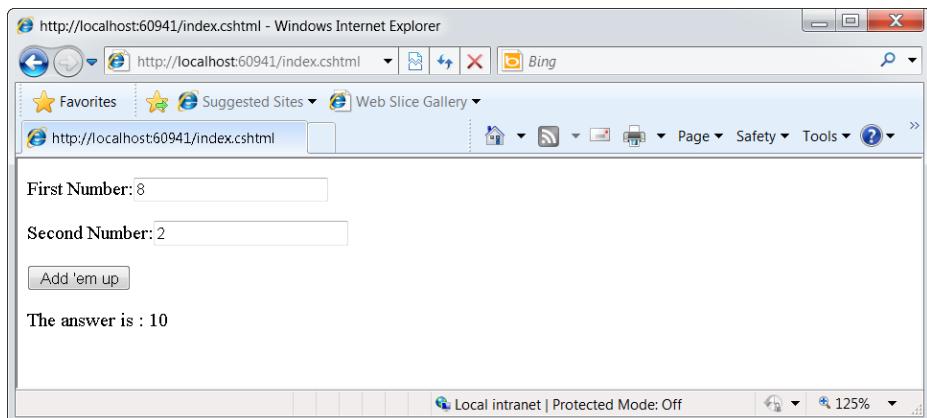


FIGURE 3-12 Using the postback to display the numbers.

This has been a pretty simple and straightforward tutorial in using *POST* values with WebMatrix. If you've done HTML development and server-side programming before, you'll understand that a lot of complexity has just been hidden from you! Though the application itself was trivial, its principles are the same as those you'll use in building more complex sites. You'll learn more about using forms and posting back data in Chapter 5, "Using Video in WebMatrix."

Summary

This chapter introduced you to server programming. You saw how HTML pages don't have to be just static text; a server can dynamically build their content. You also saw how to provide basic interactivity between your users and your server via the browser, by building a *form* that can be used to submit data to your server for processing.

In the next two chapters, you'll start exploring how you can use images and video in WebMatrix, giving your pages a little more interesting feel.

Chapter 4

Using Images in WebMatrix

In this chapter, you will:

- Create a page that uses an image.
- Create thumbnails and links.
- Program an image tag.
- Use the *WebImage* helper to add images to a webpage.

In the previous chapter, you had a look at some of the basics of HTML and writing code. In this chapter, we're going to get a little more targeted, showing how you can use images on your website.

You'll start by looking at the HTML ** tag and how you can use it for static and dynamic images. Then you'll get some hands-on experience in writing an application to upload images to the server, resize them, and render them in HTML.

Creating a Page That Uses an Image

To use an image on a webpage, you can use the HTML ** tag. This tag commonly takes two attributes, one for the path to the image, and one for the text that you want to display if the user's browser cannot render the image.

Let's take a look at building a page that uses an ** tag in Microsoft WebMatrix.

1. Launch WebMatrix and create a new site based on the Empty Site template. Name your new site **4-1**. Then switch to the Files workspace. Right-click anywhere in the files area. In the pop-up menu that appears, there is an option to add an existing file. You can see this in Figure 4-1.

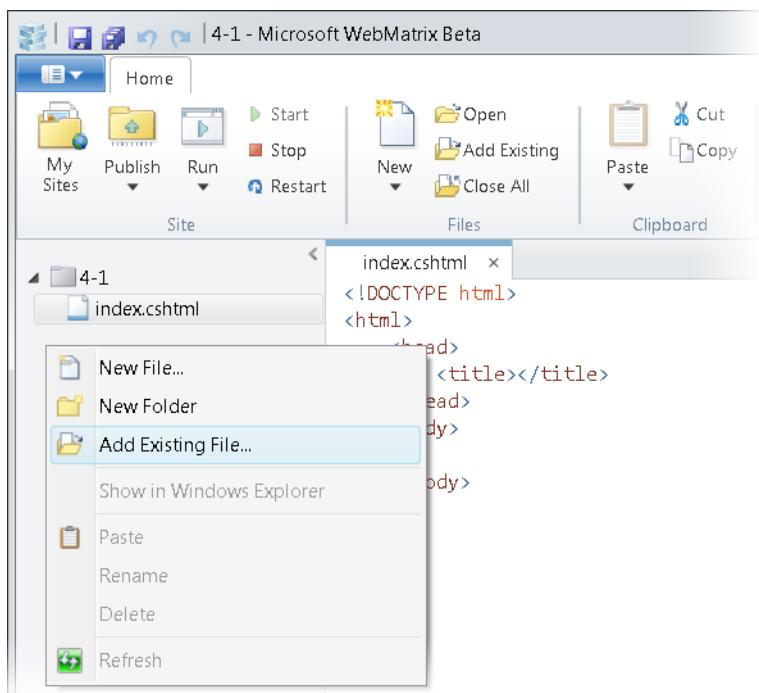


FIGURE 4-1 Adding a file to your site.

2. Browse to a picture on your hard drive and select that picture. You'll see that it is added to your files list. In the examples for this exercise, an image called fwwall.jpg is used, so you'll see that in the *src* attribute. You, of course, will use the name of the image from your machine. Note that not all browsers support all image formats, so to keep things simple, use a JPEG (or JPG) image.
3. Edit the index.cshtml file to add an ** tag that points at the image you've just added to your website. You can do so by adding the code shown in bold in the following HTML to your *<body>* tag. Remember to change the *src* attribute to match the name of your image:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <h1>A great book!</h1>
        
    </body>
</html>
```

4. Run your site. The browser displays the image (see Figure 4-2 for an example).

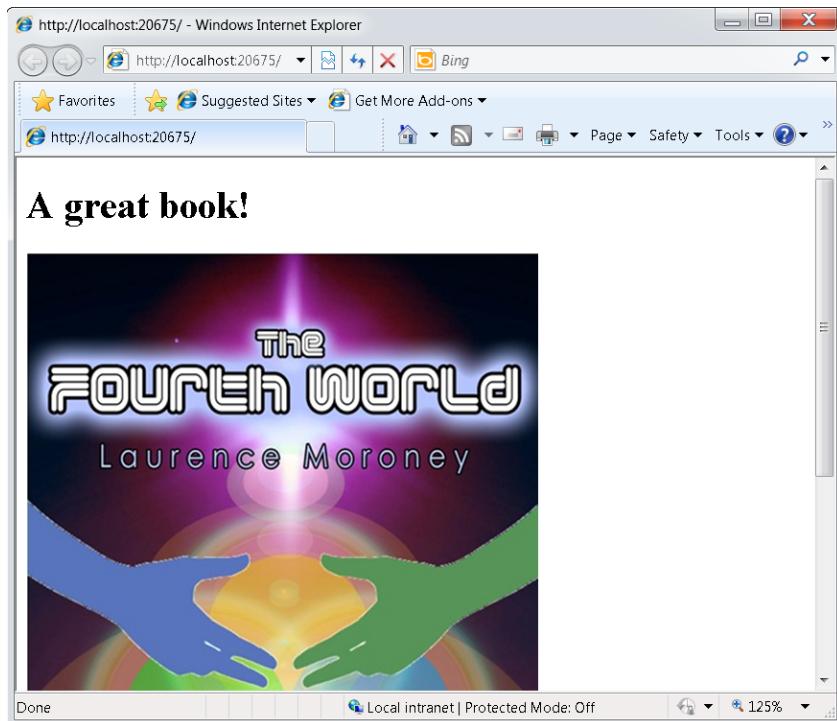


FIGURE 4-2 Rendering an image in HTML.

As you can see in Figure 4-2, this image is pretty big, and it goes beyond the boundaries of the browser. It's a book cover, so for a typical site you might want to use a smaller book cover and include more details about the book, instead of taking up the whole page with the cover.

5. To reduce the size of the image, use the *height* and *width* attributes of the ** tag, as shown here:

```

```

Now when you view the page, you're viewing a much smaller image. See Figure 4-3.

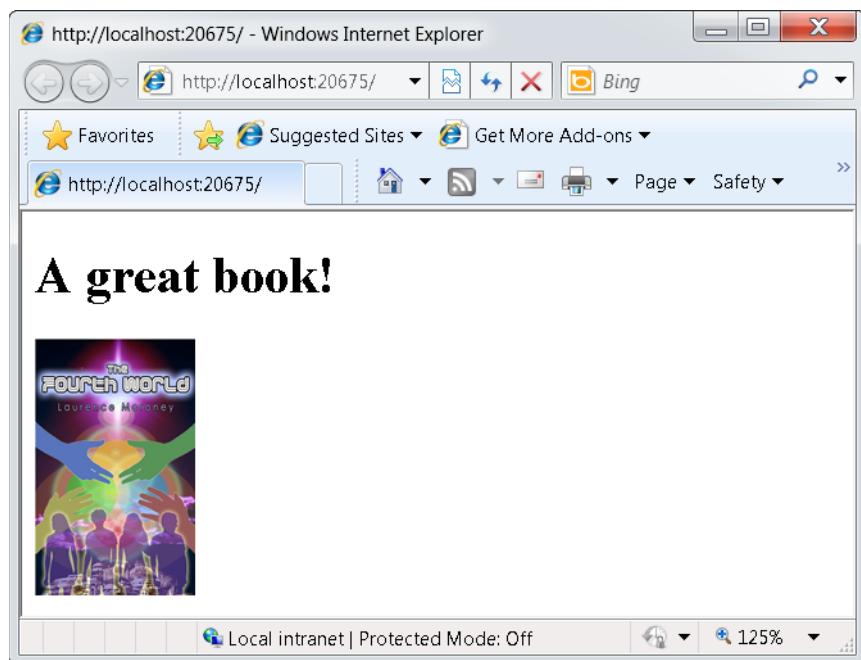


FIGURE 4-3 Setting the image size.

It's important to note here that you aren't *changing* the size of the image, you are just changing the size at which you ask the browser to render the image. When building your site, you might want to consider creating thumbnails of your images for the small versions, and use these to link to larger versions.

For example, the image used in this sample is 427 x 640 pixels, but this example is rendering it at 100 x 160 pixels. The user has to download the *full* image to view the *small* image. In this case, the image is only 92 KB, but modern digital cameras take very high resolution pictures that can be many megabytes in size. If you only want to render a small image on your page, it would be very user-hostile to make the user download the full image to just see the thumbnail. In the next section you'll look at how to do something about this.

Creating Thumbnails and Links

In the previous section, you saw how to add an image to a webpage and how to use the *height* and *width* attributes to set the size at which to render the image. This worked well for a small image (640 x 480 pixels, for example), but when you want to render a larger image (such as a typical digital image, which can be several megapixels), you end up forcing your users to download the full image in order to render it, and it's usually compressed to fit onto the screen. In this section, you'll look at creating a thumbnail, rendering the thumbnail on the page, and then creating a link to the larger image.



Important Some of the exercises in this chapter use downloadable practice files. For more information about the practice files, see "Code Samples" in the Introduction to this book.

1. On your computer, browse to a large digital image, such as one from a digital camera, or just use the moonshot.jpg image included with the download for this book.
2. Double-click the image file. It will probably open in the Windows Photo Viewer. In Photo Viewer, click to show the Open menu (see Figure 4-4).

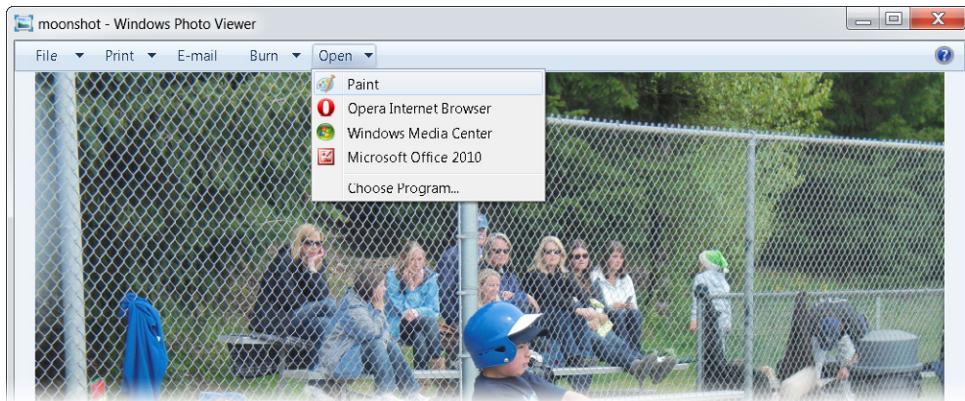


FIGURE 4-4 Opening an image in the Windows Photo Viewer.

Select Paint from the menu. (Paint is a program included with Windows that is used to manipulate images.) The image opens in Paint. The image used in this example (moonshot.jpg) is 3456 x 2592 pixels, so it's probably much larger than your screen, and you'll only see the upper-left corner of it in Paint. Can you imagine the amount of time it would take your end user to download a page with a slide show of several pictures like this? Let's resize it.

3. In Paint, click the Resize button on the ribbon, as shown in Figure 4-5.

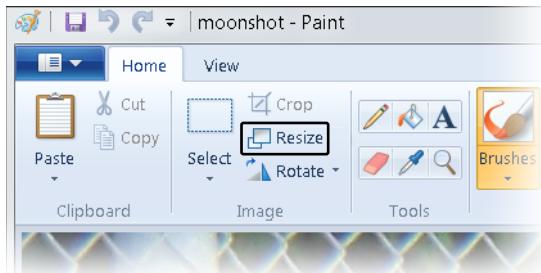


FIGURE 4-5 The Paint ribbon.

When resizing, it's best to keep the image at the same aspect ratio as the original version, meaning that the smaller image should look the same with respect to height and width as the larger image, to avoid stretching or crushing its content.

Paint can do this automatically, as you can see in the Resize And Skew dialog box shown in Figure 4-6.

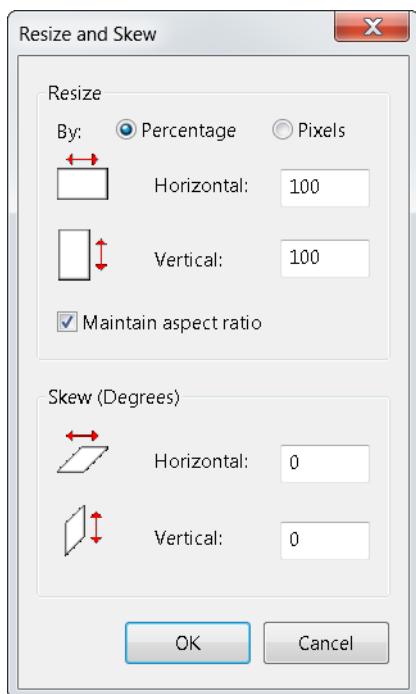


FIGURE 4-6 Resizing a picture in Paint.

4. At the top of the dialog box, click Pixels. The Horizontal and Vertical boxes change to the width and height of your picture. If you're using moonshot.jpg, these will be 3456 and 2592. Make sure that the Maintain Aspect Ratio check box is selected, and type **200** into the Horizontal box. You'll see that the value in the Vertical box changes to **150** automatically. Click OK. You'll now see a much smaller image in Paint.



Important Save this image with a *new* filename, such as **moonshot_thumb.jpg**. You do not want to overwrite your original image.

Take a look at the difference in file size. The size of the thumbnail is about 1 percent of the file size of the original and therefore will download 100 times faster.

5. Launch WebMatrix and create a new empty site called **4-2**. Add both images to your website by using the Add Existing Files menu option.
6. Edit your index.cshtml page to add an `` tag that points to the thumbnail, as shown in the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    <h1>Moonshot Home run in Little League</h1>
    
  </body>
</html>
```

7. Run your site. You'll see the page with the thumbnail.
8. Now let's add a hyperlink to this image, so that when the user clicks the thumbnail, the full image will be displayed. This makes your site a little more friendly in that the large image (close to 7 MB in the case of moonshot.jpg) is downloaded only if the user chooses to download it.

Here's the code:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    <h1>Moonshot Home run in Little League</h1>
    <a href="moonshot.jpg">
      
    </a>
  </body>
</html>
```

9. View the site again. You might notice that the thumbnail image now has a small blue border around it. This is the default behavior of the browser to indicate that an image is a link, similar to how it indicates that text is a link by underlining it. Click the image to see the full image. Note that the Windows Internet Explorer browser will scale the image to the current browser size so that you can see the full image.

Programming the Image Tag

The exercises in the previous sections used a hard-coded *src* attribute that pointed to a specific image. As you might expect, you can use WebMatrix to add code here instead, thus allowing you to have a dynamic image. In this example, you'll look at how to create *parameters* on your page to specify the name of the image and whether you want to use the thumbnail or not.

Parameters in WebMatrix pages are included in the *Request* collection. To use a parameter, you simply check if it is there (not null) by looking at the *Request["parametername"]*. If the parameter exists, you can read it as a string. So, for example, if you want to create a page that uses parameters for the image and thumbnail, you would probably use a URL such as this:

http://server/page.cshtml?imagename=whatever&thumb=yes

Parameters in a URL start with the ? character to indicate that a parameter list is following; parameters are separated by & characters.

Here's the code for a page that uses these parameters and gets the correct image based on their settings:

```
@{
    var srcPath="";
    if( Request["imagename"] != null)
    {
        srcPath=Request["imagename"];
    }
    else
    {
        srcPath="moonshot";
    }
    if( Request["thumb"] != null)
    {
        if (Request["thumb"] == "yes"){
            srcPath+="_thumb.jpg";
        }
        else{
            srcPath+=".jpg";
        }
    }
    else
    {
        srcPath+="_thumb.jpg";
    }
}
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        
    </body>
</html>
```

Let's look at this code in detail. First, there is a *var* called *srcPath* that is set to be an empty string by being initialized:

```
var srcPath="";
```

Next, the code looks at the parameter list to see if there is a parameter called *imagename*. If there is, its value is assigned to *srcPath*. If not, *srcPath* defaults to "moonshot". Remember, if the user didn't specify the *imagename* parameter, *Request["imagename"]* will be null, so the code must check that it is there before assigning it to *srcPath*, otherwise the site will hit an error if the parameter hasn't been set:

```
if( Request["imagename"] != null)
{
    srcPath=Request["imagename"];
}
else
{
    srcPath="moonshot";
}
```

Next, a similar check determines whether the user wants a thumbnail. If the user has omitted the parameter or has answered anything other than 'yes', the thumbnail is not used. Remember that the images were called moonshot.jpg and moonshot_thumb.jpg, but up to now the *srcPath* only reads "moonshot". To display the full image, the code must append *.jpg*; to display the thumbnail the code must append *_thumb.jpg*:

```
if( Request["thumb"] != null)
{
    if (Request["thumb"] == "yes"){
        srcPath+="_thumb.jpg";
    }
    else{
        srcPath+=".jpg";
    }
}
else
{
    srcPath+="_thumb.jpg";
}
```

Now either moonshot.jpg or moonshot_thumb.jpg is loaded into the *srcPath* variable. To render the image, the ** tag should have its *src* attribute set to this *var* instead of a hard-coded value:

```

```

You can see the final result in Figure 4-7. Note the URL in the browser, where it is specified that only the thumbnail is requested, not the full image.

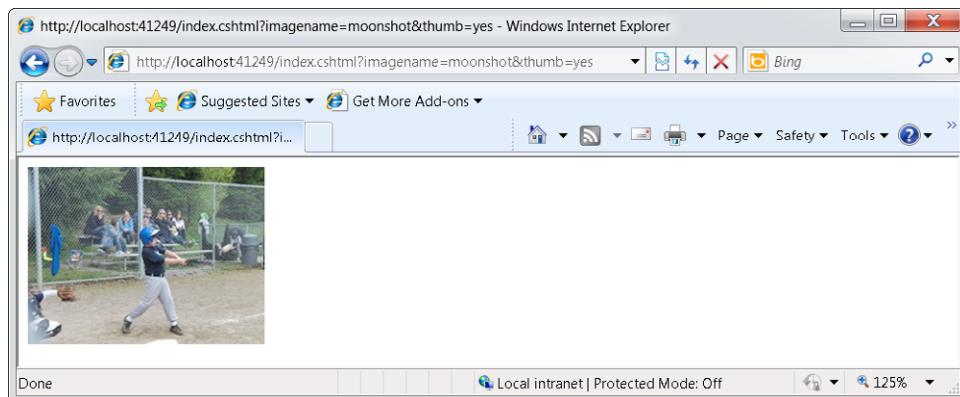


FIGURE 4-7 Using parameters to dynamically change the image.

This simple example works because we already know the image name, and we had already prepared images called moonshot.jpg and moonshot_thumb.jpg. As you work through the book and learn more, particularly when it comes to database and file system access, you'll see how you can make a site like this more flexible and dynamic.

Using the *WebImage* Helper

WebMatrix provides a helper object called *WebImage* that allows you to do some complex image manipulation, such as resizing the image, on the server. For example, in a scenario such as the previous one in which the original image is large and you want to use thumbnails, you wouldn't have to use Paint or a similar program to resize it—you could simply have your users upload the image to the server and have the server resize it into a thumbnail for end users to view.

In this section, you'll see how to do just that by using the *WebImage* helper.

First, let's take a look at the steps involved in uploading an image through the browser to the server, and how the server can save it.

1. Launch WebMatrix and create a new empty site called **4-3**. Create a new empty folder called **images** in the website.
2. Add a new file called **upload.cshtml**, and add the following code to its *<body>* tag to create an upload form:

```
<form action="" method="post" enctype="multipart/form-data">
    <p>Enter your file path...</p>
    <input type="file" name="Image" />
    <input type="submit" value="Upload" />
</form>
```



Note If you're not familiar with HTML forms, refer to Chapter 3, "Programming with WebMatrix," for a quick refresher.

This creates a form with a file input control that takes the form of a text box with a browse button beside it. When the user clicks the Browse button, a file dialog box will appear that is used to find a file to upload. The form is a *multipart/form-data* type, which means that it will submit binary data. The action is empty, so upload.cshtml will process the *POST* also. Let's implement the code for that.

3. Put the following code at the top of upload.cshtml:

```
@{  
    WebImage theImage = null;  
    var strFileName="";  
    var strImgPath="";  
  
    if(IsPost){  
        theImage = WebImage.GetImageFromRequest();  
        if(theImage!=null){  
            strFileName = Path.GetFileName(theImage.FileName);  
            strImgPath = @"images\" + strFileName;  
            theImage.Save(@"~\" + strImgPath);  
        }  
    }  
}
```

Absolute and Relative Paths

You might have noticed some strange syntax here, most notably the `~\` in the line that saves the image. This has to do with absolute and relative paths. Remember that the code running on the server will be running in a specific location, but you might not know exactly where that is. So, for example if you're running 4-3 on your development machine, your file location would be `C:\Users\<Your User Name>\Documents\My Web Sites\4-3` or something similar. This is an *absolute* path. If you were to try and save the image to the absolute path, you might have problems, because the `<Your User Name>` part will be different between machines, so it's best to use a *relative* path. The `~` just means, "move up one directory from where I am," and the `\` afterwards means "put it here." So when you upload `moonshot.jpg`, the `strImgPath` is set to `images\moonshot.jpg`, and the relative path becomes `~\images\moonshot.jpg`; in other words, the `images` directory that's in the same directory as the `upload.cshtml` file on the server.

This code sets up three variables, a *WebImage* used to store the image, and two strings, one for the file name of the image and one for the path on the server where the image will be found. These might seem to be the same thing, but they're not. The file name will be the raw location on the server, such as `C:\Websites\BookExercises\4-3\Images\Moonshot.jpg`, whereas the path is the URL, such as `http://server/4-3/images/Moonshot.jpg`. Both point to the same thing, just in different ways.

Next, the code checks to see if there is a *POST*, which is the case when the form's submit button is clicked. This example is a great way to show off how useful the *WebImage* web helper is. When you upload an image, it's generally broken into chunks, and it's a bit of work to piece it back together. Fortunately, *WebImage* can do this for you by just issuing a *GetImageFromRequest* method call. You now have an in-memory image.

Then it's pretty simple for you to get the file name of the image that was uploaded and save it out.

- Finally, after the image is uploaded, you need to render it, so, back in your page `<body>`, add the following code:

```
<h1>Uploaded Image</h1>
@if(strImagePath != ""){
    
}
```

When the page runs on a *POST*, the `strImagePath` variable will be set to something, so the `` tag will be set and the page will render the image. When you first run the page, you run it as a *GET* (if this isn't familiar, go back and read Chapter 3 again!), and the image path will not be set, so you see nothing.

- Run the page. You'll see something like the screen shown in Figure 4-8.

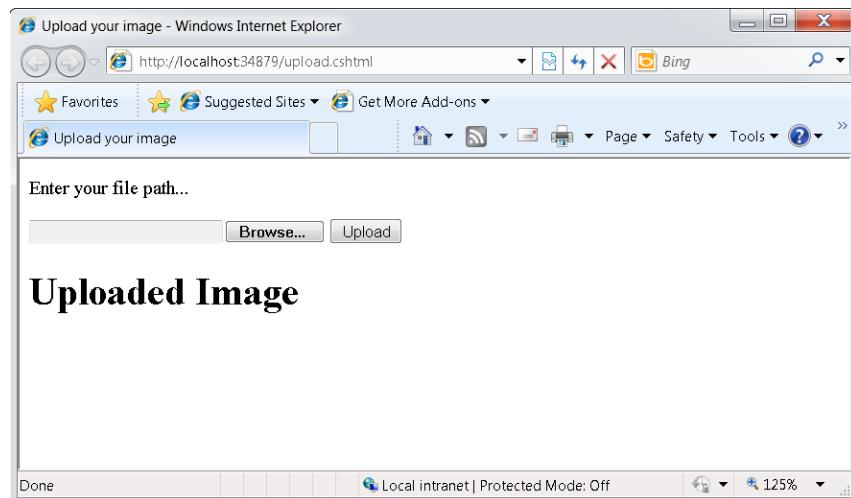


FIGURE 4-8 The Upload Image page.

6. Click the Browse button and find the moonshot_thumb.jpg file that you created earlier. (Use the thumbnail or another small image for now; we'll look at what happens with a large image in a few minutes.) Click Upload, and the image will be rendered for you (see Figure 4-9).

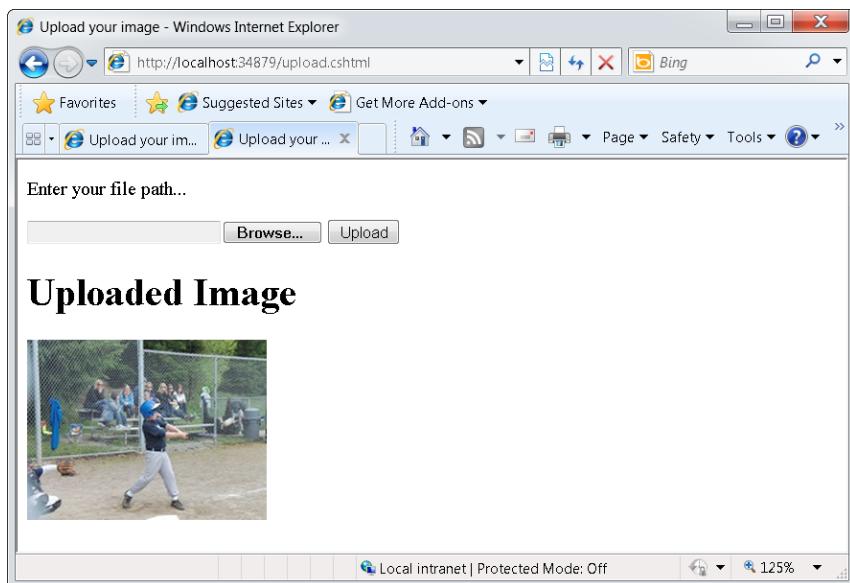


FIGURE 4-9 The uploaded image.

7. You have actually uploaded the file and stored it on the server. Take a look at the WebMatrix Files workspace, and look within the images directory that you created earlier. The file is there (see Figure 4-10).

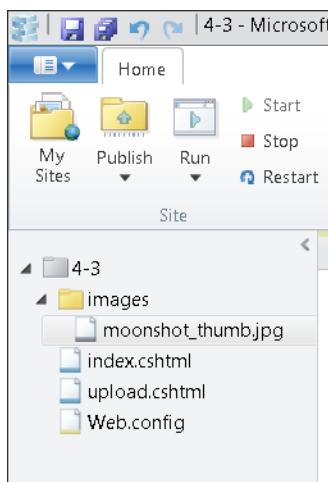


FIGURE 4-10 The file is saved on your server.

8. Now try using a large image, such as moonshot.jpg. You'll probably get a Maximum Request Length Exceeded error like that in Figure 4-11.

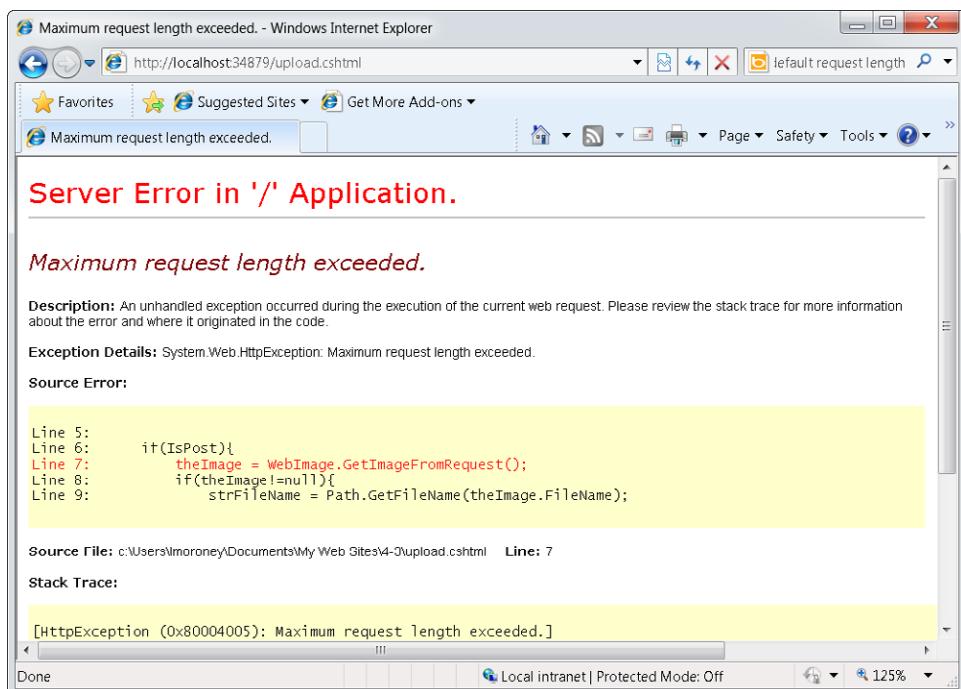


FIGURE 4-11 The Maximum Request Length Exceeded error.

You get this error because the maximum length of an item that can be placed in a *Request* is 4 MB, but the moonshot.jpg image is almost 7 MB. Fortunately, there's an easy fix, which we'll look at next.

Using Web.config to Change the Allowed Image Size

The solution for the Maximum Request Length Exceeded error is to use a configuration file for your website. You can simply tell your website to allow a bigger request length. You do this by adding a Web.config file to your site.

1. From the WebMatrix Files workspace, click the New button. In the Choose A File Type dialog box, make sure you select the All link on the left side. You'll see Web.config as one of the file types that you can create (see Figure 4-12).

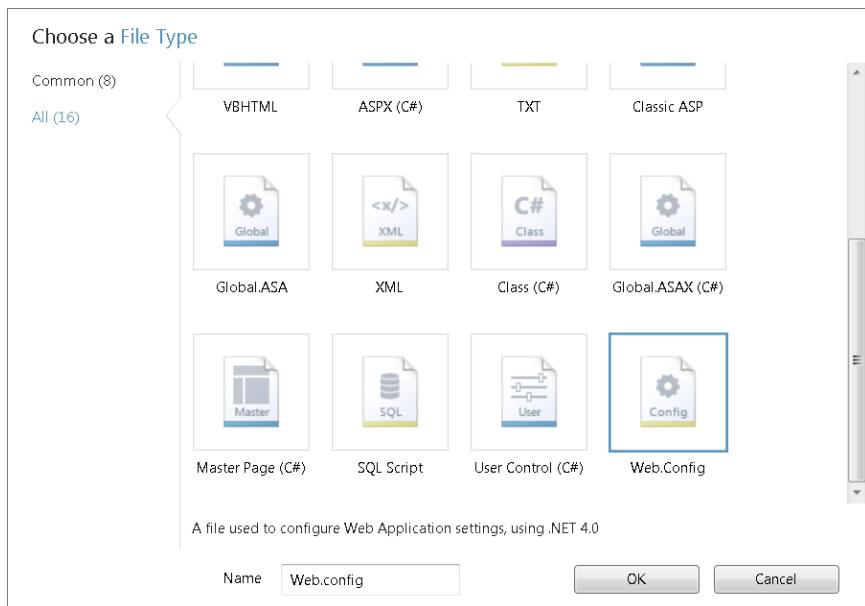


FIGURE 4-12 Adding Web.config.

2. Click OK. The Web.config file is added to your website. It's an XML file that's pretty empty right now. It looks like this:

```
<?xml version="1.0"?>

<configuration>

    <system.web>
        <compilation debug="false" targetFramework="4.0" />
    </system.web>

</configuration>
```

3. The maximum request length setting should be part of the `<system.web>` setting, so edit your Web.config to add it, like this:

```
<?xml version="1.0"?>

<configuration>

    <system.web>
        <compilation debug="false" targetFramework="4.0" />
        <httpRuntime maxRequestLength="1048576" /> -->
    </system.web>

</configuration>
```

This changes the maximum request length to 1 GB, which should be more than enough!



Important For a real site, this is probably too big—it leaves you vulnerable to a Denial of Service attack where malicious users upload a lot of 1 GB files, clogging your server. Think about this setting carefully.

4. After adding and changing Web.config, restart your server by clicking the Restart button on the ribbon. Now browse to your upload page and try to upload the large file. You can see that it succeeds, as shown in Figure 4-13.

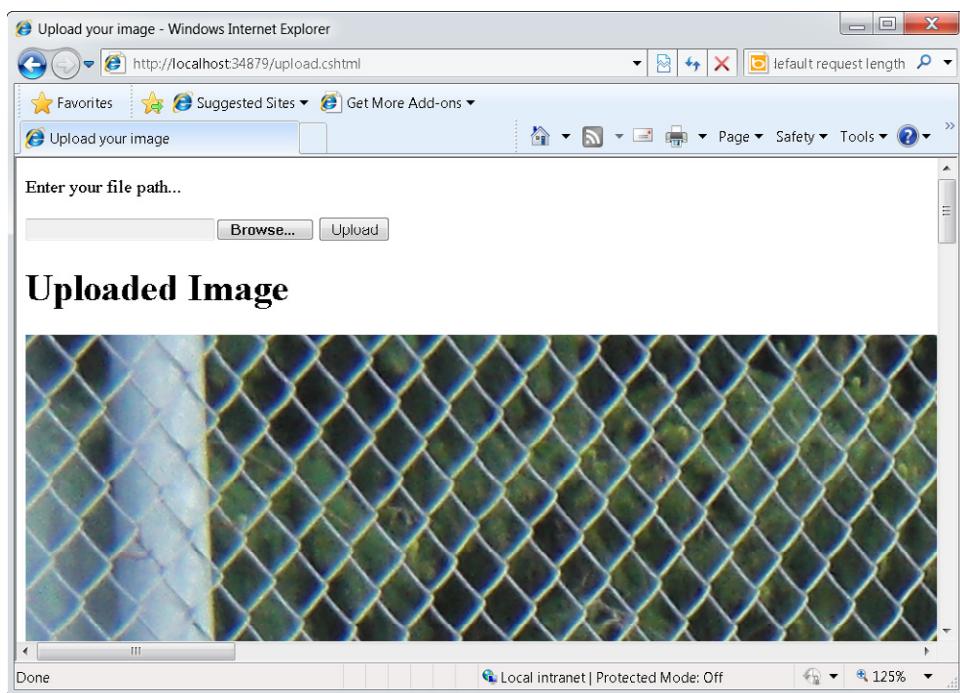


FIGURE 4-13 Uploading a large image.

You've now succeeded in building a site that will allow your end users to upload a large image, and the *WebImage* helper made the programming of this pretty straightforward. However, the image is still pretty big, and you don't want your users to have to generate their own thumbnails, so let's see how *WebImage* can help here.

Resizing an Image with *WebImage*

The *WebImage* helper supports a *Resize* command that allows you to resize an image. You can then save the resized image to make a thumbnail.

1. Start by adding a new directory called **thumbnails** inside the images directory (see Figure 4-14).

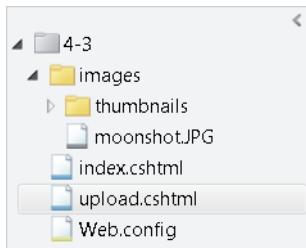


FIGURE 4-14 The thumbnails directory structure.

2. Edit your code to add a new variable for the thumbnail path at the top of your code area, beneath the current code declarations:

```
var strImgThumbnail = "";
```

3. Then, after your code that saves the image onto the server, add some new code that resizes the image and saves the results into the thumbnails folder:

```
strImgThumbnail = @"images\thumbnails\" + strFileName;
theImage.Resize(width:200, height:200,
                 preserveAspectRatio:true, preventEnlarge:true);
theImage.Save(@"~\" + strImgThumbnail);
```

The magic happens in the *Resize* command. You can see that the image is resized to have a width and/or height of 200 pixels. The *preserveAspectRatio* value is set to *true*, indicating that the aspect ratio will be preserved; and the *preventEnlarge* value is set to *true*, indicating that if the image is currently smaller than 200 on either axis, the size should not be changed. The image is then saved into the thumbnails directory.

4. Finally, change the code on the page body to render the thumbnail instead of the full image:

```
<h1>Uploaded Image</h1>
@if(strImgThumbnail != ""){
    
}
```

You can see the result in Figure 4-15.

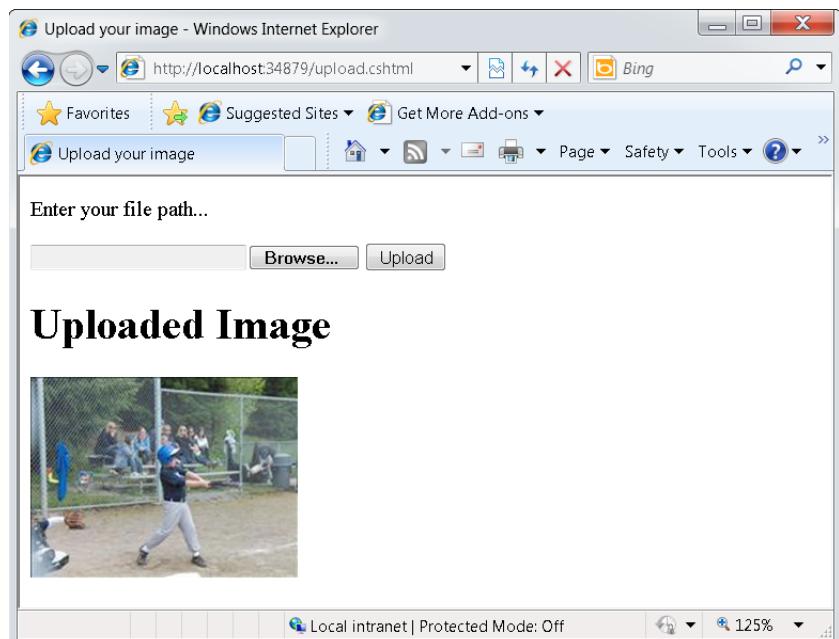


FIGURE 4-15 Uploading an image and generating a thumbnail on the server.

Here's the full source code for this page:

```
@{
    WebImage theImage = null;
    var strFileName="";
    var strImgPath="";
    var strImgThumbnail="";

    if(IsPost){
        theImage = WebImage.GetImageFromRequest();
        if(theImage!=null){
            strFileName = Path.GetFileName(theImage.FileName);
            strImgPath = @"images\" + strFileName;
            theImage.Save(@"~\" + strImgPath);

            strImgThumbnail = @"images\thumbnails\" + strFileName;
            theImage.Resize(width:200, height:200,
                            preserveAspectRatio:true, preventEnlarge:true);
            theImage.Save(@"~\" + strImgThumbnail);
        }
    }
}

<!DOCTYPE html>
<html>
    <head>
        <title>Upload your image</title>
    </head>
```

```
<body>
<form action="" method="post" enctype="multipart/form-data">
    <p>Enter your file path...</p>
    <input type="file" name="Image" />
    <input type="submit" value="Upload" />
</form>
<h1>Uploaded Image</h1>
@if(strImgThumbnail != ""){
    
}
</body>
</html>
```

And that's all you have to do to provide your users with the facility to upload images to your server and have them automatically resized!

As an exercise, see if you can make the thumbnail clickable, and have it return the full size image. It's very similar to what you did earlier with the static image.

Further Exercises

For a little more fun with your image, try the following methods:

- *theImage.FlipVertical()*
- *theImage.FlipHorizontal()*
- *theImageRotateLeft()*
- *theImageRotateRight()*
- *theImage.AddTextWatermark("Some Text", fontColor:"White", fontFamily:"Arial");*

Simply add these lines after the *theImage.Resize* line in the previous code to give them a try!

Summary

In this chapter, you looked at how to use images in WebMatrix, first looking at the HTML ** tag. You then saw how to use some programming to add dynamic images to your page. Finally, you looked at the *WebImage* helper. This helper makes tasks such as resizing an image, saving it, rotating it, and adding a watermark to it very easy to do on the server side, allowing your end users to simply upload their images by using an HTML form, and having your server do the rest!

In the next chapter, you'll start looking at web video and how you can implement it using WebMatrix, including making use of the Microsoft Silverlight and Adobe Flash video helpers.

Chapter 5

Using Video in WebMatrix

In this chapter, you will:

- Explore the use of video in your sites.
- Explore the WebMatrix Video helper.
- Find out how to use the HTML5 `<video>` tag.

Consumer use of Internet video is already huge and will only increase over time. However, delivering video over the Internet to browsers is fraught with challenges, so several methods have evolved to help deliver a satisfactory video experience.

In this chapter, you'll take a look at some of the development possibilities for building sites that deliver video. You'll start by looking at the simplest method—a hyperlink to a server-based media file that users can select to launch their default player for that media type. Then you'll look at some of the technologies that support rich Internet application (RIA) integration in a site, and how you can use them to deliver video. You'll specifically see how to use Microsoft Silverlight and Adobe Flash for this purpose.

With the basics in hand, you'll then see how the new helpers in WebMatrix can simplify the task of delivering video. The chapter wraps up with a quick look at the new `<video>` tag that's part of HTML5 and how that tag makes it easy (from a development perspective) to add video to your page. Unfortunately, although the `<video>` tag holds future promise, at the time of this writing, due to lack of widespread support and lack of established video standards, using the `<video>` tag has some challenges.

Using Video in Your Site

Over the years, a large number of video formats have evolved and are available for you to use. This chapter discusses three of the most common formats:

- **WMV** Windows Media format, used by Windows Media Player, Silverlight, and others
- **SWF and FLV** Flash Video format, used by Flash and present on many sites
- **MP4** A format that is growing in popularity and that is supported by HTML5

In this section, you'll see a straightforward method of using these video formats by including direct links in your site, letting users click them, and relying on the users' default media players to display them.

Creating a Simple Video Site in WebMatrix

1. Start WebMatrix and create a new empty site called **5-1**. This site will create a default page for you named index.cshtml.
2. Create a new folder in the site called **media**, and add a video file to the site.



Note With Windows, you'll find a sample video on your hard drive. If you're using Windows 7, it will be here: C:\Users\Public\Public Videos\Sample Videos. For the purposes of this example, you can use any video file you like, or you can use the sports.wmv file available with the downloadable code for this book.

3. Open the index.cshtml file from the Files workspace and add a hyperlink to the video file, as shown in the following code:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <a href="media\sports.wmv">Click here for the video</a>
    </body>
</html>
```

4. Run the page. Click the link on the page to launch your media player and show the video. You can see this in Figure 5-1.

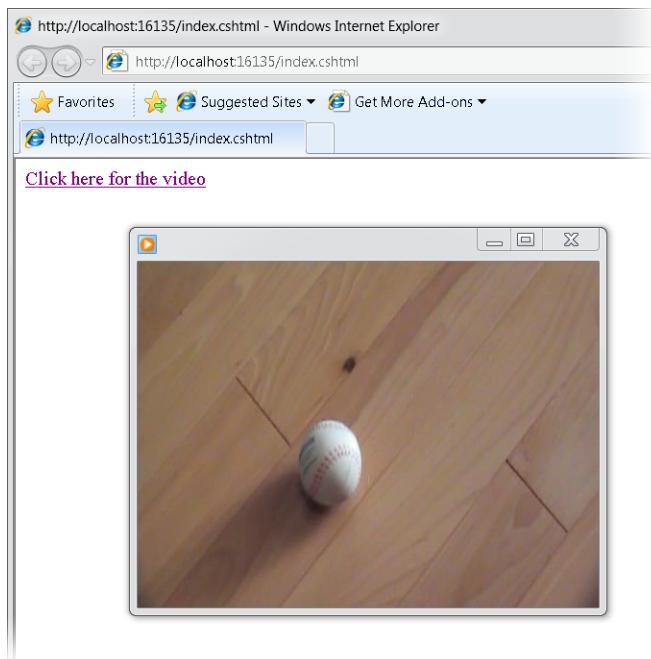


FIGURE 5-1 Running the video from a link.

This method is very simple to implement, as you can see, but it suffers from the fact that the media player is launched as a separate application. Fortunately, it's easy to embed a media player (or any other object) into a browser page by using the HTML `<object>` tag. In the next section, you'll see an example of how to do that.

Embedding a Media Player by Using the `<object>` Tag

HTML provides an `<object>` tag that allows you to embed objects in a webpage. This tag specifies the object type by using its internal identifier—a value called a *class ID* (or *classID*)—and includes a set of parameters used to initialize the embedded object.

The *classID* for Windows Media Player, which is usually the default player used to render .wmv files, is *6BF52A52-394A-11D3-B153-00C04F79FAA6*. This *classID* is an example of what is called a globally unique identifier (GUID).

The `<object>` tag has many child `<param>` tags. These define the parameters for the `<object>`. For example, a media player needs to know what media file to play, so that is a parameter. Every other setting you pass to the media player becomes a child `<param>` tag as well. At the end of your parameter list, and before your closing tag, you can include some HTML called the *fallback*, which is the HTML to display if no object with the specified *classID* exists on the computer that is browsing your site.

Changing your simple linked video to an embedded video is straightforward.

1. To embed Windows Media Player on your page, edit your HTML so it uses the `<object>` tag, as shown in the following code:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <object classid="clsid:6BF52A52-394A-11D3-B153-00C04F79FAA6" >
            <param name="URL" value="/Media/sports.wmv" />
            <param name="playCount" value="2" />
            <param name="uiMode" value="full" />
            <param name="stretchToFit" value="True" />
            <param name="volume" value="75" />
            <p>Sorry, I can't play this video!</p>
        </object>
    </body>
</html>
```

2. Run this page, and try it in Windows Internet Explorer. You'll see the video playing, embedded on the page, similar to the page shown in Figure 5-2. Note that this works only if you have Windows Media Player 10 installed on your machine, because the `classID` specified in the preceding code is specific to that version. There are different `classID` values for earlier versions of Windows Media Player.

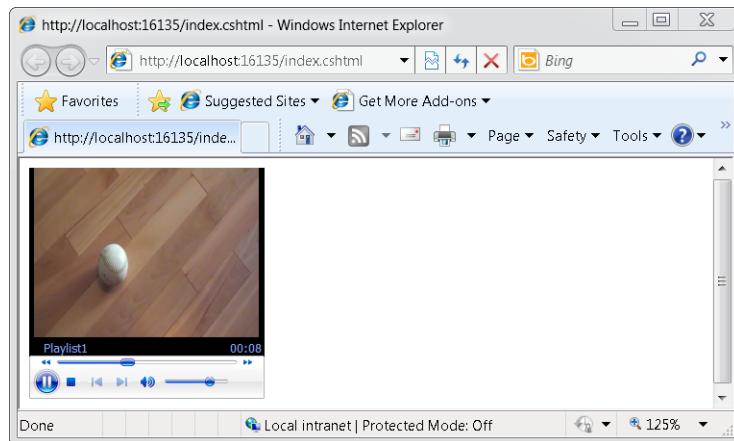


FIGURE 5-2 The video playing in Internet Explorer.

If you try to run this page in another browser, such as Opera, you might not get the same results, because other browsers might require you to explicitly install a plug-in

to allow the media to be played. For example, Figure 5-3 shows how the page looks in a fresh install of Opera. Note that because Opera requires a plug-in, the video doesn't appear; you only see the fallback text.

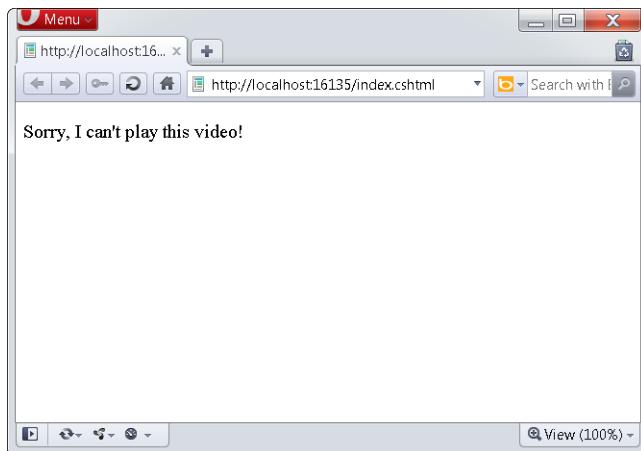


FIGURE 5-3 Running the video with an *<object>* tag in Opera.

Opera and other browsers might use the *<embed>* tag instead of the *<object>* tag for media. At this point, you're beginning to encounter some of the problems inherent in web programming; it's easy to end up having to write one piece of code for one browser and another for another browser. When you have to adapt your code to accommodate multiple browsers, you quickly reach a situation in which the code can become difficult to manage and maintain.

3. But don't give up. Try this amendment to the code, replacing the fallback text with an *<embed>* tag:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    <object classid="clsid:6BF52A52-394A-11D3-B153-00C04F79FAA6" >
      <param name="URL" value="/Media/sports.wmv" />
      <param name="playCount" value="2" />
      <param name="uiMode" value="full" />
      <param name="stretchToFit" value="True" />
      <param name="volume" value="75" />
      <embed src="/Media/sports.wmv" type="application/x-mplayer2"
        playCount="2" uiMode="full" stretchToFit="True" volume="75" />
    </object>

  </body>
</html>
```

Figure 5-4 shows the results of running the preceding code. Although the page still doesn't work, it at least shows end users what they need to do. In this case, they need to install the Windows Media Player plug-in.

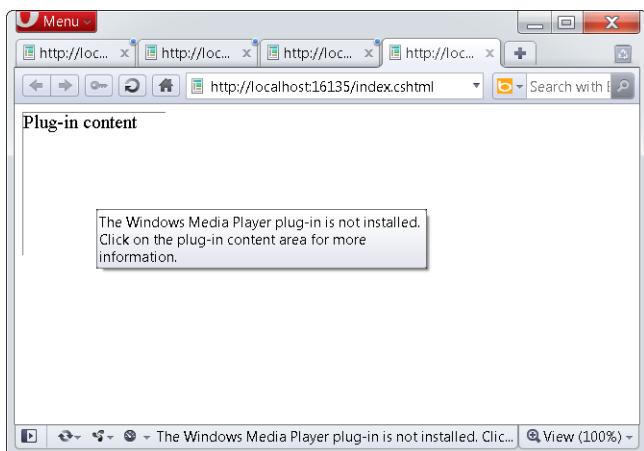


FIGURE 5-4 Opera using the `<embed>` tag fallback.

Now, when users encounter this problem, they can follow the message directions and install the plug-in. They'll eventually see the video, as shown in Figure 5-5.



FIGURE 5-5 Viewing the video with the plug-in installed.

This works because the Opera browser (as well as Firefox) prefers to use the `<embed>` tag instead of the `<object>` tag, because `<object>` tends to be bound to a *classID*, which is operating-system specific. In contrast, `<embed>` uses a concept called a *MIME type* that isn't operating-system specific. In the attributes of the `<embed>` tag for this example, the MIME type is specified by using `type="application/x-mplayer2"`. Many applications have MIME types. You can find an official list of available MIME types at: <http://www.iana.org/assignments/media-types/>.

On the surface, using MIME types seems to be a better idea, but the `<embed>` tag has been deprecated in favor of the `<object>` tag. That's because the general tendency in web programming is to use cross-browser, cross-platform objects such as Flash and Silverlight as a solution to the problem. As mentioned at the beginning of this chapter, the HTML5 specification will be updated to include a `<video>` tag. Although the `<video>` tag is not widely supported among installed browsers today, and its implementation among those browsers that do support it is currently inconsistent, you'll see how to use it later in this chapter.

As you can see, delivering video to different browsers and different machines—all of which might handle things in different ways—can quickly get complex.

WebMatrix solves some of these problems by including another, easier, way of delivering video: the *Video* helper.

Using the *Video* Helper

The *Video* helper function supports playback of various types of Windows Media files, including WMV video files and WMA audio files, as well as MP3 files. It also supports running Silverlight or Flash applications that can be used to play back videos.

It's simple to use. You simply call the `@Video.MediaPlayer` helper, passing it some properties that define the player's behavior. The available properties include:

- ***path*** Location of the media file.
- ***width*** Width of the player in pixels.
- ***height*** Height of the player in pixels.
- ***autoStart*** True or false. If true, the video will start automatically.
- ***playCount*** The number of times the video will play.
- ***uiMode*** Specifies how much of the player the user interface will show. The value can be invisible, none, mini, or full.
- ***stretchToFit*** Specifies whether the player will stretch to fit its container, overriding width and height.
- ***enableContextMenu*** True or false. If true, users can right-click the video to get a context menu.
- ***mute*** True or false. If true, the audio is muted.
- ***volume*** An integer value between 1 and 100, with 100 being the loudest.

You don't have to include every property to use the *Video* helper. The following exercise walks you through using the helper.

1. Create a new page and edit its index.cshtml file as shown in the following code:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        @Video.MediaPlayer(path:"media/sports.wmv",
                           width:"400", height:"400", autoStart:true)
    </body>
</html>
```

2. Run this page (see Figure 5-6). You'll see that it works the same way as the `<object>` tag did, in the example that included the `<embed>` tag as a fallback for browsers other than Internet Explorer.

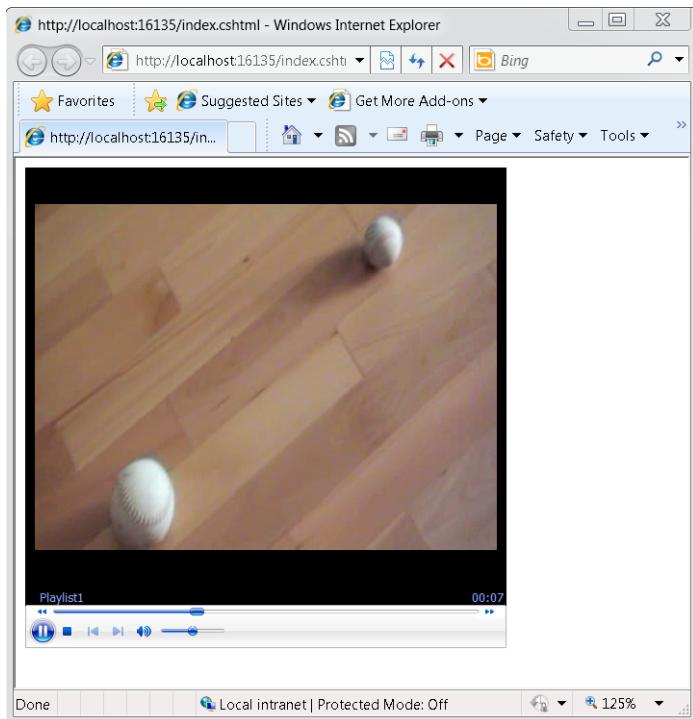


FIGURE 5-6 Running the Windows Media Player by using the `Video.MediaPlayer` helper.

3. Look at the source code of this page from your browser. You'll see that the *Video* helper has created an *<object>* tag in exactly the same way that you did manually:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <object width="400" height="400"
            classid="clsid:6BF52A52-394A-11D3-B153-00C04F79FAA6" >
            <param name="URL" value="/media/sports.wmv" />
            <embed src="/media/sports.wmv"
                width="400" height="400" type="application/x-mplayer2" />
        </object>
    </body>
</html>
```

This demonstrates that using the *Video.MediaPlayer* helper makes writing your code a lot easier as a developer than writing the *<object>* and *<embed>* tags yourself.

Using Flash Video

Adobe Flash is probably the most popular plug-in for websites; it's available on the vast majority of browsers worldwide. That popularity both reflects and drives the use of Flash for delivering video on sites. In fact, most of the major video sites, including YouTube, use Flash to deliver their video.

The *Video* helper provides a Flash helper too, but despite the name, the Flash helper can actually launch *any* Flash item, not just a video.

It's important to note that Flash is *not* a video player. It's an application platform on which developers build applications stored as .swf files. The Flash runtime, which users must have downloaded and installed to their machines, executes the .swf files. Flash videos use the FLV format. You can build a media player in Flash that renders these videos. For example, if you look at the source code for any site that uses a Flash video, you'll most likely see that it has an *<object>* tag that loads a .swf file, where the .swf is the application that encapsulates the player *and* the FLV video. You might also see a parameter specifying the FLV file that the player should load.

Because of Flash's popularity, converting other video formats to work with Flash is a common need. A very useful tool for doing this is SUPER, a freeware tool available from eRightSoft (<http://www.erightsoft.com>). SUPER lets you convert a file from one of many media formats, including video, into an .swf file. The resulting .swf file contains a media player with your converted video embedded in it. I used it to create a sports.swf file used in this example.

1. If you have an .swf file, add it to the media folder in the 5-1 project.
2. Now substitute `@Video.Flash` instead of `@Video.MediaPlayer`, and specify the path to the .swf file as the first parameter:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    @Video.Flash(path:"media/sports.swf",width:"400", height:"400")
  </body>
</html>
```

3. Run the page. You'll see the video playing in a Flash player. Right-click the video to see the Flash version information.

The following shows the code emitted by the server:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    <object width="400" height="400" type="application/x-oleobject"
      classid="clsid:d27cdb6e-ae6d-11cf-96b8-44553540000"
      codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab" >
      <param name="movie" value="/media/sports.swf" />
      <embed src="/media/sports.swf" width="400" height="400"
        type="application/x-shockwave-flash" />
    </object>
  </body>
</html>
```

Again, you can see how the `Video` helper writes both the `<object>` and `<embed>` tags for you.

Using Silverlight Video

Silverlight is the fastest growing plug-in in history. Like Flash, it is a cross-platform, cross-browser, and cross-device technology. It's a Microsoft .NET Framework-based application platform that is extremely powerful and has deep support for building rich applications easily.

Just as `@Video.Flash` can be used to render any Flash application, despite the helper prefix of `@Video`, you can use `@Video.Silverlight` to render any Silverlight application. Silverlight applications are packaged in XAP (pronounced ZAP) files.



More Info To learn more about Silverlight and how to create Silverlight applications, check out my book *Microsoft Silverlight 4 Step by Step* (Microsoft Press, 2010), which shows you how to build Silverlight applications by using the free Microsoft Visual Web Developer 2010 Express toolkit.

1. This book's downloadable code includes a simple XAP file that is similar to the preceding Flash example; it contains a Silverlight media player that plays back the sports.wmv file. Copy the Ch5Player.xap file to your media folder, and change the index.cshtml file so that it calls the `@Video.Silverlight` function to see it in action. The following shows the code:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        @Video.Silverlight(path:"media/Ch5Player.xap",width:"400", height:"400")
    </body>
</html>
```

2. Run this to see a page containing a Silverlight application that plays back the video. The following shows the source of the page as emitted by the server:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <object width="400" height="400" type="application/x-silverlight-2"
            data="data:application/x-silverlight-2,">
            <param name="source" value="/media/Ch5Player.xap" />
            <a href="http://go.microsoft.com/fwlink/?LinkId=149156"
                style="text-decoration:none">
                
            </a>
        </object>
    </body>
</html>
```

This time, you'll note that Silverlight uses the `<object>` tag, but it does so without using a class ID. This allows it to run seamlessly within other browsers such as Opera, Firefox, and Chrome.

Using the HTML5 *<video>* Tag

Because implementing video on a page has been so complex in previous versions of HTML, for the next version, HTML5, the World Wide Web Consortium (W3C) has proposed a *<video>* tag that simply specifies which video to deliver. HTML5-capable browsers have built-in media playback abilities that meet this specification. It's a simple solution, but it should be noted that HTML5 is not a ratified specification—it's a proposal, and it will be for some time. It's also unlikely that advanced video features, such as adaptive streaming or digital rights management, will make it into the upcoming HTML5 specification, so HTML5 might only enable simple video scenarios.

Despite this, the rapid growth of devices that support HTML5 (such as Apple's iPad) and growing support for HTML5 within browsers such as Google Chrome and Internet Explorer 9, make it handy for you to have some knowledge of it in your toolbox. You can use WebMatrix to build HTML5 video sites, as you'll see in this section.

To test WebMatrix with HTML5, you'll need a browser that supports HTML5. Internet Explorer 9, which supports HTML5, is available at <http://www.beautyoftheweb.com>. You can also use Google Chrome, which you can download from <http://www.google.com/chrome>.

Although the implementation of the *<video>* tag is still inconsistent between browsers, the following attributes are available:

- **Autoplay** When autoplay is used, the video will play automatically.
- **Controls** When controls is used, the video playback controls will appear.
- **Height** This setting specifies the height of the player in pixels.
- **Loop** When loop is used, the video will loop continually.
- **Preload** If preload is used, the video will be loaded before it is played. This setting is ignored if autoplay is set.
- **Src** This is the URL of the video to be played.
- **Width** This setting specifies the width of the player in pixels.

HTML5 can use several video formats, but the most common one at time of writing is H.264. Therefore, an H.264 version of the sports video, encoded as a QuickTime .mov file, is provided with the download code that accompanies this book. You can either use that file or generate your own for this exercise. The exercise uses several different browsers and assumes that you have them installed. If you don't have the browser in question, just move on to the next one.

1. Create a webpage using the following HTML5 code:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <video src="media/sports.mov" controls="controls"
               autoplay="autoplay">
            Your browser does not support the video tag
        </video>
    </body>
</html>
```

2. Run this page with Internet Explorer 8. You'll see results similar to Figure 5-7, because Internet Explorer 8 doesn't support HTML5.

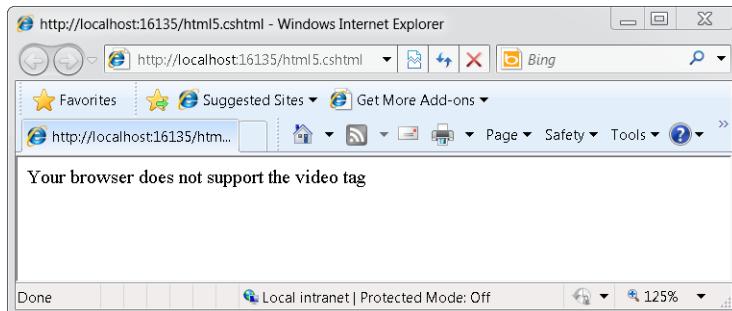


FIGURE 5-7 Using HTML5 in Internet Explorer 8.

As you can see, browsers that don't support the `<video>` tag will render the fallback text instead, letting the user know that his or her browser doesn't support the content.

3. Next, check Opera. Although at the time of this writing, Opera 10.5 does support the HTML5 `<video>` tag, it doesn't support this particular type; in other words, H.264-encoded .mov files are *not* supported in Opera, and you'll see a blank screen, as shown in Figure 5-8.



FIGURE 5-8 Trying to view an HTML5 H.264 video in Opera.

Although this browser does recognize the `<video>` tag, it can't handle the type of encoding, but because it recognizes the tag, it doesn't render the fallback text.

4. Google Chrome supports both the `<video>` tag and the H.264 encoding, so when you try to view the page in Chrome, it works. You'll see results similar to those shown in Figure 5-9.

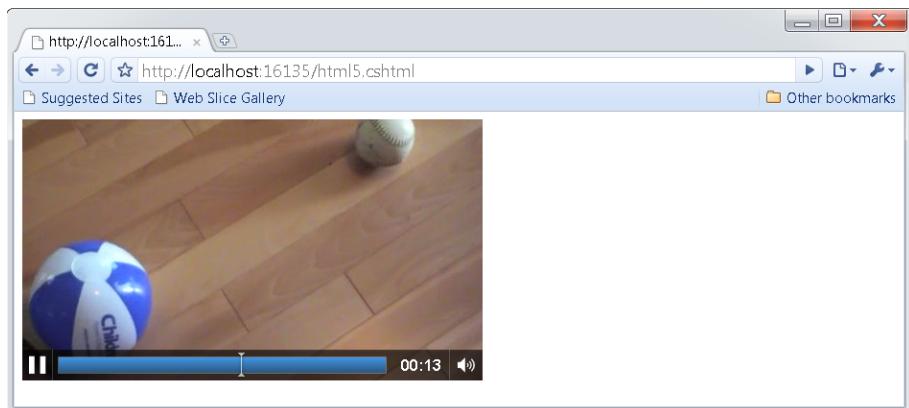


FIGURE 5-9 Viewing the video in Chrome.

5. Finally, Internet Explorer 9 also supports both H.264 encoding and the HTML5 `<video>` tag, so you can use it to view the rendered video, as in Figure 5-10.

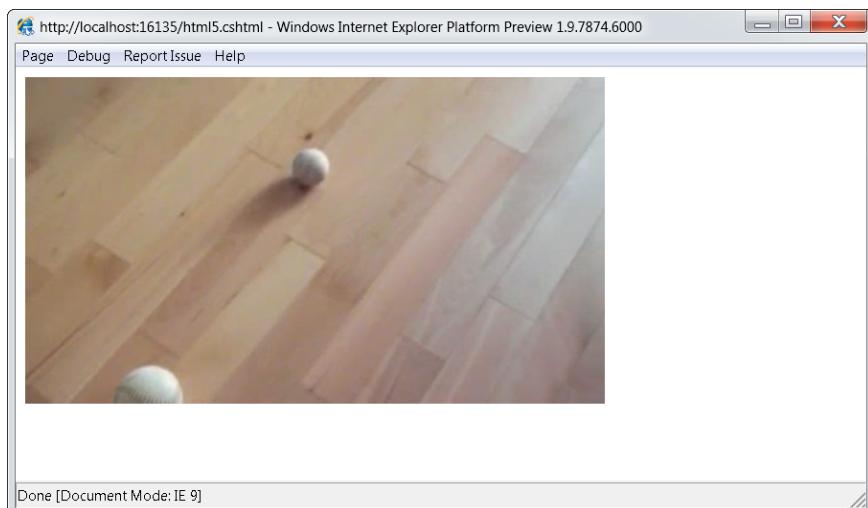


FIGURE 5-10 Viewing the video in the Preview version of Internet Explorer 9.

You can evaluate the current level of support for HTML5 and the `<video>` tag for yourself from this brief exercise. Wikipedia (http://en.wikipedia.org/wiki/HTML5_video) is a great resource for discovering which browsers support the `<video>` tag in HTML5 and which formats they support. At the bottom of the Wikipedia page, you'll see a table that shows what each browser supports.

At the time of this writing, no browser supports *all* of the various video formats listed. Therefore, you (and your users) will experience inconsistent behavior until the standards are settled and implemented more completely.

Summary

This chapter explored several ways that you can add basic video to your webpages. You saw how to embed a video by using a hyperlink that opens the user's default media player. You then looked at the `<object>` and `<embed>` tags, which you can use to embed a media player and use it to play a specific video.

Next, you saw how the *Video* helper available in WebMatrix simplifies the process of embedding media files and applications built on Flash or Silverlight into your page.

Finally, you briefly explored the emerging HTML5 technology and saw how to implement a simple media player by using the new `<video>` tag.

Chapter 6

Forms and Controls

In this chapter, you will:

- Explore how forms work.
- Take a tour of the available form controls.
- Find out how to capture user input in forms.

Back in Chapter 3, “Programming with WebMatrix,” you saw how to *POST* information back to the server by using an HTML form control. In this chapter, you’ll look at that process in a little more detail by taking a tour of the available form controls and seeing how the server captures the data that your users enter into them.

How Forms Work

The HTTP protocol has two main verbs that browsers use when interacting with a server to get data. The *GET* verb retrieves a page from a specified URL. It’s typically a passive operation, in which little or no data is passed to the server, only a request that means “send me the resource from this URL.” I say “typically” because there is a way to send a small amount of data by using request parameters added to the URL—a parameterized *GET* request. A *GET* request that passes parameterized data to the server would look something like this:

```
http://server/page.cshtml?param1=value1&param2=value2
```

The parameter list in the preceding URL begins with a question mark. Each parameter is a *name=value* pair, and the parameters are separated by ampersands.

In contrast, the *POST* verb isn’t issued from the browser address line; instead, it’s issued when a user clicks a submit button in a page. Within your page’s HTML, you can define a *<form>* to include an action that is the URL of a page on your site. If the form has a submit button, when a user clicks the button, the browser will call that page, embedding the data from the form within the call. You can also make *POST* calls programmatically. You’d typically use the *POST* verb if you have a lot of data that you want to pass up to the server, because using the parameterized *GET* statements (as shown in the previous line of code) would quickly become unwieldy. *GET* statements also have a size limit that varies by browser to some degree, so using *GET* for large amounts of data, such as uploading an image, doesn’t work.

A Simple Example

1. Launch WebMatrix and create a new empty site named **6-1**.
2. Edit the content of index.cshtml so that it contains the following code:

```
@{  
    var sum = 0;  
    var sumText = "";  
    var num1="";  
    var num2="";  
    if(IsPost) {  
        num1 = Request["fNum"];  
        num2 = Request["sNum"];  
        sum = num1.ToInt() + num2.ToInt();  
        sumText = "The answer is : " + sum;  
    }  
}  
  
<!DOCTYPE html>  
<html>  
    <head>  
        <title></title>  
    </head>  
    <body>  
        <form action="" method="post">  
            <p>First Number:<input type="text" name="fNum" value="@num1" /></p>  
            <p>Second Number:<input type="text" name="sNum" value="@num2" /></p>  
            <p><input type="submit" value="Add 'em up" /></p>  
        </form>  
        <p>@sumText</p>  
    </body>  
</html>
```

3. Run the page. You'll see something similar to Figure 6-1.

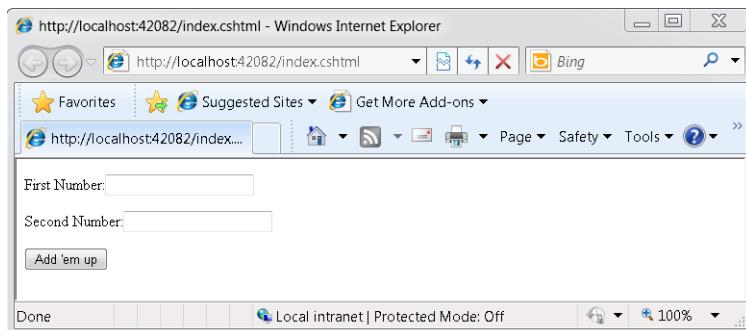


FIGURE 6-1 A simple forms-based application.

4. Type a couple of numbers into the text boxes and click the Add 'Em Up button. This is the submit button for this form, so when you click it, the form is submitted and the server responds with the result (see Figure 6-2).

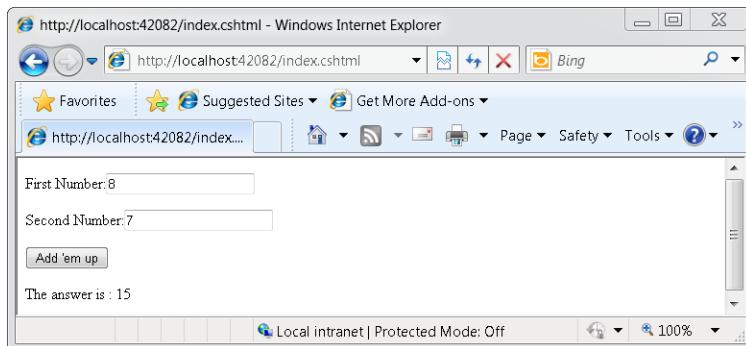


FIGURE 6-2 Running the form.

The important thing to notice in this application is the HTML `<form>` tag within the body of the page, as shown in the following code:

```
<form action="" method="post">
    <p>First Number:<input type="text" name="fNum" value="@num1" /></p>
    <p>Second Number:<input type="text" name="sNum" value="@num2" /></p>
    <p><input type="submit" value="Add 'em up" /></p>
</form>.
```

When you define a form, you can set several attributes.

- The *action* attribute is required. It defines the URL to which the browser will submit form data when a user clicks the submit button. If you leave the *action* attribute empty, the browser will submit the form data to the same page that it used to retrieve the page to begin with. This is a useful technique, because you can use the same CSHTML page both to deliver the user interface of the form and to handle any form data submitted by the user.
- The *accept* attribute specifies the type of data (called a MIME-type) that the form will submit.
- The *accept-charset* attribute specifies the character sets that the browser will use for form data. This is useful when you have localized applications that use non-English character sets.

- The *enctype* attribute specifies how (and whether) data is encoded before it is sent to a server. You can use *text/plain* for plain text. Alternatively, you can use *multipart/form-data* to break large files into chunks, such as when you're uploading an image via a form. (For more information about the *multipart/form-data* type, see "Using the *WebImage Helper*" in Chapter 4, "Using Images in WebMatrix.") It can also take the value *application/x-www-form-urlencoded*, for when the form specifies content that you want to have converted into URLs.
- The *method* attribute can take either a *get* or *post* value. Typically, you will use *post*, which is the default. These values are equivalent to the HTTP verbs discussed earlier in this book.
- The *name* attribute holds the name for the form.
- The *target* attribute defines what will happen when the browser submits the form. The value can be *_blank* to open the results in a new browser window; *_self* to open the results in the same window; *_parent* to open the results in the browser that opened the current browser (if it exists); *_top* to open in the top frame of a frameset if the current page is within a frameset; and *framename* if a frame with the name *framename* exists within the current frameset.



Note The *target* parameter is deprecated, meaning that future versions of HTML probably will not support it.

In the preceding `<form>` example, you can see that the form has a blank *action* parameter, meaning that the form's originating page (`index.cshtml` in this case) will accept the data back from the form. Also, the *method* is set to *post*, meaning that the form will use the HTTP *POST* verb, embedding the data in the HTTP header.

Setting the *action* to blank causes the form to post data back to the originating page, but to handle the submission, the page needs to determine whether it's being called by a *GET* (meaning that it will give the blank form to the user) or a *POST* (meaning that it will add up the numbers and send the result back to the user). The code at the top of the page handles this determination. It determines whether the request is a *POST* by checking `if(IsPost)`. If the request is a *POST*, it adds the two numbers and stores the answer—including some text—into the `sumText` variable. This variable is rendered whether the request was a *GET* or a *POST*, but for a *GET* request, it will be empty, so users don't see it!

Exploring HTTP Headers with Fiddler

I've mentioned HTTP headers a couple of times already in the context of HTTP *POST* verbs. But you might not understand what they are and would probably like to see them in action, so this section is a quick detour to explore them.

There's a great free tool available called Fiddler that lets you inspect the headers your browser is sending to a server. You can download this at <http://www.fiddler2.com/fiddler2/>.

1. Download and install Fiddler from the link, and then launch it. Fiddler works by acting as a *proxy* between your browser and the server; in other words, it captures web traffic and passes it on to the server.



Important When you use Windows Internet Explorer, Fiddler can't capture the *localhost* address easily. There are ways around this, but rather than going into them, for this exercise we use the Firefox browser, which doesn't proxy the *localhost* address, leaving it accessible so Fiddler can capture it.

2. Using WebMatrix, you can run your application in a specific browser—in this case, Firefox. Click Run on the ribbon to open the Run menu, and select Firefox as your browser (see Figure 6-3).

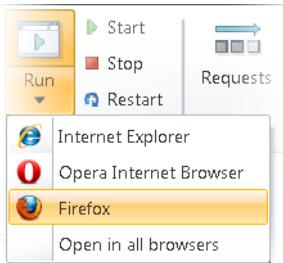


FIGURE 6-3 Running your WebMatrix site in Firefox.

If Fiddler is running, you should see requests being captured. If you don't, select File | Capture Traffic, and then run the page in Firefox again. If you have browsers or any Internet software such as music players running, you'll probably see a lot of traffic. But you're only interested in the *localhost* traffic. You should see at least one instance of this, which is the initial *GET* when the browser opened your page.

3. Your form application will be running in the browser now, so type a couple of numbers into the boxes and click the Add 'Em Up button to generate the *POST* request.

Figure 6-4 shows what Fiddler will look like with these two calls.

The screenshot shows the Fiddler application window titled 'Fiddler - HTTP Debugging Proxy'. The main pane displays a table of captured network sessions. The columns are labeled '#', 'Result', 'Proto...', 'Host', 'URL', 'Body', 'Caching', 'Content-Type', 'Process', and 'Com...'. There are two rows visible in the table:

#	Result	Proto...	Host	URL	Body	Caching	Content-Type	Process	Com...
1	200	HTTP	localhost:42082	/	304	private	text/html; charset=utf-8	firefo...	
3	200	HTTP	localhost:42082	/	323	private	text/html; charset=utf-8	firefo...	

FIGURE 6-4 Inspecting web traffic with Fiddler.

4. Select the first *localhost* call to inspect the details. On the right side of the screen, you'll see several tabs. Select the Inspectors tab. The top half of the screen will show the request, and the bottom half will show the response from the server. Click the Raw tab in each section. Your screen will look something like that shown in Figure 6-5.

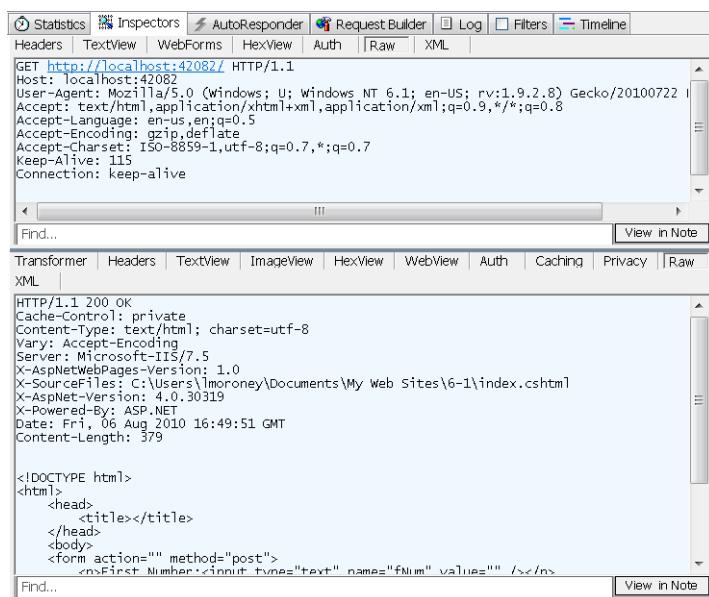


FIGURE 6-5 Inspecting the *GET* request and response.

If you look closely at Figure 6-5, you'll see that the top half shows data for the initial page *GET* request. In the very first line, right at the top of the top window, you can see the details for the *GET* verb, in which the browser asks the server to *GET* the resource that you specified. Fiddler will ask you to decode the information before inspection. After you've done this, the bottom half shows the server's response; you can see the HTML in the response that the browser renders.

5. Now select the second *localhost* request in the pane on the left side of the Fiddler screen. This will probably be the traffic that was captured when you submitted the *POST*. Again, the Inspector panes fill with the request that your *POST* request generated and the response from the server (see Figure 6-6). When you are viewing the Raw tab, you can see that the request header includes the variables in *name=value* form, in an ampersand-separated list. Also note that the HTML the server returned now includes both of the values that you typed in, as well as the result. You can see this in Figure 6-6.

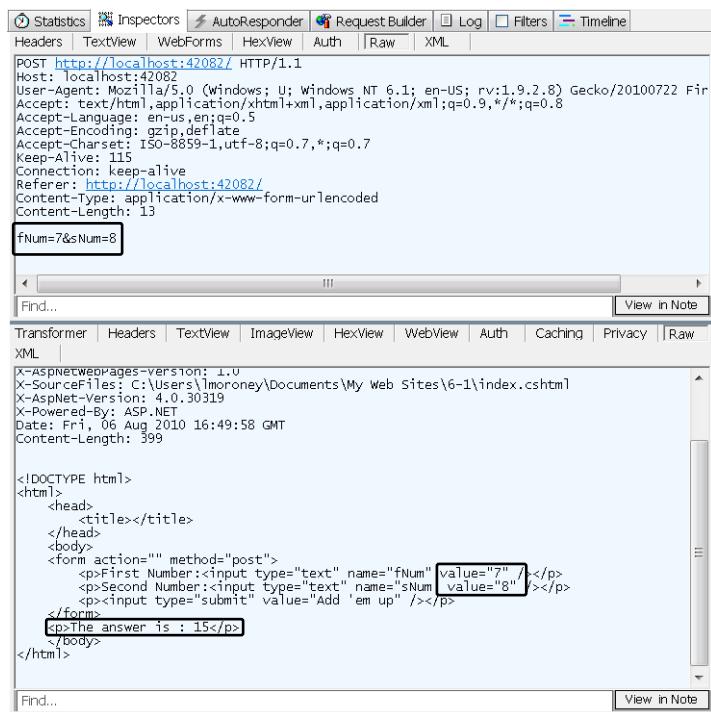


FIGURE 6-6 Inspecting the *POST* request and response.

Regardless of your expertise in web development, whether you are an absolute beginner or an established expert, you'll find that having the ability to inspect exactly what's occurring "on the wire" will make you better at your job.

Exploring the Form Controls

In the previous example you built a form that contained a couple of text boxes and a submit button. You were able to use this to type in a couple of text values, submit them, and have the server add them up. But your form palette is not limited to text boxes and submit buttons. In this section, you'll take a look at the various form control types that you can use in HTML.

Text Boxes

To define a single-line text box, you use the `<input>` control and set its `type` parameter to `text`. You should use the `name` parameter to specify the name of the text box. Although setting a name is optional, it's important to add a name if you want to capture the data, because as you saw in Figure 6-6, HTTP passes form data to the server by using the `name=value` syntax.

The following is a brief example that uses text boxes:

```
<form>
    First name: <input type="text" name="fName" />
    <br/>
    Last name: <input type="text" name="lName" />
</form>
```

Figure 6-7 shows how these are rendered in the browser.

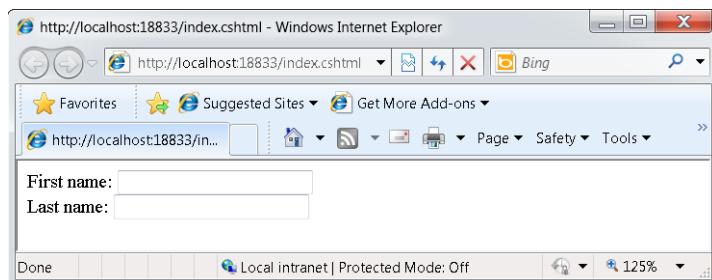


FIGURE 6-7 Rendered text boxes.

Text boxes are used for simple, single-line string input, such as your first name, a number, or anything else that is straightforward.

Password Boxes

A special type of text box that disguises its content is typically used when you need a user to enter a password. Although the password box doesn't *show* its content, it does *store* the content so that it can be used by the server. To create this type of text box, use the `<input>` tag with the `type` attribute set to `password`. The following is the same example with a password field added:

```
<form>
    First name: <input type="text" name="fName" />
    <br/>
    Last name: <input type="text" name="lName" />
    <br/>
    Password: <input type="password" name="pWord" />
</form>
```

You can see how this is rendered in Figure 6-8, which shows how the browser renders the password box and the content that is typed into it.

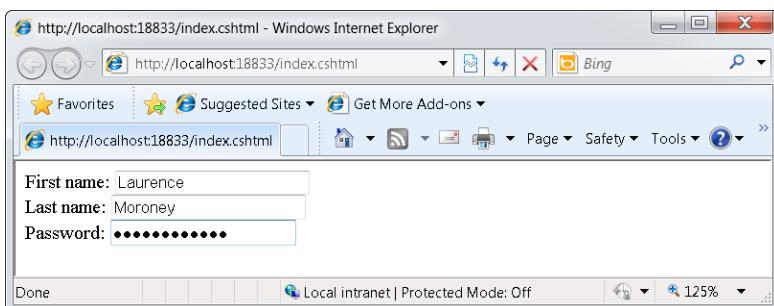


FIGURE 6-8 Using the password box.

Note that when you use a password box, its contents are *not* encrypted by the browser unless the page uses HTTPS. For example, if you were to submit the preceding form and capture the results with Fiddler, you'd see data similar to the following:

```
POST http://localhost:18833/index.cshtml HTTP/1.1
Host: localhost:18833
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722
Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:18833/index.cshtml
Content-Type: application/x-www-form-urlencoded
Content-Length: 69

fName=Laurence&lName=Moroney&pWord=secretpassword&submit=Submit+Query
```

As you can see in the last line, the password is right there, in plain text. This means that someone could potentially steal your password. In contrast, when using HTTPS, the browser encrypts the content in such a way that only the server can understand it.



Note The HTTPS protocol uses Secure Sockets Layer (SSL) encryption. A full discussion of SSL encryption and its certificate system is beyond the scope of this book.

Option Buttons

Option buttons (also called *radio buttons*) allow the user to choose one item from a selection. A classic example is when you want users to specify their gender: male or female. You don't want to allow users to choose both, so you limit them to only one answer.

You use the `<input>` control with the `type` attribute set to `radio` to specify an option button. Option buttons typically exist in groups. Users can select only one option out of each group. To make option buttons act as a group, give each button in the group the same name attribute. To create a form containing a question for gender, with the options *Male* or *Female*, you would create two `<input type=radio>` tags and give both tags the name `gender`. You might want multiple option button groups within the same form. For instance, if you also wanted to have a separate question for favorite sport, supplying a few options, you could give the sport option buttons the name `sport`. The following is an example:

```
<form action="" method="post">
    First name: <input type="text" name="fName" />
    <br/>
    Last name: <input type="text" name="lName" />
    <br/>
    Password: <input type="password" name="pWord" />
    <br/>
    Gender:
    <input type="radio" name="gender" value="Male">Male</input>
    <input type="radio" name="gender" value="Female">Female</input>
    <br/>
    Favorite Sport:
    <input type="radio" name="sport" value="1">Baseball</input>
    <input type="radio" name="sport" value="2">Football</input>
    <input type="radio" name="sport" value="3">Soccer</input>
    <input type="radio" name="sport" value="4">Basketball</input>
    <input type="radio" name="sport" value="5">Hockey</input>
    <br/>
    <input type="submit" name="submit" />
</form>
```

Figure 6-9 shows how the preceding form looks when rendered in a browser.

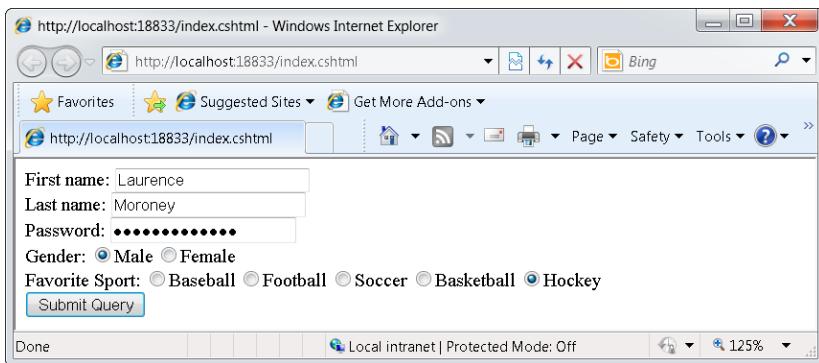


FIGURE 6-9 Using option buttons.

Note that when the form is submitted, it sends the *value* of the selected option, not the text. In the preceding example, if the user selects *Hockey* as his or her favorite sport, the form will submit a *name=value* pair of *sport=5*. Similarly, if you select the male Gender option, the form submits the value *gender=Male*.

The following is a captured HTTP request header containing those parameters:

```
POST http://localhost:18833/index.cshtml HTTP/1.1
Host: localhost:18833
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722
Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:18833/index.cshtml
Content-Type: application/x-www-form-urlencoded
Content-Length: 88

fName=Laurence&lName=Moroney&pWord=sillypassword&gender=Male&sport=5
&submit=Submit+Query
```

The *checkbox* Control

The *checkbox* control provides users with the ability to answer a question with a single click. Check boxes are similar to option buttons; you can group them by using a single name. But they're different from option buttons because the groups allow more than one answer to a question. For example, instead of asking users for their favorite sport, you might ask them to select each sport that they like, providing check boxes so they can select more than one.

You create a check box in exactly the same way as an option button, using the `<input>` tag, but you set the `type` attribute to `checkbox`. Otherwise the HTML is identical. For example, to ask users to select the sports they like, you can use the same group name and values as before, but specify a `type` value of `checkbox`. Here's the changed form:

```
<form action="" method="post">
    First name: <input type="text" name="fName" />
    <br/>
    Last name: <input type="text" name="lName" />
    <br/>
    Password: <input type="password" name="pWord" />
    <br/>
    Gender:
    <input type="radio" name="gender" value="Male">Male</input>
    <input type="radio" name="gender" value="Female">Female</input>
    <br/>
    Favorite Sport:
    <input type="checkbox" name="sport" value="1">Baseball</input>
    <input type="checkbox" name="sport" value="2">Football</input>
    <input type="checkbox" name="sport" value="3">Soccer</input>
    <input type="checkbox" name="sport" value="4">Basketball</input>
    <input type="checkbox" name="sport" value="5">Hockey</input>
    <br/>
    <input type="submit" name="submit" />
</form>
```

Figure 6-10 shows how the form looks when it's rendered in a browser. Note that check boxes have replaced the option buttons for the sports question.

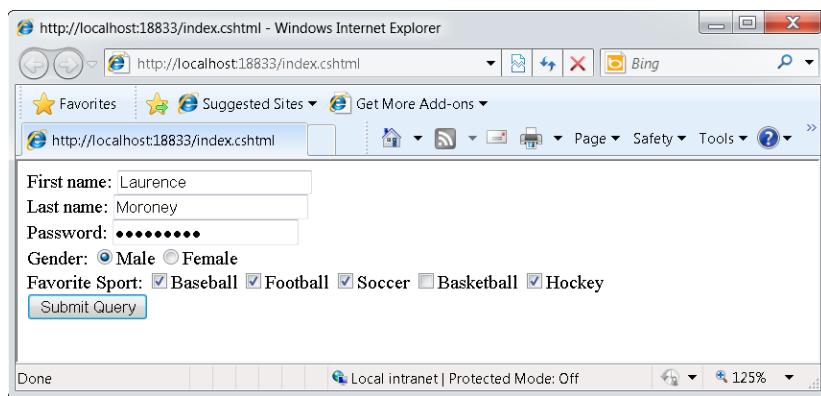


FIGURE 6-10 Using `checkbox` controls.

When you use the `checkbox` control, the HTTP header will include several instances of the check box name with its associated value—one for each checked control. In this case, the name of the `checkbox` group is `sport`, and there are five check boxes, which have the values

1 through 5. In Figure 6-10, the user selected four different sports, so when the form is submitted, the submitted header values should contain four *name=value* pairs, one for each selected control: *sport=1&sport=2&sport=3&sport=5*. You can see the captured header in the following code:

```
POST http://localhost:18833/index.cshtml HTTP/1.1
Host: localhost:18833
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722
Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:18833/index.cshtml
Content-Type: application/x-www-form-urlencoded
Content-Length: 104

fName=Laurence&lName=Moroney&pWord=pass&gender=Male&sport=1&sport=2
&sport=3&sport=5&submit=Submit+Query
```

The *TextArea* Control

Earlier you saw a text box that allowed users to enter a single line of text. HTML forms also support a *<textarea>* tag that lets users enter *multiple lines* of text. You can use the *rows* and *columns* properties to define the size of your *TextArea* control; in all other respects, it works exactly the same as the text box. The following is an example:

```
Comments:
<br/>
<textarea rows="8" columns="20" value="Enter comments here"
           name="tComments">
</textarea>
<br/>
```

I added the preceding code at the bottom of the form we've been working with throughout this section. You can see how the *TextArea* renders by looking at the *Comments* control in Figure 6-11.

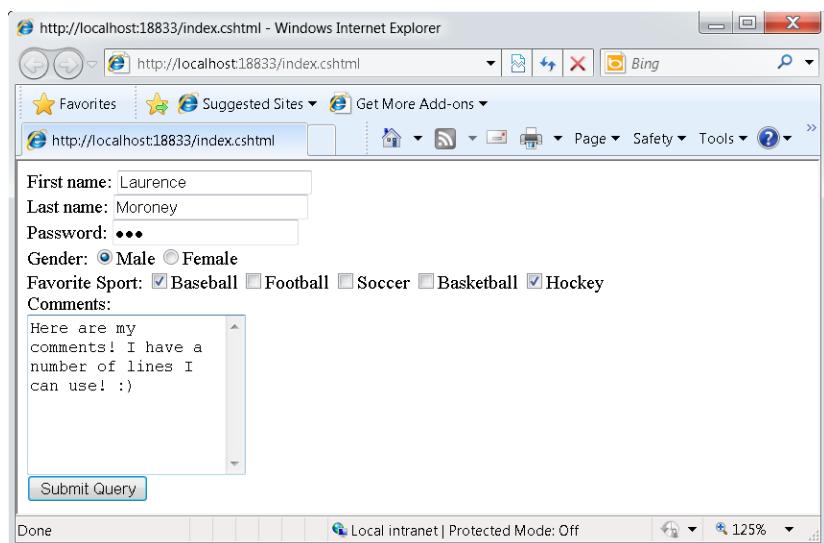


FIGURE 6-11 Using the *TextArea* control.

When this form is submitted, the browser sends the comments to the server. The following shows what the header looks like in Fiddler:

```
POST http://localhost:18833/index.cshtml HTTP/1.1
Host: localhost:18833
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722
Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:18833/index.cshtml
Content-Type: application/x-www-form-urlencoded
Content-Length: 165

fName=Laurence&lName=Moroney&pWord=213&gender=Male&sport=1&sport=5&
tComments=Here+are+my+comments%21+I+have+a+number+of+lines+I+can+use%21+%3A%29&
submit=Submit+Query
```

Just as with a single-line text box control, the browser sends a single *name=value* pair for a *TextArea* control, where *name* is the name you assigned to the control, and *value* is a long string of text. In that text, note that spaces have been replaced by plus (+) characters, and non-text characters such as the exclamation point (!), the colon (:), and the right parenthesis) have been replaced by values such as %21, %3A, and %29. This replacement operation is called *URL encoding*, and it's the default behavior for a browser form. The browser does this

to avoid sending characters that might confuse the server. The server will decode them before processing the data.

The *select* Control for Lists

You can offer your users a selection list by using the *<select>* tag. You define the options within your list by using the *<option>* tag, which supports a *selected* attribute that allows you to specify the default option. When the form is submitted, it sends the *value* of the selected item to the server. The following is an example:

```
Age:  
<select name="agegroup">  
    <option value="0" selected="selected">Prefer not to say</option>  
    <option value="1">0-12</option>  
    <option value="2">13-19</option>  
    <option value="3">20-35</option>  
    <option value="4">36-50</option>  
    <option value="5">Over 50</option>  
</select>  
<br>
```

You can see how this looks on the screen in Figure 6-12.

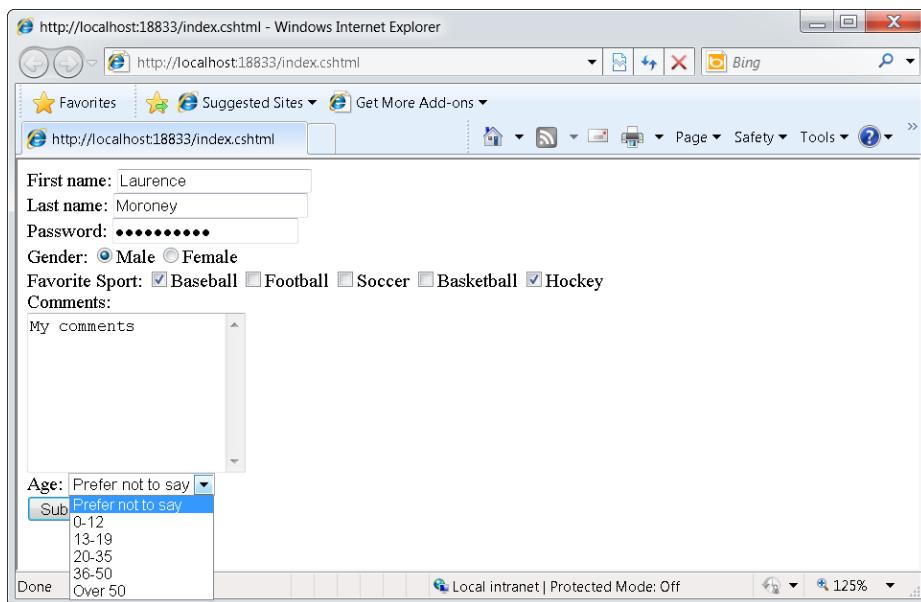


FIGURE 6-12 Using the *select* control.

As an example, if you submitted the form after selecting the Age selection 20-35, which has a value of 3, the browser sends an *agegroup=3* value pair:

```
POST http://localhost:18833/index.cshtml HTTP/1.1
Host: localhost:18833
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722
Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:18833/index.cshtml

Content-Type: application/x-www-form-urlencoded
Content-Length: 139

fName=Laurence&lName=Moroney&pWord=12123&gender=Male&sport=1&sport=5&
tComments=My+comments%0D%0Aok%21+%3A%29&agegroup=3&submit=Submit+Query
```

You can change the appearance and function of the *select* list from a drop-down list to a fully rendered list by adding a *size* attribute, which takes a numeric value that indicates the number of rows that should be visible. (Adding additional non-visible rows causes the control to render with a scrollbar if necessary.) You can also specify that users can select multiple values by specifying the tag as *<select multiple>* instead of just *<select>*. For multiple-selection lists, holding the Ctrl key as you click items lets you select multiple items. Additionally, holding the Shift key and clicking selects a range of items. The following is an example:

```
Programming Languages
<br/>
<select multiple name="lang" size="4">
    <option value="0" selected="selected">C#</option>
    <option value="1">Visual Basic</option>
    <option value="2">F#</option>
    <option value="3">PHP</option>
    <option value="4">Ruby on Rails</option>
    <option value="5">Punch Cards</option>
    <option value="6">Smoke Signals and Shouting</option>
</select>
<br/>
```

You can see how this looks in the browser in Figure 6-13.

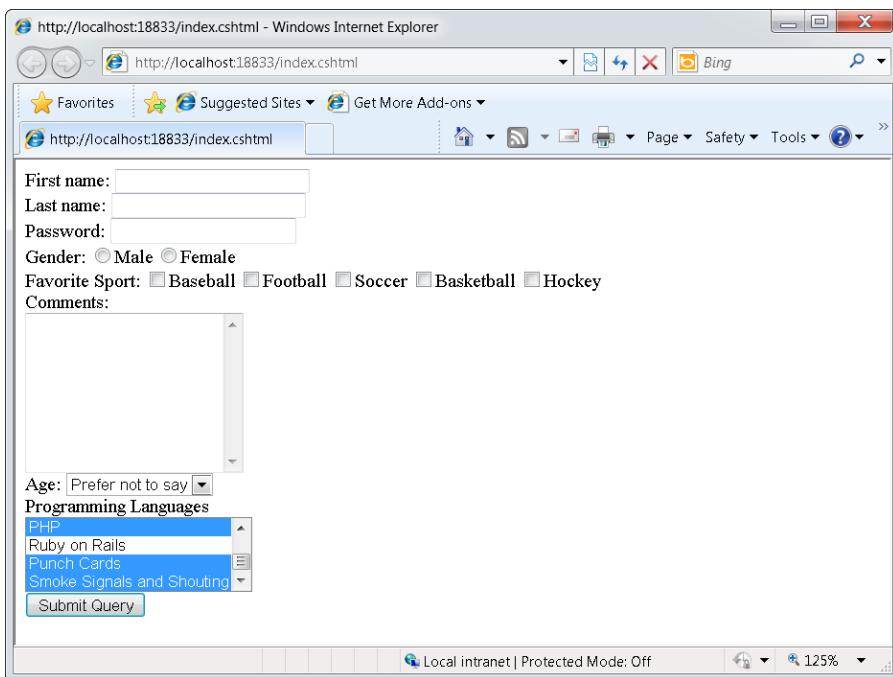


FIGURE 6-13 Using a *select* list with multiple entries selected.

When you submit this form, you'll have some ampersand-separated entries in the header, one for each selected item. The following is an example:

```
POST http://localhost:18833/index.cshtml HTTP/1.1
Host: localhost:18833
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.8) Gecko/20100722
Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://localhost:18833/index.cshtml
Content-Type: application/x-www-form-urlencoded
Content-Length: 83
fName=&lName=&pWord=&tComments=&agegroup=0&lang=3&lang=5&lang=6&submit=Submit+Query
```

This completes the tour of the HTML controls that are available to forms. In the next section, you'll look at how you can capture this input by using code on the server side.

Capturing Form Input

Capturing form input on the server is easy. First, remember to use `if(IsPost)` to separate the code that you want to run when the form is submitted with a *POST*, as opposed to a *GET*.

Then, you simply use `Request["name"]` to get the submitted values, where *name* is the name of the form field that you want to capture. If the form field has multiple items—such as items from check box groups, option button groups, or lists, you'll get a comma-separated list of values. If the input was HTML-encoded, such as the data submitted in the *TextArea* example in the preceding section, WebMatrix automatically decodes it for you.

1. Here's a simple page that uses a variety of form fields so that you can see how the data gets captured and re-rendered. Create this page by using WebMatrix:

```
@{
    var FirstName = "";
    var Gender = "";
    var Sport = "";
    var Comments = "";
    var Age="";
    var Language="";
    if(IsPost){
        FirstName=Request["fName"];
        Gender=Request["gender"];
        Sport=Request["sport"];
        Comments=Request["tComments"];
        Age=Request["agegroup"];
        Language=Request["lang"];
    }
}
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <p>You entered:</p>
        <p>@FirstName</p>
        <p>@Gender</p>
        <p>@Sport</p>
        <p>@Comments</p>
        <p>@Age</p>
        <p>@Language</p>
        <form action="" method="post">
            First name: <input type="text" name="fName" />
            <br/>
            Last name: <input type="text" name="lName" />
            <br/>
            Password: <input type="password" name="pWord" />
            <br/>
        </form>
    </body>
</html>
```

```
Gender:  
<input type="radio" name="gender" value="Male">Male</input>  
<input type="radio" name="gender" value="Female">Female</input>  
<br/>  
Favorite Sport:  
<input type="checkbox" name="sport" value="1">Baseball</input>  
<input type="checkbox" name="sport" value="2">Football</input>  
<input type="checkbox" name="sport" value="3">Soccer</input>  
<input type="checkbox" name="sport" value="4">Basketball</input>  
<input type="checkbox" name="sport" value="5">Hockey</input>  
<br/>  
Comments:  
<br/>  
<textarea rows="8" columns="20"  
          value="Enter comments here" name="tComments"></textarea>  
<br/>  
Age:  
<select name="agegroup">  
    <option value="0" selected="selected">Prefer not to say</option>  
    <option value="1">0-12</option>  
    <option value="2">13-19</option>  
    <option value="3">20-35</option>  
    <option value="4">36-50</option>  
    <option value="5">Over 50</option>  
</select>  
<br/>  
Programming Languages  
<br/>  
<select multiple name="lang" size="4">  
    <option value="0" selected="selected">C#</option>  
    <option value="1">Visual Basic</option>  
    <option value="2">F#</option>  
    <option value="3">PHP</option>  
    <option value="4">Ruby on Rails</option>  
    <option value="5">Punch Cards</option>  
    <option value="6">Smoke Signals and Shouting</option>  
</select>  
<br/>  
<input type="submit" name="submit" />  
</form>  
</body>  
</html>
```

- Type some sample data into this page and submit it. Pay close attention to how the server posts the page back, including the captured content.



Note You would typically use a form like this to enter user-supplied information into a database. You'll see how to do this in Chapter 7, "Databases in WebMatrix."

Figure 6-14 shows an example of how this page looks in a browser. In particular, note the comma-separated value for the multiple selections from the list box, and that even though the *TextArea* control's content gets URL encoded for transmission (see the *TextArea* section earlier in this chapter), WebMatrix has decoded it for you.

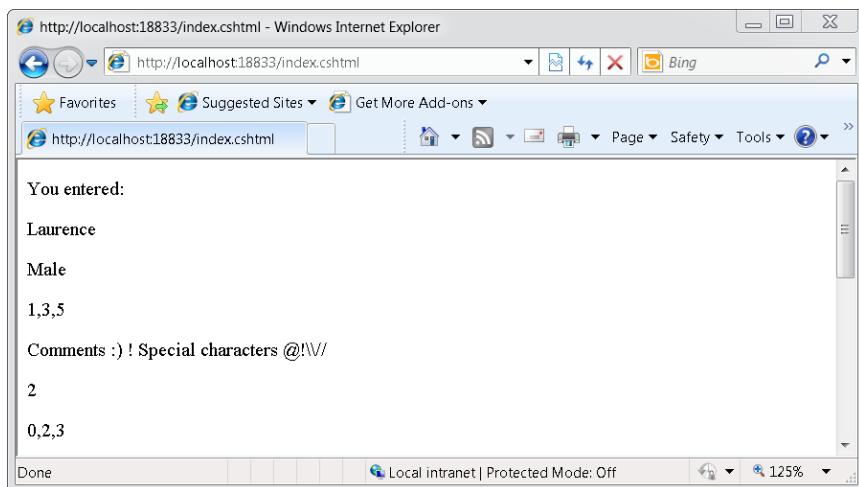


FIGURE 6-14 Capturing form input with server-side code.

Summary

This chapter provided a brief tour of HTML forms and HTML form controls. It showed you how to define form controls in your HTML and then see how browsers use the HTTP protocol to submit the data to the server. You then saw how easy it is to use WebMatrix to capture the entered data from the HTTP headers by using the *Request* object. In the next chapter, you'll build on this to see how to validate entered data and store it in a database.

Chapter 7

Databases in WebMatrix

In this chapter, you will:

- Create a database with WebMatrix.
- Explore how to query a database.
- See how to add, update, and delete database data.

Microsoft WebMatrix can connect not only to the built-in Microsoft SQL Server Compact database but to any database that provides drivers for the Microsoft .NET Framework. Programming websites to use databases is very straightforward with WebMatrix. In this chapter, you'll see how to use SQL Server Compact to create a database. You'll then see how you can use HTML forms and code to build a simple application that uses that database.

Creating a Database with WebMatrix

In this section, you'll find out how to create a new SQL Server Compact database by using WebMatrix.

1. To get started, launch WebMatrix and create a new site called **7-1** that uses the Empty Site template.
2. Select the Databases workspace, and then click the New Database button on the ribbon (see Figure 7-1).

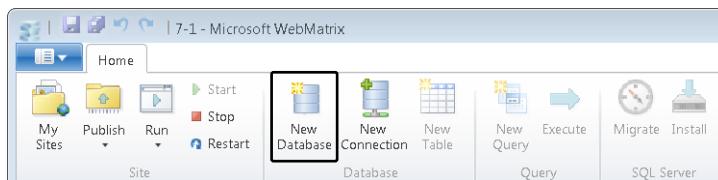


FIGURE 7-1 Creating a database with WebMatrix.

3. You'll see that WebMatrix creates a default database named **7-1.sdf**. The database is empty, as you'll see if you try to browse through the tables—there's nothing there. So click the New Table button to create a new table in your database. The table editor will open within the main workspace. The table editor allows you to specify the name and type of each data item in your database table (see Figure 7-2).

(7-1.sdf).NewTable_1		
Column Name	Data Type	Allow Nulls
<input type="text"/> id	bigint	True
Column Properties		
(Name)		
Allow Nulls		
Data Type		
Default Value		
Is Identity?		
Is Primary Key?		

FIGURE 7-2 Creating a new table.

To add columns (fields) to your table, simply type the name you want in the column field, and then select the data type that you want to use and whether you want to allow nulls.

In this exercise, you'll create a list of contacts, each of whom has a first name, a last name, and a Twitter page address. Each contact will also have a unique ID. This will be the first field that you add. You want the ID field to be unique for each contact; fortunately, SQL Server gives you a way to add unique ID values, in what it calls an *identity* field.

- Type **id** in the column field, and specify that the field is both an identity *and* the primary key. Your properties area for this field will look something like that shown in Figure 7-3.

Column Name	Data Type	Allow Nulls
<input type="text"/> id	bigint	False
Column Properties		
(Name)		
Allow Nulls		
Data Type		
Default Value		
Is Identity?		
Is Primary Key?		

FIGURE 7-3 Adding the ID field.

Assigning the *identity* specification to a field means that the database will automatically fill in that field with a unique value as new records are added. The database does this by incrementing a value, which by default starts at 0. Thus, the first record in the table will be given the value 1, the second 2, and so on.

5. Next, add three new columns, all of the type *nchar*. Name them **FirstName**, **LastName**, and **TwitterID**.
6. When you're done, click the Save button in the title bar. In the Save Table dialog box, enter the name **TwitterFriends** for your table, and click OK.

When you're done, you'll see *TwitterFriends* in your Databases workspace. The table definition should look like that shown in Figure 7-4.

The screenshot shows the Microsoft WebMatrix Beta interface. The ribbon at the top has 'Home' and 'Table' selected. The left sidebar shows a database named '7-1' containing a file '7-1.sdf' and a folder 'Tables' which contains a table named 'TwitterFriends'. The main area is titled 'Table - (7-1.sdf).TwitterFriends' and displays the table definition with four columns: 'id' (bigint, Allow Nulls False), 'FirstName' (nchar, Allow Nulls True), 'LastName' (nchar, Allow Nulls True), and 'TwitterID' (nchar, Allow Nulls True).

FIGURE 7-4 The TwitterFriends table in the 7-1 database.

7. Double-click the table within the explorer view (or click the Data button on the ribbon). You'll see a spreadsheet-like interface that you can use to enter data (see Figure 7-5).

The screenshot shows the Microsoft WebMatrix Beta interface with the 'Data' tab selected in the ribbon. The left sidebar is the same as Figure 7-4. The main area is titled 'Table - (7-1.sdf).TwitterFriends' and shows a data grid with four columns: 'id', 'FirstName', 'LastName', and 'TwitterID'. A single row is present with an asterisk (*) in the 'id' field.

FIGURE 7-5 The data entry view.

8. Don't type anything in the ID field—it will be filled in automatically for you. Go to the *FirstName* field and start typing in some data. You can move between fields by using the Tab key on your keyboard. You'll see that the fields auto-fill with *NULL* before you enter data. As you tab through the fields, you'll see that the ID field gets updated for you automatically. You can see a table with sample data in it in Figure 7-6.

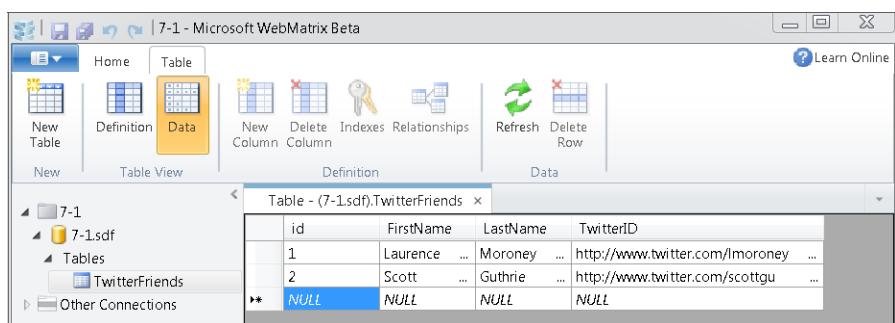


FIGURE 7-6 Entering data into a table by using WebMatrix.



Note For the purposes of this example, add the two rows shown in Figure 7-6 to your table.

That's how easy it is to use the WebMatrix integrated tools to create and populate a simple SQL Server database. In the next few sections, you'll see how to write code to manage this database.

Using a Database in Code

The standard way to get data from a database is to *query* the database for the information you want by using a language called Structured Query Language (SQL). This provides a very simple English-like language that allows you to select the information you want. You'll learn more about how to use this language in this section. SQL also allows you to add data (by using the *INSERT* command) and update existing data (by using the *UPDATE* command). You'll see how to add and insert data in the next sections.

The SQL *SELECT* command supports several uses, but the simplest and most common one you'll encounter looks like the following:

```
SELECT fields FROM table WHERE conditions ORDER BY field
```

The *fields* value specifies which fields to retrieve from the *table*. You can use the asterisk (*) character to indicate that you want to retrieve all the fields. If your table has many fields, retrieving them all—if you don’t need them—is not efficient: such queries will run more slowly and will affect overall application performance. Therefore, when you are using SQL, think carefully about which fields to select, how many entries there are in the database, how big the returned data will be, and so on. When your application is running on the web, you might have to handle many simultaneous queries, so you want each to be as fast as possible and require as few server resources as possible.

The *conditions* value specifies how to filter the rows. (If you want to return every row in the table, you can omit the *conditions*.) The *conditions* syntax is pretty straightforward. For example, you could write a query condition such as *WHERE age>21* to filter a table so that your query returns only rows for people whose ages are over 21. You can match similar conditions by using the *LIKE* keyword. When you use *LIKE* combined with wildcards in the query, you can achieve simple pattern matching. For example, the clause *LIKE A%* will match values that begin with the letter *A* (the % character is a wildcard that matches all text).

The *ORDER BY* clause indicates how to sort the query results. For example, if you have filtered your dataset by *age* using *WHERE age>21* and you want the results to be ordered by *age*, with youngest first, your query would include *ORDER BY age ASC* (where *ASC* means ascending order and *DESC* means descending order).

A query on the example table that returns all the records in the table would look like this:

```
SELECT * FROM TwitterFriends
```

Here’s another example that returns rows where the *FirstName* field contains *Laurence*:

```
SELECT * FROM TwitterFriends WHERE FirstName='Laurence'
```

And finally, to retrieve only the Twitter ID for people in the database whose first name begins with *L*, you could write the following:

```
SELECT TwitterID FROM TwitterFriends WHERE FirstName Like 'L%'
```

You can use the WebMatrix Databases workspace to test your queries. Simply click the New Query ribbon button to create a query and the Execute ribbon button to run it. You can see the result of a query in Figure 7-7.

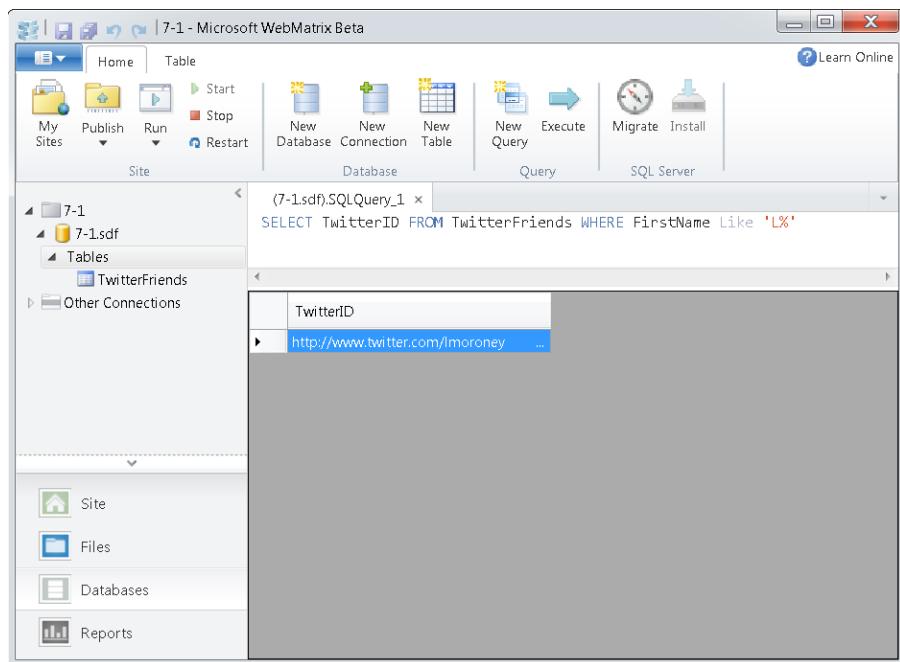


FIGURE 7-7 Testing queries with WebMatrix.

Using SQL in code is very easy. You simply create a database object and pass the SQL query string to its *Query* method.

1. To see a simple example of a page that connects to the example database and renders some results, create a new C#HTML page and call it index.cshtml. Edit it so it looks like the following:

```
@{
    var db = Database.Open("7-1");
    var query = "SELECT TwitterID From TwitterFriends";
}
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        @foreach (var item in db.Query(query))
        {
            @item.TwitterID;
            <br/>
        }
    </body>
</html>
```

Here's what that code does: At the top of the page is a code block that creates a *var* named *db* that gets initialized to a reference to the database by opening the database file 7-1.sdf.



Troubleshooting When you created the database earlier, 7-1.sdf would have been the default name. If you're using a different name, remember to use it instead when coding these examples. If you don't, the examples won't work.

Next, the code initializes a *query* variable to *SELECT TwitterID from TwitterFriends*. This query will select all the *TwitterID* values because it has no conditions set (no *WHERE* clause).

In the body of the page, you'll see the code *db.Query(query)*, which runs the query on the database and returns a collection of records. Each record has a collection of fields. In this case, the collection of records will have two items in it, because you entered two rows into the database. The *foreach* loop is designed to loop through each record in the collection and assign the current record to the variable *item* specified within the *foreach* line. The first time through the loop, *item* will be assigned the first record that resulted from the query, the second time it will be assigned the second record, and so on. The *item* will be an object that is set up and initialized with the values within the record, so if you have a record with multiple fields, the *item* object can access them by using the *item.fieldname* syntax. In this case, the query asked for only one field, *TwitterID*, so you can access it by using the *item.TwitterID* syntax, as shown.

Thus, the code that reads the returned records will loop through each row that the query returned and display its *TwitterID* value followed by a line break (*
*).

2. Run the program to see the result in your browser. You should see something like what is shown in Figure 7-8.

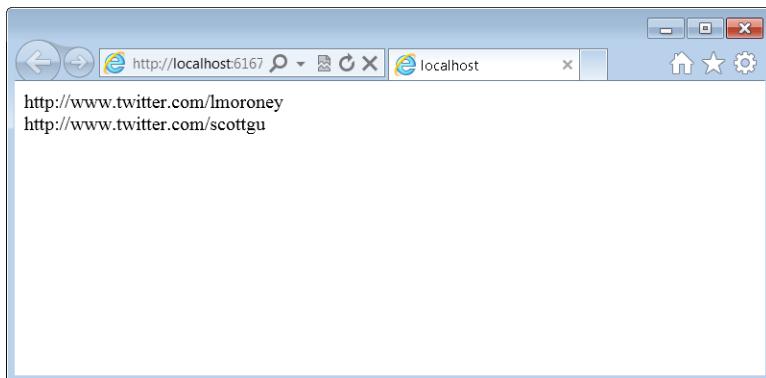


FIGURE 7-8 Getting data from a database.

This is, of course, a very simple example. In the next section, you'll build on it to create a way that your users can *enter* data into the database.

Adding Data to the Database

In the previous section, you saw how to use code to retrieve data from the database in your application and then render it back to the browser. In the exercise, you had to first enter the data by using WebMatrix. But there are plenty of scenarios in which you might want to write an application that allows users to enter data and then store their data in a database. In this section, you'll see how you can create a simple application that lets users enter Twitter names and page locations into the example database.

1. First you need to define the page that will be used for data entry. This page uses an HTML form similar to those you saw in Chapter 6, "Forms and Controls." Add a new CSHTML page to the 7-1 website and name it **input.cshtml**.
2. Edit the page to add an HTML form. The following shows the code:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <form action="" method="post">
            <p>First Name</p>
            <input type="text" name="firstname"></input>
            <br/>
            <p>Last Name</p>
            <input type="text" name="lastname"></input>
            <br/>
            <p>Twitter Address</p>
            <input type="text" name="twitter"></input>
            <br/>
            <input type="submit" name="Submit"></input>
        </form>
    </body>
</html>
```

The form provides three text boxes that users will use to enter a first name, last name, and Twitter address. The *action* attribute of the form is empty, meaning that this page will also be used to process the data that users post back when they submit the form to the server.

3. Remember that data entered into a form gets posted back to the server when a user clicks the submit button. WebMatrix makes that data available in the *Request* collection via the *Request["fieldname"]* syntax. The first thing you'll need to do is retrieve the returned data into variables. Add the following at the top of your input.cshtml page:

```
@{  
    var firstName = Request["firstname"];  
    var lastName = Request["lastname"];  
    var twitterID = Request["twitter"];  
}
```

4. Because this page handles both the HTTP *POST* and the HTTP *GET*, the code that stores the data in the database must be within an *if(IsPost)* clause. Start this clause after the last variable declaration shown above:

```
if(IsPost){}
```

5. The next thing you need to do is enable the user to add data. You add data to a database by using the *INSERT* statement in SQL. The syntax is as follows:

```
INSERT INTO tablename (field1, field2, field3,...) VALUES (value1, value2, value3, ...)
```

You can have any number of fields, but the number of fields and the number of values must match, because the database stores *value1* in *field1*, *value2* in *field2*, and so on.

Here's an example:

```
INSERT INTO TwitterNames (FirstName, LastName, TwitterID) VALUES ('Laurence',  
'Moroney', 'http://twitter.com/lmoroney')
```

Note that the values can use "placeholder" parameters rather than actual variables or values. A parameter starts with an at sign (@). This placeholder capability lets you write a statement in advance, before you know the data values you want to store. The following is an example *INSERT* statement that uses parameters:

```
INSERT INTO TwitterNames(FirstName, LastName, TwitterID) VALUES (@0, @1, @2)
```

When you execute the query, you pass it the parameters in order; the first parameter value replaces @0, the second replaces @1, the third replaces @2, and so forth.

Such statements are known as *parameterized queries*, and they're important because they give you the chance to validate the parameter values before they get stored into the database, which can help you prevent SQL injection attacks in which a hacker can try to enter SQL statements in a form rather than in valid data, and have that SQL executed by your website.

Update your code so that it includes the following lines:

```
if(IsPost){  
    var insertQ = "INSERT INTO TwitterFriends (FirstName, LastName,  
        TwitterID) VALUES (@0, @1, @2)";  
    db.Execute(insertQ, firstName, lastName, twitterID);  
}
```

6. In the preceding code, the `insertQ` query string has three parameters. The `db.Execute` call passes the user-entered `firstName`, `lastName`, and `twitterID` values as those parameters. The `Execute` method then runs the query and stores those values into the database.

Use the following complete code listing to update your page. The completed code will grab the input from the form and store it in the database using the `INSERT` query, and then redirect the user back to `index.cshtml` where the data is displayed. Note that in a real application, you should validate the values before submitting them to the database like this:

```
@{
    var db = Database.Open("7-1");
    var firstName = Request["firstname"];
    var lastName = Request["lastname"];
    var twitterID = Request["twitter"];
    if(IsPost){
        var insertQ = "INSERT INTO TwitterFriends (FirstName, LastName,
                    TwitterID) VALUES (@0, @1, @2)";
        db.Execute(insertQ, firstName, lastName, twitterID);
        Response.Redirect(@Href("~/index.cshtml"));
    }
}
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <form action="" method="post">
            <p>First Name</p>
            <input type="text" name="firstname"></input>
            <br/>
            <p>Last Name</p>
            <input type="text" name="lastname"></input>
            <br/>
            <p>Twitter Address</p>
            <input type="text" name="twitter"></input>
            <br/>
            <input type="submit" name="Submit"></input>
        </form>
    </body>
</html>
```

7. Run the page; you'll see something like Figure 7-9.

The screenshot shows a Microsoft Internet Explorer window with the URL `http://localhost:6167`. The page contains an HTML form with three text input fields and one button. The first field is labeled "First Name" and contains the value "Microsoft". The second field is labeled "Last Name" and contains the value "Web Platform". The third field is labeled "Twitter Address" and contains the value "http://twitter.com/mswebplatform". Below these fields is a blue "Submit Query" button.

FIGURE 7-9 Entering data using a form.

8. Type in the data as shown, and click the submit button. The browser will submit the data to the server, which grabs the values and stores them in the database. Finally, it redirects the browser to `index.cshtml`—which, you might remember, renders the Twitter address of everyone in the database. The newest entry will show the data you just added, as shown in Figure 7-10.

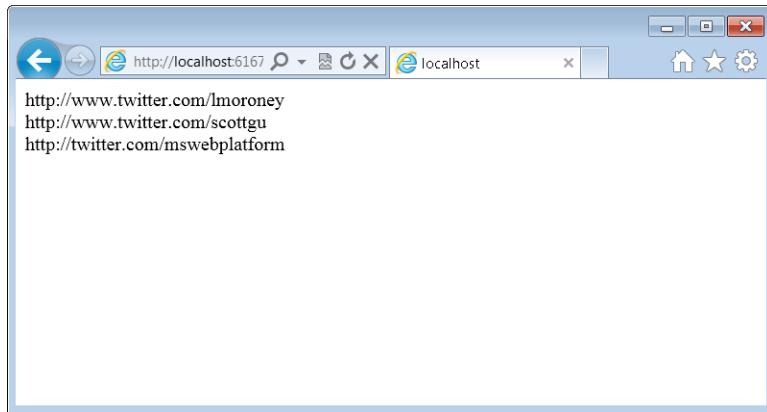


FIGURE 7-10 The new data has been added.

Now that you've seen how to add data to the database by using an HTML form, the logical next step is to see how you can let your users edit data that already exists in the database.

Editing Your Database

In the previous section, you saw how to create an HTML form that allows users to add data to your database. In this section, you'll see how to update the existing data to allow them to make edits.

To do this, you need to allow users to click an entry in the index.cshtml page so that they can edit that entry. This is accomplished by editing the page to create a hyperlink for each entry. The hyperlink will link to a page where users can edit the selected record. This edit page needs to know which record to edit, so you'll need to let it know the ID of the record. Therefore, you need to get the record ID from the database in addition to the Twitter address—which means that you'll need to edit your query.

1. Right now, the code in your input.cshtml page should look like the following:

```

@{
    var db = Database.Open ("7-1");
    var query = "SELECT TwitterID From TwitterFriends";
}
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        @foreach (var item in db.Query(query))
        {
            @item.TwitterID;
            <br/>
        }
    </body>
</html>

```

Notice that the query selects only the *TwitterID*. Update that query so that it retrieves the ID of the database row as well by changing the query to the following:

```
var query = "SELECT id, TwitterID From TwitterFriends";
```

2. Now update the HTML so that when it renders the Twitter page address for each record, it also provides a hyperlink that links to a page where users can edit that record:

```

@foreach (var item in db.Query(query))
{
    @item.TwitterID;
    <a href="edit.cshtml?id=@item.id">edit</a>
    <br/>
}

```

Because your query selected both *id* and *TwitterID*, each @item will have both an *id* and a *TwitterID* property. You can access the record ID through the *id* property. You could then create a page called edit.cshtml that takes an *id* parameter and uses that to retrieve (and update) the record with the corresponding ID. You'll see how to create that page in a moment, but to create the link to it, all you have to do is use @item.id as the parameter, as you can see in the preceding code block.

If you run index.cshtml now, it will display each record followed by an "edit" link, as shown in Figure 7-11:

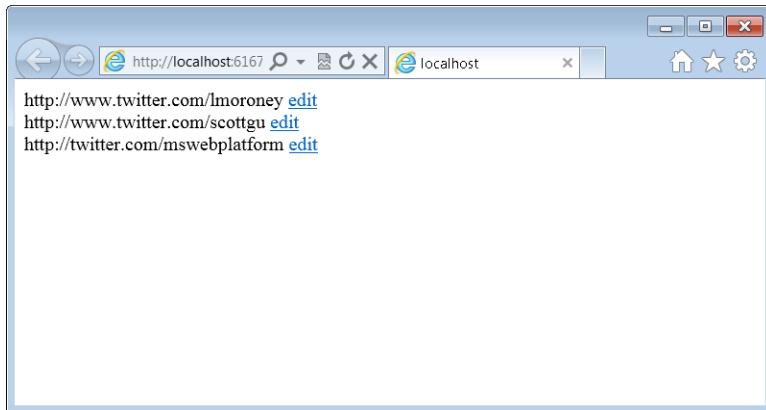


FIGURE 7-11 The list page with edit links.

If you look at the HTML behind this page, you'll see the edit links with the built-in parameters:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    http://www.twitter.com/lmoroney
    <a href="edit.cshtml?id=1">edit</a>
    <br/>
    http://www.twitter.com/scottgu
    <a href="edit.cshtml?id=2">edit</a>
    <br/>
    http://twitter.com/MingNa
    <a href="edit.cshtml?id=3">edit</a>
    <br/>
  </body>
</html>
```

3. With that code in place, here's what you need to build the edit page. First, create a new page called **edit.cshtml**.

4. Add a form to it that lets users edit the first name, last name, and Twitter address values. You can copy and paste the form code from input.cshtml if you like, because it's identical.

Your edit.cshtml page code should look like the following:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <form action="" method="post">
            <p>First Name</p>
            <input type="text" name="firstname"></input>
            <br/>
            <p>Last Name</p>
            <input type="text" name="lastname"></input>
            <br/>
            <p>Twitter Address</p>
            <input type="text" name="twitter"></input>
            <br/>
            <input type="submit" name="Submit"></input>
        </form>
    </body>
</html>
```

5. Next you need to initialize this form with data: the first and last name and Twitter address of the record the user wants to edit.

To do this, you need to query the database for the record that matches the ID that was passed in as a parameter by the hyperlink, and you need to return the results of this query into the three text boxes. You'll also add a quick check to see if the *id* parameter passed to the page is empty. If that's the case, you will just redirect back to the index.cshtml page. Here's the completed code:

```
@{
    var id = Request["id"];
    var query = "";
    var db = Database.Open("7-1.sdf");
    var firstName="";
    var lastName="";
    var twitterID="";
    if(id.IsEmpty())
    {
        Response.Redirect("index.cshtml");
    }
}
```

```
else
{
    query = "SELECT * FROM TwitterFriends WHERE id=@0";
    var item = db.QuerySingle(query, id);
    firstName = item.FirstName.Trim();
    lastName = item.LastName.Trim();
    twitterID = item.TwitterID.Trim();
}
}

<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <form action="" method="post">
            <p>First Name</p>
            <input type="text" name="firstname" value="@firstName"></input>
            <br/>
            <p>Last Name</p>
            <input type="text" name="lastname" value="@lastName"></input>
            <br/>
            <p>Twitter Address</p>
            <input type="text" name="twitter" value="@twitterID"></input>
            <br/>
            <input type="submit" name="Submit"></input>
        </form>
    </body>
</html>
```

This code retrieves the *id* from the URL, constructs a SQL query, and passes that query to a *QuerySingle* method (which works like *Execute* but returns only a single record). It extracts the *FirstName*, *LastName*, and *TwitterID* values from the returned record and uses them to initialize local variables that have the same name. It “trims” these variables to remove any trailing spaces, and then loads them into the *value* attributes of the text boxes in the form to initialize them.

6. Save this page. Now, if you browse to index.cshtml and click one of the Edit links, you’ll see something like Figure 7-12.

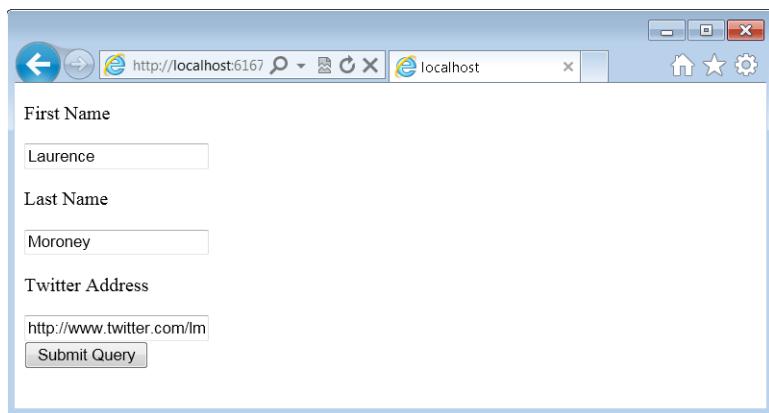


FIGURE 7-12 The Edit form.

At this point, the form has been initialized with the data from the record with the appropriate ID, but it doesn't yet change the data in the database. However, because the data is in the form, when the user makes changes and submits the form, the browser will pass the changed data back to the page using a *POST* request. You can retrieve the posted data values and use them to update the contents of the database.

7. The code to do this is very similar to the `Input.cshtml` *POST* handler you've already created. It checks to make sure that the page is in a *POST* situation; if it is a *POST*, the code pulls the data from the form, validates it, and if validation is successful, it then generates an *UPDATE* query.

An *UPDATE* query in SQL has the following syntax:

```
UPDATE table SET field1=val1, field2=val2, field3=val3 WHERE criteria
```

In this case, the *criteria* specifies that the ID field of the record must match the ID that was passed into the page for the record you want to edit, and the field names are *FirstName*, *LastName*, and *TwitterID*. The values are the text typed by the user into the text boxes.

Update your code so that it looks like this completed `edit.cshtml` page code:

```
@{  
    var id = Request["id"];  
    var query = "";  
    var db = Database.OpenFile("7-1");  
    var firstName="";  
    var lastName="";  
    var twitterID="";  
    if(id.IsEmpty())  
    {  
        Response.Redirect("index.cshtml");  
    }
```

```
else
{
    query = "SELECT * FROM TwitterFriends WHERE id=@0";
    var item = db.QuerySingle(query, id);
    firstName = item.FirstName.Trim();
    lastName = item.LastName.Trim();
    twitterID = item.TwitterID.Trim();
}
if(IsPost){
    firstName = Request["firstname"];
    lastName = Request["lastname"];
    twitterID = Request["twitter"];
    var updateQ = "UPDATE TwitterFriends SET FirstName=@0,
        LastName=@1, TwitterID=@2 Where id=@3";
    db.Execute(updateQ, firstName, lastName, twitterID, id);
    Response.Redirect(@Href("~/index.cshtml"));
}
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <form action="" method="post">
            <p>First Name</p>
            <input type="text" name="firstname" value="@firstName"></input>
            <br/>
            <p>Last Name</p>
            <input type="text" name="lastname" value="@lastName"></input>
            <br/>
            <p>Twitter Address</p>
            <input type="text" name="twitter" value="@twitterID"></input>
            <br/>
            <input type="submit" name="Submit"></input>
        </form>
    </body>
</html>
```

Now when you run the index.cshtml page and click the edit link, you'll be taken to the preloaded edit form.

8. Make a change, such as removing the *www.* part of the Twitter URL, as in Figure 7-13.

The screenshot shows a Microsoft WebMatrix browser window. The address bar displays 'http://localhost:6167'. The page content is a form for editing a record:

- First Name: Laurence
- Last Name: Moroney
- Twitter Address: <http://twitter.com/lmoroney>

A blue 'Submit Query' button is at the bottom of the form.

FIGURE 7-13 Editing a record.

9. Click the submit button. The page will save your edits and then redirect you to the index page that lists every entry. Figure 7-14 shows the index page after an edit has been made.

The screenshot shows a Microsoft WebMatrix browser window. The address bar displays 'http://localhost:6167'. The page content shows a list of Twitter addresses with 'edit' links:

- <http://twitter.com/lmoroney> [edit](#)
- <http://www.twitter.com/scottgu> [edit](#)
- <http://twitter.com/mswebplatform> [edit](#)

FIGURE 7-14 The updated list.

At this point, you have seen how to create and update records. The final task, of course, is to be able to handle deleting records. You'll see how to do that in the next section.

Deleting Records from the Database

In the previous sections, you saw how to create and update your database. To make your database experience complete, you also need to know how to delete records from your database. Fortunately, that's very simple!

1. First, just as with the editing scenario, update the index page to add a hyperlink next to each item that links to a delete page. Here's the full code for the page with the delete link added:

```
@{  
    var db = Database.Open("7-1");  
    var query = "SELECT id, TwitterID From TwitterFriends";  
}  
<!DOCTYPE html>  
<html>  
    <head>  
        <title></title>  
    </head>  
    <body>  
        @foreach (var item in db.Query(query))  
        {  
            @item.TwitterID;  
            <a href="edit.cshtml?id=@item.id">edit</a>  
            <a href="delete.cshtml?id=@item.id">delete</a>  
            <br/>  
        }  
    </body>  
</html>
```

2. Run the index page now. You'll see Delete links as shown in Figure 7-15. Clicking any of the Delete links will call the delete.cshtml page, passing it the ID of the appropriate database record.

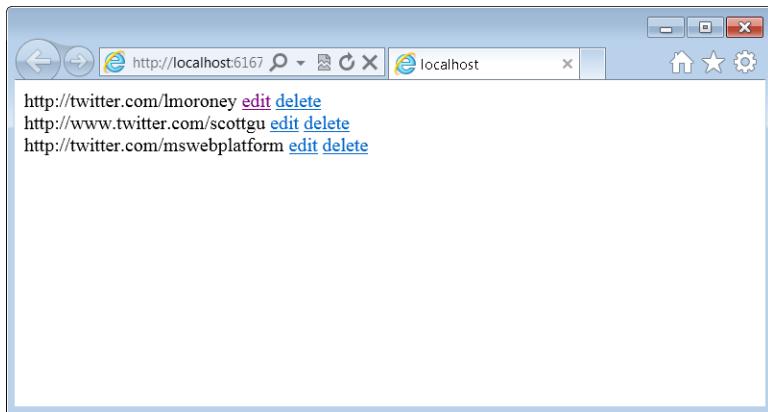


FIGURE 7-15 The Twitter Pages list with delete links added.

For example, if you point to the Delete link for the first record, you'll see that the link navigates to `http://localhost:<port>/delete.cshtml?id=1`.

Of course, the delete.cshtml page doesn't exist yet; you still need to create it.

3. Add a new page named **delete.cshtml** to your site. This time, you don't need form fields for each of the data items in the record, because you aren't editing or creating. Instead, add a very simple form that contains a check box that users can select to confirm that they want to delete this record. The following shows the code:

```
<form action="" method="post">
<p>You have chosen to delete this record. Check the box below
    if you are sure, and press submit.</p>
<p>First Name : @firstName</p>
<p>Last Name : @lastName</p>
<p>Twitter Address : @twitterID</p>
<br/>
<input type="checkbox" name="delete" value="confirm">
    I am sure I want to delete this record
</input>
<input type="submit" name="Submit"></input>
</form>
```

4. In the page code, initialize the *firstName*, *lastName*, and *twitterID* variables, which you do in exactly the same way as you did for the edit page:

```
var id = Request["id"];
var query = "";
var db = Database.Open ("7-1");
var firstName="";
var lastName="";
var twitterID="";
if(id.IsEmpty())
{
    Response.Redirect("index.cshtml");
}
else
{
    query = "SELECT * FROM TwitterFriends WHERE id=@0";
    var item = db.QuerySingle(query, id);
    firstName = item.FirstName.Trim();
    lastName = item.LastName.Trim();
    twitterID = item.TwitterID.Trim();
}
```

5. Finally, you can handle the postback from the form by checking whether the user has selected the check box; if so, you delete the record. You delete records in SQL by using the *DELETE* command, which uses the following syntax:

`DELETE FROM table WHERE criteria`

The *table* is the name of the table that you want to delete records from, and the *criteria* is a specification for a field or set of fields that identify the record or records you want to delete. In this case, you want to delete only the record with a specific ID, as follows:

`DELETE FROM TwitterFriends WHERE id=id`

But before deleting the record, you need to ensure that the check box was selected. To do that, you check the *delete* parameter in the request header (the check box was named “*delete*” in the *delete.cshtml* page code). Because the check box was given the value *confirm*, if the *delete* parameter contains *confirm*, the user has decided to delete the record and has selected the check box.

Update your code to match the complete page code, including the delete and redirect logic:

```
@{  
    var id = Request["id"];  
    var query = "";  
    var db = Database.Open("7-1 ");  
    var firstName="";  
    var lastName="";  
    var twitterID="";  
    if(id.IsEmpty())  
    {  
        Response.Redirect("index.cshtml");  
    }  
    else  
    {  
        query = "SELECT * FROM TwitterFriends WHERE id=@0";  
        var item = db.QuerySingle(query, id);  
        firstName = item.FirstName.Trim();  
        lastName = item.LastName.Trim();  
        twitterID = item.TwitterID.Trim();  
    }  
  
    if(IsPost){  
        var confirm = Request["delete"];  
        if(confirm=="confirm"){  
            var deleteQ = "DELETE FROM TwitterFriends WHERE id=@0";  
            db.Execute(deleteQ, id);  
        }  
        Response.Redirect(@Href("~/index.cshtml"));  
    }  
}  
  
<!DOCTYPE html>  
<html>  
    <head>  
        <title></title>  
    </head>  
    <body>  
        <form action="" method="post">  
            <p>You have chosen to delete this record. Check the box below if you are  
            sure, and press submit.</p>  
            <p>First Name : @firstName</p>  
            <p>Last Name : @lastName</p>  
            <p>Twitter Address : @twitterID</p>  
            <br/>  
        </form>  
    </body>  
</html>
```

```
<input type="checkbox" name="delete" value="confirm">I am sure I want to  
delete this record</input>  
<input type="submit" name="Submit"></input>  
</form>  
</body>  
</html>
```

6. Click a delete link. The page you see will look similar to Figure 7-16.

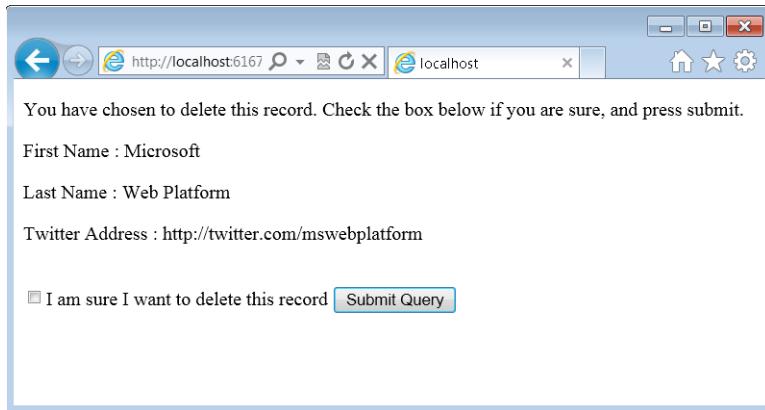


FIGURE 7-16 The delete page.

7. Select the check box and click the submit button. The record will be deleted from the database, and you are returned to the index page. You can see the index page (without the now-deleted record) in Figure 7-17.

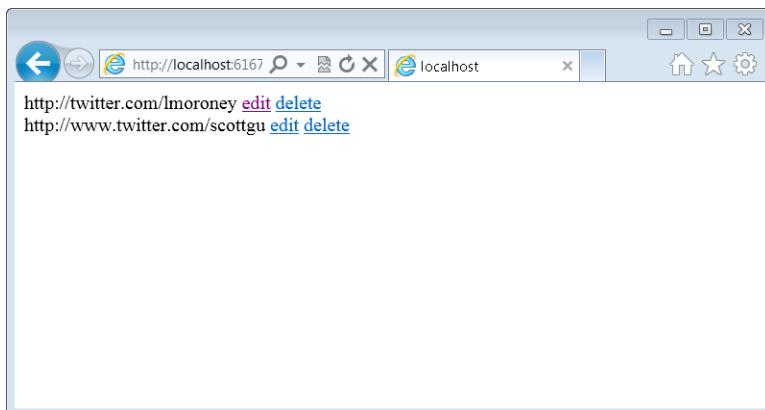


FIGURE 7-17 The deleted record no longer appears in the list of records.

At this point, you've created a database, added a page that lets users create new records in this database, added a page so users can edit existing records and update the list with edits saved to the database, and created a page that deletes a record from the table with user confirmation.

Summary

This chapter gave you the basics of working with databases in WebMatrix. You saw how WebMatrix supports the SQL Server Compact database, and how easy it is to use the integrated database editor. Using that editor, you created a new database, added a table to it, and then added some records to the table.

You also saw how to read records from the database by using the SQL SELECT statement, giving you the ability to display database contents to your users. You then saw how to update records by providing an HTML form prepopulated with a specific record. Users could edit that record and submit the results, which were then saved back to the database. Finally, you saw how to delete records from the database, again using a form, but this time for user confirmation. Together these pages give you what are called *CRUD* abilities—*Create*, *Read*, *Update*, and *Delete*—the four basic functions of any persistent storage.

Chapter 8

Exposing Your Site Through Social Networking

In this chapter, you will:

- Discover how to share your site through social sites.
- Use Twitter in your sites.
- Render Xbox Gamercards.

The Internet is an increasingly social place, and many sites are out there that make promoting your site and linking back to it very easy. Sites such as Digg and Delicious, among others, can drive a lot of traffic to your site, but you need to make it as easy as possible for people to get to you. In this chapter, you'll look into the helpers in Microsoft WebMatrix that make this a lot easier, as well as using other social sites such as Twitter and Xbox Live.

Sharing Your Site with Others

Before you start, you'll need to install the WebMatrix helpers.

1. Run your site and go to http://localhost:yourport/_admin and follow the instructions. You'll get to the NuGet feed, from which the helpers can be installed. You'll need to install the ASP.NET Web Helpers Library, which includes a helper called LinkShare that makes it easy for people to link to your page by using Digg, Reddit, Facebook, Twitter, and other sites.
2. To get started with LinkShare, create a new WebMatrix site called **8-1**. Add a new index.cshtml page and edit it so that looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    <p>My First Link</p>
    @LinkShare.GetHtml("My first link")
  </body>
</html>
```

3. Run the page. You'll see something like Figure 8-1.

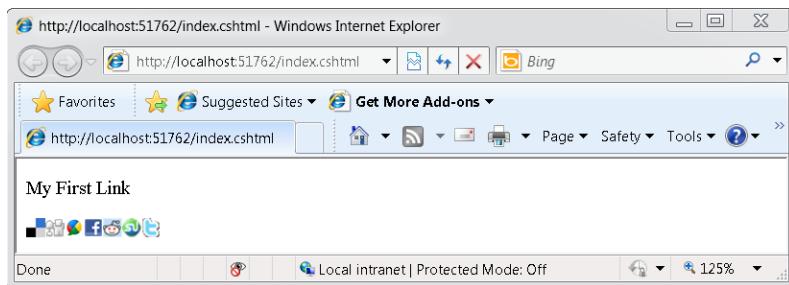


FIGURE 8-1 Using the LinkShare helper.

In the rest of this section, you'll look at how each of these social networking sites works so that you can see how your site's visitors can use them from within your site.

Using Delicious

Delicious is a social bookmarking site that allows users to save links to their favorite sites from anywhere. It offers tagging and sharing facilities to help keep those sites organized. If you, like me, have a bunch of computers, and a bunch of browsers on each, it's hard to keep all of your bookmarks organized. Delicious provides a service that does this. Naturally, you want people to bookmark your site, so you want to make it as easy as possible for them to do so.

Users can sign up for a Delicious account on their site or they can sign in with a Yahoo ID. You can see my home page on Delicious in Figure 8-2. At this point, I haven't saved any bookmarks.

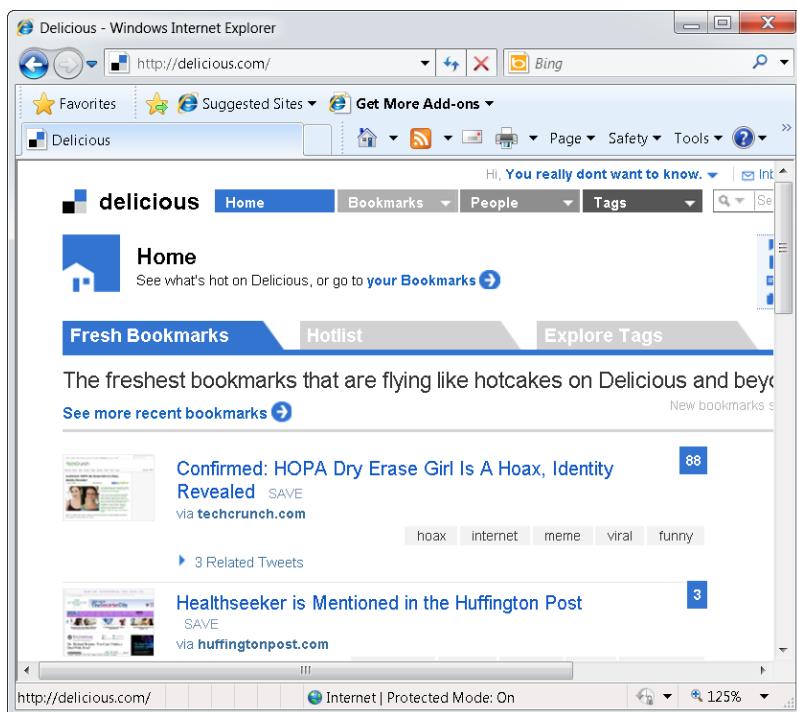


FIGURE 8-2 A Delicious home page.

When a user clicks the Delicious link on your LinkShare page, the browser will automatically launch the Delicious Save Bookmark page. You can see this in Figure 8-3.

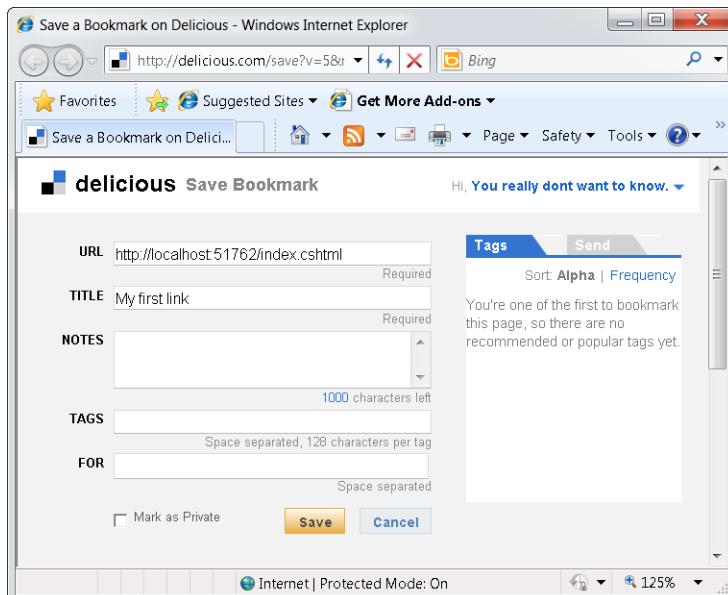


FIGURE 8-3 Adding a bookmark with Delicious.

As you can see in the image, the URL of the page is automatically populated and the title is set to the parameter that you set in your code.

Remember, when you wrote it, it looked like this:

```
@LinkShare.GetHtml("My first link")
```

As a result, the text *My first link* was loaded into the Title field. When the user clicks the Save button, the link will be added to his or her set of bookmarks on Delicious. You can see it in Figure 8-4.

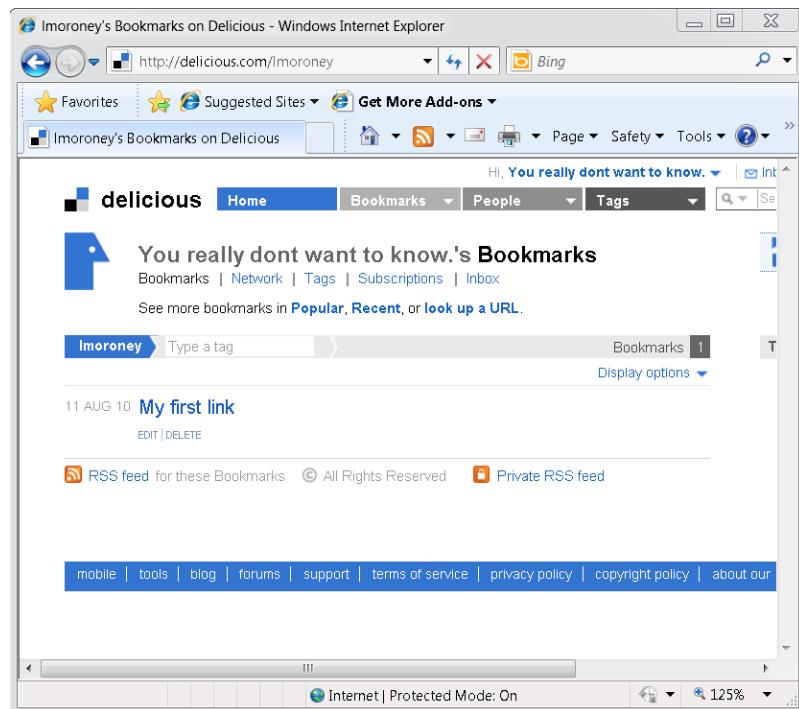


FIGURE 8-4 The bookmark has been added to Delicious.

And just as easy as that, your page has been added to the Delicious bookmarks. Users of Delicious can then share their bookmarks with each other and drive a lot of traffic to your site!

Using Digg

Digg is a social site where users share what they like with other users. Basically, if you "dig" a page (American slang for really liking it), you can tell Digg.com that you like it, and the fact that you like it will be shared with other Digg users. If a lot of people like a site and share this fact, the site can become very popular, and the snowball effect can drive a lot of traffic to the site.

To test this with a WebMatrix site, you'll need a live site. You can't share an <http://localhost> site on Digg, so, I recommend that you visit <http://www.microsoft.com/web/hosting/home/>. You'll find links to several hosting providers that you can use to host your site there. For the example site in this section, I'm using Cyttanium.

You can see my site in Figure 8-5. It's a modification of the Bakery template that is included with WebMatrix and shows how to sell something (in this case, books) using PayPal.

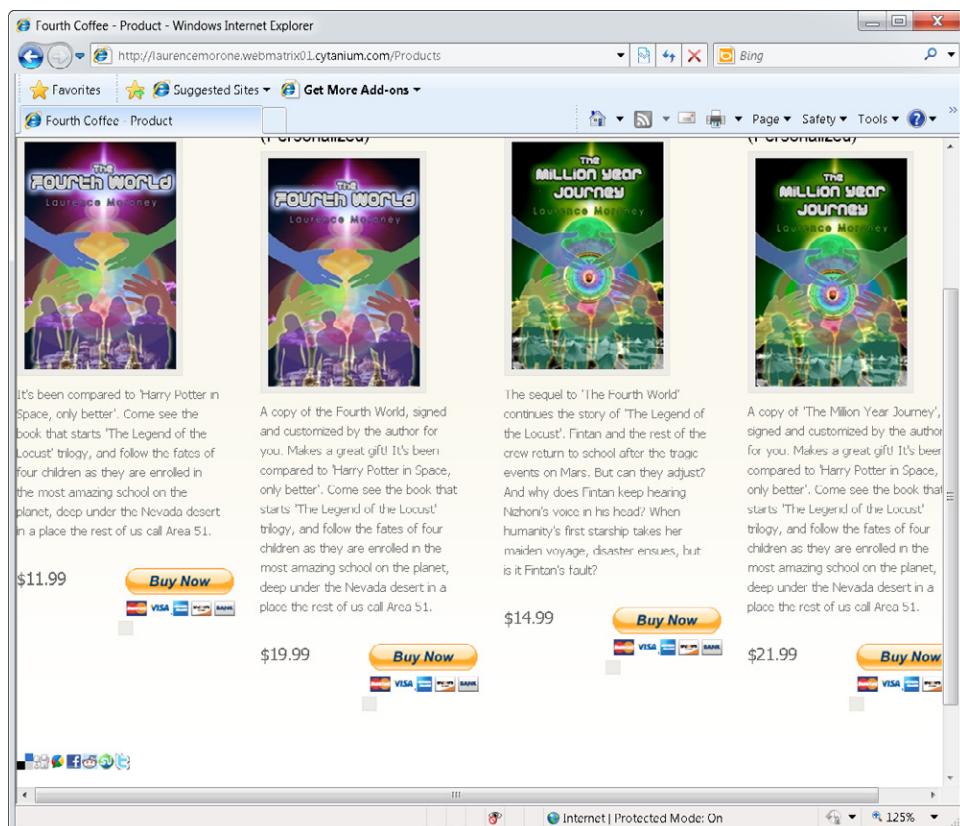


FIGURE 8-5 A live site with social links.

If the user clicks the Digg icon (second from the left), the Digg wizard will launch and allow him or her to submit your page automatically. Digg might ask if the page has a duplicate; if so, the user will end up digging the original page, which is the ultimate idea—the more digs your page gets, the higher it will appear in the Digg rankings, and the more traffic you'll receive. You want as many digs as possible!

The next stage is to describe the link. You can see this page in Figure 8-6.

The screenshot shows the 'Submit a New Link - Step 2 of 2' page in Microsoft Internet Explorer. The URL in the address bar is <http://digg.com/submit/details?key=b9614095efdd8131bae186d4f043bcb1>. The page is titled 'Submit a New Link - Step 2 of 2'. It contains three main sections: 'Describe It', 'Choose Thumbnail', and 'Choose a Topic'. The 'Describe It' section has a 'Title' field containing 'A simple online bookstore!' with 34 characters left. The 'Description' field contains the text: 'This is my page where I have edited a WebMatrix template to sell books using PayPal, and I have digged it using the LinkShare helper'. The 'Choose Thumbnail' section shows four thumbnail options: 'WORLD JOURNEY' (selected), 'POLAROID WORLD' (disabled), 'Pie Chart' (disabled), and 'No Thumbnail'. The 'Choose a Topic' section has a text input field with placeholder text: 'Tell us where to place your submission. Be as accurate as possible so that other people will be able to easily find it.'

FIGURE 8-6 Adding a site to Digg.

When you finish filling out the form, your site will be added to Digg. Figure 8-7 shows the Digg screen after my site was added.

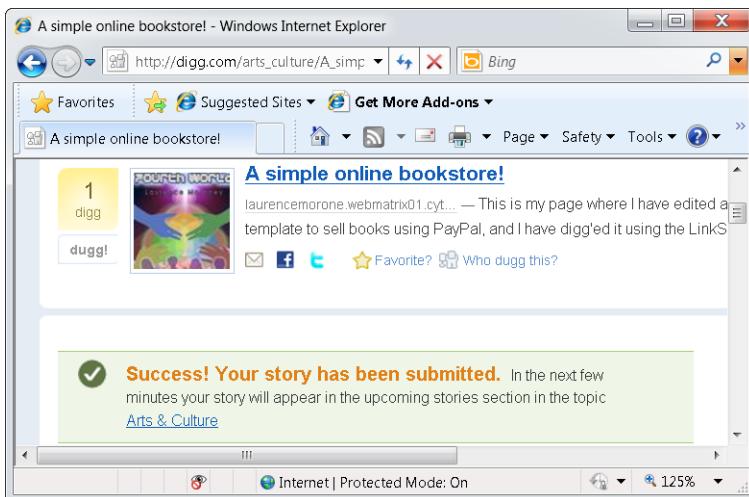


FIGURE 8-7 A site has been added to Digg.

Making it as easy as possible for someone to add your page to Digg will mean that they're more likely to do so. As you can see, the LinkShare helper made this very simple!

Using Google Reader

Similar to Delicious, Google Reader allows users to store and share bookmarks to sites. The third item in the LinkShare list allows the user to log your site on Google Reader. When your users select the item, a browser window opens, allowing them to add your site to Google Reader. You can see this in Figure 8-8.

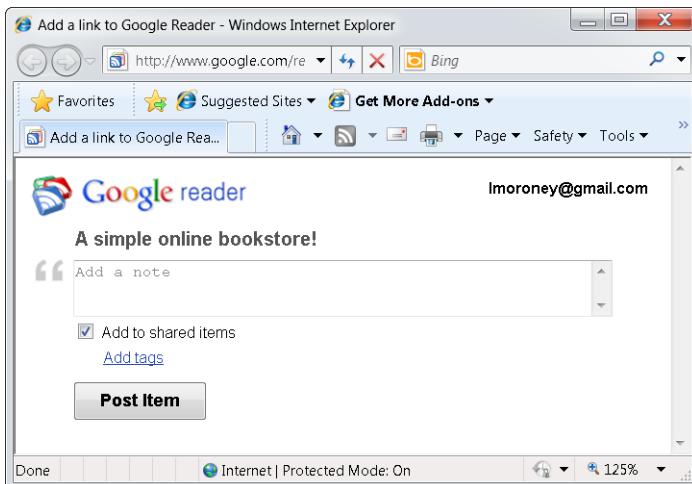


FIGURE 8-8 Adding your site to Google Reader.

This is a simple user interface that allows the user to add a note about your site and tag it. After this is done, the window closes, but the user can visit the Google Reader site at <http://www.google.com/reader>. You can see mine in Figure 8-9, where the link from the WebMatrix site has been shared.

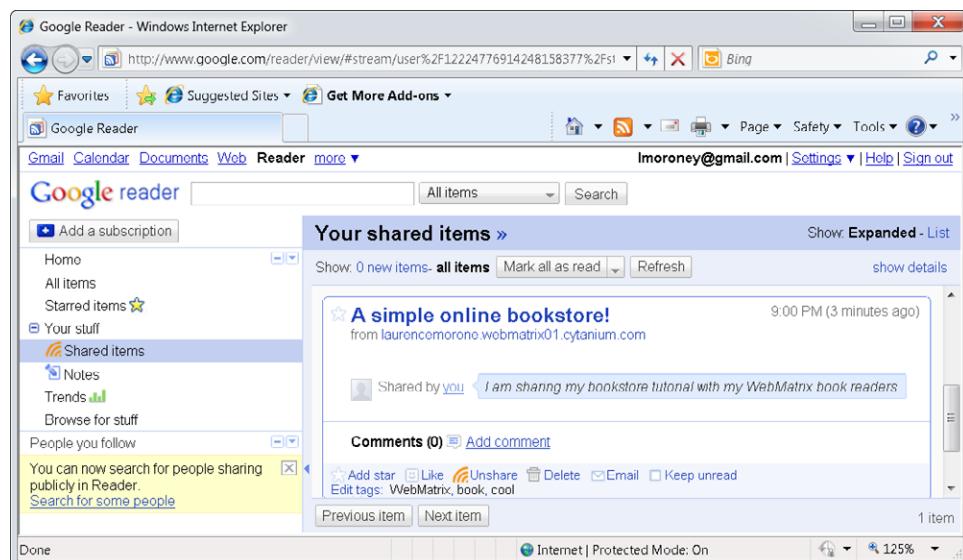


FIGURE 8-9 A shared item in Google Reader.

As you can see, it's pretty easy to allow your users to add links to their social sites. We'll continue exploring how they work by taking a look at Facebook in the next section.

Using Facebook

When your user clicks the Facebook icon on your site, his or her individual Facebook page will appear (possibly after a logon screen). This action creates a share link with your site and displays a really nice summary of your site's contents. Continuing with the site that I'm using as an example in this chapter, Figure 8-10 gives a pretty good summary. Not only that, users can add their thoughts to this summary so that it gets shared with their friends.

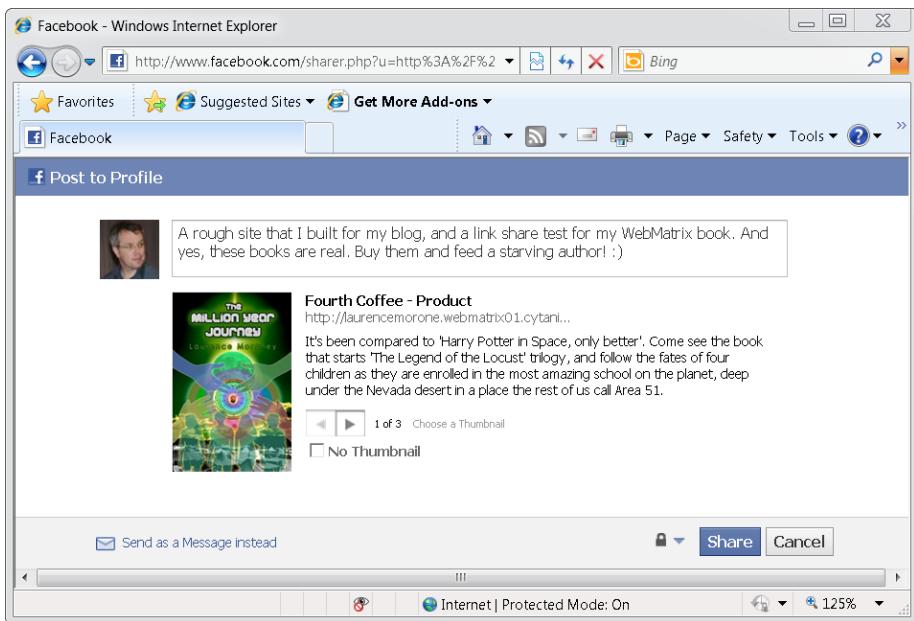


FIGURE 8-10 Sharing your site on Facebook.

Facebook can scan your page for thumbnails, so you can pick the one you want to use. In this example, I changed the thumbnail to a different one and clicked Share. On my Facebook profile, I saw what's in Figure 8-11.



FIGURE 8-11 Sharing the LinkShare content on Facebook.

As with the other sites, if you make it very easy for people to share your content on Facebook, they just might do it.

Using Reddit

Reddit is another popular site for sharing links. Again, the LinkShare helper makes it as easy as possible for your users to share your site on Reddit. When they click the Reddit link, they'll be taken directly to the Reddit submit page (see Figure 8-12).

The screenshot shows the Reddit submission form. At the top, there's a logo and a "SUBMIT" button. Below that, a message says "submit to reddit.com". There are two tabs: "link" (which is selected) and "text". A yellow box contains the instruction: "You are submitting a link. The key to a successful submission is interesting content and a descriptive title." The "title" field contains "A simple online bookstore!". The "url" field contains "http://laurencemorone.webmatrix01.cytanium.com/Products". A "suggest title" button is visible below the url field.

FIGURE 8-12 Submitting a site to Reddit.

Then, after your user has submitted your page, it will be on the Reddit list, where it can be voted up or down by the Reddit user community (see Figure 8-13).

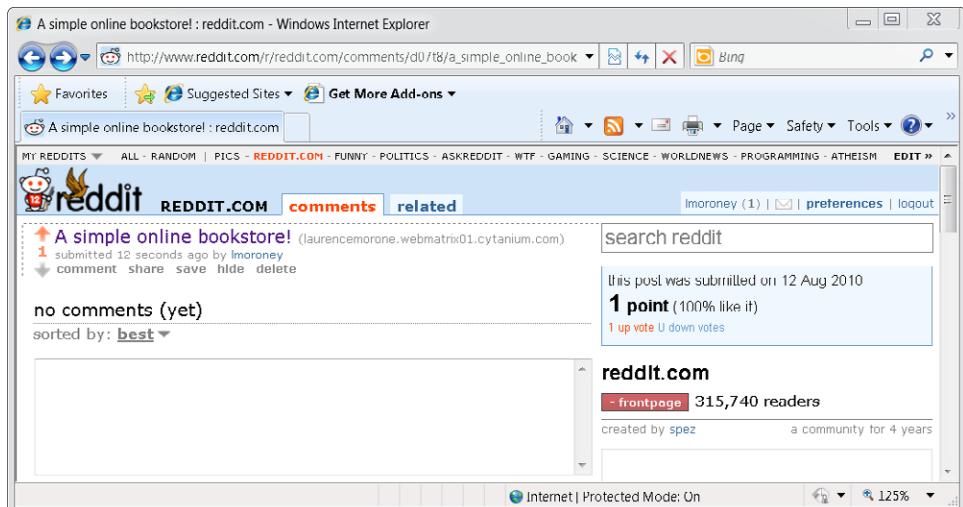


FIGURE 8-13 A site that has been added to Reddit.

Reddit users can comment on your site and help you to be successful, so be sure to post good links with good descriptions.

Using StumbleUpon

StumbleUpon is another recommendation engine like Digg or Reddit, but StumbleUpon takes into account your interests, what your friends like, and what similar users like. So, of course, you want people to be able to share your site easily with this tool, because it will raise the likelihood that their friends—and those friends' friends, and so on—will have the site recommended to them.

When users click the StumbleUpon link on your page, they'll get a browser window where they can recommend your page to the current StumbleUpon population (see Figure 8-14).

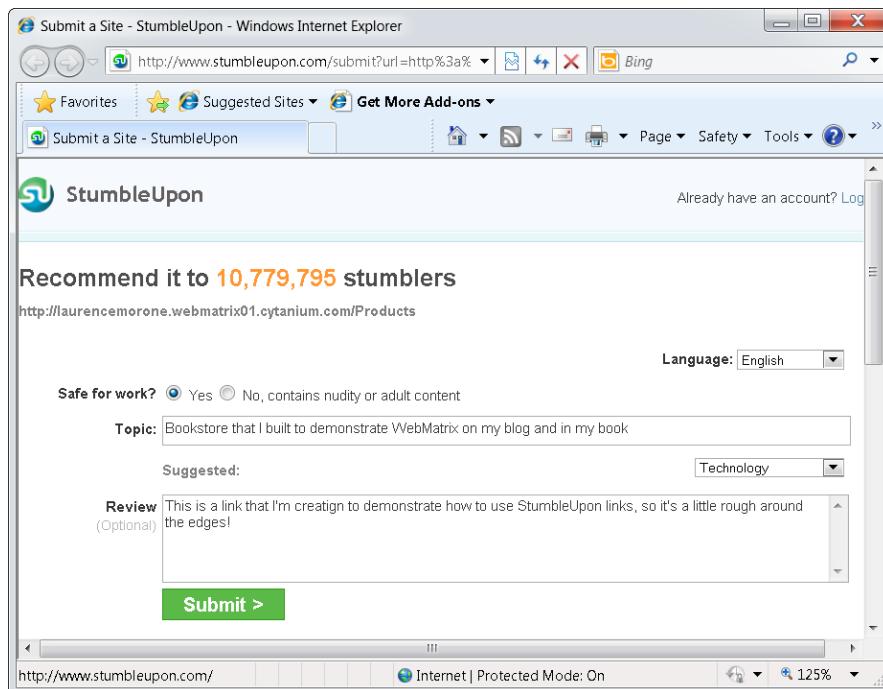


FIGURE 8-14 Adding a site to StumbleUpon.

When the user clicks Submit, the page will be submitted to the StumbleUpon engine. Then that user's friends will be more likely to have it recommended to them when they sign in! See Figure 8-15 for an example of what the StumbleUpon opening page might look like.

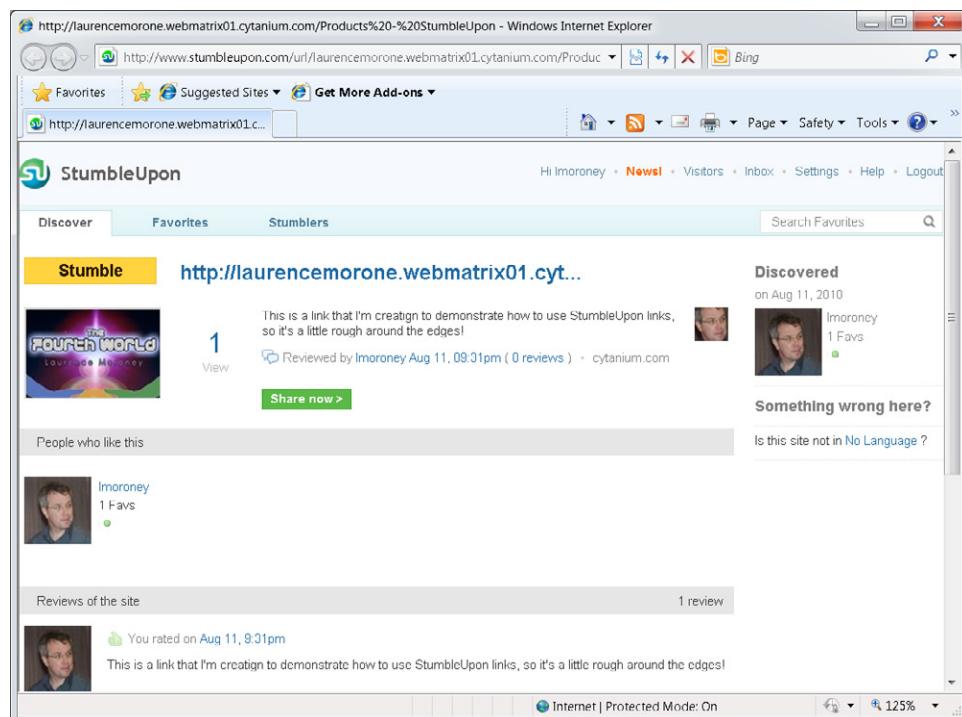


FIGURE 8-15 A site linked in StumbleUpon.

StumbleUpon also shares content with Facebook, so sites linked here can reach both populations.

Using Twitter

The last link is a link to Twitter. Again, if it's as easy as possible for your users to tweet your site, they'll be more likely to do so. Then, every one of those users' followers will see your page, giving you free marketing. If you are using the LinkShare helper, when the user clicks the icon on your site, his or her Twitter page will open, and the text you used in the LinkShare code line will be displayed, followed by the URL of your page. You can see an example in Figure 8-16.

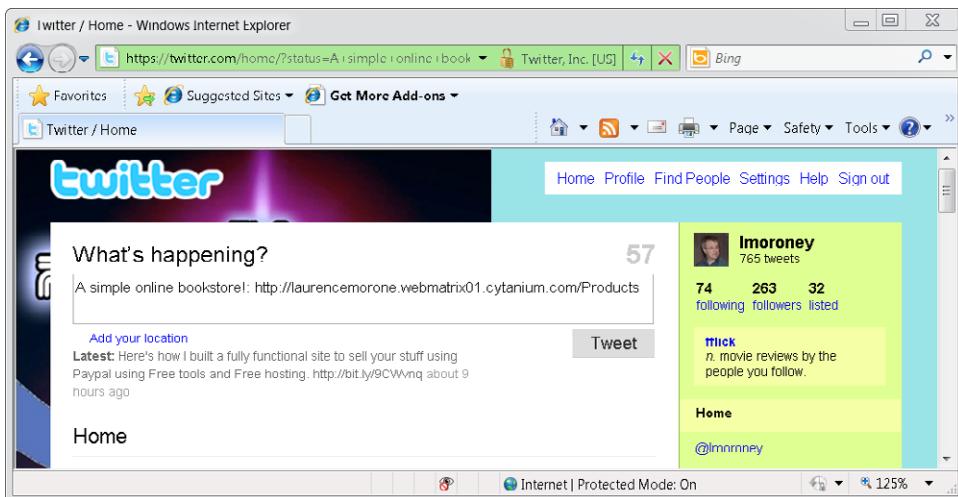


FIGURE 8-16 Sharing a site on Twitter.

All of these add up to what could be free word-of-mouth marketing on social networking sites. And all you needed to write was a single line of code!

Adding Twitter to Your Site

Now that you've seen how to add your site to the popular social networking sites, how about going the other way and adding content from them, specifically Twitter, to your site? With the Twitter web helper, you can do this very easily.

Displaying a Twitter Profile

The *Twitter.Profile* web helper allows you to display all the tweets from a particular Twitter user. For example, if you have a site, products, or just a personal account in Twitter, you can make sure that all the visitors to your site can see the tweets from that account by using the following line:

```
@Twitter.Profile(<whatever your TwitterId is>)
```

So, for example, if you want my tweets to be on your site, just use:

```
@Twitter.Profile("l moroney")
```

You can see what this looks like in Figure 8-17.



FIGURE 8-17 Using the *Twitter.Profile* helper.

Displaying Twitter Search Results

You can see what other people are saying on Twitter about a particular subject by using the *Twitter.Search* helper. This renders up to five entries at a time, the most recent first, and continues to render items with a nice scrolling effect as it adds new ones and removes the older ones. The code is very straightforward:

```
@Twitter.Search("WebMatrix")
```

You can see the results in Figure 8-18.

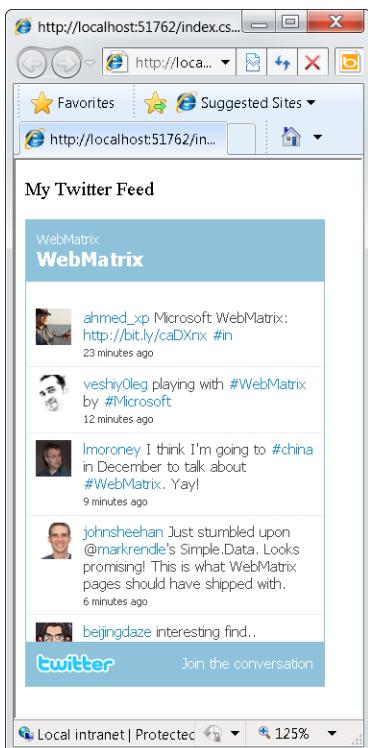


FIGURE 8-18 Using the *Twitter.Search* helper.

It might take a few moments before you see any results, while the helper searches Twitter, pulls down the results, and then starts rendering them.

Rendering Xbox Gamercards

Xbox Live is a huge and vibrant social network consisting mostly of gamers, but its population of media consumers is increasing because of applications such as Zune and Netflix.

WebMatrix supplies a helper for this too. Like the other helpers, it only takes a single line of code to add some nice functionality.

The Xbox helper looks like this:

```
@GamerCard.GetHtml("lmoroney")
```

When it's rendered, it looks like Figure 8-19.

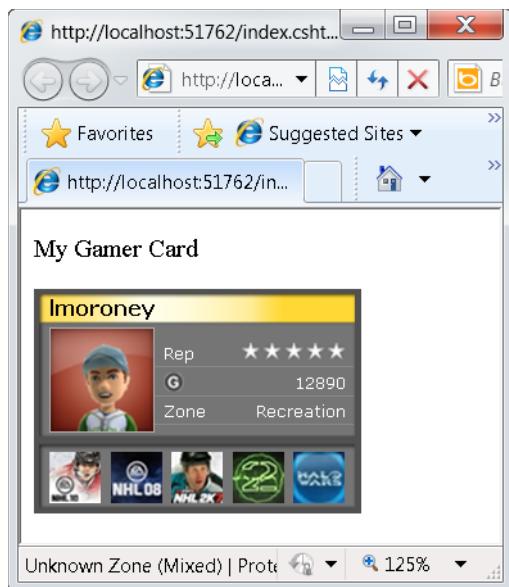


FIGURE 8-19 Rendering an Xbox Gamercard.

The Gamercard shows your reputation, your gamer points, the zone that you regularly inhabit, and the five games that you most recently played. It's clickable, so the user can see your profile and your achievements in those games. And, as you can see, the helper makes it very easy to implement!

Summary

There wasn't a lot of code in this chapter, because it wasn't necessary. The social networking web helpers in WebMatrix make it simple for you to add social functionality to your site, offering your readers a very easy way to recommend your site to others. You learned about the LinkShare helper and how it can make sharing your site via Delicious, Digg, Google Reader, Facebook, Reddit, StumbleUpon, and Twitter very easy to do.

You also saw how to add Twitter functionality to your page by embedding a specific Twitter profile or by showing the search results for a particular term on Twitter. Finally, you saw how to incorporate the Xbox Live social network into your site by using a helper to display Gamercards.

In the next chapter, you'll continue on the social theme and see how to integrate email functionality into your site so that your users can give you feedback via email messages.

Chapter 9

Adding Email to Your Site

In this chapter, you will:

- Use Simple Mail Transfer Protocol (SMTP).
- Send email messages by using the *WebMail* helper.
- Build a simple email application.

Many sites nowadays have automated email systems. You've probably encountered sites that send you automatically generated confirmation messages when you register with them. Some sites also use email to notify users when something changes on the site—if the user has chosen to receive such email notifications. Another use, of course, is when you, as the site owner, want the server to let you know when something happens on your site, such as when a new user registers or when someone posts a comment on your blog.

Microsoft WebMatrix provides a *WebMail* helper that makes building all these types of functionality straightforward. But before you start building applications that can send email messages, it's important to understand some of the terminology involved.

Using Simple Mail Transfer Protocol (SMTP)

Although email clients have become increasingly sophisticated, supporting more and more features, the basic underlying protocol for sending email messages on the Internet hasn't changed very much since 1982, when the SMTP protocol was first defined. SMTP is used for *outgoing* mail and typically uses TCP/IP port 25. There are many protocols used for *receiving* mail, including POP3 and IMAP, but this chapter just looks at *sending* mail and thus covers only the SMTP protocol—sites don't typically receive email; they only send it.

Note also that the site itself doesn't actually *send* the mail on your behalf. You don't need a mail server on your site. Instead, you use SMTP to relay your mail to a real mail server, which performs the sending operation for you.

If you have an Internet service provider (ISP), that ISP has probably provided you with the details for setting up email addresses on your domain, including the SMTP settings. If not, don't worry; you can use many free mail services, including Windows Live Hotmail, for SMTP operations. In other words, you can create an email account on Hotmail, and then your website can tell the Hotmail server to send email messages to your end users.

Thus, the underlying workflow for SMTP is to sign into your SMTP server on its assigned port (usually 25) by using secure sockets via Telnet, and to issue commands such as the following:

```
MAIL FROM: myaddress@myserver.com
RCPT-TO:youraddress@yourserver.com
DATA
From: "Me, Myself" myaddress@myserver.com
To: "You, Yourself" youraddress@yourserver.com
Date: Today
Subject: Test Message
Hello You<CR><LF>
How are you?<CR><LF>
QUIT
```

Of course, it would be a little difficult to try to do all this on your website when the user clicks a button—creating a Telnet connection, signing in, and sending all these commands. This is where the *WebMail* helper makes the process simple. You'll explore the *WebMail* helper in the next section.

Using the *WebMail* Helper

In this section, you'll explore how the *WebMail* helper can sign into your SMTP server and send a simple message. Then you'll build on this capability in the next section. If you don't have an ISP with an SMTP server, don't worry. Go to the Hotmail site and sign up for a free email account. This chapter uses a Hotmail account as an example.

The settings that you'll need for your account are:

- **SMTP server address** This is the address that your ISP gives you for your email server's outbound WebMail. For Hotmail, this is smtp.live.com.
- **SMTP port** This is the port on which the server communicates by using the SMTP protocol. The port will usually be 25 or 587. For Hotmail, it is 25.
- **SSL** This setting indicates whether the server uses Secure Sockets Layer (SSL) for communication. The answer is usually *Yes*, which indicates that you do not have to send your password across the wire in clear text. For Hotmail, this is *Yes* (or *True*).
- **User name** This is the name that you use to sign in to your mailbox. It is often, but not always, your email address. For Hotmail, it is your email address and takes the form *your_address@hotmail.com*. (Replace *your_address* with the Hotmail ID you signed up for; for example, my Hotmail user name is *Imoroney@hotmail.com*.)
- **Password** This is the password you use to sign in to your SMTP mail server.

1. Using WebMatrix, create a new empty site and name it **9-1**.
2. Create a new page called **index.cshtml**, and edit it so it contains the following code. Remember that this is for a Hotmail account. If you are using a different mailbox, you'll have to change the settings accordingly:

```
@{  
    WebMail.SmtpServer = "smtp.live.com";  
    WebMail.SmtpPort = 25;  
    WebMail.EnableSsl = true;  
    WebMail.UserName = "<your email server signin>";  
    WebMail.From = "<your email address>";  
    WebMail.Password = "<your password>";  
    WebMail.Send(to: "person-you-send-mail-to@their-mailbox.com",  
                subject: "First email sent from my Web site.",  
                body: "This is the first email sent from my Web site.");  
  
}  
<!DOCTYPE html>  
<html>  
    <head>  
        <title></title>  
    </head>  
    <body>  
  
        </body>  
</html>
```

You've used the *WebMail* helper and easily configured it to connect to your mail server and send an email!

3. Run the page. Nothing will appear to happen because you have no user interface, but, despite the fact that you're running the page locally on IIS Express, your page has just logged into your Hotmail mailbox and sent a message. If you set the *WebMail.Send to:* address to a mailbox that you control, you should see it arrive. Figure 9-1 shows the email message I received in my Microsoft Outlook Inbox after running this code with *WebMail.Send* set to my email address.

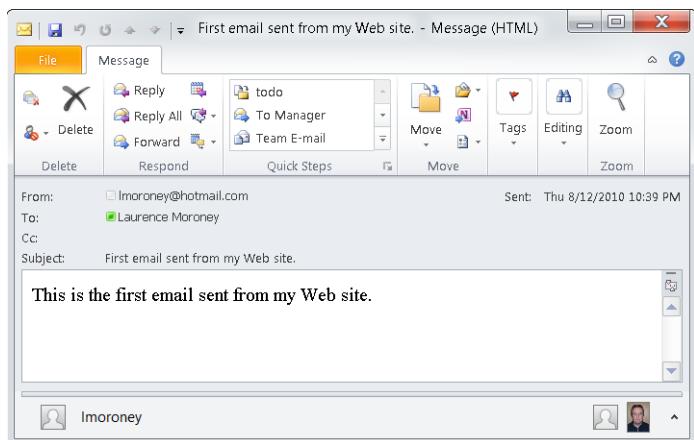


FIGURE 9-1 Receiving an email sent from WebMatrix.

Despite the fact that the message is encoded for transmission, most of the text is transmitted in plain text. You can see the email header contents if you crack open the email message. The method for doing this depends on your email client, but if you use Outlook, you can drag the message onto the desktop to make an .MSG file. If you open this file in Notepad, you'll see a lot of junk for the hexadecimal code, but you'll also see details of the mail that was sent. It's an interesting way to learn how email works.

One thing to note: When you use Hotmail to relay an email message, it isn't completely anonymous, because Hotmail embeds the IP address of the sending server into the message itself. That IP address is placed in the header as the *X-Originating-IP* property. Note that the encoding leaves spaces between each character. Here's part of the header from the previous email:

```
m c 4 - s 1 0 . b 1 u 0 . h o t m a i l . c o m      w i t h      M i c r o s o f t      S M T P S
V C ( 6 . 0 . 3 7 9 0 . 4 6 7 5 ) ;           T h u ,   1 2   A u g   2 0 1 0   2 2 : 3 9 : 0
1 - 0 7 0 0
X - O r i g i n a t i n g - I P : [ 2 x . 1 x . x . 1 x x ]
```

Warning I've obscured the true contents of *X-Originating-IP* here, but when I inspected the header, it was indeed my actual IP address as assigned by my ISP. So, if you are building a site that sends email messages to people, remember that your mail can be tracked, and that you *should not use your site to send anything that you wouldn't send from your own email address*. The messages can be traced!

Building a Simple Email Application

In this section, you'll build on the example from the previous section to create a simple page in which your user can type information into a form and have it generate an email message to you. Consider this as the basic "Send us an email message" functionality you see on many websites.

1. Use the following full listing to create a CSHTML page that provides an email form.

You'll explore this code piece by piece afterwards:

```
@{
    var eMailSent = false;
    if(IsPost)
    {
        eMailSent = true;

        var eMailSubject = Request["subject"];
        if(eMailSubject==null){
            eMailSubject = "Dummy Mail";
        }

        var eMailMessage = Request["message"];
        if(eMailMessage==null){
            eMailMessage = "Dummy Message";
        }

        var eMailAdditional = Request["fb"];
        if(eMailAdditional==null){
            eMailAdditional = "";
        }
        else{
            eMailAdditional = "By the way your site " + eMailAdditional;
        }

        eMailMessage = eMailMessage + eMailAdditional;

        WebMail.SmtpServer = "smtp.live.com";
        WebMail.SmtpPort = 25;
        WebMail.EnableSsl = true;
        WebMail.UserName = "yourusername@hotWebMail.com";
        WebMail.From ="yourusername@hotWebMail.com";
        WebMail.Password="yourpassword";
        WebMail.Send(to: "yourdestinationmail@yoursite.com",
                     subject: eMailSubject,
                     body: eMailMessage);

    }
}
```

```
<!DOCTYPE html>
<html>
    <head>
        <title>

            </title>
    </head>
    <body>
        <form method="POST" action="">
            <p>Subject</p><input type="text" name="subject" value="Feedback email from site"></input><br/>
            <p>Message</p><textarea rows="6" cols="40" name="message">Hi: I just wanted to send you some feedback!</textarea><br/>
            <p>Your site</p>
            <input type="radio" name="fb" value="Rocks" checked="yes">Rocks</input>
            <input type="radio" name="fb" value="Rules">Rules</input>
            <br/>
            <input type="submit" value="Send Email"></input>
        </form>
        @if(eMailSent){
            <p>Thanks for your submission!</p>
        }
    }

    </body>
</html>
```

2. Run this page. You'll see a form in which you can type the subject and message body of an email message (see Figure 9-2).

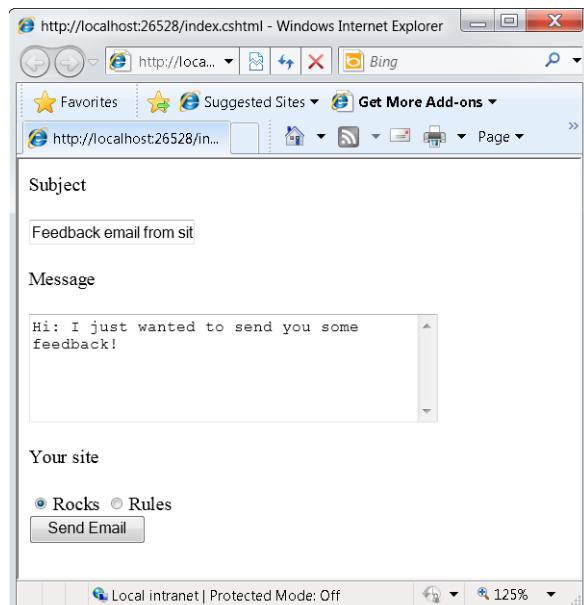


FIGURE 9-2 Your email form.

Users can change the subject and message body and select an option. Figure 9-3 shows an example of changing the option from *Rocks* to *Rules*.

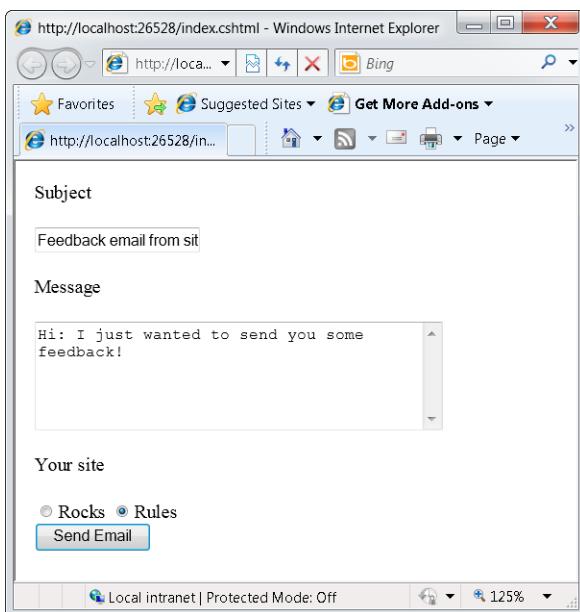


FIGURE 9-3 The email site with some details entered.

When the user clicks Send Email, the form will be submitted and the message will be sent. Figure 9-4 shows an email message in which I added some details to the message and selected the *Rules* option.

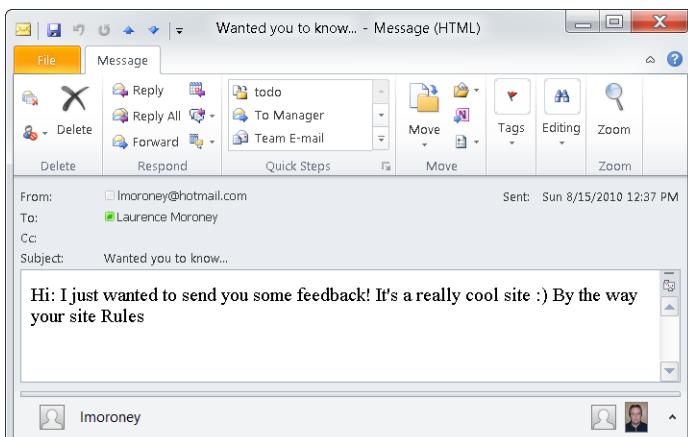


FIGURE 9-4 The mail sent from the submission form.

First of all, let's take a look at the form itself. The following code was used for the two text boxes and the option buttons that make up the email entry form:

```
<form method="POST" action="">
<p>Subject</p>
<input type="text" name="subject" value="Feedback email from site"></input>
<br/>
<p>Message</p>
<textarea rows="6" cols="40" name="message">
    Hi: I just wanted to send you some feedback!
</textarea>
<br/>
<p>Your site</p>
<input type="radio" name="fb" value="Rocks" checked="yes">Rocks</input>
<input type="radio" name="fb" value="Rules">Rules</input>
<br/>
<input type="submit" value="Send Email"></input>
</form>
```

Note the control names used in this example. The first, for the subject of the message, is naturally enough named *subject*. The body of the mail is in an HTML input control of type *textarea* and is called *message*. Finally, the option button range is called *fb*. This will have the value *Rocks* when the first option is selected and *Rules* when the second option is selected.

Now here's what happens when the server runs the form code.

The first line is outside the *if(IsPost)* block, so the server executes it every time, for both *GET* and *POST* requests:

```
var eMailSent = false;
```

This initializes the *eMailSent* variable. If the page is loading after a *GET* request, such as when you first view the page, the variable will already be *false*. Later, in the *if(IsPost)* block, the variable gets set to *true*. In the page markup, you'll see the following code, which always gets executed:

```
@{if(eMailSent){
    <p>Thanks for your submission!</p>
}
}
```

So when this variable has been set to *true* for a request, the email message is sent and the page renders the *Thanks for your submission!* text. You can see this in Figure 9-5 as a result of the *POST* operation.

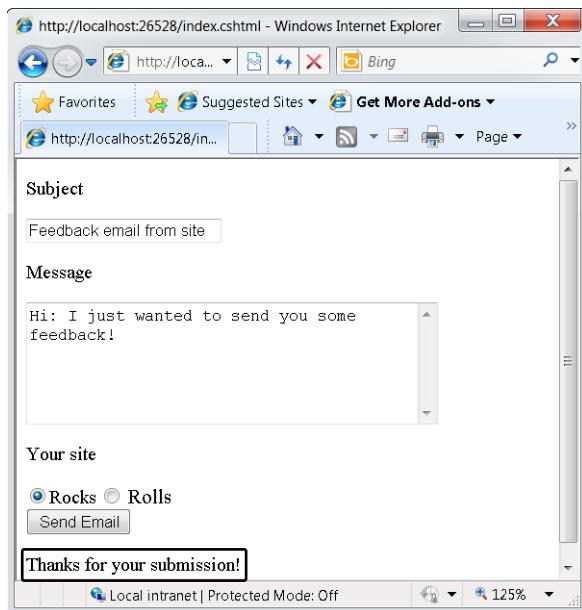


FIGURE 9-5 Receiving the notification that the message has been sent.

The rest of the *IfIsPost()* block is pretty straightforward. It simply sets up variables for *eMailSubject*, *eMailMessage* and *eMailAdditional*:

```
if(IsPost)
{
    eMailSent = true;

    var eMailSubject = Request["subject"];
    if(eMailSubject==null){
        eMailSubject = "Dummy Mail";
    }

    var eMailMessage = Request["message"];
    if(eMailMessage==null){
        eMailMessage = "Dummy Message";
    }

    var eMailAdditional = Request["fb"];
    if(eMailAdditional==null){
        eMailAdditional = "";
    }
    else{
        eMailAdditional = "By the way your site " + eMailAdditional;
    }

    eMailMessage = eMailMessage + eMailAdditional;
```

The code checks to see if each variable is null, and if so, initializes it with a default value. The idea behind the *eMailAdditional* variable is that the user will select either the Rocks or Rules option button, so an additional message of *By the way your site Rules/Rocks will be added to the email message.*

In the code, this is as simple as creating a string and concatenating the value that was passed in via the input *fb* option buttons. That passed-in value is then added to the end of the email message string.

Finally, you simply create the email message by using the *WebMail* helper, and then you send it:

```
WebMail.SmtpServer = "smtp.live.com";
WebMail.SmtpPort = 25;
WebMail.EnableSsl = true;
WebMail.UserName = "yourusername@hotWebMail.com";
WebMail.From = "yourusername@hotWebMail.com";
WebMail.Password="yourpassword";
WebMail.Send(to: "yourdestinationmail@yoursite.com",
             subject: eMailSubject,
             body: eMailMessage);
```

Notice that after the postback happens and the message gets sent, the email form is reinitialized to its default values, because the page is writing back the form HTML without the user-entered values. You could easily update it to keep the modified values. Another thing to take note of is that in this case the *action* property of the form was set to empty, meaning that this page handles the sending of the message. You might find it easier for a production application to have a dedicated “send mail” page, with a friendlier “Your mail has been sent” message, including logos and branding, and for the action of the form to point to that page.

Summary

This chapter functioned as a brief look at using the *WebMail* helper built into WebMatrix. You saw how to use the SMTP protocol to send mail before building a simple email form of your own. The *WebMail* helper is a very useful tool with which you can easily build in the functionality to let your end users contact you via your site, and as you saw from these examples, it’s very easy to implement!

Chapter 10

Building a Simple Web Application: Styles, Layout, and Templates

In this chapter, you will:

- Create and style a website.
- Add style to your site by using CSS.
- Use layout and templates.

In the previous chapters, you have been exploring some of the technologies available in Microsoft WebMatrix. Now, over the next few chapters, you'll put the knowledge you've gained into practice by building a real site. The site you'll build is a data-driven site with standard CRUD (Create, Read, Update, Delete) functionality for your users.

You'll start with the basics, by creating a page in HTML that generates a simple to-do list. This will be a static HTML page that will give you some experience in exploring the styling capabilities of cascading style sheets (CSS). You'll also see how to use the WebMatrix layout and template concepts to turn this page into a template for creating other pages. In Chapter 11, "Building a Simple Web Application: Using Data," you'll "activate" the site by making it data-driven rather than static, and then you'll use the layout and templates from this chapter to create multiple webpages for your site.

Creating and Styling Your Site

In this section, you will create a new site and style it with CSS.

1. Start with an empty site by choosing Site From Template on the WebMatrix opening screen and then selecting the Empty Site template. Name the new template **Ch10**, as shown in Figure 10-1.

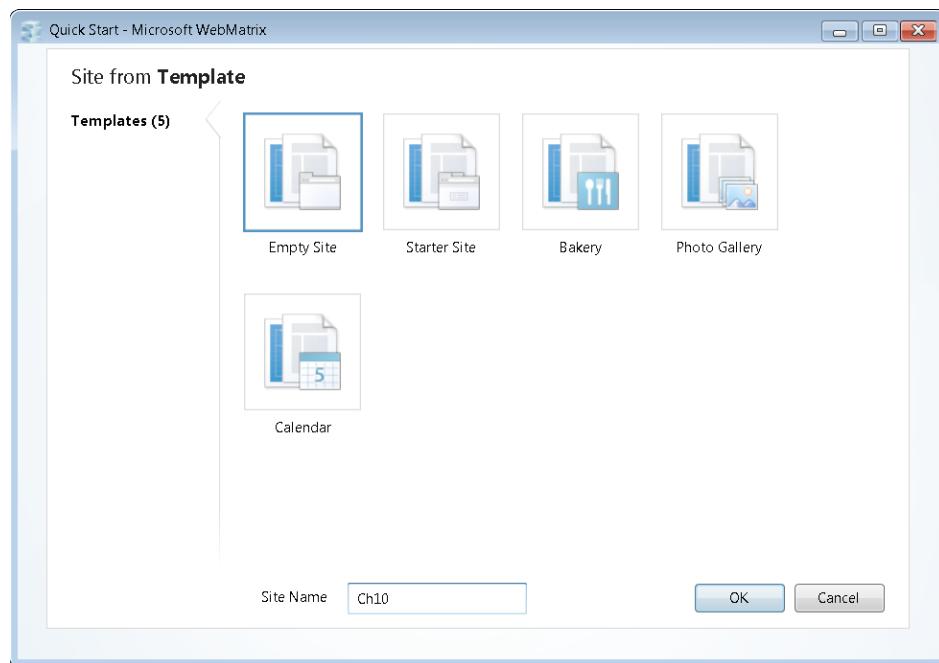


FIGURE 10-1 Creating the site.

2. Switch to the Files workspace, create a new file, and name the file **todo.cshtml**.
3. Change the contents of this file to look like the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>My Todo List</title>
</head>
<body>
    <h1>My honeydo todo list</h1>
    <ul>
        <li>Finish this chapter</li>
        <li>Mow the lawn</li>
        <li>Kids homework</li>
        <li>Change lightbulb</li>
    </ul>
</body>
</html>
```

This gives you a simple HTML unordered list (``) containing a few list items (``). When you run the site, you'll see the basic, unstyled list (see Figure 10-2).

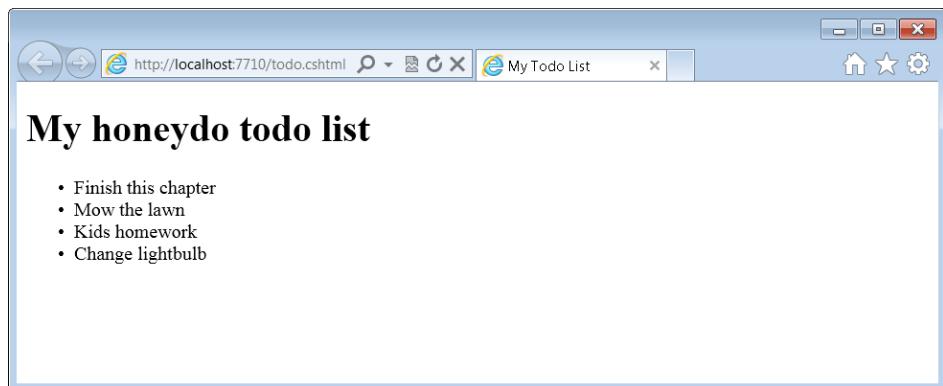


FIGURE 10-2 A simple HTML list.

In HTML, you can logically divide your page up into blocks by using the `<div>` tag. This is especially useful when you start looking at styling later in this chapter, where you can specify the style for a certain part of a page by styling its `div`.

4. The first thing to do is to wrap the list containing the to-do items into its own `<div>` as shown in the following code:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>My Todo List</title>
    </head>
    <body>
        <h1>My honeydo todo list</h1>
        <div id="todolist">
            <ul>
                <li>Finish this chapter</li>
                <li>Mow the lawn</li>
                <li>Kids homework</li>
                <li>Change lightbulb</li>
            </ul>
        </div>
    </body>
</html>
```

You can see from the preceding code that the `` list containing the items is now contained within the `<div>` tag. If you view the page now, it will look no different than it did earlier, because the `<div>` tag is a logical divider. It doesn't have any physical appearance.

Getting Your Page Ready for CSS

You're already familiar with those clickable areas on a page that link to another page. Although the term for these is *hyperlinks*, in HTML they were originally called *anchors*, so whenever you want to create one in HTML you use the anchor (*<a>*) tag.

The *<a>* tag works by making the content between the opening *<a>* tag and its matching closing ** tag clickable. When the user clicks the enclosed content, the browser will be redirected to an HREF (hypertext reference) indicated by the *href* attribute within the *<a>* tag.



Note An *attribute* is defined inside the opening tag itself, instead of with the content between the opening and closing tags, like this:

```
<tag attribute="attributevalue">content</tag>
```

Thus, to create a hyperlink, you use syntax like this:

```
<a href="http://www.philotic.com">Click Here</a>
```

The *href* doesn't have to be a website, as in the example in the preceding Note; it can also be something such as a JavaScript function that executes some code on the client side—that is, within the browser. There's a special *href* that can be used as a placeholder while you're developing your site, so that you can test that your hyperlink styles are working. To do this, use the number sign (#) character as your *href*. We'll use this *href* in the following exercise.

1. To turn all the *</i>* items containing the to-do items into hyperlinks, simply wrap the text of the item in an *<a>* tag and set the *href* attribute to #, as shown in the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>My Todo List</title>
</head>
<body>
    <h1>My honeydo todo list</h1>
    <div id="todolist">
        <ul>
            <li><a href="#">Finish this chapter</a></li>
            <li><a href="#">Mow the lawn</a></li>
            <li><a href="#">Kids homework</a></li>
            <li><a href="#">Change lightbulb</a></li>
        </ul>
    </div>
</body>
</html>
```

2. Run the page. You'll see that the elements on your list use a familiar style for hyperlinks: namely, a blue underline (see Figure 10-3).

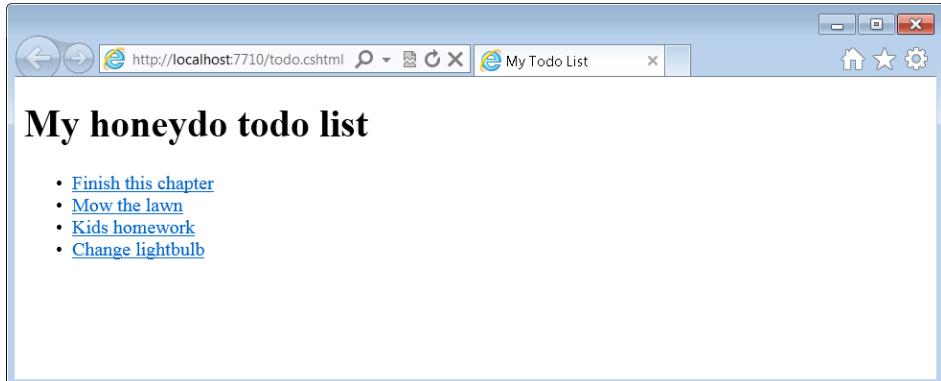


FIGURE 10-3 Adding hyperlinks.

3. The next thing to do is add a header and footer to the page. Add those by using the new `<header>` and `<footer>` tags available in HTML5, as shown in the following code:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>My Todo List</title>
    </head>
    <body>
        <header><h1>My honeydo todo list</h1></header>
        <div id="todolist">
            <ul>
                <li><a href="#">Finish this chapter</a></li>
                <li><a href="#">Mow the lawn</a></li>
                <li><a href="#">Kids homework</a></li>
                <li><a href="#">Change lightbulb</a></li>
            </ul>
        </div>
        <footer>Built by <a href="http://www.philotic.com">me</a></footer>
    </body>
</html>
```

As you can see, these are simply straightforward pieces of HTML. For the header, you just wrapped the `<h1>` tag that you created earlier in a `<header>` tag. For the footer, you just needed to create a little text and a hyperlink.

When you look at the page within the browser, it will now look something like that shown in Figure 10-4.

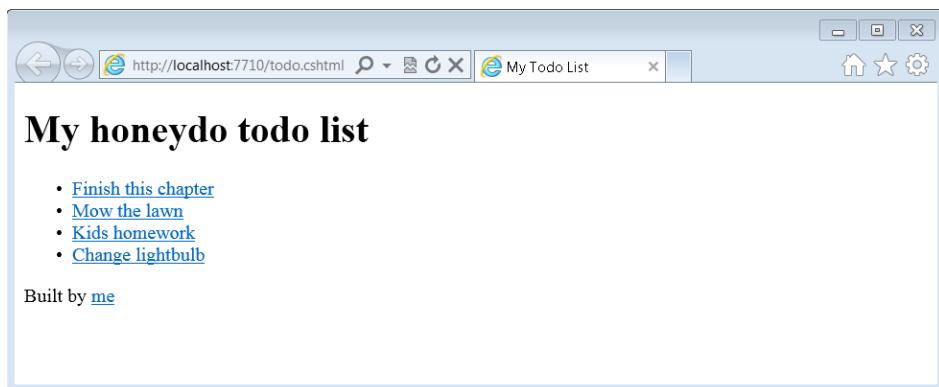


FIGURE 10-4 The site with the header and footer added.

Other than the footer, it's not much different, but don't worry, that will soon change!

Adding Some Style with CSS

In the previous section, you created an HTML page and tweaked its content a little to add headers and footers. You also changed the list items in a `` list to make them hyperlinks. All that was done both to make styling the list a little easier and to make it prettier than the default HTML. For example, because you made the list items into hyperlinks, the browser will provide rollover functionality, and you can style that functionality to improve the list's appearance as the users roll over items. In this section, you'll take a quick tour of CSS to see how you can use this technology on your to-do list.

As you might expect, you can use attributes to specify how any page element looks, changing characteristics such as font style, font size, colors, borders, and much more.

For example, you could change the font and color of the `<h1>` element defined on the example page as follows:

```
<h1 style="color:blue; font-size:32; font-family:Verdana;  
text-decoration:underline">My honeydo todo list</h1>
```

As you can see, the `style` attribute of the `<h1>` tag contains a list of properties that control the rendering style. The preceding markup sets the color to *blue*, the font size to 32, the font family to *Verdana*, and the text decoration to *underline*, giving you a result like what is shown in Figure 10-5.

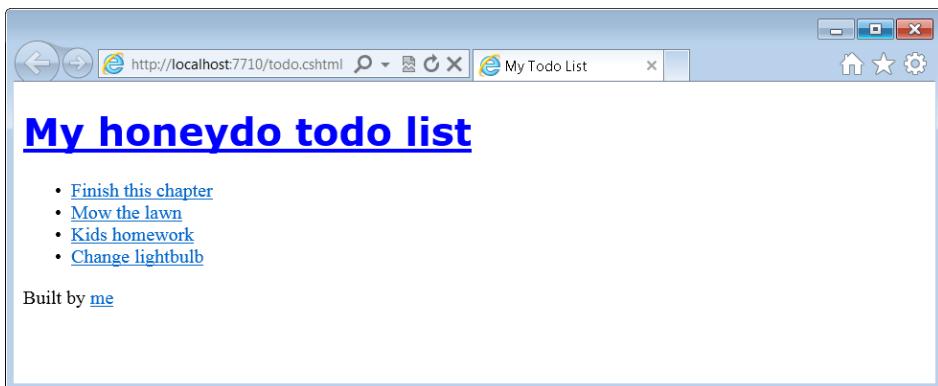


FIGURE 10-5 Changing the style of the title.

Although using attributes directly within tags for styling (a technique called *inline markup*) works just fine, it isn't the best way to style the page. Consider what would happen if you had to style *every* element that way. You'd end up with a lot of styling text on your pages, which would slow down both the download and the browser.

Fortunately, there is another way, and that is to use a style sheet on your page. Style sheets in HTML are called *cascading style sheets* (CSS), in which a style that is set on an element can be inherited by a child of the element. So, for example, if you put a style on a `<div>` and that `<div>` has child `` and `` elements, then that style will also apply to them—unless you override that inherited style with a more specific style.

With that in mind, here's what it takes to define the style on the `<h1>` tag in this chapter's example, without using a lot of inline markup for the `style` attribute.

1. First, rather than putting all the styling markup into the `<h1>` tag itself, just specify its `class` attribute, as shown in the following code:

```
<h1 class="Title">My honeydo todo list</h1>
```

2. Now that the tag has a class, you can tell the browser to use a specific style for *everything* that has this class. In CSS code syntax, you do that using the following:

```
.Title {  
    font-size: xx-large;  
    font-weight: normal;  
    padding: 0px;  
    margin: 0px;  
}
```

The CSS style "language" consists of a list of properties separated by semicolons and contained within curly braces (`{...}`). If you want the style thus defined to apply to a *class*, you define the class name by using the "dot" syntax, which is nothing more than the class name preceded by a dot.

Place this code within a `<style>` tag in the *header* of the page. The complete page markup should now look like the following:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>My Todo List</title>
        <style type="text/css">
            .Title {
                font-size: xx-large;
                font-weight: normal;
                padding: 0px;
                margin: 0px;
            }
        </style>
    </head>
    <body>
        <header><h1 class="title">My honeydo todo list</h1></header>
        <div id="todolist">
            <ul>
                <li><a href="#">Finish this chapter</a></li>
                <li><a href="#">Mow the lawn</a></li>
                <li><a href="#">Kids homework</a></li>
                <li><a href="#">Change lightbulb</a></li>
            </ul>
        </div>
        <footer>Built by <a href="http://www.mysite.com">me</a></footer>
    </body>
</html>
```

3. Run the page. The style will take effect, and you'll see something like Figure 10-6.

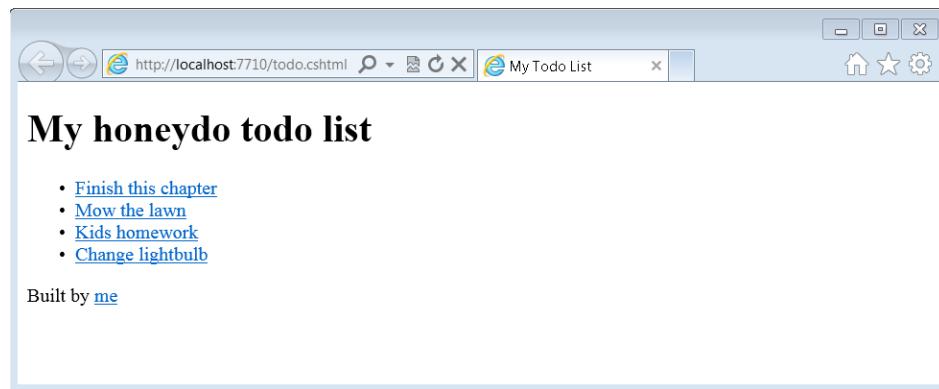


FIGURE 10-6 Using the `style` tag.

Remember that the `<h1>` had a *class* of *Title*, so by setting the *.Title*, you can set the style for every element that has the same class.

4. When you want to set the style for a specific element, you can either use a class for that element, knowing that there's only one instance of that class, or you can name the element by using an ID, and then set the class for that ID. Take a look at your HTML; you'll notice that the list of items is contained within a `<div>` that has been given the ID `tololist`. You can set the style for this ID by preceding its name with a number sign (#) in your style sheet definition. Add the following lines to the style sheet definition, like this:

```
#tololist{  
    font-family: Geneva, Tahoma, sans-serif;  
}
```

This defines a style for the `tololist` `<div>`, and because style sheets *cascade*, any element within this `div` will *also* have this style applied to it. So even though you haven't explicitly set a style for the `` elements containing the text, the `tololist` style will still be applied.

5. Remember that the browser defaults to rendering `<i>` objects in a `` list as bulleted items. You can override that by setting the style. Because these objects are inside the `div` named `tololist`, you can address them easily to change their style.

Update your style sheet definition so that it looks like the following:

```
#tololist{  
    font-family: Geneva, Tahoma, sans-serif;  
}  
#tololist ul{  
    list-style: none;  
    margin: 0;  
    padding: 0;  
    border: none;  
}
```

This simply states that for each `` within `#tololist`, set the list style to be *none* (no bullets), no margin (0), no padding (0), and no border (*none*).

The result, as you can see in Figure 10-7, is that the bullets are now gone.

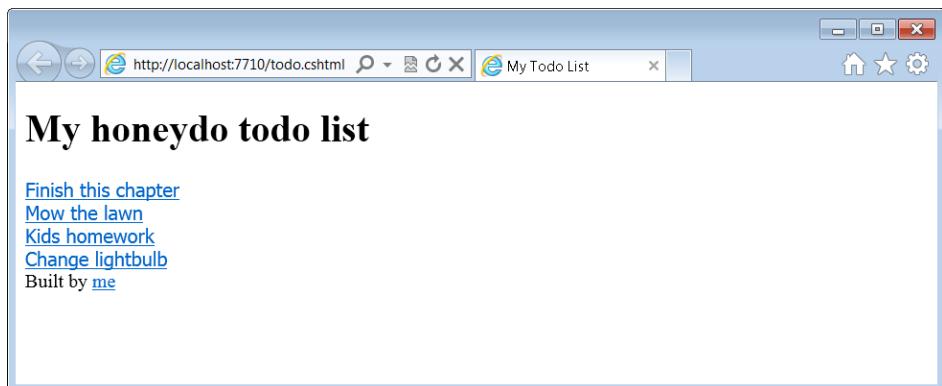


FIGURE 10-7 Editing the list style.

6. The text of each list item was held within an `<a>` tag, so you can also use CSS to define the appearance of every `<a>` tag within every `<i>` tag within `tololist`. To do that now, add this:

```
#tololist li a {  
    font-size: medium;  
    color: #002222;  
    display: block;  
    padding: 5px;  
}
```

The settings here pretty much speak for themselves. Figure 10-8 shows what the page looks like when you run it now.

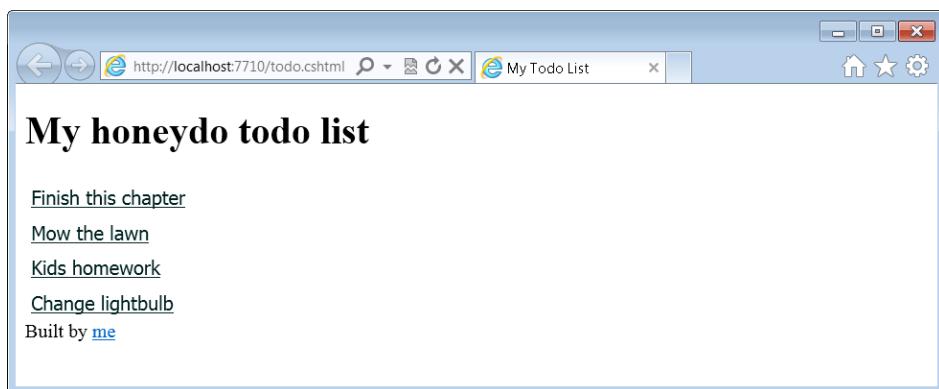


FIGURE 10-8 Editing the list items.

7. The `<a>` tag provides another behavior when users point to the link with the mouse. You can use this feature to “hot track” the mouse and change the style of the element when the mouse pointer is over it. CSS also supports this, using the following syntax:

```
#tololist li a:hover{}
```

Define what to do when the mouse pointer is over an anchor tag by adding the following code:

```
#tololist li a:hover {  
    border-left: 10px solid #0000d4;  
    padding-left: 10px;  
    background-color: #DDDDDD;  
    text-decoration: none;  
}
```

This CSS style gives the text a 10-pixel border on the left and a background color to the item so that it appears highlighted.

Figure 10-9 shows how this looks when a user points to one of the options in the list.

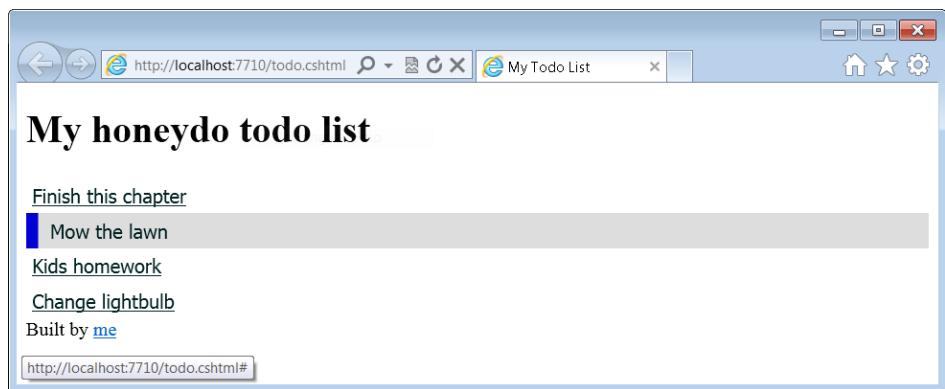


FIGURE 10-9 The mouse-pointing highlight.

The following is the complete listing for your page up to this point:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>My Todo List</title>
    <style type="text/css">
        .Title {
            font-size: xx-large;
            font-weight: normal;
            padding: 0px;
            margin: 0px;
        }
        #todolist {
            font-family: Geneva, Tahoma, sans-serif;
        }
        #todolist ul {
            list-style: none;
            margin: 0;
            padding: 0;
            border: none;
        }
        #todolist li a {
            font-size: medium;
            color: #002222;
            display: block;
            padding: 5px;
        }
        #todolist li a:hover {
            border-left: 10px solid #0000d4;
            padding-left: 10px;
            background-color: #DDDDDD;
            text-decoration: none;
        }
    </style>
</head>
```

```
<body>
<header><h1 class="title">My honeydo todo list</h1></header>
<div id="todolist">
    <ul>
        <li><a href="#">Finish this chapter</a></li>
        <li><a href="#">Mow the lawn</a></li>
        <li><a href="#">Kids homework</a></li>
        <li><a href="#">Change lightbulb</a></li>
    </ul>
</div>
<footer>Built by <a href="http://www.mysite.com">me</a></footer>
</body>
</html>
```

Using CSS Files

It's fun to experiment with CSS—and WebMatrix makes it easy. As you've been working through these exercises, you probably thought, "Wait a minute—this CSS stuff is all very nice, but what if my site has more than one page?"

That's a great question. In our example, the CSS has been integrated into the page `<head>` by using a `<style>` block. The good news is that a webpage doesn't have to use a `<style>` block to use CSS—it can point to an external CSS file by using a `<link>` tag instead. That way, any page that points to that file will be able to take advantage of the same styles.

It's easy to do this with WebMatrix.

1. With the Files workspace open, click the New button and select New File. The New Files dialog box opens.
2. Select CSS as the file type, and give the file the name **todo.css**. Click OK, and WebMatrix will create the CSS file for you.
3. The file will contain an empty `<body>` tag, like the following:

```
body {  
}
```

Replace this with the following CSS. I've tidied up some of the CSS that you created as you worked along through this chapter, and I have created specific CSS styles for the header instead of using the `class` attribute on the `<h1>` tag, which produces cleaner HTML on your page, as you'll see in a moment:

```
body {  
    font-family: Tahoma, Verdana, Geneva, sans-serif;  
    width: 85%;  
    margin: 20px auto;  
}  
header {  
    padding: 10px;  
    border-bottom: 1px solid #dddddd;  
}  
  
header h1 {  
    font-size: xx-large;  
    font-weight: normal;  
    padding: 0px;  
    margin: 0px;  
}  
#todolist{  
    font-family: Geneva, Tahoma, sans-serif;  
}  
#todolist ul{  
    list-style: none;  
    margin: 0;  
    padding: 0;  
    border: none;  
}  
#todolist li a {  
    font-size: medium;  
    color: #002222;  
    display: block;  
    padding: 5px;  
}  
#todolist li a:hover {  
    border-left: 10px solid #0000d4;  
    padding-left: 10px;  
    background-color: #DDDDDD;  
    text-decoration: none;  
}
```

4. Because this CSS now explicitly sets the styles for both the `<header>` and the `<h1>` tag within the header, you can now simplify your header code to look like this, by removing the `class` attribute:

```
<header><h1>My honeydo todo list</h1></header>
```



Note The `<header>` tag is an HTML5 construct and therefore will work only in browsers that support HTML5.

5. To use the styles defined in `todo.css` instead of the code on your page, delete the `<style>` block in your `todo.cshtml` page and replace it with the following code:

```
<link rel="stylesheet" type="text/css" href="todo.css" />
```

The `<link>` tag tells the browser to load the referenced style sheet instead of getting the styles from a style block directly within the page. Your page markup should now look something like the following, which I'm sure you'll agree is much cleaner:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <link rel="stylesheet" type="text/css" href="todo.css" />
        <meta charset="utf-8" />
        <title>My Todo List</title>
    </head>
    <body>
        <header><h1>My honeydo todo list</h1></header>
        <div id="todolist">
            <ul>
                <li><a href="#">Finish this chapter</a></li>
                <li><a href="#">Mow the lawn</a></li>
                <li><a href="#">Kids homework</a></li>
                <li><a href="#">Change lightbulb</a></li>
            </ul>
        </div>
        <footer>Built by <a href="http://www.mysite.com">me</a></footer>
    </body>
</html>
```



Important I'll repeat this once more: note that this code is HTML5, so you'll need a browser that supports HTML5, such as Windows Internet Explorer 9, to render it properly.

- When you run this, you'll see that the footer looks a little untidy. Next, you'll add some styles to the CSS file so the footer looks a little nicer. Add the following code to your todo.css file:

```
footer {
    font-size: smaller;
    color: #dddddd;
    text-align: center;
    padding: 10px 10px 10px 10px;
    border-top: 1px solid #232323;
}
```

This CSS centers the footer text and assigns it a small font in a gray color, as well as a border that separates it from the rest of the list. You can see the results in Figure 10-10.

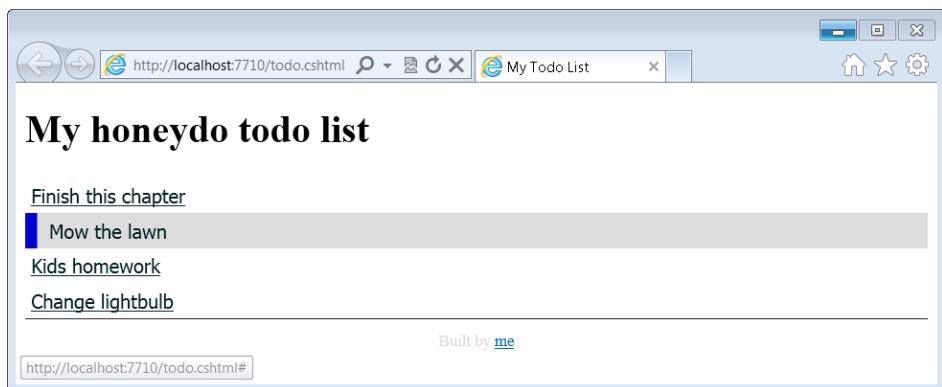


FIGURE 10-10 The fully styled list.

At this point, you have a basic static HTML page that has been styled as you would like the final site to look. Now I'll switch gears and show you how you can turn this page into a template for all the pages that the to-do list website uses, so that you don't have to write out the styles and content for every page.

Using Layout Pages and Templates

Imagine for a moment that your site has several pages like the page that you have right now, which lists your to-do items, but that you would like to have other pages for creating an item, deleting an item, or editing an item on your to-do list. Each of these pages would use the same header ("My honeydo to-do list") and footer ("Built by me") as the page you've just created; only the part of the page that's between the header and footer will be different.

In this section, you'll see how to create a *layout page* that contains the header and footer, and you'll learn how to edit your existing page to get it to use that layout page as its template.

Using *RenderBody()*

The key to the Razor view engine is the *RenderBody()* command. With this command, you can tell a template page to render the contents of your desired page at a specific location in the template. Don't worry if this is a little confusing! It's best understood by example.

Look at your existing todo.cshtml page. The only *distinctive* content in the page is the *div* containing the list:

```
<div id="todolist">
  <ul>
    <li><a href="#">Finish this chapter</a></li>
    <li><a href="#">Mow the lawn</a></li>
    <li><a href="#">Kids homework</a></li>
    <li><a href="#">Change lightbulb</a></li>
  </ul>
</div>
```

The rest of the markup and code is simply HTML “plumbing” plus the header and footer. This illustrates an important concept: When you want to create a template, it’s best to make the template do the “plumbing,” and have it render any distinctive content at a particular location. In other words, a template that renders the to-do list would contain the HTML header, the page header, and the footer within it, and would use the *RenderBody()* command to display the page body (the *<div>* containing the to-do list, in this case). The following shows what such a template would look like:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" type="text/css" href="todo.css" />
    <meta charset="utf-8" />
    <title>My Todo List</title>
  </head>
  <body>
    <header><h1>My honeydo todo list</h1></header>
    @RenderBody()
    <footer>Built by <a href="http://www.mysite.com">me</a></footer>
  </body>
</html>
```

Note that everything you had on your todo.cshtml page is here *except* the *<div>* that contains the distinctive content. In place of that content, you now have a *RenderBody* command that, when the template is complete, will render the distinctive content for you.

1. Create a new file named **_sitelayout.cshtml** and copy the preceding code into it.
2. Then go back to todo.cshtml and delete everything except the *<div>* containing the to-do list, so that it looks like the listing you saw earlier.

- Finally, you'll need to create another page, named **_PageStart.cshtml**, which is a special page that WebMatrix always calls when rendering a page. You can use this page to tell the system what the layout for any page is. The entire content of this page should use the following code:

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

With all that in place, here's what happens when you now visit todo.cshtml. WebMatrix will look and see that there's a layout template for the page (**_sitelayout.cshtml**). It will then render that page (and not todo.cshtml), which causes the *RenderBody()* command to be called. This command then renders the contents of todo.cshtml, providing the *<div>*.

- Run the page. You'll see something like Figure 10-11.

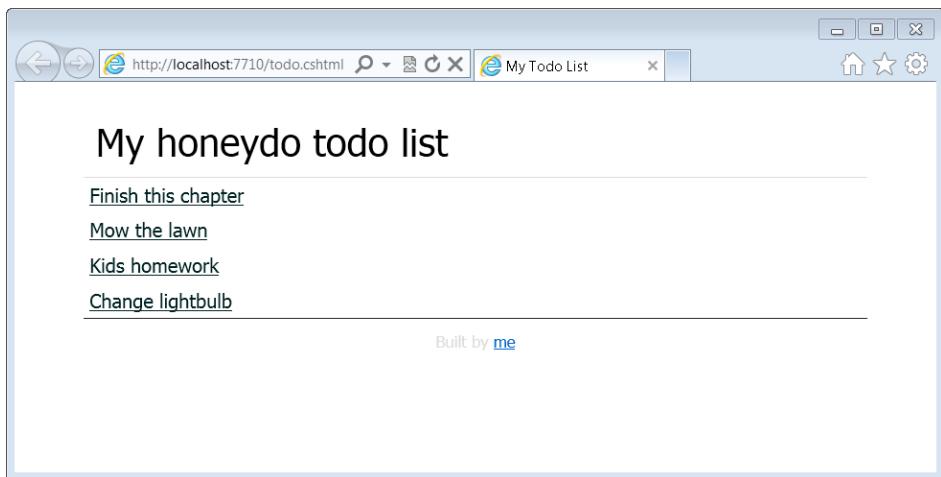


FIGURE 10-11 The todo.cshtml page using a layout page.

Of course, the page doesn't look any different than it did before. That's because the layout page that acts as the template uses the same code as the earlier todo.cshtml did. But what's nice now is that all the parts of the content are externalized so that you can reuse them easily.

- As an example, create another page called **about.cshtml**, and give it just the following content:

```
<h1>About this site</h1>  
<h2>Created while reading Chapter 10 of Introducing Microsoft WebMatrix</h2>
```

You don't need any *<html>*, *<head>*, or *<body>* tags or anything—just the two preceding lines of code.

- Run this page now, and you'll see something like Figure 10-12.

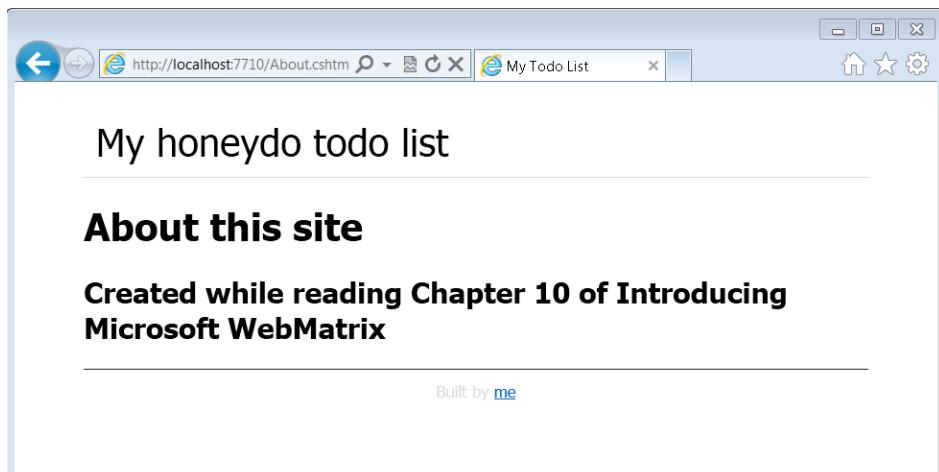


FIGURE 10-12 A templated page.

As you can see, you have created and styled a page, and then created a layout page that can be used as a template for all the pages on your site.

In Chapter 11, you'll add a database to this site and use the template page to create the rest of the pages, giving you a fully active to-do list.

Summary

This chapter walked through the first steps of using WebMatrix to put a site together end to end. To start the site, you created a webpage that rendered static HTML. You then saw how to use CSS to style the HTML and make it more attractive. You externalized the CSS to make that reusable. Then you used a layout page and template to turn the HTML into a template that you can easily reuse across many pages. The template contains the HTML common to all pages and uses the *RenderBody* command to render distinctive content for each page.

In the next chapter you'll build on these techniques, adding a database to create an active site that retrieves data from that database and uses the data to build dynamic pages for creating to-do items. Eventually, you'll be able to add, update, and delete items from the to-do list as well as render the list as you did in this chapter.

Chapter 11

Building a Simple Web Application: Using Data

In this chapter, you will:

- Create a database.
- Build data retrieval.
- Add, edit, and delete pages for the database.

In Chapter 10, "Building a Simple Web Application: Styles, Layout, and Templates," you started building a web application. In that chapter, you saw how to use CSS as well as templates and layout pages to give your pages a common look and feel. In this chapter, you'll flesh out the application by using what you learned about databases back in Chapter 7, "Databases in WebMatrix."

Creating the Database

The first thing to do is create your database in the Ch10 website you created in Chapter 10.

1. Using the Databases workspace in Microsoft WebMatrix, create a new database and name it **todo.sdf**.
2. Add a new table to the database, and then add three fields to it. Name the first field **id**. Make sure that it's an identity field and also the primary key field. Name the second field **details** and make it an *ntext* type. The third field should be called **duedate** and should be a *datetime* type. When you're done, your table should look like the one shown in Figure 11-1.

(todo.sdf).NewTable_1*		
Column Name	Data Type	Allow Nulls
id	bigint	False
details	ntext	True
duedate	datetime	True

FIGURE 11-1 Your to-do list table.

3. Save the table and name it **todolist**.

4. Open the table and add a dummy entry to the table. Don't set the *id* value, because WebMatrix will create that for you automatically. Just place your cursor in the details column and type something such as **Finish this Chapter**. In the *duedate* field, enter a date, such as **March 1, 2011**. WebMatrix will convert the date you entered to a date and time. In Figure 11-2, you can see how I filled out the rest of the list items that you saw in the previous chapter.

Table - (todo.sdf).todoist			
	id	details	duedate
	1	Finish this Chapter	3/1/2011 12:00:00 AM
	2	Mow the Lawn	3/1/2011 12:00:00 AM
	3	Kids Homework	3/1/2011 12:00:00 AM
▶	4	Change lightbulb	3/1/2011 12:00:00 AM
*	NULL	NULL	NULL

FIGURE 11-2 To-do list entries in the database.

In the next section, you'll see how to replace the static list from Chapter 10 with a dynamic one driven from this data.

Creating a Data Retrieval Page

In Chapter 10, your site used a layout page to create the HTML *<head>*, styling, *<body>*, and everything else. If you've been following along, you'll have the two files *_siteLayout.cshtml* and *_PageStart.cshtml* already set up. If you don't, the rest of this code won't work, and you'll have to go back to Chapter 10 and follow the instructions in order to get it to work.

What's nice about using the layout page is that you now only need to write the specific code for the specific page. The *@RenderBody()* call in the template will inject that content into the full page.

1. With that in mind, create a new CSHTML page and name it **viewtodolist.cshtml**.
2. Replace all the code in this new page with the static HTML shown here (this is the same HTML that you used in the previous chapter):

```

<div id="todolist">
    <ul>
        <li><a href="#">Finish this Chapter</a></li>
        <li><a href="#">Mow the lawn</a></li>
        <li><a href="#">Kids homework</a></li>
        <li><a href="#">Change lightbulb</a></li>
    </ul>
</div>

```

When you look closely at this, you'll see that it's a *<div>* that specifies an unordered list (**). The static list has four items in it. If you want a fifth item, you can add a new ** entry.

When the page is pulling the data from the database, it doesn't know how many items are in the database, so it would have to produce n list ($</i>$) elements, where n is the number of records in the database. This repeated operation is perfect for a code *loop*, and in a moment, you'll write that loop.

3. But before you do that, you first need to tell the page about the database. At the top of `viewtodolist.cshtml`, enter the following code:

```
@{  
    var db= Database.Open("todo");  
    var sq1Q = "SELECT * FROM todolist";  
    var data = db.Query(sq1Q);  
}
```

In this case, you have multiple lines of code, so instead of putting an @ in front of each, you can use an at sign and opening brace (@{}) and then write all the code, ending the code block with a closing brace ()�.

The following is a line-by-line look at the code.

In this line, `var` stands for *variable*, which is an addressable item containing data. The code is telling the server to open the database called `todo` and store the reference to this in a variable that we'll call `db`:

```
var db= Database.Open("todo");
```

The next line of code then creates a `var` called `sq1Q` and stores the SQL command `SELECT * FROM todolist` in it:

```
var sq1Q = "SELECT * FROM todolist";
```

Structured Query Language (SQL) is the most common language used to find data in a database. In fact, SQL goes beyond just querying the data and can be used to insert new data, as you'll see later. For now, we'll just use it to get data.

When data is retrieved from a database, the `SELECT` command is commonly used. The syntax is `SELECT <something> FROM <somewhere> WHERE <condition>`.

The `<something>` can be a list of fields or a wildcard (*) character that means "everything." We haven't set a condition, so `SELECT * FROM todolist` commands the database to get all the columns and all the records from the `todolist` table.

This next line is the workhorse of the page. Earlier you opened the database and used the variable `db` to refer to it:

```
var data = db.Query(sq1Q);
```

WebMatrix was smart enough to know that what you are opening is a database. A database object in WebMatrix has a lot of functions (typically called *methods*) that you can call on it. One of these is the `Query` method, which runs a SQL command. You pass that command to it as a string, and the database executes it, returning a set of records. This set of records is then loaded into a variable called `data`.

We've gotten the data from the database, but we haven't done anything with it yet. We still have the static HTML on the page. Let's change that.

4. First, delete all but one of the `</i>` fields, so that your `<u>` looks like the following:

```
<ul>
    <li><a href="#">Finish this Chapter</a></li>
</ul>
```

5. Remember earlier in this section when we talked about having n database fields, and how we should then have n entries in the list? This is achievable by using a loop, so let's write a loop here.

Using Razor, add your code by using the @ symbol, as shown here:

```
@foreach(var row in data)
{
    <li><a href="#"> Finish this Chapter</a></li>
}
```

The code is telling the server to count the number of rows in the data, and for each one of them, write out the `</i>` tag.

The result of this would be, of course, that *Finish this Chapter* would get written out four times, because I have four entries in my database. You can see this in Figure 11-3.

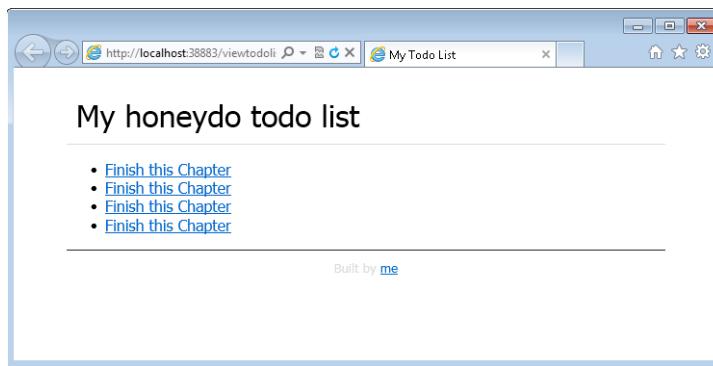


FIGURE 11-3 Running the loop.

The server only knows what it is told to do, and the code tells it to write out the details for each row in the data.

6. So let's get a little smarter. What we really want to do is, for each row in the data, write out the name of the item *in that row*. In other words, lose the *Finish this Chapter* text, and replace it with the name of the element in the row that the loop is currently looking at.

When you created the database, you called the column that contains the task *details*, and you already know that the current record is called *row*, so you can deduce that the

value `row.details` will contain the item that you want to render. Therefore, change your code to the following:

```
@{
    var db= Database.Open("todo");
    var sqlQ = "SELECT * FROM todolist";
    var data = db.Query(sqlQ);
}
<ul>
    @foreach(var row in data)
    {
        <li><a href="#">@row.details</a></li>
    }
</ul>
```

The code now tells the server that for each row in the data, it should write out the value in the `details` field from the row into the `<i>` element. You can see how this looks in Figure 11-4.

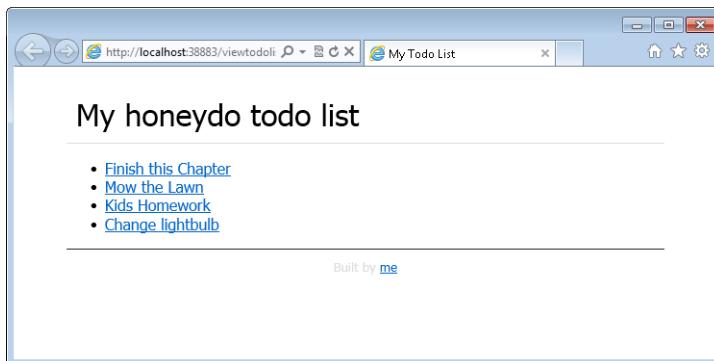


FIGURE 11-4 Viewing the list.

You can see that we have all the to-do items again!

7. Now that the page is *dynamic* and loading the items based on the data in the database, let's add another item to the database and see what happens.

Simply go back to the database editor, open the table, and add a new row of data, as shown in Figure 11-5.

Table - (todo.sdf).todolist *			
	id	details	duedate
	1	Finish this Chapter	3/1/2011 12:00:00 AM
	2	Mow the Lawn	3/1/2011 12:00:00 AM
	3	Kids Homework	3/1/2011 12:00:00 AM
	4	Change lightbulb	3/1/2011 12:00:00 AM
	5	Repair the roof	3/1/2011 12:00:00 AM
▶*	NULL	NULL	NULL

FIGURE 11-5 Adding an item to the database.

8. Run your page. You'll see a new item on the list—without writing any HTML—as shown in Figure 11-6.

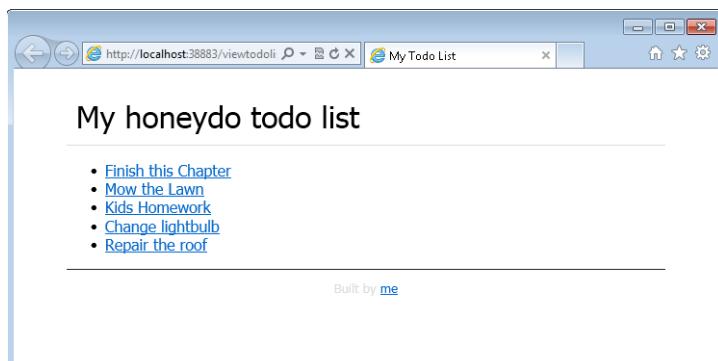


FIGURE 11-6 The new list.

If you've written code in inline languages like this before, you might have noticed that WebMatrix is smart enough to write the content out when you place it inline in the ``. You didn't have to specify `write row.details` or anything like that; you just added `@row.details` in the place where you wanted it to go. This capability makes editing and maintaining the CSHTML files much easier than when you use languages such as PHP, in which you would need to use `<?php echo(row.Name);>` or something similar.

9. This also allows you to mix dynamic and static content seamlessly. To see how this works, change the code to the following:

```
@{
    var db= Database.Open("todo");
    var sqlQ = "SELECT * FROM todolist";
    var data = db.Query(sqlQ);
}
<ul>
    @foreach(var row in data)
    {
        <li><a href="#">@row.details,
            @String.Format("{0:MMM-dd-yyyy}", row.duedate)
        </a>
    </li>
}
</ul>
```

The result is a seamless integration of the static HTML (the commas between the fields) and the dynamic database-driven content (the fields themselves) that looks like Figure 11-7.

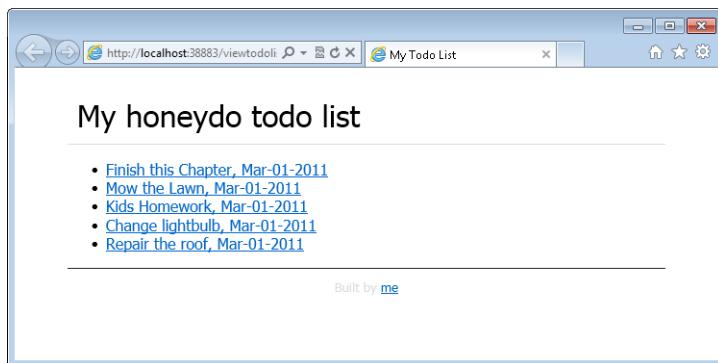


FIGURE 11-7 The list with dates added.

And with that, you've created the 'R' in CRUD (Read). In the next section, you'll see how to create a form that will allow you to add new items to the to-do list by using the browser, instead of editing the database directly by using WebMatrix.

Creating an Add Data Page

In this section, you'll see how to create a page that allows you to add items to the database, and therefore to your list, by using the browser. In a real application, you'd have some type of security and membership system set up so that users would manage only their own lists. To keep things simple, this one is just a single list that is managed by any user.

1. Create a new page and call it **addtodo.cshtml**.
2. Remove all of the page contents and replace them with this single line of code:

```
<h1>Add a new todo item</h1>
```

3. Now go back to the **viewtodolist.cshtml** file that you've been working on and look at how the list items are rendered. Add a new link at the bottom of the code page, outside of the **** list. Point this to the **addtodo.cshtml** page that you just created:

```
@{  
    var db= Database.Open("todo");  
    var sqlQ = "SELECT * FROM todolist";  
    var data = db.Query(sqlQ);  
}  
<ul>  
    @foreach(var row in data)  
    {  
        <li><a href="#">@row.details, @String.Format("{0:MMM-dd-yyyy}", row.duedate)</a></li>  
    }  
</ul>  
<a href="addtodo.cshtml">Add a new item...</a>
```

- Now run `viewtodoist.cshtml` and click the new link. You'll be taken to the `addtodo.cshtml` page that you've just created. You can see this in Figure 11-8.

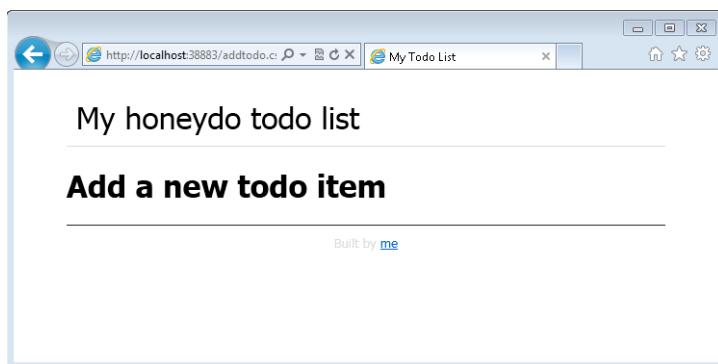


FIGURE 11-8 The add new item page.

- Your users now have a way to add content to your list, so let's create the form that allows them to specify the details and date for their to-do items.

We'll start with a simple form. It's not very pretty, but it gets the job done. Add the form code to your `addtodo.cshtml` page:

```
<h1>Add a new todo item</h1>
<form action="" method="post">
    <p>Details:<input type="text" name="inputDetails" /></p>
    <p>Due Date:<input type="text" name="inputDueDate" /></p>
    <p><input type="submit" value="Add Item" /></p>
</form>
```

You can see how this looks in Figure 11-9.

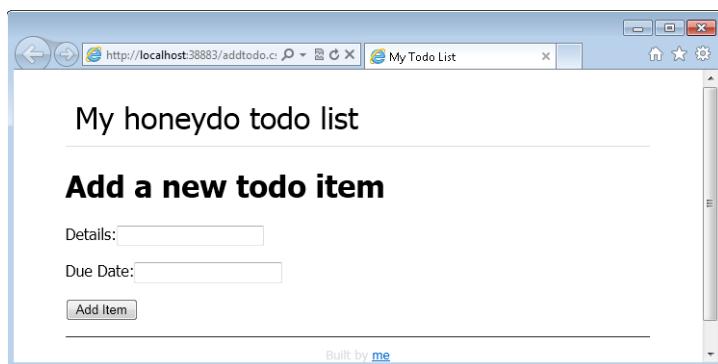


FIGURE 11-9 Your add form.

Now let's take a look at the HTML we've just written to create this form.

The first new thing is the `<form>` tag. This defines the form and tells the server that when the user clicks the `<submit>` button, which must be present in the form, the action will be an HTTP *POST*. Because the *action* parameter is empty, this same page (`addtodo.cshtml`) will process the post from the form:

```
<form action="" method="post">
```

Within the `<form>` tag, you'll see that there are two `<input>` controls. These use the HTML `<input>` control, which can have several *type* settings. In this case, the *type* is *text*, which displays a basic text box into which the user can input text. Each control is given a *name* that will be the variable that the server uses to store the value that the user enters into the corresponding text box before submitting:

```
<p>Details:<input type="text" name="inputDetails" /></p>
<p>Due Date:<input type="text" name="inputDueDate" /></p>
```

Finally, we have an `<input>` control of type *submit* that defines the submit button. When the user clicks this button, the HTTP *POST* action will be invoked and the data that the user entered will be sent to the server:

```
<p><input type="submit" value="Add Item"/></p>
```

Right now if you click the button, nothing will happen. This is because you haven't written the code to handle the postback from the server yet. You'll do that next.

Handling Submitted Data from an Add Form

When you created the form, you left the *action* parameter empty, which means that you are specifying that the same page should accept the form submission.

This is done by updating your `addtodo.cshtml` page with some code that will execute whenever the page loads.

1. Add a code block at the top of the page, as shown here:

```
@{
    // Code to execute
}
<h1>Add a new todo item</h1>
<form action="" method="post">
    <p>Details:<input type="text" name="inputDetails" /></p>
    <p>Due Date:<input type="text" name="inputDueDate" /></p>
    <p><input type="submit" value="Add Item" /></p>
</form>
```

Earlier you saw that the first time you call the page, by typing its address in your browser (or by clicking Run on the ribbon to launch the file), your browser requests the pages by using the HTTP *GET* verb to get the page. Later, when you clicked the submit button, because the *action* property was blank, the HTTP *POST* operation just called the *same* page.

In your code, you'll need a way to figure out whether the page is being retrieved by using a *GET* or being executed by using a *POST*.

2. Fortunately, the Microsoft .NET Framework lets you check the verb without getting into the complexity of breaking up the HTTP headers to inspect what type of message you're getting; you simply use the *if(IsPost)* check which, when true, means that the user has used a form to post the information. Add the following lines to addtodo.cshtml:

```
@{  
    if(IsPost)  
    {  
        // Do something on the post  
    }  
}
```

3. When you set up the form, you gave names (*inputDetails*, *inputDueDate*) to the values that the user submits. When the browser calls the server, it will use these names by sending a message that contains *inputDetails=something* and *inputDueDate=something*.

In this step, you'll set up some server variables to hold these, and then you'll read them off the HTTP post. This is a lot easier than it sounds. Just add the following lines to the top of addtodo.cshtml:

```
@{  
    var details = "";  
    var duedate = "";  
    if(IsPost)  
    {  
        details = Request["inputDetails"];  
        duedate = Request["inputDueDate"];  
  
    }  
}
```

Here you can see that two variables have been set up and initialized with the values that the user submitted. In the next step, you'll add the code to open the database and add this information to it.

Adding Data to the Database

Earlier, when you retrieved data from the database, you wrote a SQL *SELECT* query that selected the data from the database for you to read. In this case, you are *adding* data to the database, which uses an *insert* query.

The SQL *INSERT* command uses the following syntax:

```
INSERT INTO Table (Column1, Column2, ...ColumnN) VALUES (Value1, Value2, ... ValueN)
```

1. Use that syntax to add this functionality to the top of `addtodo.cshtml`:

```
@{  
    var details = "";  
    var duedate = "";  
    if(IsPost)  
    {  
        details = Request["inputDetails"];  
        duedate = Request["inputDueDate"];  
        var SQLINSERT = "INSERT INTO todolist (details, duedate) VALUES (@0, @1)";  
        var db = Database.Open("todo");  
        db.Execute(SQLINSERT, details, duedate);  
  
    }  
}
```

This code creates a string called `SQL/INSERT` to hold the command. Razor allows you to specify parameterized values in this string, so instead of trying to add the values for `details` and `duedate` to the string, you can simply use `@0` and `@1`. When you execute the query, the framework allows you to substitute them in.

2. Open the database, and then call the `Execute` method on the database to tell it to run this insert query. The values are substituted in and are added to the database.
3. Let's add one more line after `db.Execute()` to redirect users back to the page where they can view their to-do list and see the results of what they just added:

```
Response.Redirect("viewtodolist.cshtml");
```



Note When adding data to a database like this, you should make sure that the values are sanitized and validated first. I'm not doing that for this example, just to keep things simple, but you should ensure that users don't use a form like this to perform SQL injection by passing SQL code within the parameters, and you should verify that they send data (such as the date) in the correct format, so as not to cause an error.

4. Run the page. You'll see the form. Add some data to it, as shown in Figure 11-10.

The screenshot shows a web browser window titled "My Todo List". The main content is a form titled "Add a new todo item". It contains two text input fields: "Details: Plant flowers" and "Due Date: 3/12/2011". Below the inputs is a blue "Add Item" button. At the bottom of the page, there is a footer line that reads "Built by me".

FIGURE 11-10 Adding a new item.

5. Click the Add Item button. The server will add the data to the database, and then redirect you back to the list to see the results. You can see that your new to-do item has been added (see Figure 11-11).

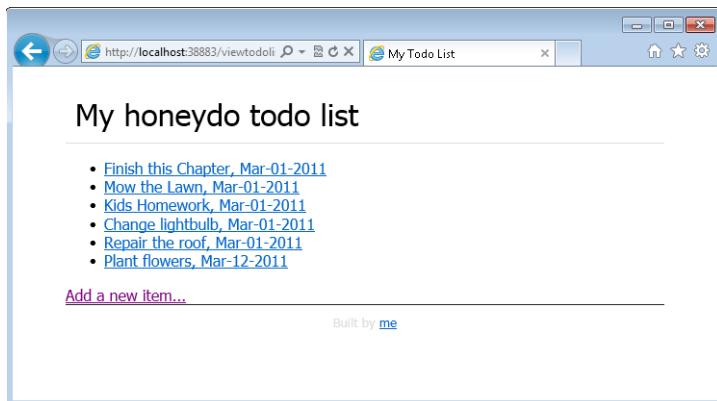


FIGURE 11-11 The new item has been added.

Now that you've added an item to your database, the next step is to allow your users to edit items. You'll see how to do this in the next section.

Creating an Edit Page

At this point, you've created your todolist page, styled it, made it data-driven, and then created a form that can be used to add items to your database. The next step is to create a very similar form that you can use to *edit* your existing list of to-do items.

Handling Submitted Data from an Edit Form

On your list of to-do items, you made each item in the list a hyperlink by using the `<a>` tag. It would make sense that if you want to edit an item, you should just use this hyperlink. Let's follow that process here.

1. Create a new CSHTML page in WebMatrix and call it **edittodo.cshtml**. This page will eventually have a form that is populated with the details of the item the user selected, and when the user *changes* those details, the changes will be submitted back to the database.

- Now replace the default content in edittodo.cshtml with the following form. This form will look very similar to the one you created for adding a to-do item:

```
<h1>Edit a todo item</h1>
<form action="" method="post">
    <p>Details:<input type="text" name="inputDetails" /></p>
    <p>Due Date:<input type="text" name="inputDueDate" /></p>
    <p><input type="submit" value="Add Item" /></p>
</form>
```

- You now have the basic foundation for the edit form. But how do you initialize the form with the contents of the database for the particular to-do item the user selected? Well, first let's figure out how to tell this page which to-do item to edit. To do that, go back to the viewtodolist.cshtml page.

As you might recall, you had the list items written out as shown here:

```
<li><a href="#">@row.details, @String.Format("{0:MMM-dd-yyyy}", row.duedate)</a></li>
```

The hyperlink went nowhere, because the *href* is just #. So let's make the hyperlink go to your edittodo.cshtml page:

```
<li><a href="edittodo.cshtml">@row.details, @String.Format("{0:MMM-dd-yyyy}", row.duedate)</a></li>
```

- This is nice, but whichever to-do item the user selects, this code will always call edittodo.cshtml, and that page will have no idea what to-do item is being edited. However, the viewtodolist.cshtml page *already* knows which item is being edited, because the user has selected it, so you can use this to pass the ID of the selected item to edittodo.cshtml by editing the *href* to this:

```
edittodo.cshtml?id=<something>
```

- Because you already know what the ID of the current row is (@row.id), you can use Razor to write out the ID as you write out the list, so change your ** to look like the following:

```
<li><a href="edittodo.cshtml?id=@row.id">@row.details, @String.Format("{0:MMM-dd-yyyy}", row.duedate)</a></li>
```

- Now look at viewtodolist.cshtml in the browser. It doesn't really look any different, but look at the HTML code for this page. This isn't the .cshtml page that you saw in the WebMatrix editor, but instead is the HTML that gets generated by the server (from the code in the CSHTML) and sent to the browser.

In Windows Internet Explorer 9, you can view this by right-clicking anywhere on the page and selecting View Source:

```

<!DOCTYPE html>
<html lang="en">
    <head>
        <link rel="stylesheet" type="text/css" href="todo.css" />
        <meta charset="utf-8" />
        <title>My Todo List</title>
    </head>
    <body>
        <header><h1>My honeydo todo list</h1></header>
        <ul>
            <li><a href="edittodo.cshtml?id=1">Finish this Chapter, Mar-01-2011</a></li>
            <li><a href="edittodo.cshtml?id=2">Mow the lawn, Mar-01-2011</a></li>
            <li><a href="edittodo.cshtml?id=3">Kids homework, Mar-01-2011</a></li>
            <li><a href="edittodo.cshtml?id=4">Change lightbulb, Mar-01-2011</a></li>
            <li><a href="edittodo.cshtml?id=5">Repair the roof, Mar-01-2011</a></li>
            <li><a href="edittodo.cshtml?id=6">Plant flowers, Mar-12-2011</a></li>
        </ul>
        <a href="addtodo.cshtml">Add a new item...</a>
        <footer>Built by <a href="http://www.philotic.com">me</a></footer>
    </body>
</html>

```

You can see how, when the page was created, the value of the ID for this particular row was written out to the HTML. Now, when `edittodo.cshtml` loads, you can take this ID and use it to find the particular record you're interested in.

So let's go back to `edittodo.cshtml`.

7. Remember that earlier, when you created the add page, you saw that if you put a @{/ at the top of the page and wrote code within that, the code would execute when the page loaded. That's the perfect place to put code that will read the ID that was in the URL of the page and then use that ID to find the to-do item to be edited.

When you call a page with the syntax that we're using,

`PageName.cshtml?<Parameter>=<Value>`

you can find out the value of the parameter by using the `Request` object. So, for example, for `edittodo.cshtml?id=6`, you can use the following code:

```
var id=Request["id"];
```

This code creates a local variable called `id` and uses the value of the parameter (also called `id`) to initialize it. WebMatrix is smart enough to realize that both `ids` are different based on their context.

Now that you have the `id`, you can use it with a `SELECT` query in SQL to find the record for that item. Add the following code to the top of the edit page:

```
@{
    var id=Request["id"];
    var SQLSELECT = "SELECT * FROM todolist where ID=@0";
    var db = Database.Open("todo");
    var item = db.QuerySingle(SQLSELECT,id);
    var details=item.details;
    var duedate=String.Format("{0:MMM-dd-yyyy}", item.duedate);
}
```

Pretty straightforward, right? This code says “select * from *todolist* where the ID field is equal to the ID that we passed in” and then runs that against the database. Because you only want one record, you say *db.QuerySingle* to get a single record.

Then the query is executed, and the *details* and *duedate* values of the to-do item are loaded into local variables.

8. This is all very well, but the problem is that the values are in variables and not in the form, so how does the user edit them? The answer is simple: remember that this code executes before the page renders, so you have the variables before you write out the HTML. And because of that, you can *initialize* the form with these values. The form uses *<input>* fields to provide text boxes, and these support a *value* property. You can now use this with the variables directly in order to initialize them.

Update your page to match the full code for the page so far, as shown here. Note the *value* attributes in the form elements:

```
@{
    var id=Request["id"];
    var SQLSELECT = "SELECT * FROM todolist where ID=@0";
    var db = Database.Open("todo");
    var item = db.QuerySingle(SQLSELECT,id);
    var details=item.details;
    var duedate=String.Format("{0:MMM-dd-yyyy}", item.duedate);
}
<h1>Edit a todo item</h1>
<form action="" method="post">
    <p>Details:<input type="text" name="inputDetails" value="@details" /></p>
    <p>Due Date:<input type="text" name="inputDueDate" value="@duedate" /></p>
    <p><input type="submit" value="Edit Item" /></p>
</form>
```

9. Now when you run the page, you’ll see that it is initialized with the to-do list values. Try it by running *viewtodolist.cshtml* first, and then select any item. You’ll see the edit form as in Figure 11-12.

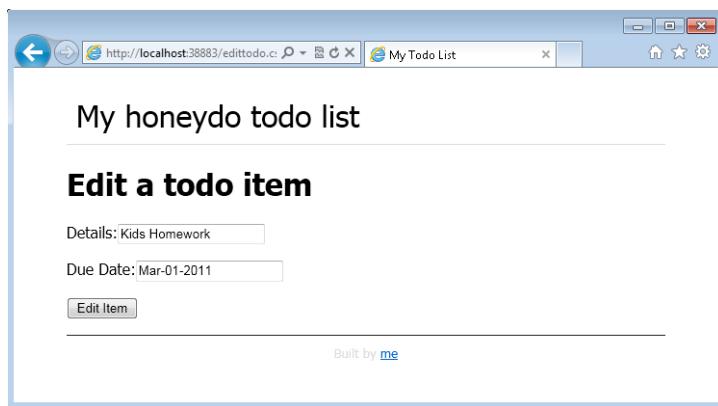


FIGURE 11-12 Editing an item.

You can type into the boxes to change any of the values, but when you click Edit Item, nothing happens. You might recall from the Add Item page earlier that the button triggers a form submit, which is an HTTP POST (as opposed to the HTTP GET, that occurred when the page was called from the hyperlink), which needs to be processed.

Updating the Database

To update a database, you use the SQL *UPDATE* command, which has the following syntax:

```
UPDATE table SET column=value, column=value, column=value... WHERE key=value
```

1. In this exercise, you are updating two columns for the ID that you already know, so here's the code to create the query and execute it:

```
if(IsPost)
{
    details=Request["inputDetails"];
    duedate=Request["inputDueDate"];
    var SQLUPDATE = "UPDATE todolist set details=@0, duedate=@1 WHERE id=@2";
    db.Execute(SQLUPDATE, details, duedate, id);
    Response.Redirect("viewtodolist.cshtml");
}
```

This uses parameters in SQL (@0, @1, and so forth, not to be confused with the @ that marks the start of a Razor block) to make the SQL easier to read. Just remember that the @ values are replaced, in order, with the values within the *db.Execute()* code, so in this case *details* is @0, *duedate* is @1, and so on.

2. Finally, once the item is edited, it's a good idea to redirect back to the list page so that users can see the results of their editing, updated dynamically. You did the same in the add to-do item page. Here's the code:

```
Response.Redirect("viewtodolist.cshtml");
```

For your convenience, here's the full code for the `edittodo.cshtml` page:

```
@{
    var id=Request["id"];
    var SQLSELECT = "SELECT * FROM todolist where ID=@0";
    var db = Database.Open("todo");
    var item = db.QuerySingle(SQLSELECT,id);
    var details=item.details;
    var duedate=String.Format("{0:MMM-dd-yyyy}", item.duedate);
    if(IsPost)
    {
        details=Request["inputDetails"];
        duedate=Request["inputDueDate"];
        var SQLUPDATE = "UPDATE todolist set details=@0, duedate=@1 WHERE id=@2";
        db.Execute(SQLUPDATE, details, duedate, id);
        Response.Redirect("viewtodolist.cshtml");
    }
}

<h1>Edit a todo item</h1>
<form action="" method="post">
    <p>Details:<input type="text" name="inputDetails" value="@details" /></p>
    <p>Due Date:<input type="text" name="inputDueDate" value="@duedate" /></p>
    <p><input type="submit" value="Edit Item" /></p>
</form>
```

3. Now let's see what happens when you run the page. Assuming that you've started with the `viewtodolist.cshtml` page and clicked a hyperlink to edit one of the items, you should see the edit item screen. Change something and click the Edit Item button. The database will be updated, and you'll be redirected to the list screen, where you can now see the changes.

Creating a Delete Data Page

You've reached the point now where you have created a data-driven to-do list, styled it, and added the ability to add and edit items in the database. The next step in creating this application is to give your users the ability to delete records from the database.

1. To start, create a new CSHTML page and call it **deletetodo.cshtml**.
2. Replace the HTML in it with the following:

```
<h1>Delete an item</h1>
<p>Are you sure you want to delete the item <strong>@item.details ?</strong></p>
<form action="" method="post">
    <input type="submit" value="Yes"/>
    <input type="button" value="No"
          onclick="window.location = 'viewtodolist.cshtml'" />
</form>
```

This creates a basic form containing two buttons: a submit button that triggers an HTTP *POST* (just as in the edit page in the previous section), and another button that redirects the user back to the to-do list when it is clicked.

Just like the edit page, this page will be called and passed a parameter that is the ID of the item to delete. As you can see, the text is *Are you sure you want to delete the item @item.details?* where the value of *item.details* will be inserted by the server. So you need to tell the server how to get this value.

3. To do that, as before, put some Razor code at the top of the page to catch the input parameter:

```
@{  
    var id=Request["id"];  
    var SQLSELECT = "SELECT * FROM todolist where ID=@0";  
    var db = Database.Open("todo");  
    var item = db.QuerySingle(SQLSELECT, id);  
    var details=item.details;  
    var duedate=String.Format("{0:MMM-dd-yyyy}", item.duedate);  
}
```

Here you can see that the parameter was passed to the page as *id* (by using *deletetodo.cshtml?id=<whatever>*), and this is used to find the specific item. A query is run against the database, and the record for that item is retrieved. Now you can derive the details of that item and show it when the page is rendered.

4. Give it a try. Run *deletetodo.cshtml?id=<something>*, and you'll see the screen shown in Figure 11-13, as long as the *<something>* is a valid ID in your database.

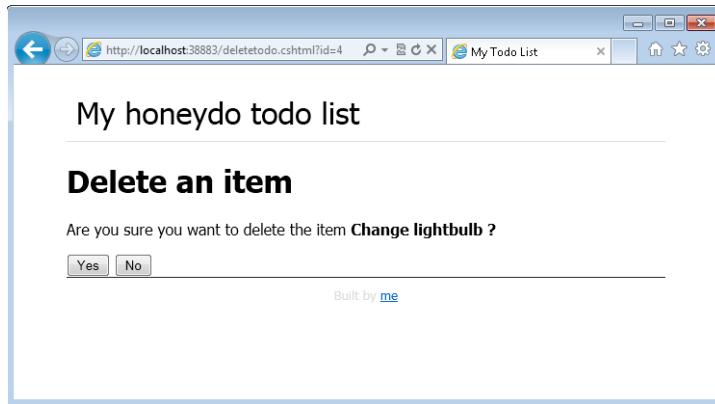


FIGURE 11-13 Deleting a list item.

If you click the No button, you'll be redirected back to the *viewtodolist.cshtml* page. If you click Yes, nothing happens, because you haven't written the code to handle the deletion yet.

5. To delete a record from a database, you use the *DELETE* SQL command. This has the following syntax:

```
DELETE FROM <Table> WHERE <Field>=<Value>
```

For example, if you wanted to delete the *id=2* item, you would use the following:

```
DELETE FROM todolist WHERE id=2
```

When the user clicks the Yes button, the form will be submitted and the deletion will occur. Update your code as follows, so that this can easily be done on the postback:

```
if(IsPost)
{
    var SQLDELETE = "DELETE FROM todolist WHERE ID=@0";
    db.Execute(SQLDELETE,id);
    Response.Redirect("viewtodolist.cshtml");
}
```

This will delete the item and redirect users back to the listing page so that they can see that the item is gone.

The following shows the full listing for deletetodo.cshtml:

```
@{
    var id=Request["id"];
    var SQLSELECT = "SELECT * FROM todolist where ID=@0";
    var db = Database.Open("todo");
    var item = db.QuerySingle(SQLSELECT,id);
    var details=item.details;
    if(IsPost)
    {
        var SQLDELETE = "DELETE FROM todolist WHERE ID=@0";
        db.Execute(SQLDELETE,id);
        Response.Redirect("viewtodolist.cshtml");
    }
}

<h1>Delete an item</h1>
<p>Are you sure you want to delete the item <strong>@item.details ?</strong></p>
<form action="" method="post">
    <input type="submit" value="Yes"/>
    <input type="button" value="No" onclick="window.location = 'viewtodolist.cshtml'" />
</form>
```

6. Now that you have your working delete page, let's wire it up to the to-do list page so that the user can select an item from that list and request to delete it.

On the list page, you simply have to add a hyperlink to each list entry. This hyperlink will link to the deleteitem.cshtml page and pass it the ID of the current item.

Update your code to add this functionality. Here's the complete code for viewtodolist.cshtml:

```
@{  
    var db= Database.Open("todo");  
    var sqlQ = "SELECT * FROM todolist";  
    var data = db.Query(sqlQ);  
}  
<ul>  
    @foreach(var row in data)  
    {  
        <li><a href="edittodo.cshtml?id=@row.id">@row.details,  
        @String.Format("{0:MMM-dd-yyyy}", row.duedate)</a></li>  
        <a href="deletetodo.cshtml?id=@row.id">Delete</a>  
    }  
</ul>  
<a href="addtodo.cshtml">Add a new item...</a>
```

7. Run this page to see the workflow for the item deletion. Figure 11-14 shows the viewtodolist.cshtml page with the delete links.

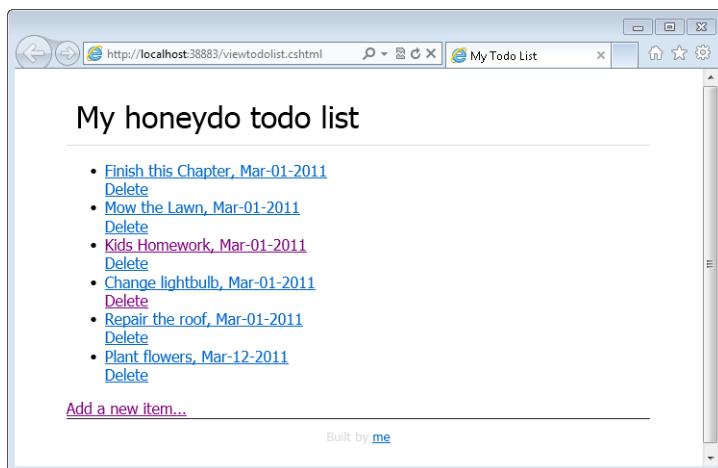


FIGURE 11-14 The view to-do list page with delete links.

8. Select the delete link on any of these items. You'll be taken to the Delete An Item page, where you'll be asked if you really want to delete that item from the list. You can see this in Figure 11-15.

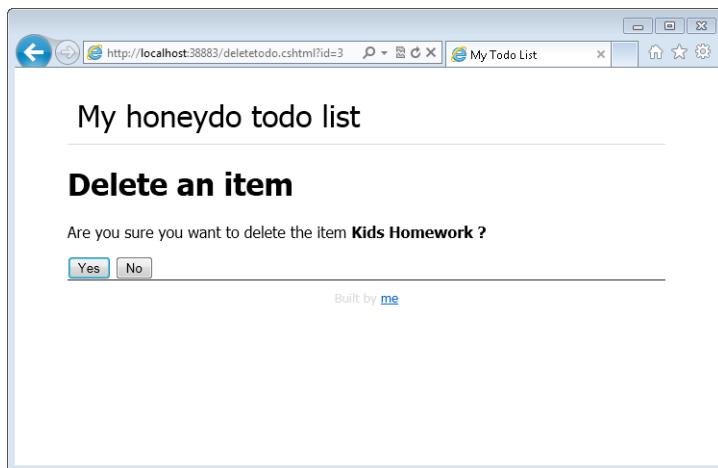


FIGURE 11-15 Viewing the delete page.

If you click No, you'll be returned to the list with the item still there. If you click Yes, you'll be returned to the list and the item will be gone. Figure 11-16 shows the result of clicking Yes.

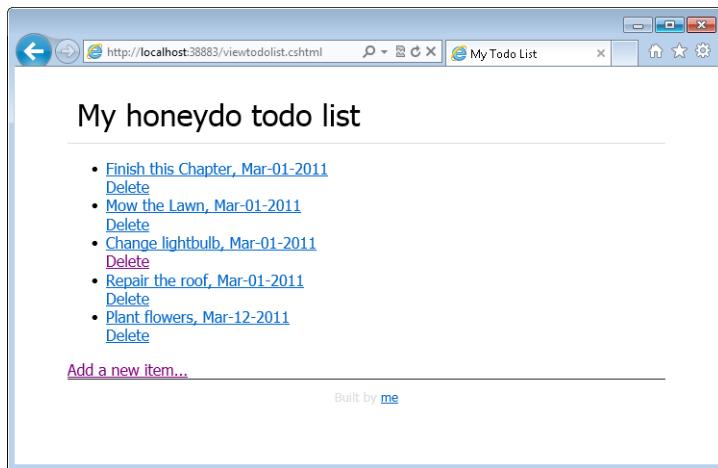


FIGURE 11-16 The updated to-do list.

And with that, you've now created a full CRUD application by using WebMatrix. Pretty simple, right?

Summary

Between the previous chapter and this one, you've now created a full Create, Read, Update, Delete (CRUD) application by using WebMatrix. Although this application is pretty basic, the concepts are valid across most web applications, demonstrating how you can retrieve, edit, and delete data, as well as add new data to a database.

Chapter 12

WebMatrix and Facebook

In this chapter, you will:

- Use the ASP.NET Web Pages Administration feature.
- Install and use the Facebook helpers for WebMatrix.
- Use a Live Stream feed.

In Chapter 8, "Exposing Your Site Through Social Networking," you used the LinkShare helper, which is an easy way to add functionality to your site to allow your users to share their content on various social networks, including Facebook. In this chapter, you'll look at some of the deeper integration that can be done with Facebook with the Facebook helpers, allowing you to create a truly social site.

Accessing ASP.NET Web Pages Administration

Before you can use them, you have to install the Facebook helpers, which are available from the Web Site Packages administration page via a feed from NuGet (<http://nuget.org>). In the next section, you'll see how to access this. NuGet is a new mechanism and feed that developers can use to share packages such as extensions and templates with each other.

1. Create an empty website by using Microsoft WebMatrix. Call it **SocialSite**.
2. Add a new CSHTML page to your site. By default, this page will be called Page.cshtml. Run your site.
3. In the address bar of the browser window, replace Page.cshtml with **_Admin**; in other words, if your address is *http://localhost:64570/Page.cshtml* it should now be *http://localhost:64570/_Admin*.

You'll now be on the Microsoft ASP.NET Web Pages Administration page (see Figure 12-1).

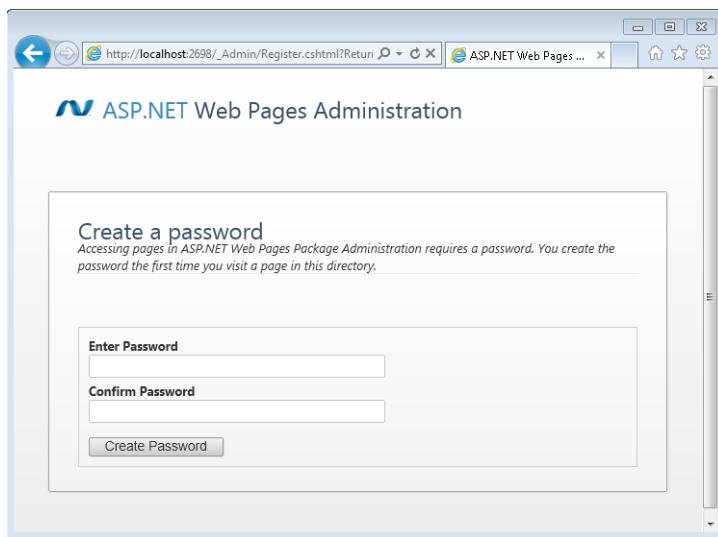


FIGURE 12-1 The ASP.NET Web Pages Administration screen.

4. You'll need to create a password here. This prevents users from coming to your live site and entering _Admin in order to change your site. Type a password, confirm it, and click Create Password.

WebMatrix writes the password to your website and gives you, the site administrator, instructions for how to finish setting it up (see Figure 12-2).



FIGURE 12-2 The ASP.NET Web Pages Administration Security Check page.

As the owner of the site, you have access to the files on the site. When the password file is created, an additional security step to ensure that you are the owner is taken: to have you change the name of the file within your site. This prevents a user from coming to your site via the browser, setting up a password, and then administering your site. As you can see, WebMatrix creates the password configuration in the file /App_Data/Admin/_Password.config.

You can see this in Figure 12-3. If you can't see the App_Data folder, simply right-click your top-level folder (in this case, SocialSite) and select Refresh from the menu.

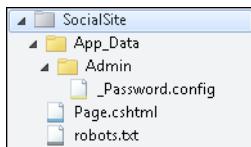


FIGURE 12-3 Editing the Password.config file name.

5. Now right-click `_Password.config` and rename it as **Password.config**, removing the leading underscore.
6. Run your site again and navigate to the `_Admin` screen. You'll now see that you have the ability to enter your password to sign in (see Figure 12-4).

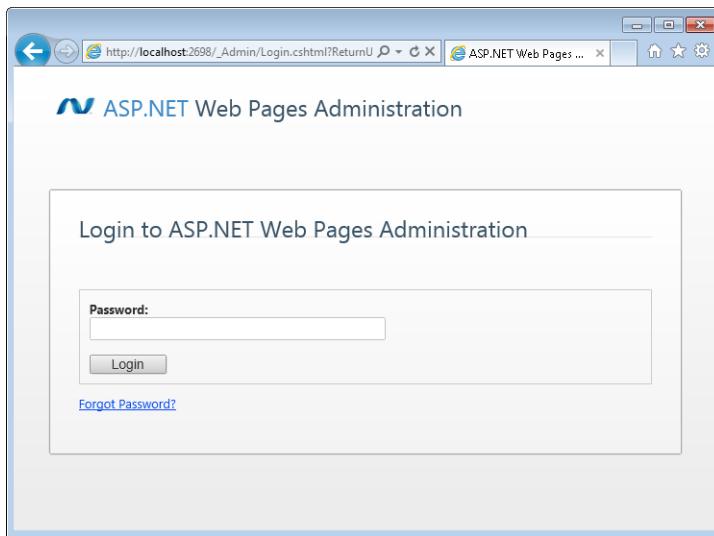


FIGURE 12-4 The ASP.NET Web Pages Administration sign-in screen.

7. Sign in with the password that you created earlier. You'll see the package manager in Figure 12-5.

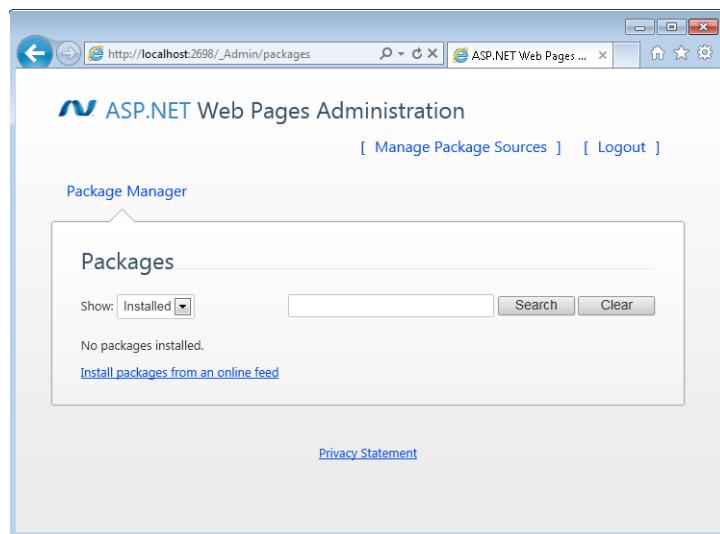


FIGURE 12-5 The Package Manager.

8. By default, the Package Manager shows you the packages that are already installed. Because you don't have any, it's a little bare, so select Online from the Show list and wait while the feed is read. You'll now get a list of available packages, an example of which is shown in Figure 12-6.

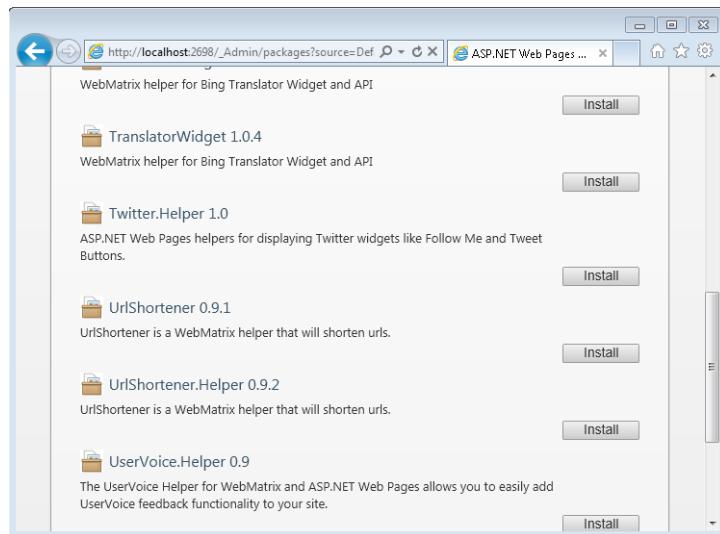


FIGURE 12-6 Available Packages from NuGet.

You're now ready to customize your WebMatrix development environment with new helpers, templates, and more!

Installing the Facebook Helpers from NuGet

Because this is a hosted list of packages, it's changing and growing all the time, so don't worry if your experience doesn't match that from Figure 12-6.

1. Find the Facebook helper (version 1.0 as of this writing, but if there's a later version, use that), and click the Install button. You'll see a confirmation screen like that in Figure 12-7.

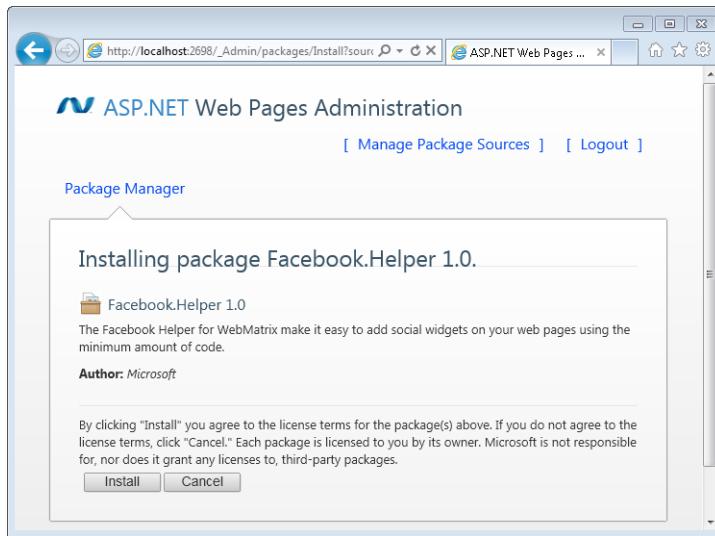


FIGURE 12-7 Confirming the Facebook helper installation.

2. Click Install again. The package will be downloaded and installed. You'll see the successful install confirmation, similar to that shown in Figure 12-8.

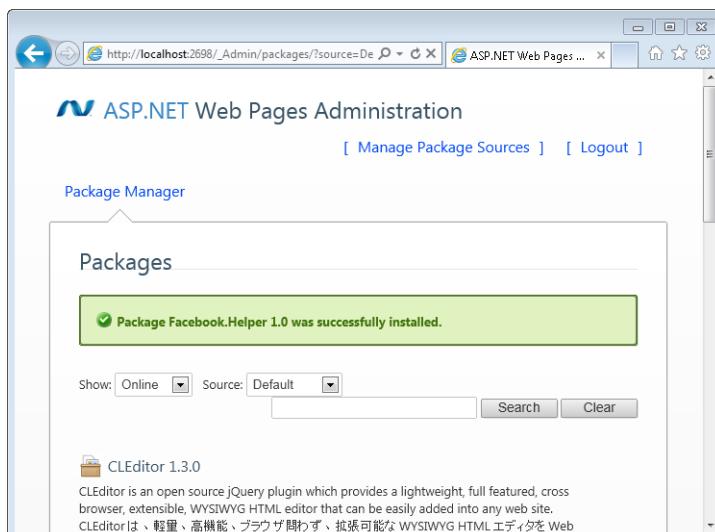


FIGURE 12-8 The confirmation screen for the Facebook helper installation.

3. Now that your package has been installed, take a look back at your Files workspace. Refresh the workspace by right-clicking SocialSite, and selecting Refresh from the menu. You'll see a lot of new content, including the helpers and documentation on how to use them, as shown in Figure 12-9.

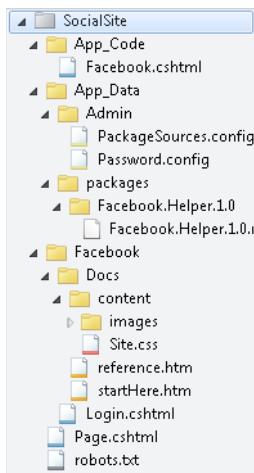


FIGURE 12-9 Your site with the Facebook helpers added.

You can view the documentation by browsing to it in your website by using the following URL: <http://localhost:<yourport>/Facebook/Docs/startHere.htm>.

Getting Started with the Facebook Helpers

Now that you've installed the Facebook helpers, let's take a look at using them.

1. Create a new site that uses the Bakery template. Call it **SocialBakery**. Go through the same process as earlier to set up package management and add the Facebook helpers.
2. The Bakery site uses a site layout page called _SiteLayout.cshtml, which defines the common header and footer of each page in the site. You can use this to add a simple Like button to your page.

To do this, open the _SiteLayout.cshtml page and take a look at the code. At the bottom of the page, you'll see the footer content. Add a Facebook Like button by adding the following code, changing *13091* to the port that your site is running on:

```
<div id="footer">
    &copy;@DateTime.Now.Year - Fourth Coffee
    <p>@Facebook.LikeButton("http://localhost:13091")</p>
</div>
```

3. Now run the site, and take a look at the footer. You'll see a Facebook Like button on it (see Figure 12-10).



FIGURE 12-10 The Facebook Like button in the SocialBakery website.

This is a fully functional Facebook Like button. You and your friends can now like the site, and it will show you how many people like this site!

Configuring and Initializing Facebook

The previous, very simple example added a Like button to the SocialBakery site. Much of the Facebook functionality requires you to have an App ID set up at Facebook.com in order for it to work. In this section, you'll see how to set this up.

1. Go to <http://www.facebook.com/developers/createapp.php>.
2. You'll be asked to give your site a name, and you'll have to accept the Facebook terms of usage (see Figure 12-11). If you've never built a Facebook application before, you'll be asked to allow access to the Developer application. You'll need to accept this too.

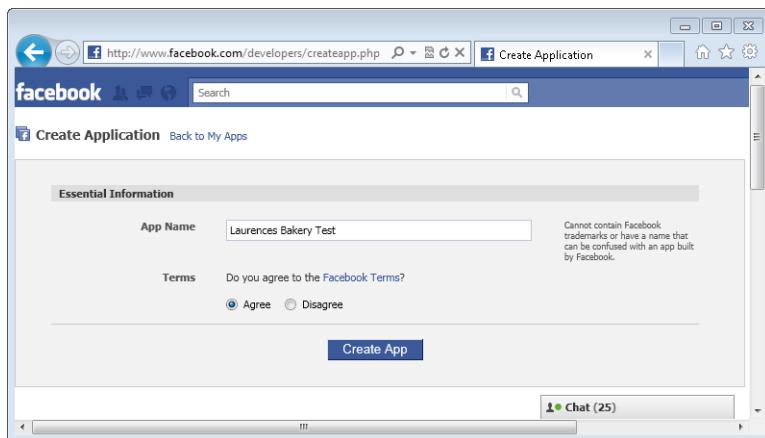


FIGURE 12-11 Setting up a Facebook application.

3. Select Create App. You'll go through a final CAPTCHA security verification, after which you will be taken to the App configuration screen on Facebook. Select the Web Site tab. You'll see a screen where you can read your App ID and App secret. These are needed to initialize the full Facebook functionality.

Make sure you enter the URL of your site here (note that the URL is at the top of the Site workspace in WebMatrix), and then click Save Changes. Your screen should look something like Figure 12-12, but be sure to use your URL instead of mine!

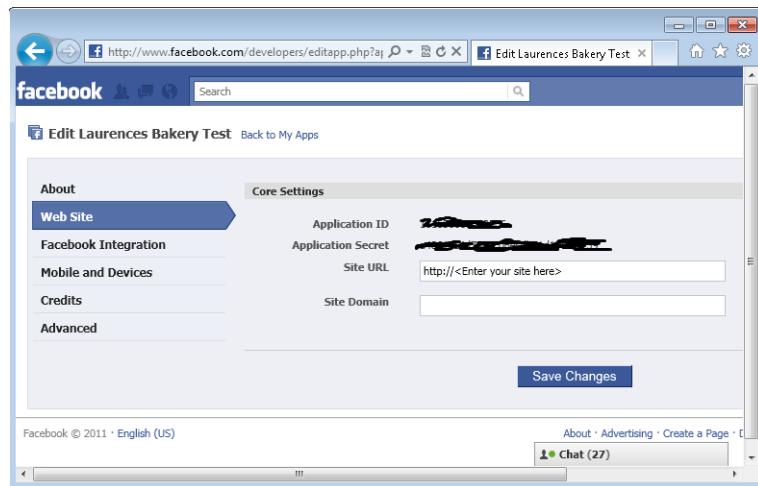


FIGURE 12-12 Getting the App ID and App Secret.

4. Take note of your App ID and App Secret—you'll need them to use the helper on your site.

Using a Facebook Comments Box

The Facebook comments box allows your visitors to comment on your site and have that comment added to their wall. It's a useful tool to have them share your site for you.

1. Create a new file in your bakery site called **_AppStart.cshtml** and replace its content with the following:

```
@{  
    Facebook.Initialize("{Your App ID}" , "{Your App Secret}");  
}
```

Change the values to the App ID and Secret that you obtained from Facebook.

2. Now go back to **_SiteLayout.cshtml** and edit the **<html>** tag at the top of the page to add the Facebook namespaces as shown in the following code. This adds the Facebook Markup Language (FBML) namespaces so that the code generated by the helpers, which use these namespaces, is understood by the browser:

```
<html lang="en" @Facebook.FbmlNamespaces()>
```

3. Next, Facebook uses some JavaScript libraries to load the FBML libraries that are used by the social plug-ins in your application. Instead of trying to figure out what the latest version or the correct syntax is, the helper does this for you, so add the following call directly below the `<body>` tag, as shown:

```
<body>
    @Facebook.GetInitializationScripts()
```

4. Finally, replace the Like button that you added to the footer earlier with a Facebook comments box, as shown here:

```
<p>@Facebook.Comments()</p>
```

For your convenience, here's the full `_SiteLayout.cshtml` listing:

```
<!DOCTYPE html>
<html lang="en" @Facebook.FbmlNamespaces()>
    <head>
        <meta charset="utf-8" />
        <title>Fourth Coffee - @Page.Title</title>
        <link href="@Href("~/Styles/Site.css")" rel="stylesheet" />
        <link href="@Href("~/favicon.ico)" rel="shortcut icon" type="image/x-
icon" />
    </head>
    <body>
        @Facebook.GetInitializationScripts()

        <div id="page">
            <div id="header">
                <p class="site-title"><a href="@Href("~/")">Fourth Coffee</a></p>
                <ul id="menu">
                    <li><a href="@Href("~/")">Home</a></li>
                    <li><a href="@Href("~/About")">About Us</a></li>
                </ul>
            </div>
            <div id="body">
                @RenderBody()
            </div>
            <div id="footer">
                &copy;@DateTime.Now.Year - Fourth Coffee
                <p>@Facebook.Comments()</p>
            </div>
        </div>
    </body>
</html>
```

5. Now run your site. You'll see the SocialBakery site with a Facebook comments box (see Figure 12-13).

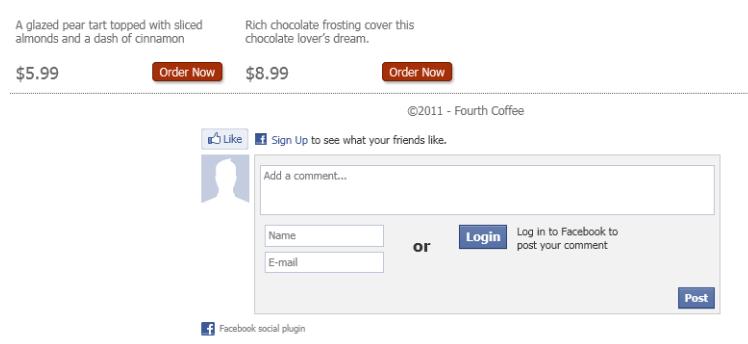


FIGURE 12-13 Your Facebook comments box.

6. Click the Login button to sign in to Facebook so that you can leave comments. The application will request permission to access your Facebook profile, as shown in Figure 12-14.



FIGURE 12-14 Accessing your profile.

7. Click Allow. You'll now go back to the bakery site, where you can see that you can make a comment on the page and share the comment with your Facebook friends (see Figure 12-15).



FIGURE 12-15 The comments box in your bakery site.

You can see how the comment looks in Figure 12-16.



FIGURE 12-16 Comment added to your site.

You can see how the comment will look on my Facebook wall in Figure 12-17.

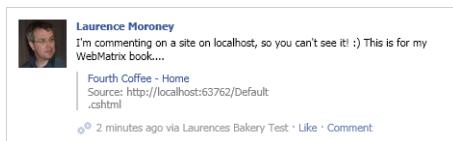


FIGURE 12-17 The posting on my wall.

Note that anybody viewing my stream will now see the comment, the URL, the name of the site, and the name of the application, Laurences Bakery Test. They can like the posting or comment on it.

The comment box is a great and powerful tool for getting the word about your site out to the Facebook community.

Using the Facebook Activity Feed

The Facebook activity feed displays stories both when users like content on your site and when users share content from your site back to Facebook. Of course, your *localhost* site probably doesn't have a lot of users other than yourself, but, believe it or not, the domain *localhost* is registered for a lot of applications on Facebook, so when you put an *ActivityFeed* on it, you'll see a lot of content. This is OK for testing it! When you deploy your site to a real server and give it a domain name, and then register the application on that domain, you'll just see your own activity.

To see the activity feed, you simply use the *ActivityFeed* method on the Facebook helper, as shown here:

```
@Facebook.ActivityFeed()
```

In this example, I've replaced the comment box from the previous section, in the SocialBakery site, with an activity feed by using the previous code. You can see how it looks in Figure 12-18.



FIGURE 12-18 The Facebook activity feed.

You can override many of the attributes of the activity feed. Full details for doing so can be found in the documents that were downloaded with the helper. You can see them at <http://localhost:<yourport>/Facebook/Docs/reference.htm>.

Using Facebook Recommendations

Similar to the activity feed, the recommendations engine will display items when users like content on your site and when they share content from your site back to Facebook. This plug-in shows personalized recommendations to your users based on their likes, and if the user is already logged into Facebook, it will include content from their friends.

To implement it, simply use the following:

```
@Facebook.Recommendations()
```

In this example, I've used it in the footer of the layout file in the SocialBakery sample. This runs on *localhost*, and the application is configured for *localhost*. You can see the Recommendations view from Facebook in Figure 12-19.



FIGURE 12-19 The Facebook recommendations feed.

You can override many of the attributes of the recommendations feed. Full details for doing so can be found in the documents that were downloaded with the helper. You can see them at <http://localhost:<yourport>/Facebook/Docs/reference.htm>.

Using the Facepile Feed

The Facepile plug-in shows the Facebook profile pictures of the user's friends who have already liked your site. It's a nice way to show who likes your site and who is contributing. As with the others, this is pretty easy to implement with the Facebook helper. You simply use the *Facepile()* method of the helper, as shown here:

```
@Facebook.Facepile()
```

When you run the site, you'll see simple tiles with the faces of the people who have liked your site. In the example in Figure 12-20, my site only has one person liking it (me).



FIGURE 12-20 Using the Facepile social plug-in.

You can override many of the attributes of the Facepile feed. Full details for doing so can be found in the documents that were downloaded with the helper. You can see them at <http://localhost:<yourport>/Facebook/Docs/reference.htm>.

Using the Live Stream Feed

Facebook Live Stream gives you real-time chat functionality on your site, where logged in users can chat with each other through Facebook. Implementing it is simple; as before, you simply use the *LiveStream()* method of the Facebook helper, as shown here:

```
@Facebook.LiveStream()
```

Now when you run your site, you'll see the Facebook Live Stream where your users can chat. This is particularly useful for events such as webcasts or other live events such as sporting events (see Figure 12-21).

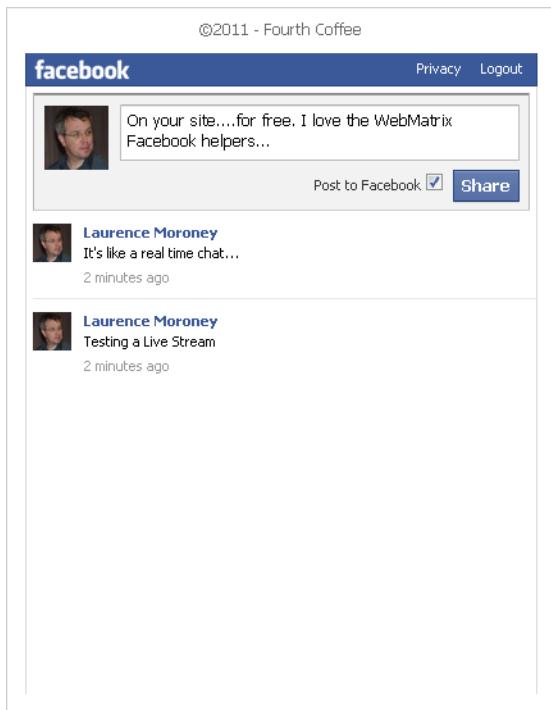


FIGURE 12-21 The Live Stream feed on my site.

Note that comments will also, by default, be posted on the user's home page so that they are shared with friends. As with the comments box, Live Stream is a great way to share your site with the world.

You can override many of the attributes of the Live Stream feed. Full details for doing so are in the documents that were downloaded with the helper. You can see them at <http://localhost:<yourport>/Facebook/Docs/reference.htm>.

Summary

This chapter introduced the Facebook helpers for WebMatrix. You saw how to install them by using ASP.NET Web Pages Package Administrator and how to set up an application on the Facebook developer site. You then saw how to add Facebook functionality to your site, including the following Facebook social helpers:

- Like box
- Comments box
- Activity feed
- Recommendations
- Facepile feed
- Live Stream feed

With these tools, you now have everything you need to build social sites with WebMatrix and access Facebook functionality easily. In the next chapter, you'll see how to use another useful helper—PayPal.

Chapter 13

WebMatrix and PayPal

In this chapter, you will:

- Sign up for a PayPal account.
- Create a PayPal Sandbox for testing purposes.
- Discover how to use PayPal with WebMatrix to accept payments.
- Understand how to switch from a PayPal Sandbox to a live PayPal environment.

In Chapter 12, “WebMatrix and Facebook,” you saw the Facebook helper, and how you can use it to build social applications that integrate Facebook into your site. Another common API frequently used to enhance websites is PayPal. Using PayPal, you can easily integrate e-commerce functionality into your site, including payment buttons and shopping carts.

In this chapter, you’ll see how to create an account on PayPal. You’ll start out using PayPal’s Sandbox site, and then integrate PayPal into a Microsoft WebMatrix site. At the end of the chapter, you’ll see how to switch from the Sandbox to a live account so that you can accept real PayPal payments on your site.

Signing Up for PayPal

Before you can use PayPal, you need to sign up for a PayPal account. You can do this using any email address that you control, such as a Windows Live Hotmail or Yahoo email account. The email address you select will be your primary way to sign into your account.

1. To begin, visit <https://www.paypal.com> and click Sign Up. You’ll be asked if you want a Personal, Premier, or Business account (see Figure 13-1).

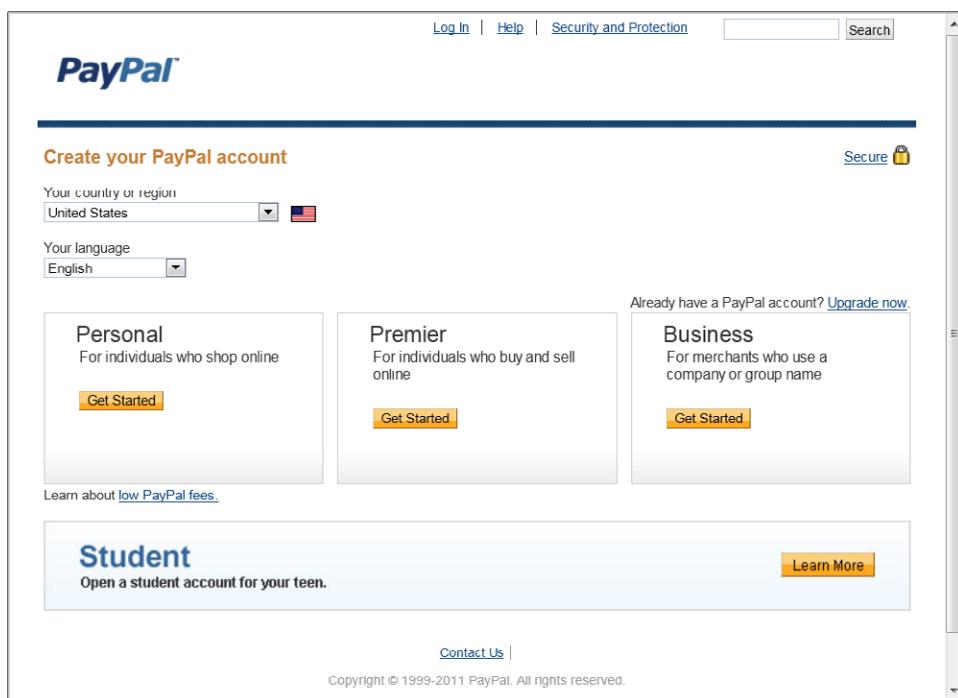


FIGURE 13-1 Selecting a PayPal account.

2. For the purposes of testing, select the Personal account. If you're setting up a live site, you should click the option that best meets your needs.
3. When you click Get Started, you'll be asked to set up the details of the account. Fill out the form with your own details (see Figure 13-2 for an example).

The screenshot shows the PayPal sign-up interface. At the top, the PayPal logo is visible. Below it, a header reads "Enter your information". A "Secure" link with a lock icon is located in the top right corner. The form contains several input fields:

- Email address: "ljpm@philotic.com" (highlighted in blue)
- Choose a password: "*****" (highlighted in blue)
- Re-enter password: "*****" (highlighted in blue)
- First name: "Laurence"
- Last name: "Moroney"
- Address line 1: (empty field)

A note at the bottom left says "Please fill in all fields."

FIGURE 13-2 Signing up for a PayPal account.

4. On the next screen, you'll be asked if you want to pay with a bank account, credit card, or with cash. Just ignore these options for now, and click Go To My Account at the bottom of the screen.
5. On the right side of the screen, you'll see a link asking you to confirm your email address. When you click this link, PayPal sends an email message to the account you registered with. Click the Activate link in this message. You'll see a screen where you can sign in to your new account. You'll be asked to set up some security details, and then you're good to go.

Creating a PayPal Sandbox

Now that you've set up your account, the next thing to do is to set up a Sandbox account where you can send and receive dummy payments without using real money.

1. To start, select <https://developer.paypal.com>. At the bottom of the screen shown in Figure 13-3, you can see that you can sign up for a Sandbox test environment.



FIGURE 13-3 The PayPal developer site.



Note At the time of this writing, PayPal Developer Central is migrating to x.com. Although that site is live, it appears that you still need to sign up for a Sandbox at <https://developer.paypal.com>. This might change by the time this book is in your hands, so if the screen you see doesn't match Figure 13-3, take a look at x.com to see if you can sign up there.

2. When you click "Sign Up Now", you'll need to fill out another form, this time for the Sandbox. Be sure to use a *different* email account for this than you did for your main PayPal account. After you've filled out the form, an email message will be sent to that address. Click the link within the message to finish signing up for the Sandbox.
3. Now return to <https://developer.paypal.com> and sign in with the credentials that you've just set up. The PayPal Sandbox page is displayed (see Figure 13-4).



FIGURE 13-4 The PayPal Sandbox account.

4. On the left side of the screen, click Test Accounts. You'll see options to create either a preconfigured test account or one that you can create manually (see Figure 13-5).

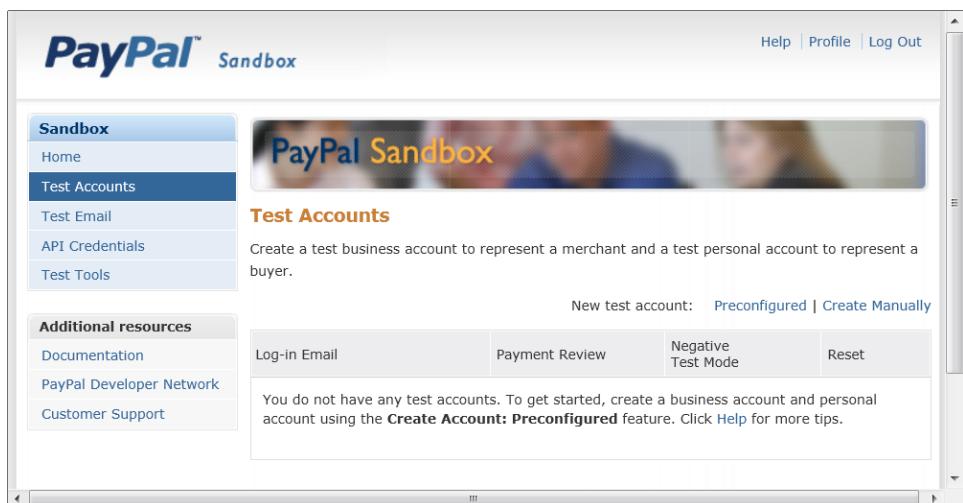


FIGURE 13-5 Creating a PayPal test account.

5. Select Preconfigured. Then on the next screen, fill out the details for a Seller account. Note that for the email address, you can only specify the first six characters of the address. You'll see in a moment how these are used. Don't worry if you can't configure your full email address.

When you're done, you'll be returned to the list, and you'll see that an account has been set up for you (see Figure 13-6).

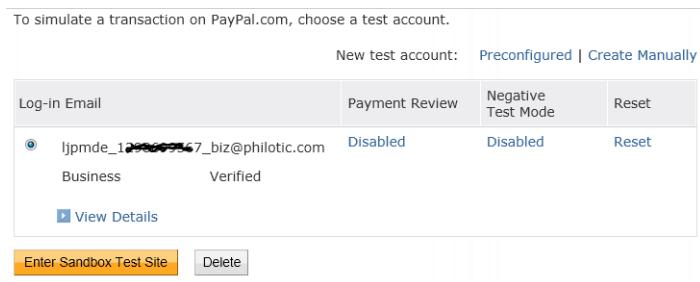


FIGURE 13-6 Your Sandbox account.

6. Next, on the left side of the screen, click API Credentials. You'll be taken to a screen with your API Username, API Password, and Signature (see Figure 13-7).

The screenshot shows the "API Credentials" section of the PayPal Sandbox. The left sidebar has a "Sandbox" menu with options: Home, Test Accounts, Test Email, API Credentials (which is selected and highlighted in blue), and Test Tools. The main content area has a header "PayPal Sandbox" and a sub-header "API Credentials". It states: "You must have credentials to test APIs for Website Payments Pro and Express Checkout in the Sandbox. In most cases, you will use API signatures and not download certificates." Below this, it says: "The test accounts identified below are enabled for API access." A note states: "Note: These credentials will not work outside the Sandbox. You will need new credentials from paypal.com to go live." A table lists the test account details:

Test Account	Date Created
Test Account: ljpmd...@philotic.com	Feb. 25, 2011 21:52:56 PST
API Username: ljpmd..._biz_api1.philotic.com	
API Password: 1234567890	
Signature: A.6JS...8321...8321...HIM6	

At the bottom, a note says: "To download the certificate, log into the sandbox test account profile and remove the 3 token credentials associated with the account. For more information, refer to the Sandbox User Guide."

FIGURE 13-7 Your API credentials.

Take note of these—you'll need them in the next section when building your application to use PayPal.

7. While you are here, create another test account, this time a personal one, and take note of the Test Account user name and password.

Using PayPal with WebMatrix

In this section, you'll see how to use PayPal with WebMatrix. You'll use the Bakery template and adjust it to be used with PayPal.

1. Create an instance of the Bakery site by launching WebMatrix and selecting New Site From Template.
2. Call the site **BakeryPP**. Run the site and enter its administration mode by appending `_Admin` to the default URL. Use this to then download the PayPal helpers package. If you aren't familiar with this process, look back at Chapter 12 where you stepped through how to do it for the Facebook helpers, and just do the same thing with the latest versions of the PayPal helpers.
3. If you're not familiar with the Bakery sample, run it now and take a quick look. You'll see that you can browse and order products from a fictitious bakery site called Fourth Coffee (see Figure 13-8).

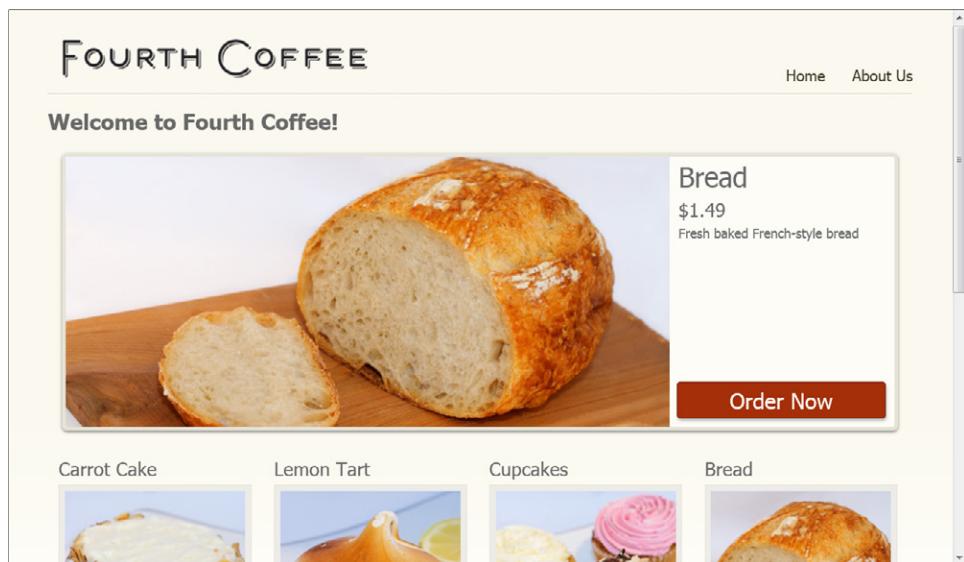


FIGURE 13-8 The Bakery site.

When you click the Order Now button, you'll be taken to a screen where you can fill out a form to order the product (see Figure 13-9).

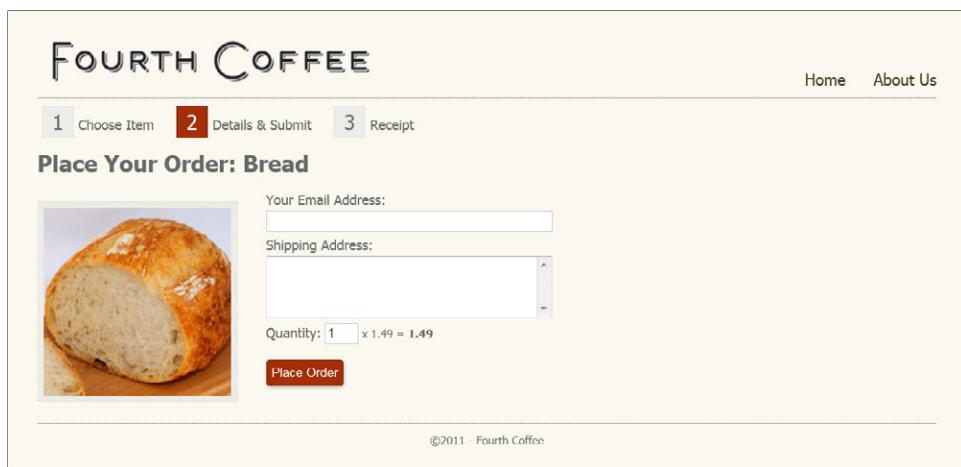


FIGURE 13-9 Ordering an item from Fourth Coffee.

Instead of this custom form, which doesn't have any type of "real" payment or realization system, you'll replace this with one that uses a shopping cart from PayPal.

Initializing the PayPal Helper

1. The first step is to initialize the PayPal helper by using the API credentials that you got from the PayPal Sandbox. To do this, create a new file called **_AppStart.cshtml** in your Bakery site.
2. Delete the contents of this file, and replace them with the following:

```
@{
    PayPal.Profile.Initialize(
        "[Value of API Username]",
        "[API Password]",
        "[API Signature]",
        "sandbox");
    // General Adaptive Payments' properties
    PayPal.Profile.Language = "en_US";
    PayPal.Profile.CancelUrl = "http://www.mystore.com/ohtoobad.cshtml";
    PayPal.Profile.ReturnUrl = "http://www.mystore.com/thanks.cshtml";
    PayPal.Profile.IpnUrl = "http://www.mystore.com/notifications.cshtml";
    PayPal.Profile.CurrencyCode = "USD";
}
```

The first parameter is always confusing. Make sure that you use the API Username, which is the second value on the screen in Figure 13-7. The API Password and the Signature are quite straightforward. This configures the environment for you.



Note The PayPal API provides the ability to add URLs for where the flow should redirect in multiple payment circumstances.

- ❑ The *CancelUrl* is the address of the page that PayPal will go to if the user cancels the transaction. You can host a page at this URL to tell the user what to do next.
- ❑ The *ReturnUrl* is the address of the page that PayPal will go to after the payment is successful. It should be a "Thank You" style page.
- ❑ PayPal has a system called Instant Payment Notification (IPN), which fires an asynchronous notification after payment completes successfully. IPN is beyond the scope of this book, so check the PayPal documentation for more details, but you can set up the IPN URL with the *lpnURL* property.

Creating a Shopping Cart

1. Create a new CSHTML page called **PayPalOrder.cshtml**. Delete its entire content and replace it with the following code:

```
@{  
    Page.Title = "Place Your Order";  
  
    var db = Database.Open("bakery");  
    var productId = UrlData[0].AsInt();  
    var product = db.QuerySingle("SELECT * FROM PRODUCTS WHERE ID = @0",  
        productId);  
  
    if (product == null) {  
        Response.Redirect("~/");  
    }  
  
    var paypalButton = PayPal.ButtonManager.AddToCartButton.Create(  
        "[Value of test account]",  
        product.Name,  
        string.Format("{0:f}", product.Price));  
  
    HtmlString paypalButtonHtml = new HtmlString(paypalButton.WebSiteCode);  
}  
<h1>Place Your Order: @product.Name for $@product.Price</h1>  
  
  
  
@paypalButtonHtml
```



Note To get the value for the test account, look back at the API Credentials screen (see Figure 13-7).

2. Now go back to Default.cshtml and edit the two instances of the following:

```
href="@Href("~/order", featured.Id)"
```

Replace them with this:

```
href="@Href("~/paypalorder", featured.Id)"
```

This simply changes the URL that the browser opens when you select the Order Now buttons on the main screen, and points them to the new PayPalOrder page that you just created. Let's take a look at the behavior first, in the next section, and then you'll dissect the PayPalOrder page to see how it works.

Running the PayPal-Enabled Bakery

1. Run the Bakery site, and click the Order button for any of the products on the default screen. You'll see the new PayPal-powered page, which should look something like Figure 13-10.

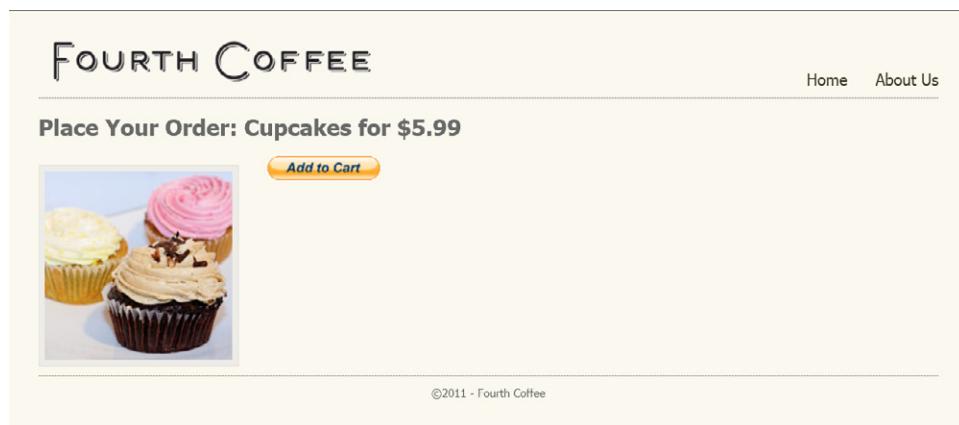


FIGURE 13-10 The PayPal-enabled order page.

2. The implementation of this page is a lot simpler than the original, because you don't need to have a form capturing the order details, delivery address, and so on. It's all done for you by PayPal. Click the Add To Cart button to see the shopping cart with this product and its cost (see Figure 13-11).

The screenshot shows a shopping cart page titled "Laurence Moroney's Test Store". The cart contains one item, "Cupcakes", with a quantity of 1 and a total price of \$5.99 USD. The page includes a "PayPal" logo and a "Secure Payments" link. Buttons for "Update cart", "Continue shopping", and "Proceed to checkout" are visible. A note at the bottom encourages using PayPal for safety and ease of payment.

FIGURE 13-11 The PayPal shopping cart.

The shopping cart maintains its state, so if you close the window and go back to the bakery to add some more products, you'll be able to see them added to the cart, along with a running total.

- When you proceed to the checkout, you're given the option to sign in to PayPal to then pay the merchant (see Figure 13-12). Sign in with the Personal account that you set up earlier in the chapter. This is a test user who is signing in to the Sandbox to pay your test business.

The screenshot shows the PayPal login interface for the Sandbox. It features a "LOG IN TO PAYPAL" form with fields for "Email" (containing "lpmde_1298702373_per@phil") and "Password" (containing masked text). A "Log In" button is present. Below the form, links for forgot email or password are shown. The background includes a note about PayPal securely processing payments for "Laurence Moroney's Test Store" and icons for various payment methods like VISA, MasterCard, American Express, and Discover.

FIGURE 13-12 Signing in to the Sandbox to pay your business.

4. You'll see a confirmation page. Click Pay Now at the bottom, and the simulated payment is sent. You can see the results of this in Figure 13-13.

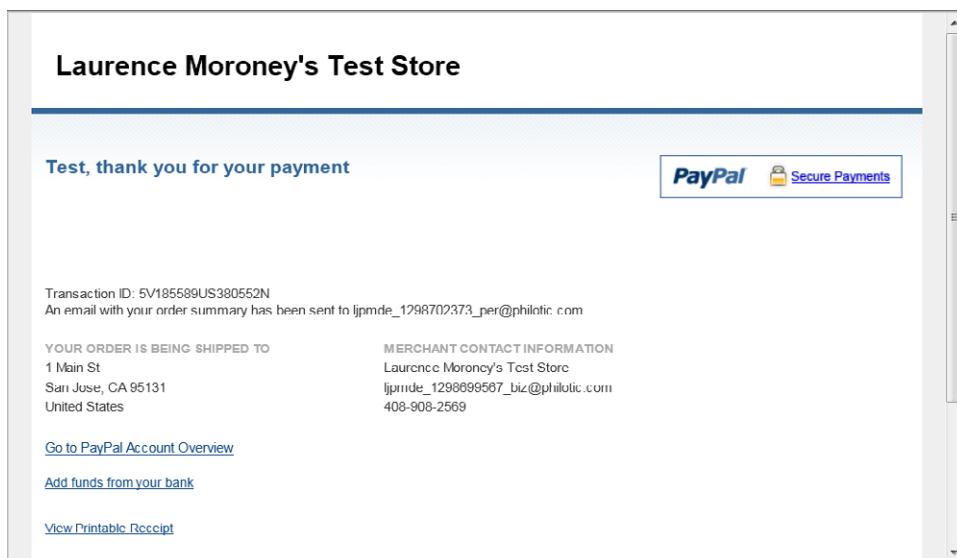


FIGURE 13-13 Confirmation page for the test payment.

5. Verify that the payments worked by looking at the Sandbox site and clicking Test Email. You'll see all the accounts' emails, including the receipt for the payment sent from one account and the payment received by the other. Remember, you set up two accounts, the business one for receiving the payment and the personal one for sending the payment (see Figure 13-14).

Test Email			
Test account email addresses are not real. Email sent to them is never delivered outside the Sandbox.			
Below are the most recent messages sent to your Sandbox test accounts.			
Inbox			
To	From	Subject	Date
ljmpmde_1298702373_per@phiservice@paypal.com lotic.com		Receipt for Your Payment to Laurence Moroney's Test Store	Feb. 25, 2011 22:43:27 PST
ljmpmde_1298699567_biz@phi lotic.com	r@philotic.com	Notification of payment received	Feb. 25, 2011 22:40:50 PST
ljmpmde_1298702373_per@phiservice@paypal.com lotic.com		Receipt for Your Payment to Laurence Moroney's Test Store	Feb. 25, 2011 22:40:13 PST
ljmpmde_1298699567_biz@phi service@paypal.com lotic.com		You have successfully lifted your PayPal withdrawal limit	Feb. 25, 2011 21:52:55 PST

FIGURE 13-14 The email paper trail.

Let's now take a look at how this all works. You'll explore your PayPalOrder.cshtml page in the next section.

Exploring the PayPalOrder.cshtml Page

Earlier, you created the PayPalOrder.cshtml page and added code to it without really looking at what the code did or how it worked. So in this section, you'll look at it in a little more detail to get an understanding of how it works.

First of all, the page was loaded using a friendly URL. You might have noticed that it wasn't called using a `/PayPalOrder.cshtml?param1=something¶m2=something` methodology—instead it was called using `/PayPalOrder/X`, where `X` is a number. This number is the ID of the product that will be rendered for sale on the page:

```
The ID of the product can be retrieved using the UrlData list.  
var productId = UrlData[0].AsInt();
```

This takes the first parameter in the URL and loads it into a variable called `productId`.

Next, the product with this ID is selected from the database:

```
var product = db.QuerySingle("SELECT * FROM PRODUCTS WHERE ID = @0", productId);
```

Now it's time to use the PayPal helper for the first time. The PayPal helper generates HTML for you, so it's a good idea to load this into a variable, like the following:

```
var paypalButton = PayPal.ButtonManager.AddToCartButton.Create(  
    "ljpmdes_1298699567_biz@philotic.com",  
    product.Name,  
    string.Format("{0:f}", product.Price));
```

The PayPal Button Manager provides several different buttons. You'll see the others later in this chapter, but in this case, it generates an `AddToCartButton`, which works with a server-side shopping cart, as you saw demonstrated in the previous section. It takes three parameters:

1. The email address of the owner of the store that you are buying from
2. The name of the product as it will be rendered on the shopping cart
3. A string with the price of the product

The name of the product is easily accessed using the `product` object; using `product.Name` gets the string containing the name. The price should be passed to PayPal as a string, so `string.Format` is used to do the conversion. All this is used to create an object called `payPalButton`:

```
HtmlString payPalButtonHtml = new HtmlString(payPalButton.WebSiteCode);
```

This object has a `WebSiteCode` property, which can then be used to initialize an `HtmlString` called `payPalButtonHtml`. This can be added to the page by using `@payPalButtonHtml` to render the HTML. The rest of the HTML is here, and it's pretty straightforward. It just shows an `<h1>` tag asking you to place an order for the product at the price for that product. It then renders the image of the product from the database and writes out the HTML contained in the `payPalButtonHtml` variable that was generated for you by the helper:

```
<h1>Place Your Order: @product.Name for $@product.Price</h1>



@payPalButtonHtml
```

And that's it! It's pretty simple, right?

Setting Up Other Types of Payment

In the previous section, you saw how to create a shopping cart by using PayPal. The `ButtonManager` can be used to set up other forms of payment styles too. These are the Buy Now button, which is a one-off payment for a single item, where no shopping cart is needed; the Donate button, which is used to donate money for something as opposed to using it to purchase something; and the Subscribe button, which is used to set up a periodic payment for something.

The Buy Now Button

Building an application for a single-item purchase as opposed to a shopping cart is very straightforward. For the purposes of this sample, you can simply replace the Add To Cart button with a Buy Now button by changing one line of code.

1. Open the PayPalOrder.cshtml page that you created earlier and find the following code:

```
var paypalButton = PayPal.ButtonManager.AddToCartButton.Create(  
    "1jpmde_1298699567_biz@philotic.com",  
    product.Name,  
    string.Format("{0:f}", product.Price));
```

2. Replace the *AddToCartButton.Create* method with a *BuyNowButton.Create* method:

```
var paypalButton = PayPal.ButtonManager.BuyNowButton.Create(  
    "1jpmde_1298699567_biz@philotic.com",  
    product.Name,  
    string.Format("{0:f}", product.Price));
```

3. Now run the site and try to order a product. Instead of an Add To Cart button, you'll see a simple Buy Now button, as shown in Figure 13-15.

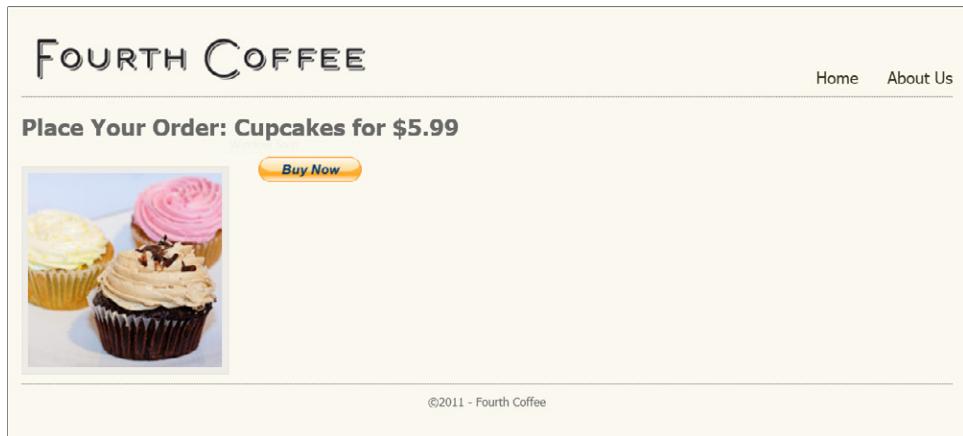


FIGURE 13-15 The Buy Now button.

4. Click the Buy Now button. You'll have a slightly different experience than earlier. Instead of displaying a shopping cart, you'll go directly to the PayPal sign-in to pay for the transaction. You can see this in Figure 13-16. Note the line "Cupcakes Total: \$5.99 USD" at the top of the screen, which matches the order.

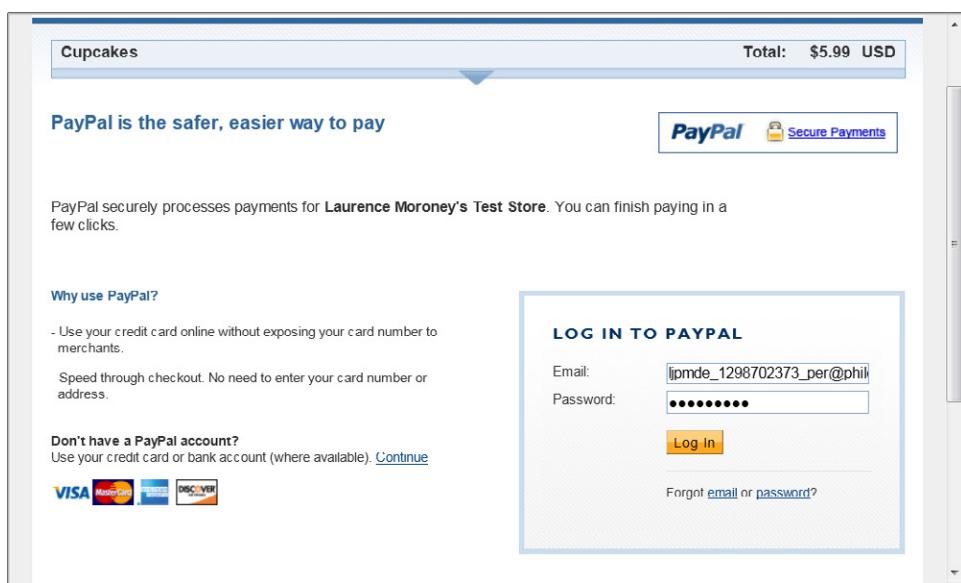


FIGURE 13-16 Paying for the Buy Now item.

And then, as before, you'll get a confirmation screen on which you can click the Pay Now button to make the payment.

This is useful if you are just selling a single item to your end users and don't want to give them the overhead of a shopping cart for multiple items.

The Donate Button

Many websites don't offer goods for sale but want to accept money from their customers. For example, an open source community might accept donations from people who use its product, or a charitable organization might want to use the Internet to accept financial help.

Implementation using the *ButtonManager* is very straightforward. In fact, it works in exactly the same way as the *AddToCart* and *BuyNow* scenarios. You'll handle it a little differently though, because you probably won't have a product associated with the donation, so you don't need to pass a product name as one of the parameters.

For example, the following shows what a Donate button could look like on the About page for the Fourth Coffee site:

```
var paypalButton = PayPal.ButtonManager.DonateButton.Create(
    "lpmde_1298699567_biz@philotic.com",
    "Donate $5 to Fourth Coffee",
    "5.00");

HtmlString paypalButtonHtml = new HtmlString(paypalButton.WebSiteCode);
```

Here you can see that the message “Donate \$5 to Fourth Coffee” is used in place of a product, and the amount is hardcoded to 5.00.

So, if the `paypalButtonHtml` is output on this page, it will look something like Figure 13-17.

A screenshot of a website page. The title is "A little bit about Fourth Coffee". Below the title is a paragraph of text: "Fourth Coffee was founded in 2010 and delivers coffee and fresh baked goods right to your door. In another life, Bill Baker was a developer by day and pastry chef by night. But soon Bill's innate skills with all things involving butter, flour and sugar put him even more in demand than his programming talents and what started out as a way to satisfy his own sweet tooth became all-consuming. Fourth Coffee is not only a candy-coated wonderland of coffee, pastries, cookies and cakes, it also honors his tech background by employing a state of the art online ordering system that makes it easy for anybody with internet access to order his all natural, locally-sourced confections and have them delivered to their door within 24 hours. If you like our site, please click here to donate \$5 and buy us a coffee!" At the bottom of the text block is a yellow "Donate" button.

FIGURE 13-17 Using the Donate button.

Now when you click the Donate button, the PayPal window will be tailored for the donation experience. There’s no concept of *buying* anything, and the text reads “Donate \$5 to Fourth Coffee,” as shown in Figure 13-18.

A screenshot of a PayPal donation page. The header says "Laurence Moroney's Test Store". The main heading is "Donate \$5 to Fourth Coffee" with "Total: \$5.00 USD" to its right. Below this is a "PayPal Secure Payments" button. A note says "PayPal securely processes donations for Laurence Moroney's Test Store. You can complete your payment with just a few clicks." To the left, there's a "Why use PayPal?" section with a bulleted list: "It's easy to send money and shop online", "You can donate without sharing your financial information", and "Over 50,000 online merchants accept PayPal". Below that is a "Don't have a PayPal account?" section with a note "Use your credit card or bank account (where available). [Continue](#)". To the right is a "LOG IN TO PAYPAL" form with fields for Email (lpmde_1298702373_per@phil) and Password (redacted), and a "Log In" button. Below the log in form is a link "Forgot [email address or password?](#)".

FIGURE 13-18 The experience of donating with the Donate button.

You’ll also see that the text and the confirmation button on the final verification screen are associated with donations instead of purchases. You can see this in Figure 13-19.

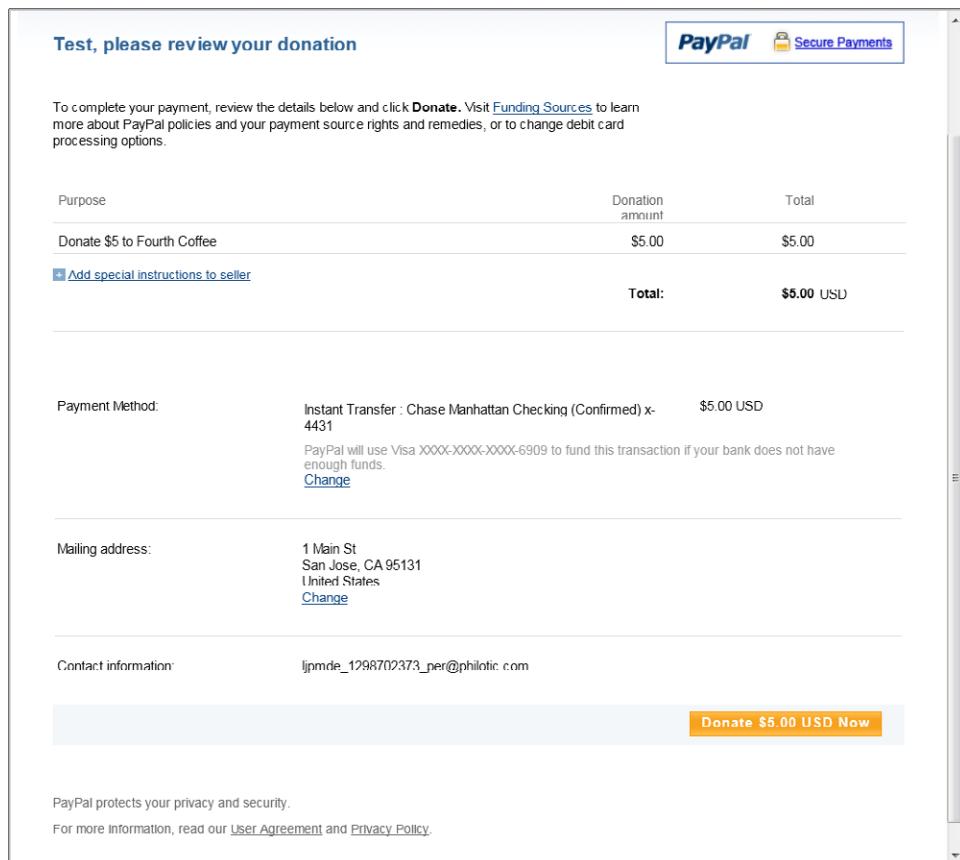


FIGURE 13-19 The Donation confirmation screen.

The design goal is for the developer to have a similar experience regardless of the button type. As you can see, this is achieved, and it's pretty straightforward to do, but differences in type of payment will dictate how you use the button—such as in the preceding “donate” example, where you wouldn't want to use a product name, but something a little more friendly!

The Subscribe Button

PayPal can also be used to trigger periodic payments for subscriptions or dues; you give PayPal permission to take money out of your account and send it to the payee on a periodic basis.

This works a little differently than the buttons you've seen earlier, in that specifying a price isn't enough—you also have to specify the number of payments and the frequency of those payments.

So, consider the following code:

```
var paypalButton = PayPal.ButtonManager.SubscribeButton.Create(
    "ljpmdc_1298699567_biz@philotic.com",
    "Test Subscription",
    "5.00", "12", "M");

HtmlString paypalButtonHtml = new HtmlString(paypalButton.WebSiteCode);
```

The parameters are *5.00* (for cost), *12* (for payments), and *M* (for period). This sets up a subscription of \$5.00 per month for 12 months, after which it would have to be renewed.

The allowed periods are:

- **Days ("D")** Can be from 1 to 90
- **Weeks ("W")** Can be from 1 to 52
- **Months ("M")** Can be from 1 to 24
- **Years ("Y")** Can be from 1 to 5

So, for example, if you want to do \$5.00 per month for 12 months, you would have *5.00*, *12*, and *M* for the final three parameters.

When you run this and click the Subscribe button, you'll see something like Figure 13-20.

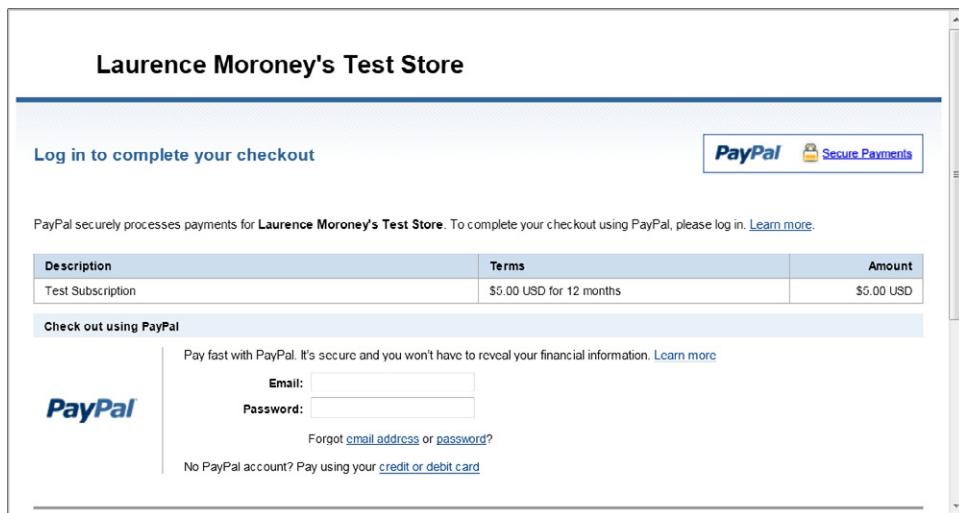


FIGURE 13-20 Using a subscription.

Now when the users sign in, they are setting up to have PayPal pay your business regularly as specified.

Going Further

The previous sections showed how you can use the PayPal helper with WebMatrix to create buttons for shopping carts, single-item purchases, donations, and subscriptions. To create them, you just used the default buttons. The good news is that the PayPal helper provides a lot more sophisticated functionality for custom buttons, and the ability to tweak everything about them to your heart's desire. It also has an adaptive payments interface that provides more sophisticated scenarios, such as having incoming payments trigger a payment to your suppliers and more. To explore in more detail, take a look at the PayPalHelper/Docs folder in your WebMatrix project after you've installed the helpers.

Going Live

Up to now, you've been looking at building PayPal sites that use the PayPal Sandbox.

To go live, you'll need to get API details that work in production. You can get these at https://www.paypal.com/us/cgi-bin/webscr?cmd=_profile-api-add-direct-access.

1. Sign in to your real PayPal account (not the Sandbox), and you should see something like Figure 13-21.

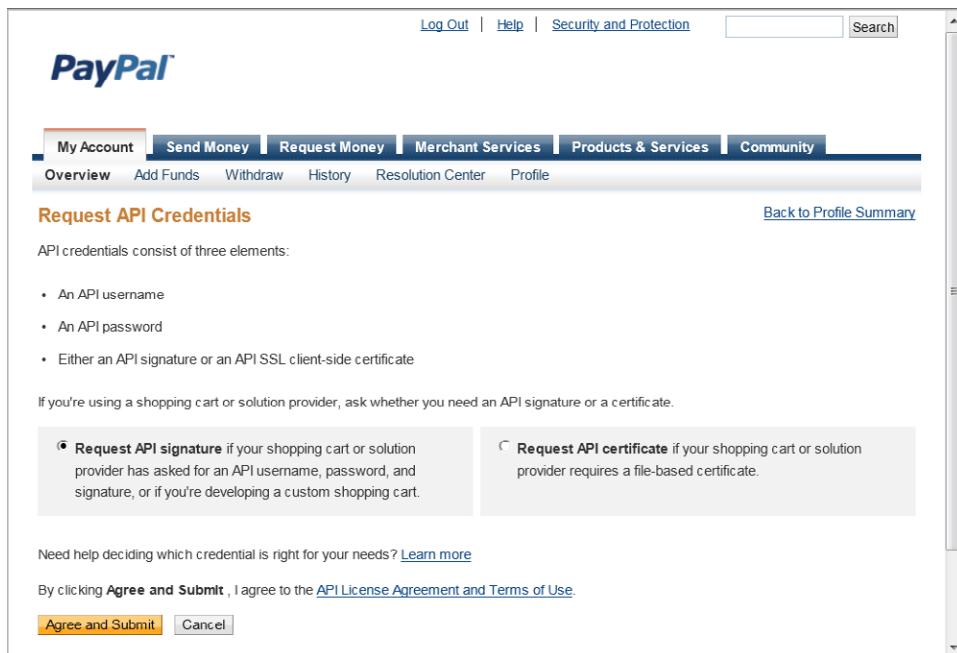


FIGURE 13-21 Getting live API access.

2. Select Request API Signature, and then click Agree And Submit. You'll now get your production API details, to replace the Sandbox ones that you created earlier. In `_AppStart.cshtml`, be sure to use *production* rather than *sandbox* for the final parameter of the `PayPal.Profile.Initialize()` call.

You're now ready to accept real payments from real people over the Internet, securely!

Summary

This chapter introduced the PayPal helpers for WebMatrix. You saw how to use them to easily add e-commerce functionality to your websites. You adapted the WebMatrix Bakery sample to remove the custom realization and replace it with a hosted shopping cart on PayPal. You also saw how to use Buy It Now buttons for single item payments; Donation buttons to allow for nonpurchase payment; and Subscribe to set up multiple regular payments. Although that's a considerable amount of functionality already, you've only scratched the surface of what's possible using PayPal and WebMatrix, but hopefully you've seen enough to get you interested in exploring more on your own!

Over the last couple of chapters, you've been using some of the more popular helpers available with WebMatrix. It's easy to build your own too, and you'll see how to do this in the next chapter!

Chapter 14

Building Your Own Web Helpers

In this chapter, you will:

- See how to use the Microsoft Translator widget.
- Create a helper for the Microsoft Translator widget in CSHTML.
- Create a helper by using the Translator API.

In earlier chapters, you looked at some of the web helpers that come with Microsoft WebMatrix or that are available in the NuGet gallery. In this chapter, you'll explore how you can use the Microsoft ASP.NET Web Pages Framework to create your own helpers. The helpers in the NuGet gallery are compiled as DLL files and packaged using NuGet, which is beyond the scope of this book. (This book just focuses on WebMatrix, but you will understand the concepts for building such helpers by using Razor after reading this chapter.) You'll proceed step by step through the process of building a helper that provides webpage translation.

Web helpers are designed to encapsulate common, complex functionality into simple solutions that often require only a single line of code. On execution, this simple code typically writes HTML to your page. In this chapter, you'll see how to add a Microsoft Translator widget to your page. The Microsoft Translator widget is implemented using a chunk of HTML and some JavaScript. You'll create two helpers for the widget, one by using the Translator API.

To work through the examples in this chapter, you'll be using the Bakery site template. You'll add translation features to that site.

Before exploring the following topics, go ahead and create a new site based on the Bakery template, and name it **BakeryTrans**.

Using the Microsoft Translator Widget

First, you'll look at the widget and what it does, and then you'll see how to implement it as a helper.

1. To get the Microsoft Translator widget, go to <http://www.microsofttranslator.com/widget/>. On this webpage, you can configure the widget and its behavior, as shown in Figure 14-1.

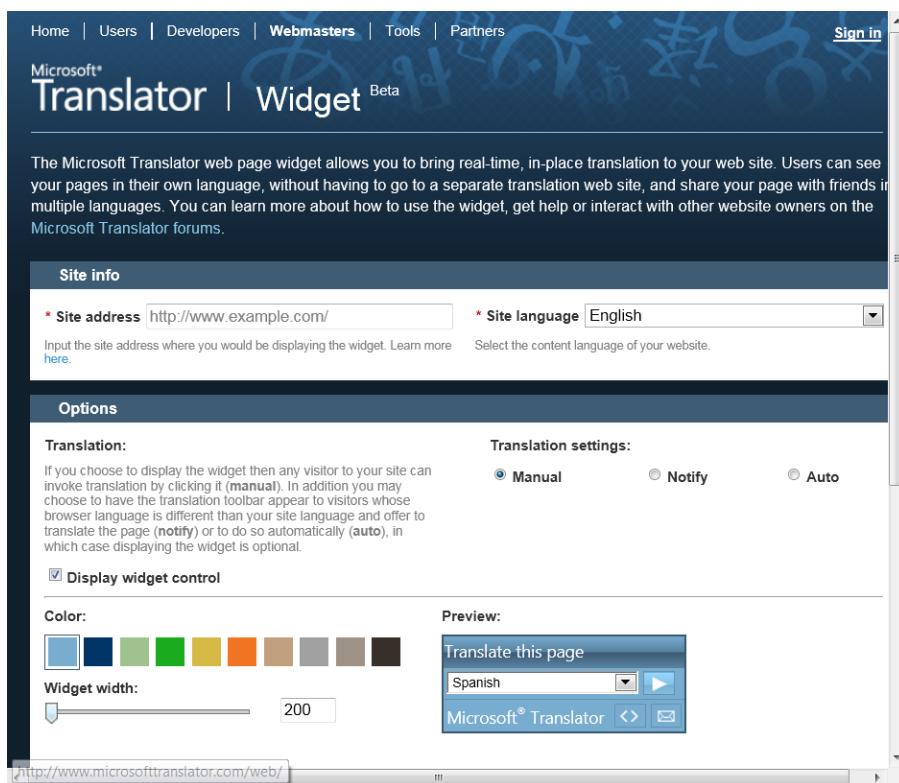


FIGURE 14-1 The Microsoft Translator widget page.

2. From here, you can set up the details for your page. Enter the site address (`http://localhost` is fine if you're developing) and the default language for your site. Then specify the different translation settings:
 - Manual** Renders the widget on the page, and allows users to select the language that they want to translate to.
 - Notify** Detects if the browser is a different language from the page, and if so, requests whether the user wants the page to be translated. In this case, the widget is not rendered.
 - Auto** Detects if the browser is a different language from the page, and if so, translates the page. In this case, as with Notify, the widget is not rendered.

For the purposes of this exercise, select Manual, and then pick a color for the widget.

3. Now scroll to the bottom of the page, where you'll see a check box that you need to select to agree to the terms of use for the widget. If you agree with the terms, select the check box, and then click Generate Code. You'll see some HTML that is generated for you (see Figure 14-2).



FIGURE 14-2 Generating the widget code.

4. Copy the HTML. Then go back to the BakeryTrans project and open the About.cshtml file.
5. At the bottom of the page, before the closing `</p>` tag, paste in the HTML that you just copied. Your page should look something like Figure 14-3.

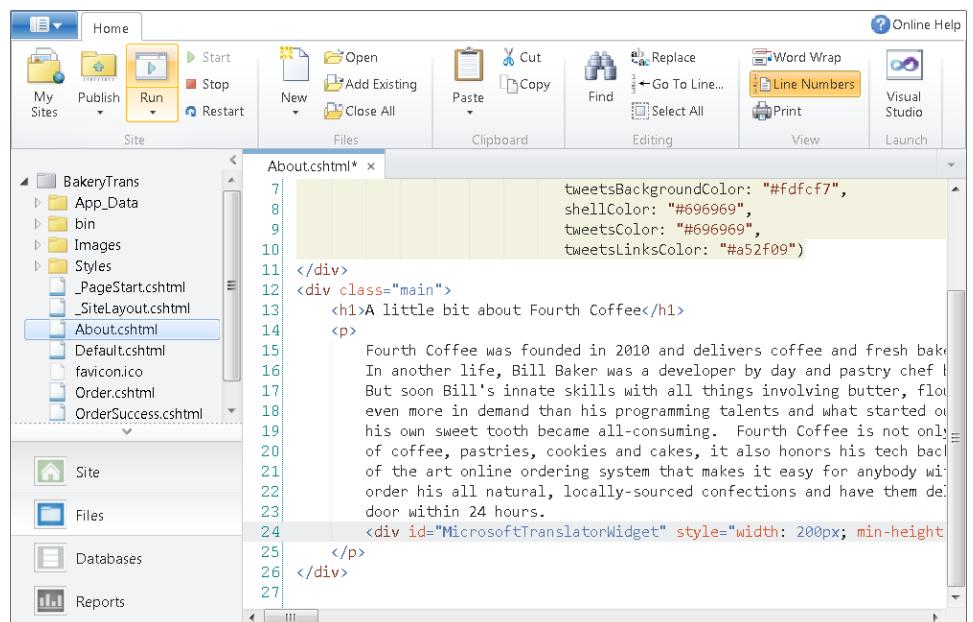


FIGURE 14-3 Adding the widget to your About.cshtml page.

6. Run the bakery app and go to the About page. You'll see the widget at the bottom of the About text (see Figure 14-4).

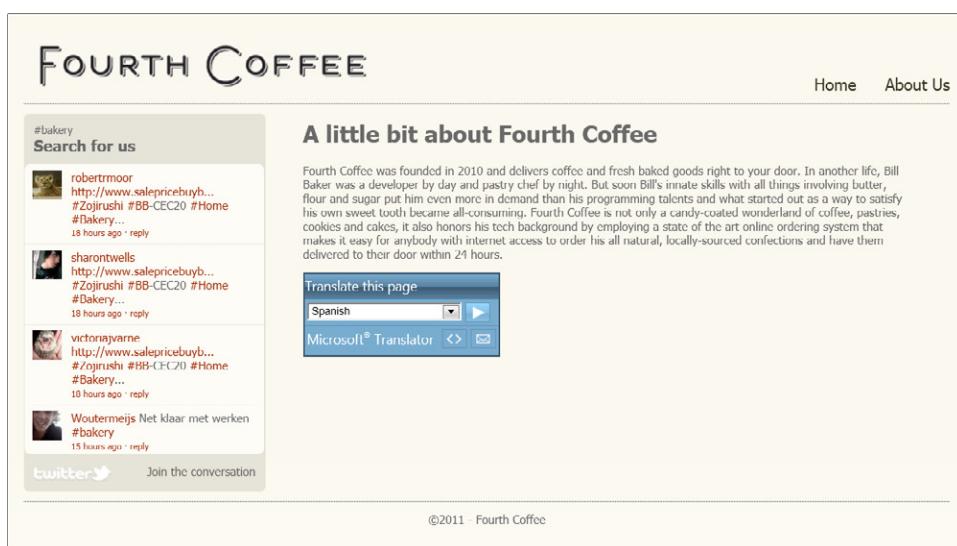


FIGURE 14-4 The widget on your page.

Now users can choose a language from the Translate This Page list, and click the right-pointing arrow next to the list to automatically translate the page into their language of choice. Figure 14-5 shows the page translated into Japanese.



FIGURE 14-5 The translated Fourth Coffee page.

A couple of things to note here. First, the page logo *Fourth Coffee* is a picture, which of course cannot be translated by this engine, so if you want to support machine translation on your page, you should consider this when designing the site. More impressively, the widget is able to translate the Twitter stream too—because the stream was implemented as a web helper, which outputs HTML and JavaScript. This is another great reason to use Razor and web helpers for building your site. If your Twitter feed had been implemented as a binary object using something like Adobe Flash, as is commonly done, you wouldn’t be able to translate it this easily.

Creating a Helper for the Widget

Helpers can be implemented very easily by using CSHTML or VBHTML pages, which use the Microsoft Visual C# and Microsoft Visual Basic syntaxes respectively (though we will focus on the former in this book) using the Razor syntax. The only thing you’ll need to remember is that they must be created within the App_Code folder. If you’re using the BakeryTrans site as created earlier in this chapter, you won’t have this folder.

1. Go ahead and create the App_Code folder now by right-clicking the project name in the Files workspace (BakeryTrans in this case) and selecting New Folder. After you’ve done this, rename your new folder **App_Code**.
2. Now right-click the App_Code folder, and select New File. In the New File Type dialog box, select CSHTML as the file type, and call your new file **Translator.cshtml**.

Your folder structure should now look something like Figure 14-6.

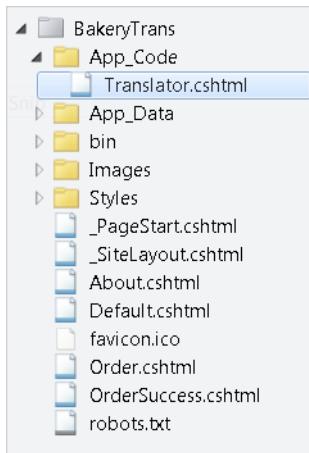


FIGURE 14-6 The BakeryTrans folder structure.

3. At this point, you're ready to implement the code for `Translator.cshtml`. First, remove everything that is in the file and replace it with the following code:

```
@helper GetWidget()  
{  
}
```

4. Now, within the braces, paste the code that you got on the Microsoft Translator widget site. Your helper should now look something like the following:

```
@helper GetWidget()  
{  
    <div id="MicrosoftTranslatorWidget" style="width: 208px; min-height: 83px;  
        border-color: #3A5770; background-color: #78ADD0;"><noscript>  
        <a href="http://www.microsofttranslator.com/bv.aspx?a=http%3a%2f%2flocalhost%  
        2f">  
            Translate this page</a><br />Powered by  
            <a href="http://www.microsofttranslator.com">  
                Microsoft® Translator</a></noscript></div>  
            <script type="text/javascript"> /*  
            <![CDATA[ /*  
                setTimeout(function() {  
                    var s = document.createElement("script");  
                    s.type = "text/javascript";  
                    s.charset = "UTF-8";  
                    s.src = ((location && location.href && location.href.indexOf('https') == 0)  
                        ? "https://ssl.microsofttranslator.com"  
                        : "http://www.microsofttranslator.com" ) +  
                            "/ajax/v2/widget.aspx?mode=auto&from=en&layout=ts";  
                    var p = document.getElementsByTagName('head')[0] ||  
                        document.documentElement; p.insertBefore(s, p.firstChild); }, 0); /* ]]>  
            */ </script>  
    }  
}
```

This gives you everything you need to have the helper produce the widget for you.

5. Now, go back to your `About.cshtml` page and replace the HTML for the widget with the following:

```
@Translator.GetWidget()
```

The syntax starts with `Translator` because that's the name of the file (`Translator.cshtml`), and the method (`GetWidget()`) is the name of the helper function you just created. So now if you run the site and visit the `About` page, you'll see the widget in exactly the same way as you did when you pasted in the raw code, but I'm sure you'll agree that the code in this version is a lot cleaner.

Congratulations, you've just created your first helper. In the next section, let's build something a little more complex than a helper that just spits out static HTML.

Creating a Helper by Using the Translator API

In the previous section, you created a simple helper that was used to render the Microsoft Translator widget. In addition to the widget, Microsoft Translator offers an API that can be used to translate anything, giving you more fine-grained control over your translation. In this section, you'll create a helper that uses this API to translate text from one language to another. The API offers a lot of functionality, and we're just going to barely scratch the surface of it, but this discussion will give you the foundation for not just doing more translation work but also building other helpers.

Getting an API Key

To get started, you'll need an API key from Bing in order to use the Translator API.

1. To get an API key, simply visit <http://www.bing.com/developers/appids.aspx> and sign in with a Windows Live ID. You'll see a screen like Figure 14-7. In this case, I have several IDs, so the screen might look a little different if you don't have an App ID yet.

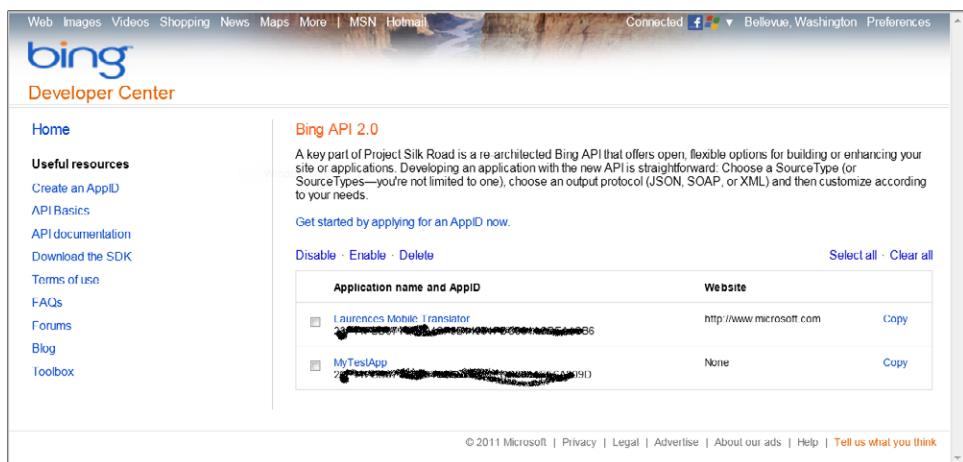


FIGURE 14-7 Getting a Bing ID.

2. Select the Get Started By Applying For An AppID Now link if you don't already have an App ID. You'll be asked to fill out a form with details of your application (see Figure 14-8).

Create a new AppID

To create a new AppID, enter your information below and click Agree to accept the API terms of use.

Required information

angular 500

Application name (60 character limit)

Description (500 character limit)

Company name (100 character limit)

Country/region (100 character limit)

Email address (e.g., webmaster@example.com)

We will use this address to notify you of issues that affect API usage.

I also want to receive promotional offers from Bing at this email address.

Optional information

Website (80 character limit)

FIGURE 14-8 Creating a new AppID.

3. After you've filled out the form, and if you agree to the terms and conditions, select the Agree button at the bottom of the page. You'll then get your new AppID, as you can see in Figure 14-9.

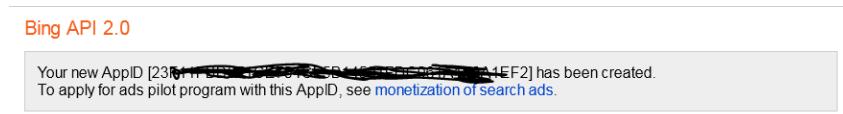


FIGURE 14-9 The AppID appears at the bottom of the screen (obscured here for security reasons).

This is now your Bing API key. Take note of it; you'll be needing it shortly.

Using the Translator API

Developers can use the Translator API in several ways. Each methodology is documented at <http://www.microsofttranslator.com/dev/>, which you can see in Figure 14-10.

The screenshot shows the Microsoft Translator Developer Offerings page. At the top, there's a navigation bar with links for Home, Users, Developers, Webmasters, Tools, Partners, a user name 'Laurence', and Sign out. Below the navigation is the Microsoft Translator logo and the title 'Translator | Developer Offerings'. A main content area contains a paragraph about the philosophy of translation as a tool, followed by sections for AJAX, SOAP, and HTTP interfaces, each with a brief description and a 'More' link. To the right of these are links for the Interactive SDK, Getting started guide (ASP.NET), News, Feedback & Support, and MSDN reference for Microsoft Translator APIs. At the bottom, there's a URL bar showing 'http://www.microsofttranslator.com/web/.../val', a 'Translator Help | Feedback' link, and a 'Feedback' link.

FIGURE 14-10 The Microsoft Translator Developer Offerings page.

There's a lot of great documentation and samples here, including how to use the different interfaces (AJAX, SOAP, and HTTP).

1. Our helper will use the HTTP interface, so select the HTTP Interface link at the left of the page.

You'll be taken to an MSDN page containing documentation on the HTTP interface. You can see this in Figure 14-11.

The screenshot shows a MSDN page for the Microsoft Translator API. At the top, there's a navigation bar with links for Home, Library (which is selected), Learn, Downloads, Support, Community, Sign in, United States - English, and Preferences. A search bar says "Search MSDN with Bing". On the left, a sidebar lists categories like MSDN Library, Web Development, Microsoft Translator, V2, and HTTP, with "HTTP" expanded to show sub-methods: AddTranslation Method, AddTranslationArray Method, BreakSentences Method, Detect Method, DetectArray Method, GetAppIdToken Method, GetLanguageNames Method, GetLanguagesForSpeak Method, GetLanguagesForTranslate Method, GetTranslations Method, GetTranslationsArray Method, Speak Method, Translate Method, and TranslateArray Method. The main content area has a title "HTTP" and a sub-section "Microsoft Translator V2". It contains instructions for using the HTTP API, mentioning XML requests to `http://api.microsofttranslator.com/V2/HTTP.svc` and parsing REST responses. Below this is a section titled "Public Methods" with a table:

Name	Description
Microsoft.Translator.AddTranslation Method	Adds a translation to the translation memory.
Microsoft.Translator.AddTranslationArray Method	Adds an array of translations to the translation memory.
Microsoft.Translator.BreakSentences Method	Returns an array of sentence lengths for each sentence of the given text.
Microsoft.Translator.Detect Method	Detects the language of a

FIGURE 14-11 MSDN documentation for the Microsoft Translator API.

2. Select the [Translate Method](#) link on the left, and you'll see the instructions on how to use it. At the bottom of the instructions page, there's a code sample that demonstrates how to use the method in C#. This is perfect, and you can use it as the basis for your helper (see Figure 14-12).

<i>contentType</i>	The format of the text being translated. The supported formats are "text/plain" and "text/html". Any HTML needs to be well-formed.
<i>category</i>	The category of the text to translate. The only supported category is "general".

Return Value

A string representing the translated text

Example

C#

```

string appId = "myAppId";
string text = "Translate this for me";
string from = "en";
string to = "fr";

string uri = "http://api.microsofttranslator.com/v2/Http.svc/
Translate?appId=" + appId +
"&text=" + text + "&from=" + from + "&to=" + to;
HttpWebRequest httpWebRequest = (HttpWebRequest)WebRequest.Cr
eate(uri);
WebResponse resp = httpWebRequest.GetResponse();
using (Stream strm = resp.GetResponseStream())
{
    System.Runtime.Serialization.DataContractSerializer dcs =
new System.Runtime.Serialization.DataContractSerializer(text.Get
Type());
    string translation = (string)dcs.ReadObject(strm);
    Response.Write("The translated text is: '" + translation
+ "'.");
}

```

FIGURE 14-12 MSDN documentation including sample code.

Take a note of this code, and together with your API key, you'll use it to finish off your helper.

Creating the Helper

You are going to use the `Translator.cshtml` file that you created in the previous section. Earlier, you created a function that returned raw HTML by using the `@helper` syntax. To get a little more control and consistency with standard programming, there's also an `@functions` syntax, which can be used to define typical functions with return values.

1. Let's create a function that returns a translation for a piece of text where we are translating from one language to another. Add the following code to `Translator.cshtml`:

```

@functions{
    public static string GetTranslation(string texttotrans,
        string langfrom,
        string langto)
    {
    }
}

```

It's pretty straightforward if you're familiar with programming in C# (or Visual Basic or Java, for that matter), where you create a public function that returns a string and that takes three parameters—the text to translate and the language from and to. Each of these is a string. You'll see how they're used in a moment.

2. Now let's add some initialization variables:

```
string appId = "[Put your appId value here]";
string translatedText = "";
string uri = "http://api.microsofttranslator.com/v2/Http.svc/Translate?appId=" +
    appId +
    "&text=" + texttotrans +
    "&from=" + langfrom +
    "&to=" + langto;
```

The first line simply sets up the *appId* that you created earlier. The second creates a string to hold your final translated text. The third creates a string to hold the URI for the service call to the Microsoft Translator service. The URI looks like this:

<http://api.microsofttranslator.com/v2/Http.svc/Translate>

The parameters are shown in the following table.

Parameter	Description
<i>appId</i>	Your AppId
<i>text</i>	The text to translate
<i>from</i>	The code for the language that the text is in (such as <i>en</i> for English)
<i>to</i>	The code for the language that you want to translate to (such as <i>jp</i> for Japanese)

The third line of code creates this URI by using the parameters that you pass in for the *text*, *from*, and *to* parameters, as well as the *appId* that you just created.

3. The next lines of code initiate the request to the Translator service and then call it. This generates a web request to the URI that you created earlier and gets its response. Add the following lines to the file:

```
HttpWebRequest request = (HttpWebRequest)WebRequest.Create(uri);
WebResponse response = request.GetResponse();
```

4. Now that you have the response, the rest of the operation is simply a matter of reading its contents. The service returns XML, so add the following code to strip out the surrounding angle bracket tags of the XML, giving you just the contents:

```
using (Stream strm = response.GetResponseStream())
{
    System.Runtime.Serialization.DataContractSerializer dcs =
        new System.Runtime.Serialization.DataContractSerializer(texttotrans.GetType());
    translatedText = (string)dcs.ReadObject(strm);
}
```

5. You now have the translated text, so it's simply a case of returning it to whatever called this function. Add the following line next:

```
return translatedText;
```

Before you look at how to use this helper, here's the full listing for the helper, including the widget:

```
@helper GetWidget()
{
    <div id="MicrosoftTranslatorWidget" style="width: 208px; min-height: 83px;
        border-color: #3A5770; background-color: #78ADD0;"><noscript><a href=
            "http://www.microsofttranslator.com/bv.aspx?a=http%3a%2f%localhost%2f">
            Translate this page</a><br />Powered by
            <a href="http://www.microsofttranslator.com">Microsoft® Translator</a>
            </noscript></div> <script type="text/javascript"> /* <!CDATA */
            setTimeout(function() { var s = document.createElement("script"); s.type =
                "text/javascript"; s.charset = "UTF-8"; s.src = ((location && location.href
&&
                location.href.indexOf('https') == 0) ? "https://ssl.microsofttranslator.com"
                :
                "http://www.microsofttranslator.com" ) +
                "/ajax/v2/widget.aspx?mode=auto&from=en&layout=ts"; var p =
                document.getElementsByTagName('head')[0] || document.documentElement;
                p.insertBefore(s, p.firstChild); }, 0); /* ]]> */ </script>
            }
@functions{
    public static string GetTranslation(string texttotrans,
                                         string langfrom,
                                         string langto)
    {
        string appId = "Enter your appID here]";
        string translatedText = "";

        string uri =
            "http://api.microsofttranslator.com/v2/Http.svc/Translate?appId=" +
        appId
        + "&text=" + texttotrans
        + "&from=" + langfrom
        + "&to=" + langto;

        HttpWebRequest request = (HttpWebRequest)WebRequest.Create(uri);
        WebResponse response = request.GetResponse();
        using (Stream strm = response.GetResponseStream())
        {
            System.Runtime.Serialization.DataContractSerializer dcs =
                new System.Runtime.Serialization.DataContractSerializer(
                    texttotrans.GetType());
            translatedText = (string)dcs.ReadObject(strm);
        }
        return translatedText;
    }
}
```

And that's the full helper. It's pretty straightforward. In the next section, you'll see how to use it.

Using the Helper

Using the helper couldn't be easier. You simply use the syntax `@HelperName.FunctionName(params)`. In this case, the *HelperName* is the name of the file containing the helper (*Translator.cshtml*), and the function is *GetTranslation*. The parameters are the text that you want to translate and the codes for the languages to translate from and to.



Note The list of supported languages is constantly being updated, but you can see the most current list by using the Microsoft Translator SDK at <http://sdk.microsofttranslator.com/HTTP/GetLanguagesForTranslate.aspx>.

For example, the English language code is *en* and the French one is *fr*, so to translate a piece of text from English to French using the helper, you'd use the following code:

```
@Translator.GetTranslation("Hello, I am French", "en", "fr")
```

This returns text (not HTML), so you should mark it up with HTML if you want it styled with your page. So, for example, you could use the following:

```
<h1>
    @Translator.GetTranslation("Hello, I am French", "en", "fr")
</h1>
```

The preceding code would render your translated text in an `<h1>` tag. Figure 14-13 shows an example of how it would look on the About page in the Bakery sample.

The screenshot shows the 'About Us' page of the Fourth Coffee website. At the top, there's a header with the logo 'FOURTH COFFEE'. Below the header, there's a sidebar on the left with a 'bakery' tag and a search bar labeled 'Search for us'. The main content area has a heading 'A little bit about Fourth Coffee'. Below this, there's a paragraph of text about the company's history and offerings. On the right side of the main content area, there's a section titled 'Bonjour, je suis français' with some text. At the bottom of the page, there's a footer with social media links for Twitter and Facebook, and a link to 'Join the conversation'.

FIGURE 14-13 Using the Microsoft Translator helper.

The text “Bonjour, je suis francais” is the translation of “Hello, I am French,” and it is rendered using an `<h1>` style from the Bakery site shown in Figure 14-13.

So now you’ve built your first helper using real C# code. You’ve seen how C# can be used within WebMatrix and how the Razor `@helper` and `@function` tags can encapsulate it.

The next step would be to use Microsoft Visual Studio to create a DLL file with your helpers in it. You can use NuGet to package your helper and publish it in a NuGet feed, but as mentioned earlier, that’s beyond the scope of this book; however, you can find good information on how to do that on the Nuget.org website.

Summary

This chapter introduced you to web helpers and how you can build them using CSHTML pages, C#, and Razor within WebMatrix. You first saw how to use the `@helper` syntax to create a simple helper that generates HTML, and then used that to render the Microsoft Translator widget. Next, you saw how to use the Microsoft Translator API with an API Key from Bing, and how to use the HTTP service exposed by the API to get a translation of any piece of text. Finally, you wrapped the code to translate text with a helper, using the `@function` syntax, and saw how to use that helper within your website.

In the next chapter, you’ll take a look at the integrated hosting gallery within WebMatrix and see how you can find a hosting provider that supports WebMatrix, as well as how to easily deploy your site and data.

Chapter 15

Deploying Your Site

In this chapter, you will:

- Discover how to find a web hosting provider.
- Use the Publish Settings dialog box to specify deployment parameters.
- Publish ASP.NET Web Pages and PHP/MySQL-based applications.

Though it's all very nice to have built your site and run it on *localhost*, often the most difficult part of building your site is *deploying* it to the Internet so that others can see and use it. Difficulties often arise with incompatibilities between the operating system you are running on your development machine (often Windows or Mac), what your Internet service provider (ISP) is running on its servers to service your website (often Windows Server or Linux), and the versions of the software that are being used. You might have one version of MySQL or ASP.NET or PHP on your development computer, but your ISP might have quite a different one on its servers.

Microsoft WebMatrix aims to help avoid this problem by putting a variety of ISP choices at your fingertips as you develop; and by including smart tools that can check for site compatibility against your ISP, helping to verify that your site will work after deployment. Finally, WebMatrix introduces a new technology called *Web Deploy* that, when used with compatible servers, makes deploying everything required for your site as easy as possible.

Finding Web Hosting

On the Sites workspace of WebMatrix, you'll find an option to find web hosting for your site. You can see this in Figure 15-1.

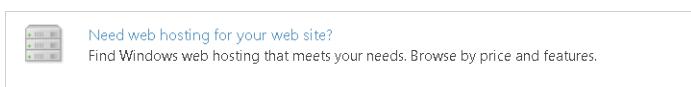


FIGURE 15-1 Using WebMatrix to find hosting for your site.

1. Click this link. You'll be taken to Microsoft's Hosting Gallery website, where you can see numerous offers for different types of hosting services. Typically, hosting services fall into three different categories:
 - ❑ **Shared hosting** In this model, one physical or virtual machine hosts many different websites. Therefore, this tends to be the cheapest approach.

- ❑ **Virtual hosting** In this model, the ISP assigns a dedicated virtual machine (VM) for your website. That VM might be shared with several other VMs on a single physical machine or on a farm of VM hosts.
- ❑ **Dedicated hosting** In this model, you get a dedicated physical server for your website. Of course, this tends to be the most expensive form of hosting.

Within the hosting gallery, you can see that the available offers are broken down into these three types (see Figure 15-2).

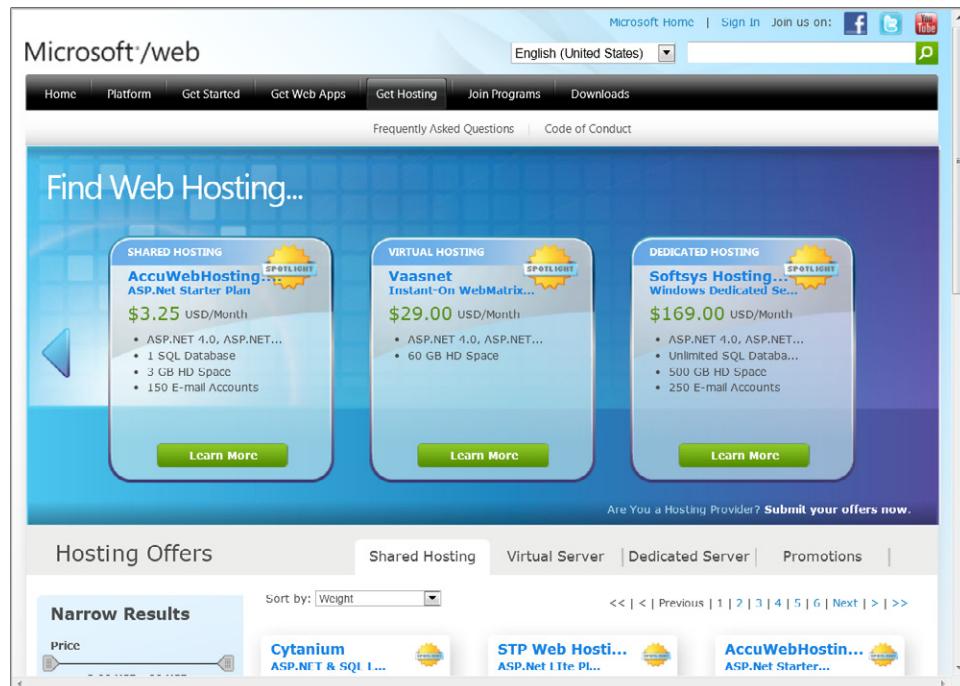


FIGURE 15-2 The Microsoft hosting gallery.

If you accessed the gallery via WebMatrix, you'll see that the URL that WebMatrix called is very interesting. For this example, I created a website by using the Bakery template, so the URL to the hosting gallery looks like the following:

```
http://www.microsoft.com/web/Hosting/Home?language=en-US&notify=true&technology=ASPNET  
4.0;MSSQL;&appId=Bakery
```

Note that by using the parameters in this URL, WebMatrix tells the hosting gallery that the site was built with ASP.NET and Microsoft SQL Server and that it passed an *appld* that told the service that it's building the bakery store. By knowing this information, the hosting gallery can provide hosting options compatible with this application.

2. You could simply choose one of the "spotlight" providers at the top of the page, but for now, scroll down to see how you can narrow the options by using the Narrow Results tool provided on the site, which you can see in Figure 15-3.

The screenshot shows the 'Hosting Offers' page with the 'Shared Hosting' tab selected. At the top left, there's a 'Narrow Results' section with various filters:

- Price:** 0.99 USD - 99 USD, with dropdowns for 'United States' (selected), 'USD', 'Keyword Search' (with a magnifying glass icon), and 'ASP.NET 4.0'.
- Storage (GB):** 1 GB - Unlimited.
- Number of SQL Databases:** 0 - Unlimited.
- Number of E-mail Accounts:** 0 - Unlimited.
- Bandwidth per month:** 2 GB - Unlimited, with checkboxes for '24/7 Support' and '99.9% Uptime'.

On the right, the results are displayed in a grid:

- Applied Innov... WebMatrix Hosti...** \$4.95 USD/Month
 - ASP.NET 4.0, ASP.NET...
 - 5 SQL Databases
 - 5 GB HD Space
 - 50 E-mail Accounts
 - 24/7 Support
 - 99.9% Uptime
- Arvixe PersonalClass A...** \$6.00 USD/Month
 - ASP.NET 4.0, ASP.NET...
 - Unlimited SQL Database...
 - Unlimited HD Space
 - Unlimited F-mail Acc...
 - 24/7 Support
 - 99.9% Uptime
- Softsys Hosti... Silver Shared P...** \$6.67 USD/Month
 - ASP.NET 4.0, ASP.NET...
 - 2 SQL Databases
 - 2 GB HD Space
 - 100 E-mail Accounts
 - 24/7 Support
 - 99.9% Uptime
- WinHost.com WinHost Max** \$8.29 USD/Month
 - ASP.NET 4.0, ASP.NET...
 - 10 SQL Databases
 - 4 GB HD Space
 - 500 E-mail Accounts
 - 24/7 Support
 - 99.9% Uptime
- Applied Innov... ValuePlus Windo...** \$9.95 USD/Month
 - ASP.NET 4.0, ASP.NET...
 - 1 SQL Database
 - 2 GB HD Space
 - 50 E-mail Accounts
 - 24/7 Support
 - 99.9% Uptime
- Adhost WebMatrix Basic** \$9.95 USD/Month
 - ASP.NET 4.0, ASP.NET...
 - 2 SQL Databases
 - 1 GB HD Space
 - 5 E-mail Accounts
 - 24/7 Support
 - 99.9% Uptime

Each listing includes a 'Sign Up' button, a 'Learn More' button, and a star rating and review count (e.g., 11 reviews).

FIGURE 15-3 Narrowing the results.

With the Narrow Results tool, you can precisely specify the configuration your site needs, such as the storage size, the number of databases, the number of email accounts, and the allowed bandwidth per month, as well as, of course, a desired price.

3. On each result, you'll see a Learn More button. By clicking this button, you can learn about the offers in more detail. This exercise uses the Applied Innovations service shown in the upper-left corner in Figure 15-3. Use the Narrow Results tool to find this service, and click Learn More. The screen shown in Figure 15-4 appears.

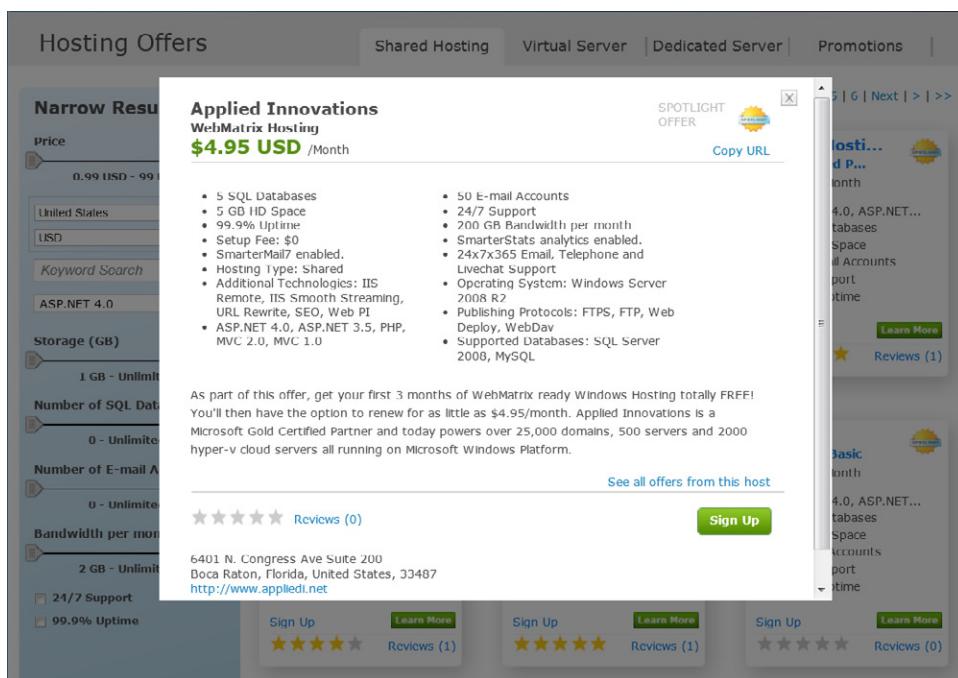


FIGURE 15-4 Signing up for an offer.

This ISP offers free hosting for three months, so it's a good way to test the service and explore the process of publishing sites from within WebMatrix.



Note At the time of this writing, Applied Innovations was offering three months for free, but by the time you have this book in your hands, there might be some differences beyond our control. If you can't get free hosting from this provider, there might be others; use the gallery to find them.

4. Click Sign Up to go to the Applied Innovations site, where you can sign up for website hosting that is compatible with WebMatrix. Figure 15-5 shows the Applied Innovations signup page.

The screenshot shows a web browser window displaying a hosting sign-up form. At the top, a red banner highlights "Your 3 Months of FREE Hosting Includes:" followed by a bulleted list of features. Below the banner, a message encourages renewing at \$4.95/month. A large blue header bar contains the text "Signup Below". The main form area has sections for "Enter Domain Name & Username" and "Contact Information", each with input fields. A "Create Account" button is located at the bottom left of the form. On the right side of the page, there is a small decorative image of a woman's face.

Your 3 Months of FREE Hosting Includes:

- 5 GB Diskspace
- Full FTP, FTPS, WebDeploy, Webmatrix
- Full ASP.NET, MVC, Razor support.
- Full PHP w/FastCGI support
- Supports all popular Web Applications: DotNetNuke, WordPress, Joomla, etc
- 200GB Monthly Bandwidth
- 5 FREE SQL2008 Databases
- 5 FREE MySQL5 Databases
- 24x7 Expert Technical Support.

Plus you'll be able to renew from just \$4.95/month!

Signup Below

Enter Domain Name & Username

Please enter a username and domain below. If you don't have a domain name, fill in username only and we'll create a temporary domain for you.

Username

Domain Name

Contact Information

First name

Last name

Company

Email

Create Account

FIGURE 15-5 Signing up for a WebMatrix site.

5. After you fill out the form, select Create Account. The service provider creates your site and sends you a welcome email message with details on how to use it. You can see what that looks like in Figure 15-6.

Getting Started with WebMatrix Instructions: webmatrixbook.com

support@appliedi.net

If there are problems with how this message is displayed, click here to view it in a web browser.

Sent: Tue 2/22/2011 9:26 PM

To: Laurence Moroney

Message | Imoroney.publishsettings (1 KB)

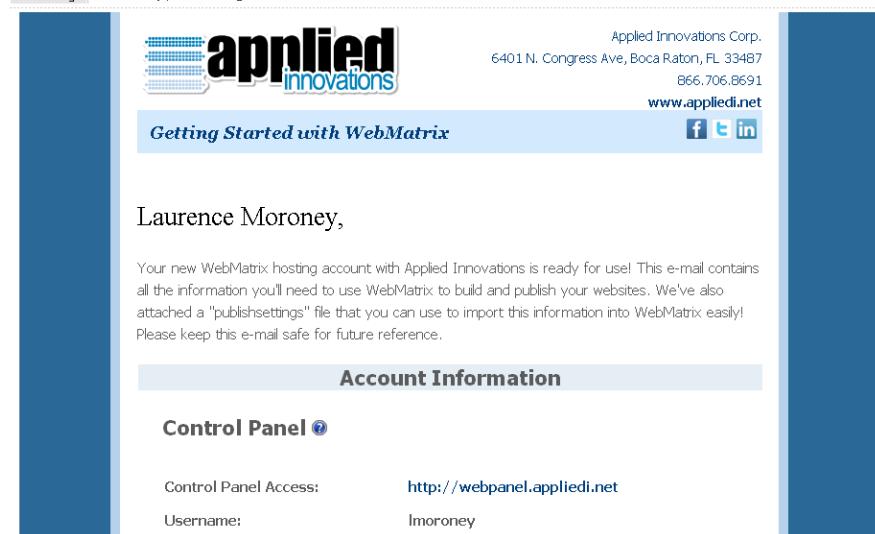


FIGURE 15-6 The welcome email message.

You might have noticed that some of the sites in the gallery were called "spotlight" sites. In addition, supporting everything a WebMatrix user needs, a spotlight site also sends you an email message containing not just your sign-in settings but also an attachment that WebMatrix can use to auto-configure publishing. You can see this in Figure 15-6 as *Imoroney.publishsettings*. It's a good idea to save this attachment, because you'll need it in the next section.

Using the Publish Settings Dialog Box

In the previous section, you signed up for a hosting provider, and the provider sent you an email message that contained an attachment with the .publishsettings extension. WebMatrix can use this to configure itself to deploy to your account on that hosting provider.

1. On the Site workspace in WebMatrix, find the setting that lets you set up remote publishing for your website (see Figure 15-7).

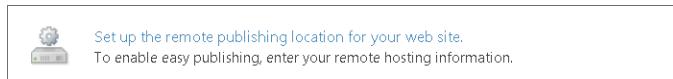


FIGURE 15-7 Setting up the remote publishing settings for your site.

2. Click this link to go to the Publish Settings dialog box, as shown in Figure 15-8.

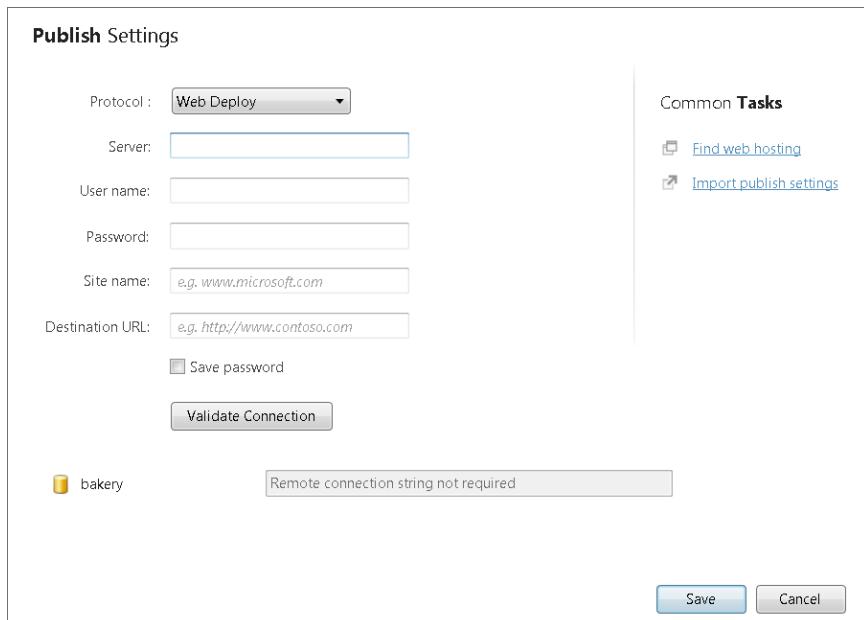


FIGURE 15-8 The Publish Settings dialog box.

3. On the right side of the screen, click the Import Publish Settings option. In the dialog box that appears, browse to the .publishsettings file that you received in the email message from your service provider. WebMatrix uses the information in that file to populate the publish settings with everything that you need to deploy successfully (see Figure 15-9).

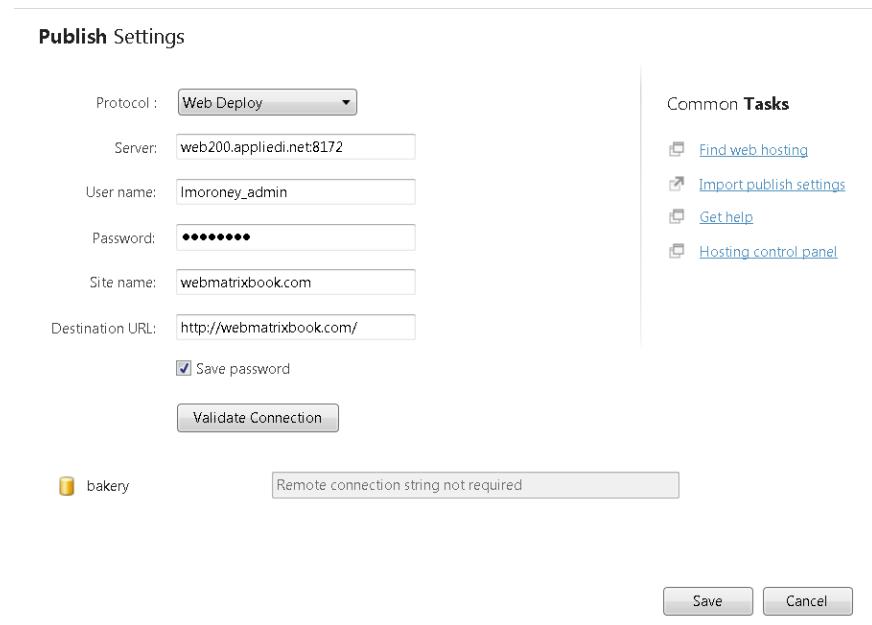


FIGURE 15-9 Properly configured publish settings.

You might notice in Figure 15-9 that although you have a database (*bakery*), a connection string is not required. If you are familiar with database programming with SQL Server or MySQL, you'll know that databases typically run on a separate server, which requires your application to log in to it. The settings for the location of the server, the database to access, and the credentials to access it are usually bundled into a settings string called a *connection string*. Because SQL Server Compact isn't a separate database but instead is a file that runs on your website, you don't need a connection string. Later, when you deploy a WordPress site, you'll see how to use a connection string to access a MySQL database.

4. Click the Validate Connection button. WebMatrix communicates with your server to double-check whether the connection is valid. After a few moments, you should see a confirmation message that WebMatrix was able to connect to the ISP successfully, as in Figure 15-10.



FIGURE 15-10 Successful connection to server.

5. At this point, save your settings. The dialog box closes.
6. On the WebMatrix ribbon, click the Publish button.

Because this is probably your first time using the account, WebMatrix will double-check to ensure that your site will work on the hosting provider, so you'll see the warning message in Figure 15-11.



FIGURE 15-11 Checking publish compatibility.

7. Click Yes. WebMatrix will begin uploading and testing files for you. When the testing process is complete, click Continue. WebMatrix will evaluate the items on your site, comparing them with the server to determine what needs to change or be updated on the server. Because this is your first time publishing, nothing exists on your server yet, so you'll see something like Figure 15-12.

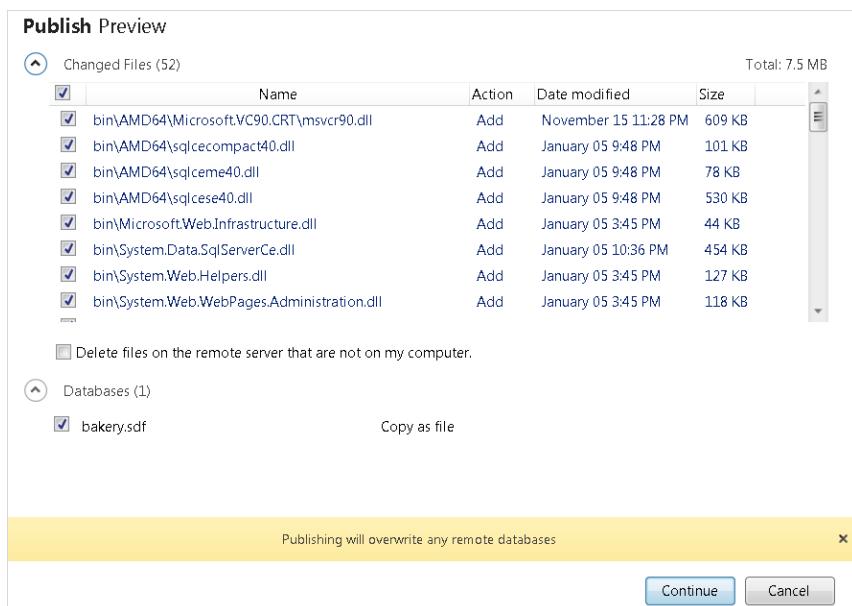


FIGURE 15-12 The Publish Preview screen.

- Click Continue. WebMatrix will begin uploading and publishing your files and your database. You'll see a progress indicator at the bottom of the WebMatrix window in the yellow status bar (see Figure 15-13).



FIGURE 15-13 Publication progress status bar.

When the publish process is complete, you'll see the completion status as shown in Figure 15-14. Note that the status shows the *domain name* that you associated with the account when you signed up. In this case, I signed up using the domain name *webmatrixbook.com*. I haven't mapped that to my Applied Innovations account, so clicking the link won't take me anywhere.



FIGURE 15-14 Completed publishing.

- Even if you haven't mapped the domain name, you can still view the site—just use the domain name you provided, postfixed with the address of the Applied Innovations server, like this: *http://webmatrixbook.com.webmatrix-appliedi.net/*.

Use that URL to display your site in the browser, as shown in Figure 15-15.

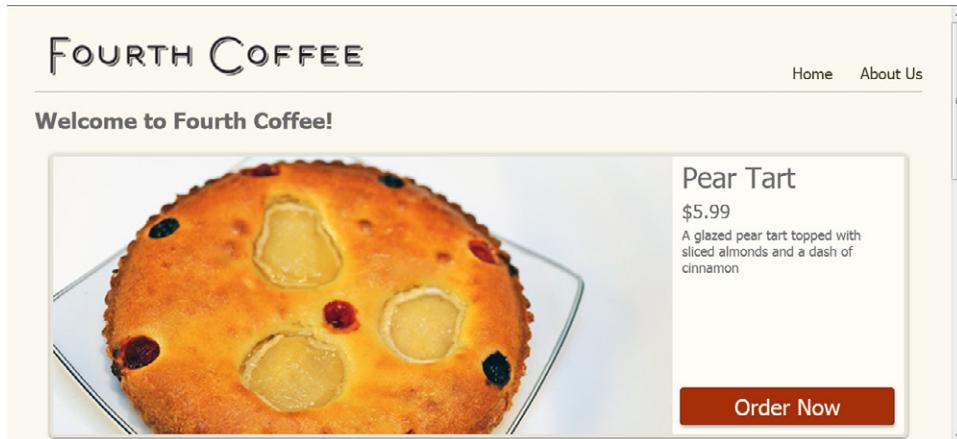


FIGURE 15-15 Viewing your site in the browser.



Important It's worth noting that different hosting providers work in different ways. If you're having trouble, check with your hosting service provider.

In this example, you saw how to deploy a WebMatrix site built by using ASP.NET Web Pages, Razor, and SQL Server Compact to your hosting provider. In the next section, you'll see how the process works for a WordPress-based site built with PHP and MySQL.

Creating a WordPress-Based Site

In addition to creating sites by using the ASP.NET Web Pages framework and hosting them on ASP.NET, SQL Server, and other Microsoft technologies, open source applications are immensely popular for building websites. Though there are many open source applications that use ASP.NET, in this section we'll look at one that uses PHP and MySQL—WordPress—and show just how well this runs on a Windows host, too!

1. Use the New Site From Gallery option to create a WordPress-based site in WebMatrix. If you aren't familiar with doing this, skip forward to Chapter 16, "WordPress, WebMatrix, and PHP," which walks you through the process step by step.
2. Follow the same publishing steps as you did for the Bakery site: choose a provider, open the Publish Settings dialog box, and import the publish settings file that you received from the ISP. You can see what this will look like in Figure 15-16.

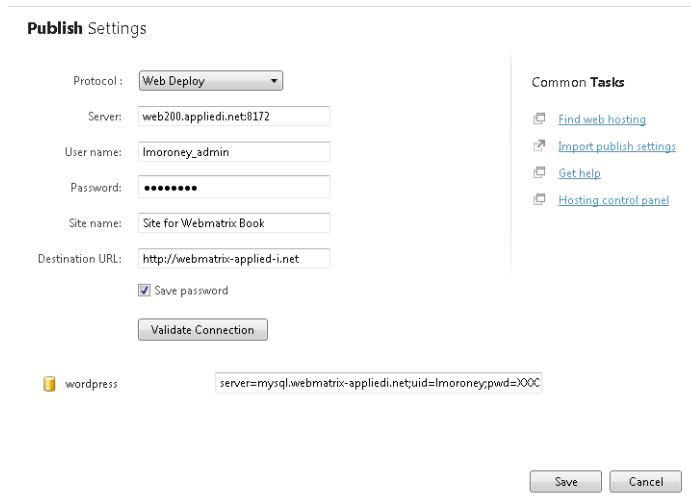


FIGURE 15-16 The publish settings for a PHP and MySQL-based application.

Notice that the main difference is that this time, the connection string is set. Earlier, when you deployed the bakery site, you didn't need a connection string because the site used an embedded SQL Server Compact database.

This is a key advantage of using WebMatrix and Web Deploy. If you have ever deployed PHP/MySQL-based sites, you'll know that when File Transfer Protocol (FTP, the de facto standard) is used, site data can be very hard to deploy. The files that make up your site—HTML, CSS, graphics, code, and so on—are easy to deploy with FTP, but the data isn't. You typically have to use a tool such as phpMyAdmin to export the table structure and content to .sql files, which you then use FTP to upload to your server. Then you use the tool on your server to import them and regenerate your tables and content on the server. For complex databases, this process can get very complicated, very quickly!

With Web Deploy, you don't have to worry about this—WebMatrix performs the entire process for you. It scans your local and remote databases and synchronizes them.

3. Save the publish settings, and then publish your site. To run it, you simply need to use the same address as earlier, but add index.php, as shown here (otherwise, the default Bakery site will run):

`http://webmatrixbook.com.webmatrix-appliedi.net/index.php`

You can see the result in Figure 15-17.

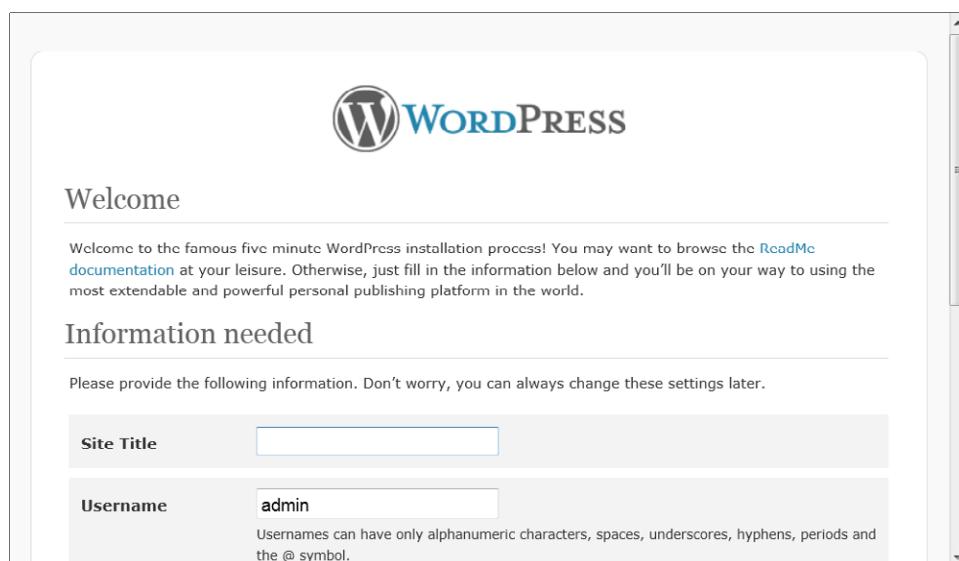


FIGURE 15-17 Running WordPress on the Internet.

Figure 15-7 shows the initial WordPress setup screen where you set up the site title, username, and so forth. You'll see more about these tasks in Chapter 16.

Note that if you choose to use a service provider that you already subscribed to, you should still be able to deploy your site. The de facto standard for publishing to the web is to use FTP. WebMatrix supports this in the Publish Settings tool; you just need to set the *Protocol* setting to *FTP*. However, you won't get all the advantages of using Web Deploy if you do this, so I recommend that you choose a spotlight provider that provides Web Deploy as well as all the other technologies you might need when building sites with WebMatrix.

Chapter 16 looks at WordPress sites in more detail and includes a walk-through of how to create a site with WebMatrix and how to use the WordPress administrative console to change the theme. Finally, you'll write some PHP code to integrate Facebook!

Summary

This chapter explained how the Microsoft Hosting Gallery works with WebMatrix and how you can use it to find a hosting provider. You saw how to sign up with a hosting provider (in this case, one that provides a free trial account) and how providers in the gallery specify publish settings that make configuring WebMatrix for deployment very straightforward. You then saw how you can use Web Deploy to publish both an ASP.NET Web Pages application and a PHP/MySQL based application, a process that included managing the synchronization of local and remote databases with a single click.

Chapter 16

WordPress, WebMatrix, and PHP

In this chapter, you will:

- Create a WordPress site.
- Configure the WordPress site.
- Use WebMatrix with PHP to edit the WordPress site.

For most of this book, you've been using the Microsoft ASP.NET Web Pages Framework and the Razor syntax to program, build, and deploy websites. But you can also use the popular PHP programming language with Microsoft WebMatrix through its integration of popular open source PHP applications such as WordPress.

In this chapter, you'll take a look at the WordPress application, exploring how you can customize the WordPress open source PHP code by using WebMatrix.

Creating a WordPress Site

Microsoft's Web Application Gallery is a collection of open source applications with installers that not only install each application on Windows but *also install their dependencies*. I can't stress the importance of the latter part of that sentence enough, particularly if you are a relatively new developer. Configuring a web stack that integrates a programming framework, a web server, and a database is hard enough without also needing to install and configure any add-on modules that the web stack components need to work together. When you use WebMatrix on Windows, the experience is a lot simpler, as you'll see in this chapter when you build a WordPress application example.

1. To get started, launch WebMatrix and select the Site From Web Gallery option (see Figure 16-1).



FIGURE 16-1 New site options in WebMatrix.

2. When you click Site From Web Gallery, the Web Application Gallery browser opens, and you can browse through the gallery of apps. For this example, select the WordPress site type, and name your new site **WPSite**, as shown in Figure 16-2.

This screenshot shows the "Site from Web Gallery" dialog box. On the left is a sidebar with categories: All (43), Blogs (11), CMS (27), eCommerce (6), Forums (2), Galleries (4), Tools (8), and Wiki (2). The main area displays a grid of app icons with their names and download counts. The "WordPress" icon, which is white with a blue 'W', is highlighted. Below the grid, there is a detailed description of the WordPress app, stating it started in 2003 and has grown to be the largest self-hosted blogging tool in the world. At the bottom, there is a "Site Name" input field containing "WPSite", and "Next" and "Cancel" buttons.

FIGURE 16-2 Creating a WordPress site.

3. Click Next. When you do so, WebMatrix determines the dependencies you need. For example, the WordPress site requires both PHP and MySQL. Your experience might vary from mine—you might need to install PHP. I already had PHP installed, but I didn't have MySQL. (The following steps assume that you need to install MySQL. If you already have MySQL installed, you can skip those steps.) WebMatrix detects that you don't have MySQL and it lets you know that you need it (see Figure 16-3).

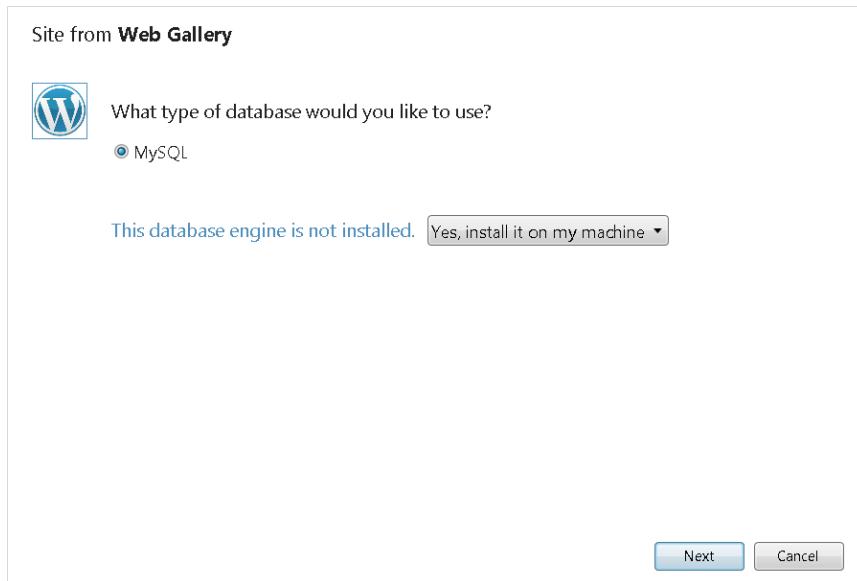


FIGURE 16-3 WebMatrix detects that you need MySQL.

4. Click Next. Because you need to install MySQL, you also need to configure the MySQL root password (*root* is the Linux-oriented term for *system administrator*), so WebMatrix displays a dialog box where you can specify the password (see Figure 16-4).

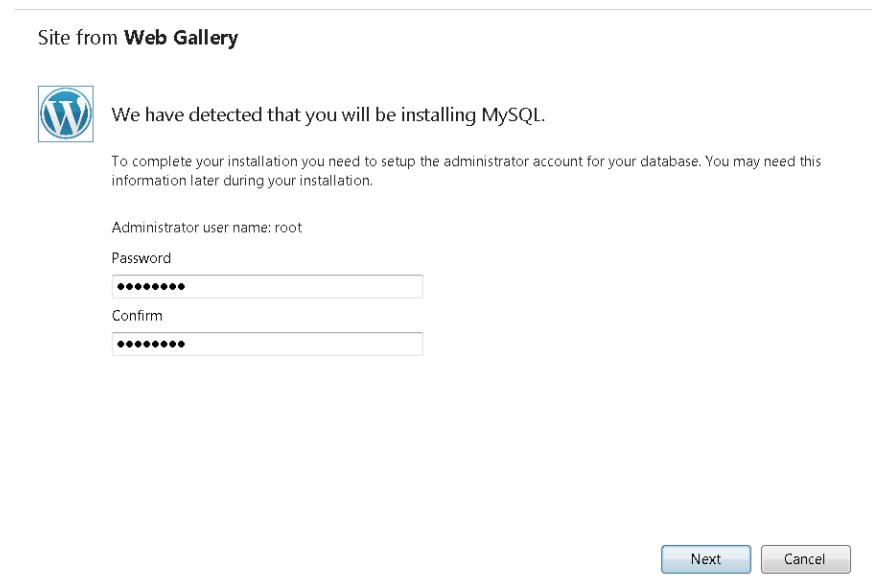


FIGURE 16-4 Configuring the MySQL root password.



Warning You need to manage the root password you create carefully. Read the sidebar “MySQL Root Password Management” before you continue.

MySQL Root Password Management

If you’re asked for a root password during any installation, you should take note of this password and remember it. You’ll need it when you configure WordPress (*or any other PHP application that uses MySQL*).

The root password is for the *database* and not for the application. It’s not uncommon for a developer to install an application using WebMatrix and configure a MySQL root password during that installation, and then come back weeks later and install a *different* application. When the installation asks for the root password, the developer thinks it is asking him or her to *configure* the root password rather than to provide the existing root password—and the installation ends up with errors.

This scenario is made even worse by the fact that MySQL configures the password file *after* installation, in a hidden directory. If you uninstall MySQL and then reinstall it, the root password from the *original* installation is still in effect. So the only way to totally “clean” your system of MySQL is to first uninstall it and then find the hidden C:\ProgramData directory and delete the MySQL subdirectory from it. You can reach the hidden directory by opening Windows Explorer and typing **C:\ProgramData** into the address bar.

5. After you have configured the MySQL root password, click Next, and WebMatrix will give you the EULA for each of the components you need to install. Note that the installation detects the dependency and provides the EULA for the MySQL Connector/Net component as well as for WordPress, as shown in Figure 16-5.

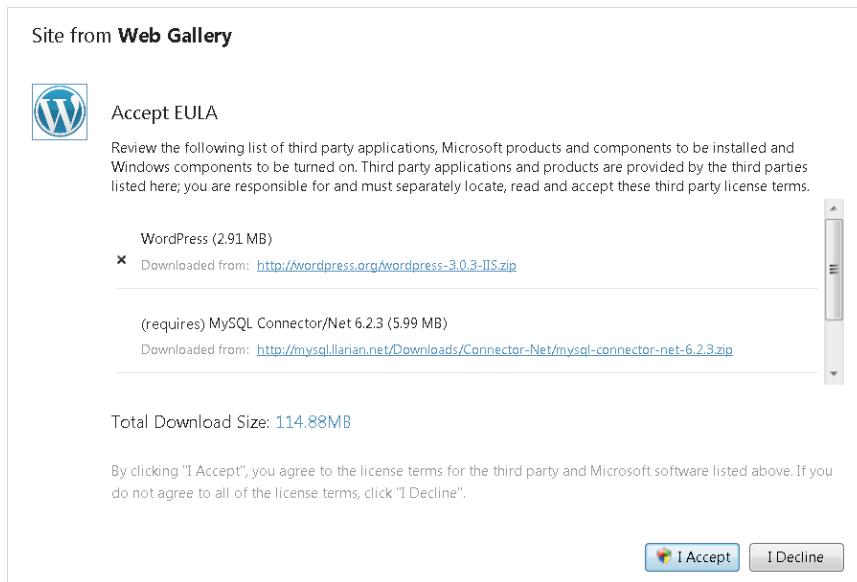


FIGURE 16-5 The EULA screen.

MySQL requires the MySQL Connector/Net component, so WebMatrix installs that component as well.

6. Click Accept. WebMatrix downloads and installs all the components that you need. You'll see an Installing screen similar to Figure 16-6.

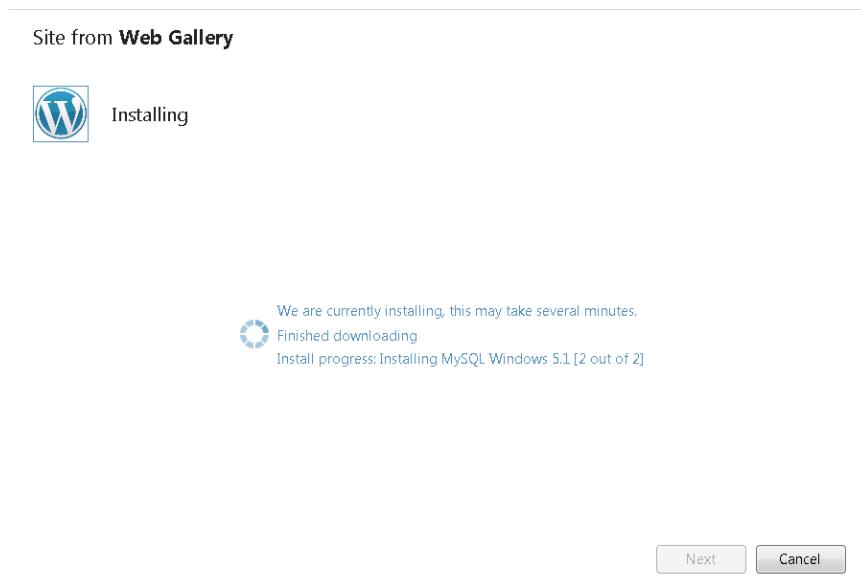


FIGURE 16-6 WebMatrix downloads and installs WordPress and MySQL.

7. After the download is complete, you need to set up WordPress. The setup involves two steps: first, you must deploy all the scripts and create the MySQL database structure. WordPress itself executes the second step, writing default values to the new database structure. You'll see that in a moment.

For WordPress to write to MySQL and perform administrative tasks such as creating tables and so on, it needs root access, so WebMatrix will need to log on to MySQL with your root password. You'll also need to specify a new user account for the database; WordPress will use that account to write *content* to its database tables, which of course, does not need root access.

WebMatrix provides the dialog box shown in Figure 16-7 so that you can configure all this in one place.

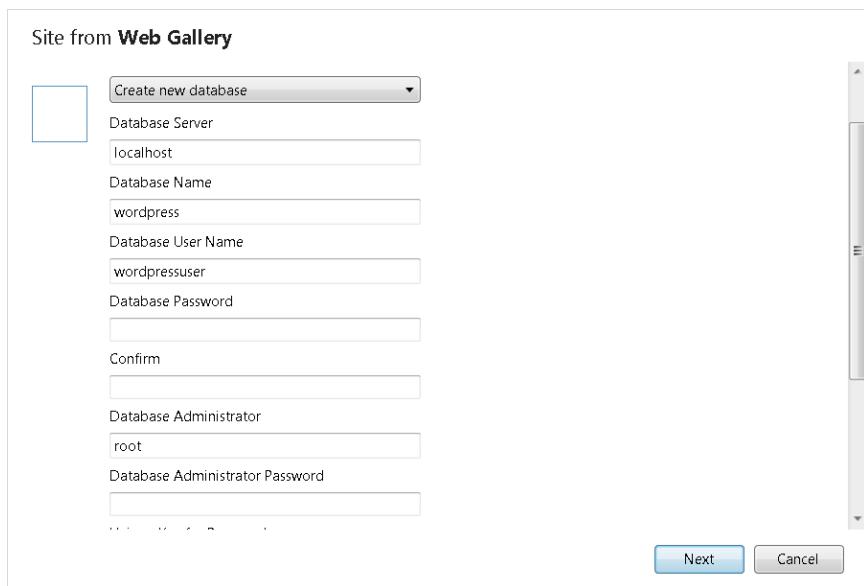


FIGURE 16-7 Configuring your WordPress installation.

In the top part of this dialog box, in the Database User Name field, type a name for the non-root user account that WordPress will use to log on to the database at run time so that it can write data (such as blog posts or page configurations). You'll need to specify a password for that account as well. Completing this dialog box creates a new user for WordPress only.

8. The dialog box isn't very clear here, but the Database Administrator and Database Administrator Password settings are asking for your *existing* root user password. Enter that information into those text boxes. If you don't have the password, the installation will fail with an error similar to that shown in Figure 16-8.

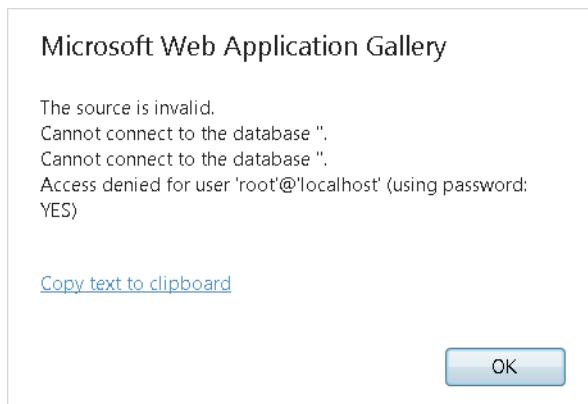


FIGURE 16-8 The error that occurs when the root password is wrong.

Remember that *uninstalling* MySQL isn't enough to remove the root password; you also have to remove the hidden C:\ProgramData\MySQL directory (see the sidebar "MySQL Root Password Management" for details).

Assuming that you have entered the root password correctly, the WordPress installation continues, creating the required WordPress tables in MySQL. When the installation is complete, you'll see a success screen similar to Figure 16-9.

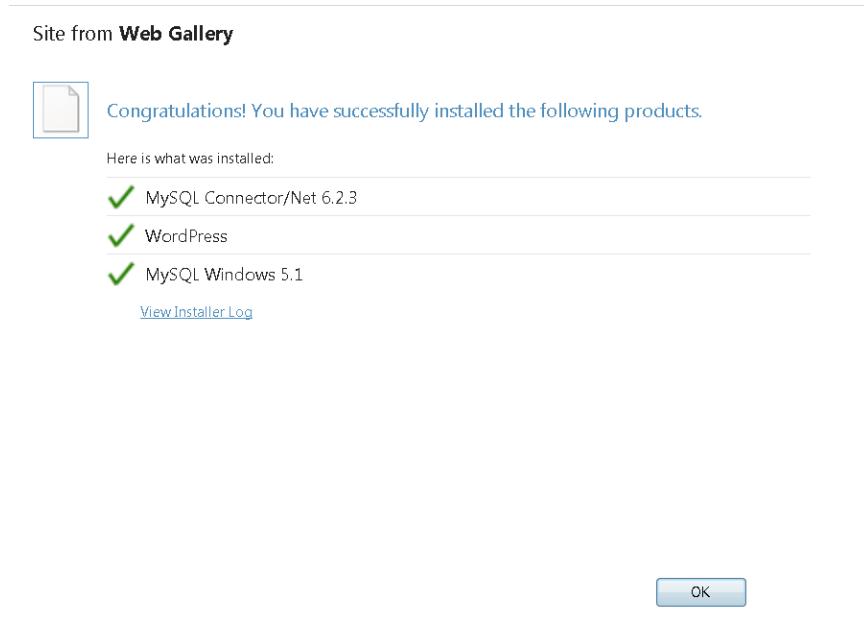


FIGURE 16-9 A successful WordPress installation.

9. Click OK, and WebMatrix launches with your new WordPress site loaded.

But your new WordPress installation isn't fully complete yet. The files are deployed and the database structure is in place, but WordPress needs to write some sample data to these, so you need to set up a site administrator user and password. WordPress will ask you for this information the first time you run the site, as shown in Figure 16-10.

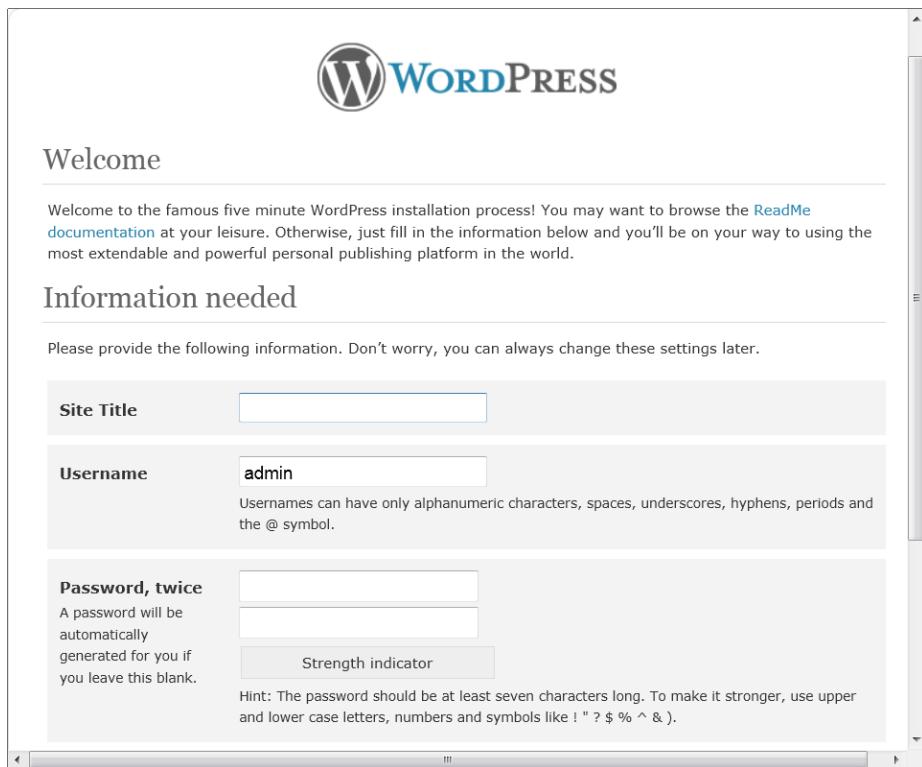


FIGURE 16-10 Setting up WordPress.

The WordPress site involves several users and passwords, as listed here:

- ❑ The *root user* is for the database. WordPress requires the root user account for managing the database itself and for performing structural tasks, such as creating and deleting tables.
- ❑ The *wordpressuser user* (shown previously in Figure 16-7) is the *database* user that WordPress itself uses to log in to the database and add and remove data.
- ❑ The *site administrator* is the account that *you* will use to sign in to and administer your WordPress site. You don't want strangers going in and changing your code, themes, and content, so you need credentials.

10. Complete the form shown in Figure 16-10 and click the Install WordPress button at the bottom of the screen. WordPress will finish the initial setup, and you'll see a success screen similar to Figure 16-11.

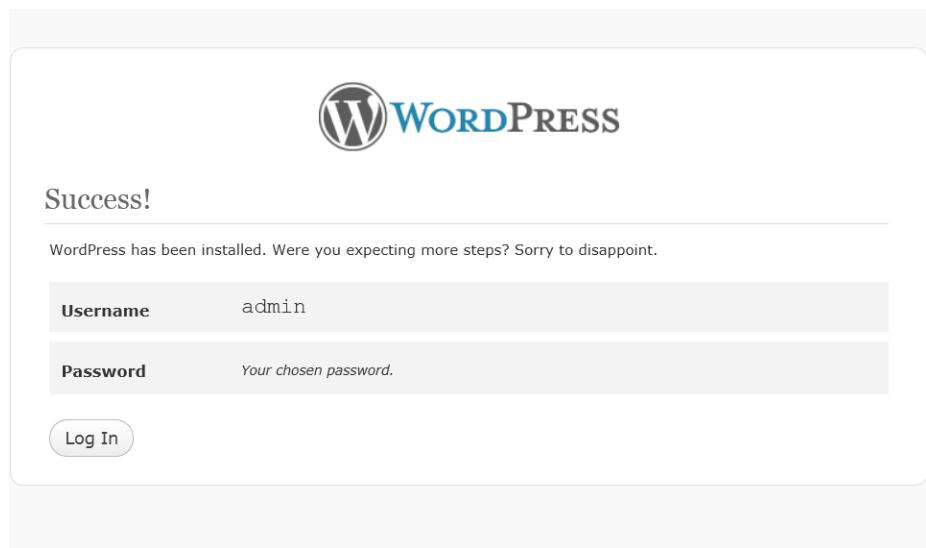


FIGURE 16-11 A successful WordPress installation.

11. At this point, you're ready to run and configure WordPress. Click Log In, and you'll be asked to sign in with the site administrator user name and password that you just configured.

When you complete the login, you'll see the WordPress administrator dashboard in Figure 16-12.

A screenshot of the WordPress administrator dashboard titled "Laurence's Blog". The dashboard features a sidebar on the left with links to "Dashboard", "Updates", "Posts", "Media", "Links", "Pages", "Comments", "Appearance", "Plugins", "Users", "Tools", and "Settings". The main content area is titled "Dashboard" and includes sections for "Right Now" (Content: 1 Post, 1 Page, 1 Category, 0 Tags; Discussion: 1 Comment, 1 Approved, 0 Pending, 0 Spam), "QuickPress" (Title, Content, Tags, Save Draft, Reset, Publish buttons), "Recent Drafts" (empty), and "WordPress Blog" (post titled "WordPress 3.1, lots of fun" dated February 22, 2011).

FIGURE 16-12 The WordPress dashboard.

This screen is where you will perform most of your configuration of WordPress. You can also change code from this screen—but as you'll see later in this chapter, you'll probably find it easier to code in WebMatrix.

Configuring Your WordPress Site

The administrator dashboard in WordPress can be used to configure your site completely, including everything from pages and blog posts to navigation, plug-ins, and other settings. It's comprehensive.

Posts and Pages

1. To see your site, just click the name of the site at the top (for example, *Laurence's Blog* in Figure 16-12). You'll see what your site looks like in the default theme. In the version of WordPress at the time of this writing, the screen looked like Figure 16-13.

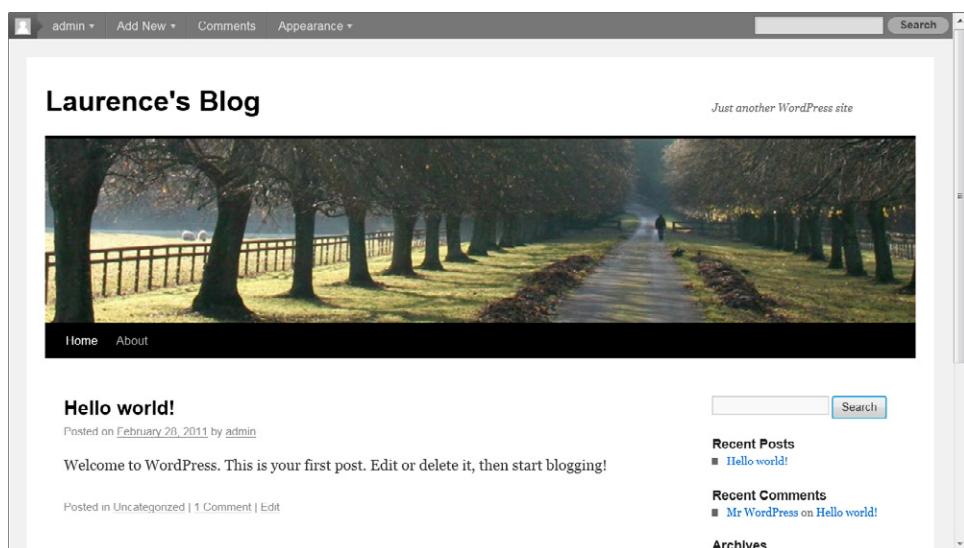


FIGURE 16-13 Viewing the site with the default theme.

2. "Hello world" is included on this page as a default *post*. WordPress was originally designed as a blogging engine, and blogs are typically made up of a sequence of posts. Over time, though, WordPress has evolved into a full-fledged site content management system (CMS), so you can also have "pages," which are typically more static than posts. Click the About link here to link to an example of a webpage built in WordPress. You can see this in Figure 16-14.

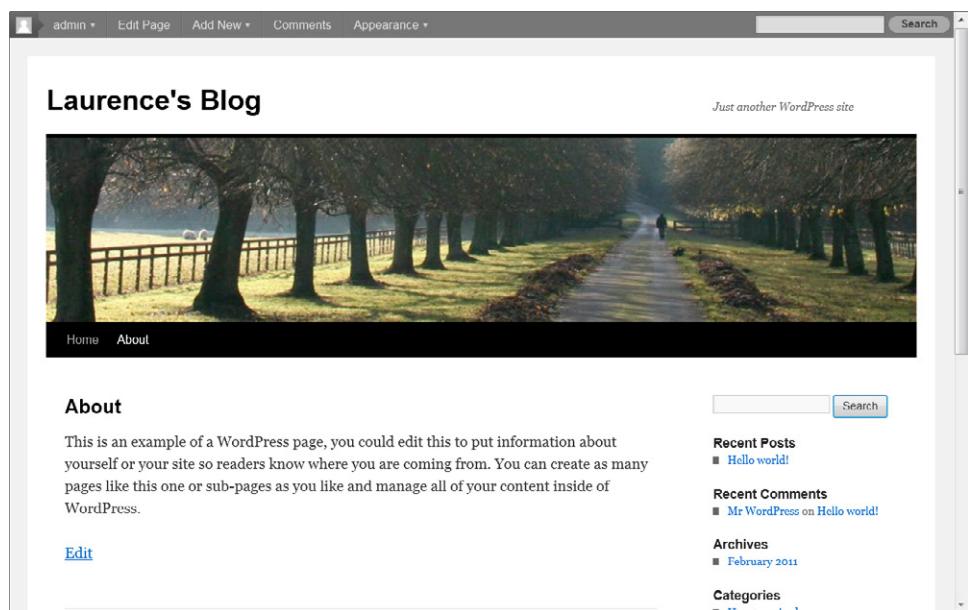


FIGURE 16-14 A webpage in WordPress.

There might not appear to be much difference between posts and webpages, but you'll typically use a post for content that is updated frequently and dated, and you'll use a page for content that will be updated less frequently and thus remain more static. I'm showing both to you now because, as you'll see in a moment, WordPress handles them differently.

3. Scroll down the page. In the Meta section, you'll see a link to Site Admin; click that link to go back to the dashboard.

On the left side of the dashboard, you'll see links to Pages and Posts. Using these, you can create a new page or a new post. Both processes are straightforward and similar, so you'll create a page here, and then you can later create a post by yourself.

4. To create a new page, select the Pages link in the dashboard. You'll see an Add New link appear underneath it.



Note This chapter was written using WordPress 3.1. If you are using a different version, the screens might be somewhat different, but the concepts are the same.

5. Select the Add New link. You'll see a form via which you can add a new page, as shown in Figure 16-15.

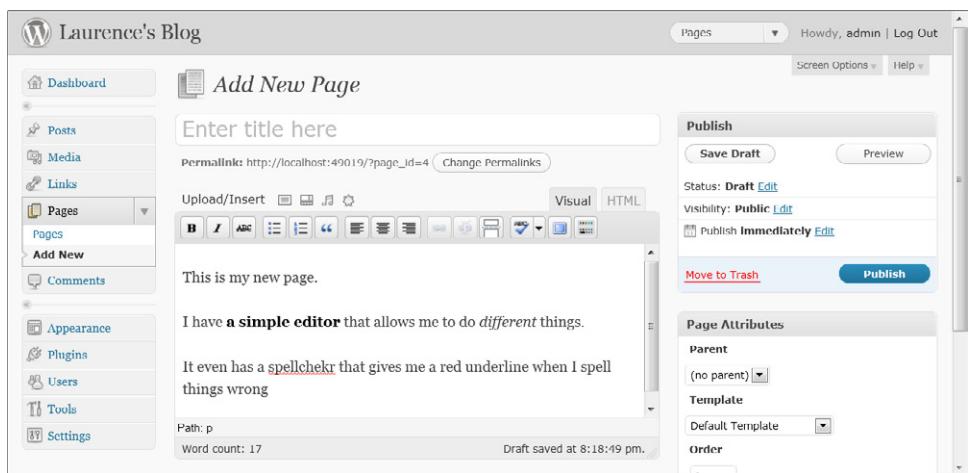


FIGURE 16-15 Adding a new page in WordPress.

6. The Add New Page screen provides the options you need to create a new page. Give the page a title, and enter some page content here by using the in-browser editor. When you're done, click the Publish button on the right side of the page. That saves the page and makes it available live.
7. An easy way to see the page you just created is to click the site title in the upper-left corner of the screen (*Laurence's Blog*, in this case). The browser will open your site to the new page.

The page you just created is added to the navigational content for the site. For example, I used the title *New Simple Page*, which appears in the navigation as shown in Figure 16-16.



FIGURE 16-16 The new page in the navigational structure.

8. Use the same procedure to create a new post.

Configuring the Site Theme

Now that you have created your site and added some simple content (a page and a post), you can tweak the look and feel of the site by picking a template.

1. To do this, click Appearance in the site dashboard. You'll see a Themes link.
2. Click the Themes link. At the top of the screen, you'll see the Manage Themes and Install Themes tabs. Select Install Themes to display the online theme searcher. There are thousands of themes online; the theme search page lets you search, filter, and preview them, as well as download selected themes (see Figure 16-17).

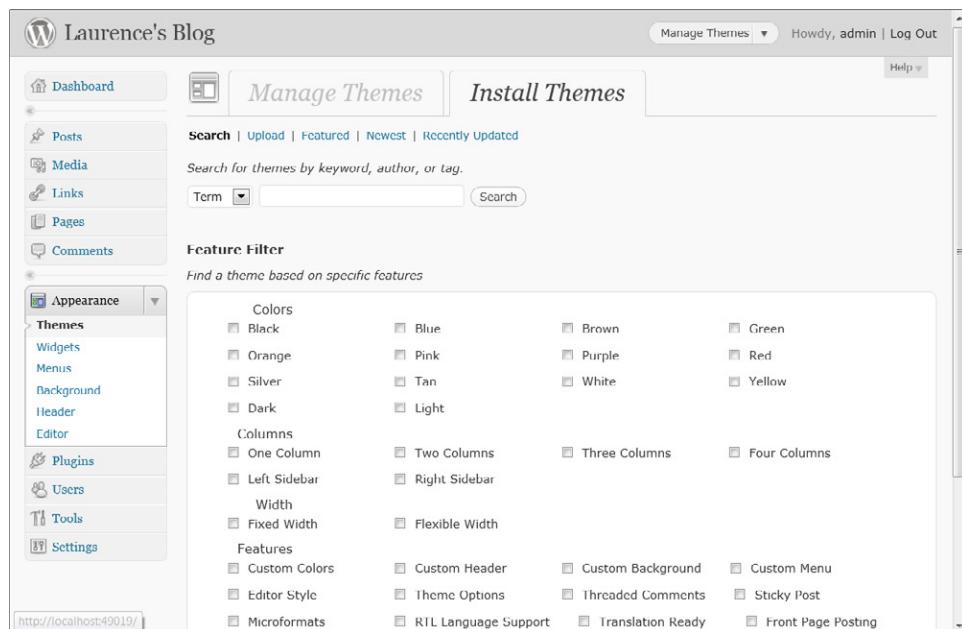


FIGURE 16-17 Installing a theme.

3. You can search or browse for themes you like. I chose one named PrimePress. Find this theme now by typing its name into the search box and clicking Search.

When you find a theme, you'll see that it has Install and Preview links, as shown in Figure 16-18. Remember that installing the theme only installs it—it won't *activate* the theme, but you'll see how to do that in a moment.

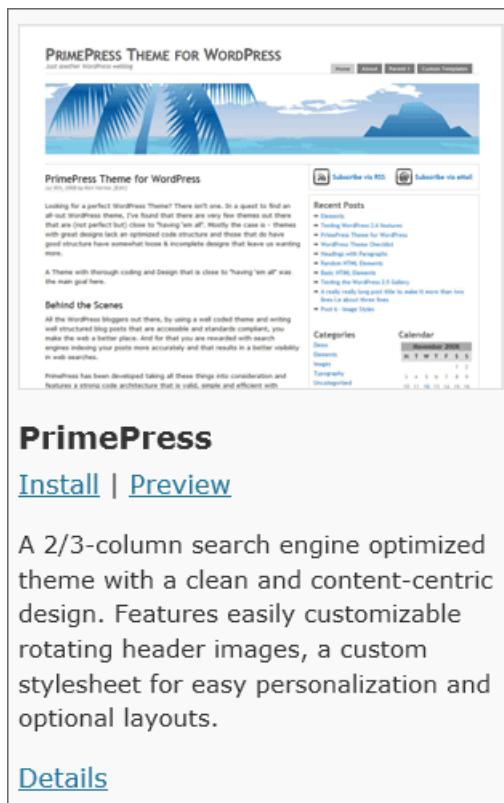


FIGURE 16-18 Finding a theme.

4. Click the Install link. You'll see a quick preview of the theme. Click the Install Now button on the preview page to download and install the theme. When that process is complete, you'll see a confirmation screen showing that the theme successfully downloaded and installed (see Figure 16-19). Click the Activate link on this screen to activate the theme and make it the default theme for your site.



FIGURE 16-19 Installing your theme.

5. After you've activated the theme, go back to view your site. You'll see the new theme in action. You can see an example in Figure 16-20.



FIGURE 16-20 The new theme applied to the WordPress site.

As you can see, considerable customization is possible, and you can generate a lot of content for your website by using the WordPress interface—without ever writing a line of code. If you do want to change the code, the WordPress console allows you to do this too. Remember that the workhorse for code on your site is the *theme* that you've just installed. The theme manages how pages, posts, headers, footers, and so forth look; and how WordPress handles them. For example, you can see that the navigation in the new theme in Figure 16-20 is on the upper right, but in the default theme you saw earlier, it was in the center of the page.

Using the Code Editor

Because of the theme-centric construction, if you as a site builder modify WordPress code, you'll most likely modify the theme you're using. Of course, the WordPress *core* code is also open source, and some developers also modify that, so if you need to modify the core code, that option is available. This difference between theme code and core code is reinforced by the fact that, when you are using the administrator dashboard, you can modify the theme code but not the core code.

This book is aimed at site builders, so first you'll focus on what it takes to edit your site by using code. First you'll explore editing themes by using WordPress, and then later you'll look at modifying the code by using WebMatrix.

1. To edit themes from within the dashboard, select Appearance from the navigation on the left. Notice that Editor is one of the options in the list, as shown in Figure 16-21.

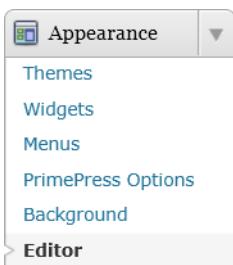


FIGURE 16-21 Using the code editor.

2. Select Editor to open a code editor screen. Notice that each of the files available for editing in the current theme is listed on the right side of the screen. A text box that you can use to edit the code is in the center, as shown in Figure 16-22.

The screenshot shows the 'Edit Themes' screen for the 'PrimePress' theme. On the left, there is a large text area containing the contents of the 'style.css' file. The file includes metadata about the theme, such as its name, URI, and version, followed by CSS code. On the right, there is a sidebar titled 'Select theme to edit:' with a dropdown set to 'PrimePress'. Below this are lists of template files with their corresponding file paths:

- Templates
 - 404 Template (404.php)
 - Archives (archive.php)
 - Archives Page Template (template-archives.php)
 - Blog Page Template (template-blog.php)
 - Category Template (category.php)
 - Comments (comments.php)
 - Footer (footer.php)
 - Header (header.php)
 - HomePage Page Template (template-home.php)
 - Image Attachment Template (image.php)

At the bottom left is a blue 'Update File' button.

FIGURE 16-22 Editing a theme from within WordPress.

3. Although the textbox-in-the-browser approach works, it's a little ungainly if you want to write a lot of code. As an example, try to edit the footer by selecting the Footer.php file on the right of the screen. You'll see the contents loaded into the text box (see Figure 16-23).



```

PrimePress: Footer (footer.php) Select theme

<div id="footer">
    <p class="left">&#169; <?php echo date('Y');?> <strong><?php bloginfo('name'); ?></strong> | Powered by <strong><a href="http://wordpress.org/">WordPress</a></strong></p>
    <p class="right">A <strong><a href="http://www.techtrot.com/primepress/" title="PrimePress theme homepage">WordPress theme</a></strong> by <strong>Ravi Varma</strong></p>
</div><!--#footer-->

</div><!--#container-->

</div><!--#page-->
<?php wp_footer(); ?>
</body>
</html>

```

FIGURE 16-23 Editing the footer.

4. The code is a little difficult to read because of indentations and line breaks, but you can still do it. Change the code a little, so that it reads as follows (the new content is shown in bold text):

```

<div id="footer">
    <p class="left">&#169; <?php echo date('Y');?> <strong><?php bloginfo('name'); ?></strong> | Powered by <strong><a href="http://wordpress.org/">WordPress</a></strong></p>
    <p class="right">A <strong><a href="http://www.techtrot.com/primepress/" title="PrimePress theme homepage">WordPress theme</a></strong> by <strong>Ravi Varma</strong> Edited by <strong> me </strong></p>
</div><!--#footer-->
</div><!--#container-->

</div><!--#page-->
<?php wp_footer(); ?>
</body>
</html>

```

5. When you're done, click the Update File button and view the site. At the bottom of every page, you'll see the content that you just edited (see Figure 16-24).



FIGURE 16-24 Your edited theme.

Editing code within the browser by using a text box has serious limitations. Part of the design principle for WebMatrix is to work better with applications such as WordPress, so in the next section, you'll see how using WebMatrix greatly improves the coding experience.

Using WebMatrix to Edit WordPress

So far in this chapter, you've built a WordPress-based site, added content, and then changed and edited the theme. You did all this with the administrator dashboard in WordPress. WebMatrix makes it even easier for you to edit the code in your theme. In this section, you'll step through a code-editing scenario using WebMatrix.

One of the problems with a blog engine is that most publishers like people to make comments on their site, but when users need to sign in to your site just to make a comment, they're less likely to do so. On the other hand, if you don't force people to sign in, then comments can be anonymous—and the ability to make anonymous comments is often abused. A nice solution to this is to avoid having custom authentication for your site and instead use a common sign-on that can be used elsewhere. Facebook is perfect for this; instead of forcing users to register a user name and credentials to post on your site (and subsequently have to remember them), it makes much more sense for you to use Facebook credentials to let people make comments. This has the added benefit that people posting comments on your site can share them with other users on Facebook, driving more traffic to your site.

In this section, you'll remove the built-in comments engine in your theme and replace it with a Facebook comments box.

Creating a Facebook Application

To use Facebook comments on your site, you'll have to create an Application on Facebook.

1. Go to <http://www.facebook.com/developers>. On the right side of the screen, you'll see a link named Set Up New App (see Figure 16-25).



FIGURE 16-25 the Facebook Developer site.

2. Click the Set Up New App link. You'll need to enter an App Name and accept the Facebook terms of use. The form is shown in Figure 16-26.

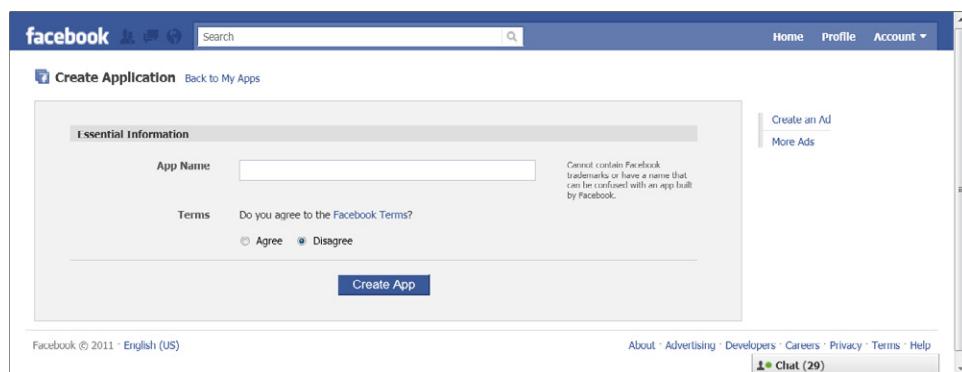


FIGURE 16-26 Creating your app.

3. Enter the required information and agree to the terms, and then click the Create App button. You'll go through a CAPTCHA check, and then you'll be taken to the app configuration screen.
4. Select Facebook Integration on the left. You'll see the Application ID. You'll need this in a moment, so make a note of it (see Figure 16-27).

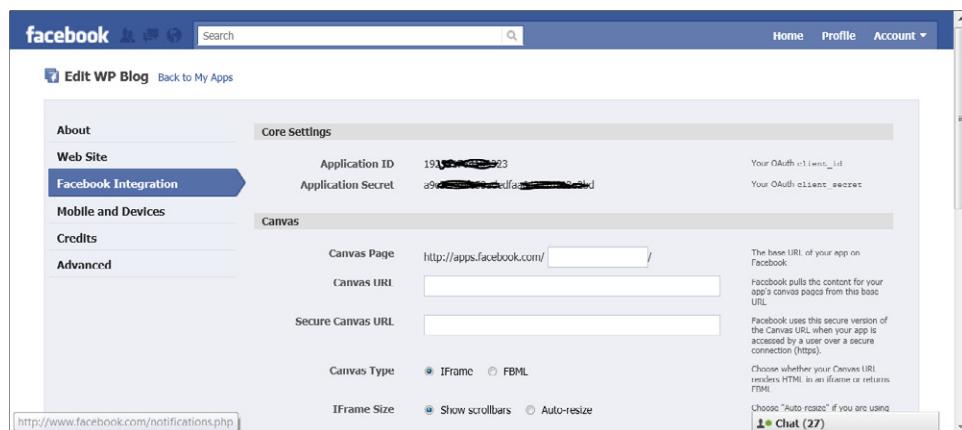


FIGURE 16-27 Getting your Facebook Application ID.

5. You should also click the Web Site tab at this point and configure the website that your comments will run on. You can simply type in *http://localhost* and click Save Changes.
6. Now go to *http://developers.facebook.com/plugins* to see the Social Plugins page. From here you can view the various social plug-ins that are available. You saw many of these and implemented a web helper earlier in this book. Scroll down to find the Comments plug-in and select it. Click its link and you'll be taken to *http://developers.facebook.com/docs/reference/plugins/comments/*.
7. You can configure your plug-in here. Simply give it your site domain (or *localhost*) and specify the number of comments you want to see on each page, as well as the width you want the comments box to take up on the screen. Then click the Get Code button. Facebook will generate the JavaScript you need to implement your comments (see Figure 16-28). Be sure to add your *appId* to this code in the places shown.

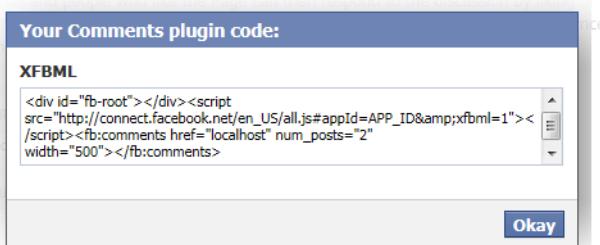


FIGURE 16-28 Facebook plug-in code.

Now that you have the code, let's go back to WebMatrix and edit your theme to implement it.

Editing Your Code with WebMatrix

In the previous section, you saw how to get the code for the Facebook Comments plug-in. Now let's go to WebMatrix and see what it takes to incorporate it.

1. WordPress stores themes in the wp-content/themes folder. Open this in the Files workspace in WebMatrix. You'll see each of your themes in a subdirectory. If you've been following along, you will have the PrimePress theme installed and configured as default. You can see this in Figure 16-29.

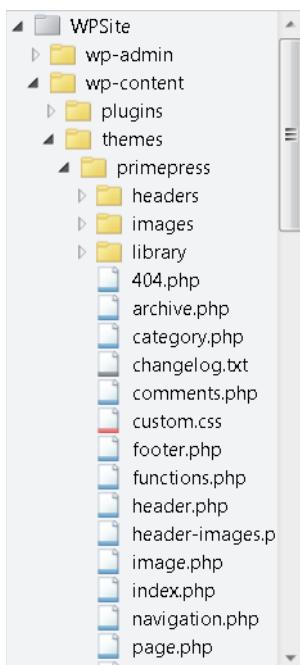
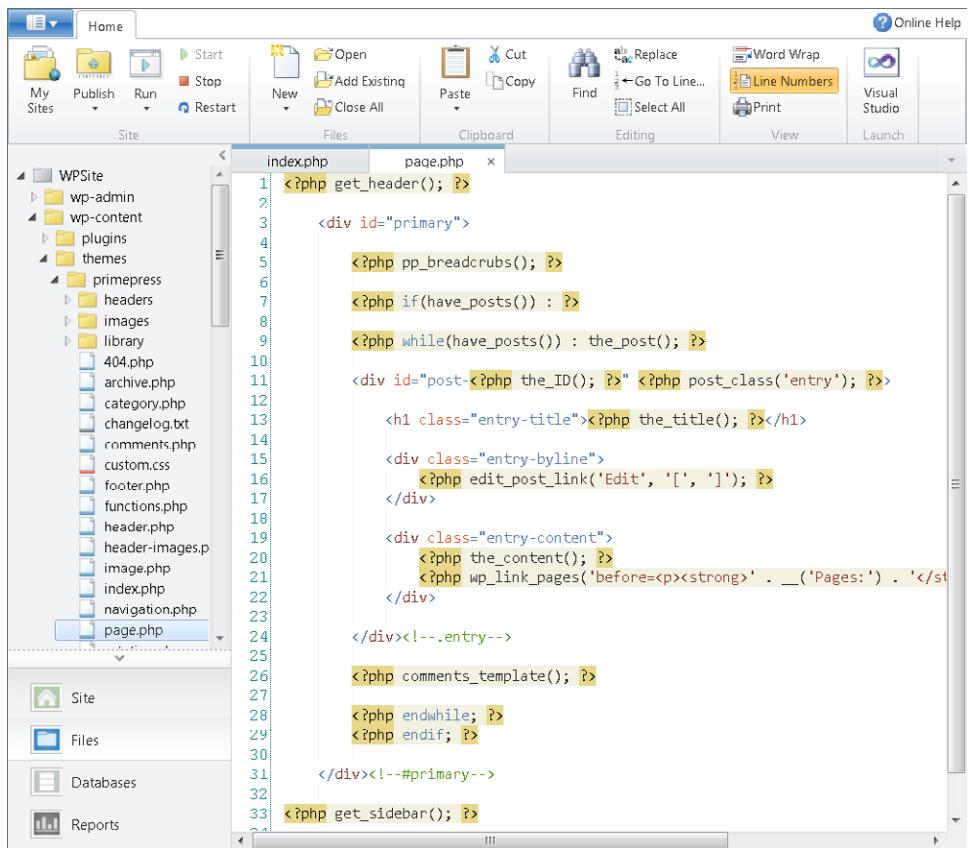


FIGURE 16-29 The WordPress PrimePress theme directory.

Now you can edit the page theme to remove the default comments and replace them with the Facebook Comments box.

2. Double-click page.php to open it. The editor will open your PHP file, complete with syntax highlighting to make editing easier (see Figure 16-30).



The screenshot shows the WebMatrix interface. The left sidebar displays a site structure for 'WPSite' under 'My Sites'. The main area shows two tabs: 'index.php' and 'page.php'. The 'page.php' tab is active, displaying PHP code for a blog post template. The code includes standard WordPress functions like `get_header()`, `pp_breadcrumbs()`, and `the_post()`. A note in the code indicates the removal of the `comments_template()` function. The right side of the interface includes a toolbar with various icons for file operations like Open, Save, Cut, Copy, Paste, Find, Replace, and Print.

FIGURE 16-30 Editing a PHP theme file in WebMatrix.

3. Remove the line that reads:

```
<?php comments_template(); ?>
```

4. Replace the deleted code with the JavaScript that you got from Facebook. Save and run your site.



Note If you click the Run button with page.php open, you'll get an error, because WebMatrix will try to run the template file directly, which doesn't work. Instead, you should open index.php in the root directory of your site, select its tab, and then click Run.

5. Browse to any page on your site. You'll now see that you (and your users) can sign in and leave comments by using your Facebook account. You can see this in Figure 16-31.



FIGURE 16-31 Adding Facebook comments to the page template.

It's just as easy to add the same functionality to a blog post. Just edit the single.php file to make the same change. You'll need to find and replace the following code:

```
<?php comments_template(); ?>
```

When you find it, delete the code and replace it with the JavaScript provided by Facebook.

That's all that it takes to integrate Facebook into a WordPress site. As you can see, the coding experience for this simple change is considerably easier when you use WebMatrix. Over time, as you need to write more complex code, you'll come to appreciate the capabilities that WebMatrix provides when working with WordPress.

Summary

This chapter demonstrated how WebMatrix and WordPress work well together. First, you saw how to install WordPress and configure it to create a site by using WebMatrix. Then you used WordPress itself to add a sample page and a sample post, and you changed the theme. Then you made a simple change to the theme by using the built-in editor within WordPress before embarking on something a bit more complicated—integrating a Facebook-driven comments engine. During that example, you saw how the coding environment in WebMatrix makes such tasks much easier.

Appendix

WebMatrix Programming Basics

Getting Started with WebMatrix Programming

In Chapter 3, “Programming with WebMatrix,” you saw how you can use HTML and the Microsoft .NET Framework to build a couple of simple pages. The first active example provided dynamic content that rendered the current time and date, and the second demonstrated how you can post data to a server, have it manipulate that data, and then update your page accordingly. If you aren’t familiar with web programming, some of the concepts might have been a little difficult to grasp, but don’t worry—they’ll become familiar in time. In this appendix, you’ll review some basic programming concepts and see how they work within Microsoft WebMatrix.

Variables and Data Types

All programming languages use variables to handle data. A variable is simply a named object that is used to store data. You can name variables just about anything you like, limited only by the restriction that the first character must be alphabetic and that the name cannot contain a space or certain reserved characters.

Specifying Variables

You can make your variables store only specific types of data by declaring them as storage for that particular data type. For example, consider the following variable declaration:

```
var foo;
```

This statement creates a variable you can use to store any type of data. Its type is initialized the first time you load something into it. In contrast, this specifically typed variable declaration:

```
string foo
```

can be used only to store a string. If you try to assign a non-string value to this variable, WebMatrix will generate a run-time error.

WebMatrix uses the .NET Framework, which supports many object types, any of which can be used as variable types. For example, Chapter 3 uses a *DateTime* object. *DateTime* is a type that can be used as a variable too—for example, you could specify tomorrow's date as follows:

```
DateTime tmrw = DateTime.Now.AddDays(1);
```

Note that this declaration and the value assignment occur on the same line; the line creates a variable named *tmrw* that can hold a *DateTime* value and then assigns it tomorrow's date by adding one day to the current date.

Converting Variable Types

To prevent errors, if you have a variable of a specific type, you should make sure that you load that type into it. Consider the example that posted data back to the server; it used two variables named *num1* and *num2* that were to be added together. The problem was that *num1* and *num2* were *not* numbers, they were strings. Performing a mathematical operation on two strings is a problem.

The solution was to *convert* these strings to numbers before adding them together:

```
if(IsPost) {  
    num1 = Request["fNum"];  
    num2 = Request["sNum"];  
    sum = num1.ToInt() + num2.ToInt();  
    sumText = "The answer is : " + sum;  
}
```

The page's code accomplished this by using the *ToInt()* property of string objects, which returns an integer, so *num1.ToInt()* returned a number and the code added that to *num2.ToInt()*, which also returned a number. The result (also a number) was loaded into *sum*.

Some other common data formats besides *int* (for a number) and *string* (for a text value) are:

- **Bool** This is a Boolean value that holds one of two states, which can be interpreted as *true/false*, *on/off*, *right/wrong*, and so on. You can convert a value to a *bool* by using *AsBool()*, and you can test whether a value is a *bool* by using *IsBool()*. You have used a *bool* already without realizing it. In the sample where you added two numbers, the code checked whether the request was a *POST* by using *if(IsPost)*. *IsPost* is a Boolean value set by the .NET Framework. When the server detects a *POST* request, it sets *IsPost* to *true*; otherwise, it sets *IsPost* to *false*.
- **Float** This is a number that has a floating decimal point. The term *floating* means that the decimal point can appear anywhere in the number. For example, both 123.45 and 1.2345 are floating-point numbers. You can convert a value to a floating-point number by using *AsFloat()*, and you test to see if a value is a floating-point number by using *IsFloat()*.

- **Decimal** A *decimal* is very similar to a *float*, except that it uses more memory and is more accurate. In most cases, you can just use a *float*. To convert a number to a decimal, you can use *AsDecimal()*. To test if a value is a decimal, use *IsDecimal()*.
- **DateTime** As you've seen, this type stores date and time information. To convert a value to a *DateTime*, use *AsDateTime()*. To check if a value is a *DateTime*, use *IsDateTime()*.

These are just a few of the more common data types you'll use in your variables. As you work through this book, you'll use some more. For more information on programming with WebMatrix, go to <http://www.microsoft.com/web/category/learn>.

Common Programming Concepts

In addition to variables, some other programming constructs you'll commonly use within applications are *conditional statements*, *flow statements*, and *loop statements*. This section explores the most common constructs.

Testing Conditions with *if* and *switch*

The obvious conditional statement—in English as well as in programming languages—is *if*. You use the *if* statement to test a value or condition. If the condition evaluates to *true*, you execute one piece of code; if the condition evaluates to *false*, you execute another. You've seen a simple conditional statement example already when you posted data to the server:

```
if(IsPost) {  
    num1 = Request["fNum"];  
    num2 = Request["sNum"];  
    sum = num1.ToInt() + num2.ToInt();  
    sumText = "The answer is : " + sum;  
}
```

The preceding code begins with a statement that checks the *IsPost* variable. If *IsPost* is *true*, then the next line or block of code will execute. (In C#, a *block* of code is contained between curly brackets.) If, instead, the value of *IsPost* is *false*, execution continues at the end of the conditional block. In this case, nothing happens because there's no code after the end of the block. If you want something to happen when the condition (*IsPost*) is *false*, you can use an *else* statement, placing the line or block of code you want to execute after it. The following shows an example:

```
if(IsPost)  
{  
    //Code to run on Post  
}  
  
else  
{  
    //Code to run otherwise  
}
```

Beginners tend to make several common programming errors when using the *if* statement. The most common is omitting the parentheses around the condition. You must always place the condition inside parentheses. You can't write *if IsPost*; you have to write *if(IsPost)* instead.

A second common error occurs when checking whether values are equal. In a spoken language, assigning a value to a variable (*x=6*) and checking whether a variable holds a value *if(x=6)* sound the same. But to computers, the latter also looks like you are assigning 6 to *x*. To avoid this confusion, C# uses two equal signs (==) to check for equality and only one for assignment. So, to check whether *x* is equal to 6, you'd write *if(x==6)*.

Finally, remember that the value within the parentheses after the *if* (the condition) must actually evaluate to *true* or *false*, not to some other value.

Using *if* statements is useful, but you might encounter a situation where you want to choose code to run based on one of several different conditions. For example, suppose you want to set the price of a car based on its color, and there are 10 colors to choose from. You could do that with multiple *if* statements, writing code that essentially means "if the car is red, use this price; if it is black, use this price; if it is blue, use this price;" and so forth. But that's tedious. Instead, you can use a condensed form of multiple *if* statements called a *case* statement to do this. Here's a code example:

```
switch(car_color)
{
    case "Red":
        //Do something
        break;
    case "Blue":
        //Do something
        break;
    case "Green":
        // Do something
        break;
    default:
        // Do something
        break;
}
```

You specify the value that you want to check by using the *switch* statement, and within that statement's code block you test each possible value by using *case*. Note that after you've finished the code that executes for any individual case, you use a *break* statement. The *break* causes execution to continue after the end of the *switch* block. If none of the conditions for the *case* statements evaluate to *true*, you can use an optional *default* section at the end to run code, after which you *break* again.

Repeating Code with Loops

Computers are very good at repeating tasks; they use a construct called *loops* to perform repetitious tasks. There are several different types of loops, each of which could probably be made to work for any situation, but some are easier to use in some situations than others.

The *for* loop When you know exactly how many times you want to run through a loop, you use the *for* loop construct. This basically specifies that you start counting at one value and keep going until you get to another value or condition, incrementing the value by a certain amount each time the loop executes.

Here's an example that writes out even numbers between 0 and 100:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        @{
            int n;
            for(float i=0;i<=100;i++){
                n=(int) i;
                if( (i/2) == (n/2) ){
                    <p>@i</p>
                }
            }
        }
    </body>
</html>
```



Note Those of you with more programming experience will realize that this isn't the most efficient way to create a loop to write out even numbers, but bear with me; this example is purposefully inefficient.

One interesting point is that WebMatrix is very smart at determining what is code and what is HTML markup, which helps keep your job as a developer as simple as possible. In the preceding listing, you can see that the `<p></p>` tags function as HTML within the code block. You don't need to write any special characters or markup on the page to separate them, you simply write the HTML markup and add the Razor variable within it as shown.

This code writes out the numbers by defining a loop that runs from 0 to 100 by using the command `for(float i=0; i<=100;i++)`. You can read this as "Execute the following code block repeatedly (the *for* command). Start the value of the float variable *i* (called a *loop counter*) at 0, and continue while the value of *i* is less than or equal to 100. Add 1 to *i* each time you complete the block."

To check for an even number, the trick is to divide i by 2 and see if the result is an integer (whether the result of the division has a decimal place). For example, 4 divided by 2 is 2, with no decimal; thus 4 is an even number. In contrast, 5 divided by 2 is 2.5, which has a decimal place, so you know that 5 is not an even number. So, to perform the check, load an integer with the value of i and divide that by 2 as well. An integer can't have a decimal, so it is rounded down. Thus, when you divide 5 by 2, instead of getting 2.5, you'll get 2. So you can determine whether a number is even if the *float* version divided by 2 is the *same* as the *int* version divided by 2. If so, you write out the value of i .

You can see the results in Figure A-1.

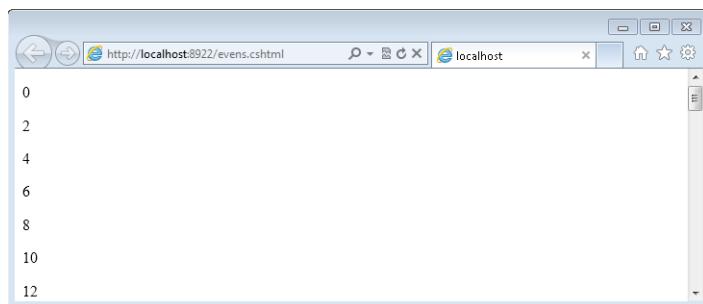


FIGURE A-1 Using a *for* loop to find even numbers.

Even if you know only a little about programming, you'll have realized that this is a silly example because it assumes that loops can only be incremented by the value 1—so the example has to check whether all the numbers are odd or even. A far easier way is to start at 0 and increment by 2. That way, you already know that the loop counter is even, which saves a lot of bother. The revised loop looks like the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    @{
      for(float i=0;i<=100;i+=2){
        <p>@i</p>
      }
    }
  </body>
</html>
```

This new version uses both far less code and less server time and memory, while still getting the same results!

The *while* loop Similar to the *for* loop, but a little easier to read, is the *while* loop. This basically executes code within the *while* block as long as a certain condition is *true*. It's much like repeating an *if* statement until the condition fails.

So, for example, to write out the even numbers between 0 and 100 by using a *while* loop, you could do something like this:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        @{
            var val=0;
            while(val<=100)
            {
                <p>@val</p>
                val+=2;
            }
        }
    </body>
</html>
```

Here the code begins by creating a variable called *val*, assigned an initial value of 0. While *val* is less than or equal to 100, the code inside the block will repeat. This code writes out the variable value and then increments it by 2. After the block executes 51 times, *val* will have been incremented until it's 102. At that point, the *val < 100* condition fails, and thus the code inside the block won't execute any more.

Try this exercise. See what it would take to list all the even numbers from 100 down to 0, instead of the other way around. How do you think you would do that?

The *foreach* loop The .NET Framework has a special kind of loop that allows you to loop through a collection of objects. If you know that an object is a collection (or an array, which contains a list of values, if you are familiar with that), there's an easy way to examine each of the items in the collection without knowing what they are called. So, for example, all servers have several variables associated with them that an advanced programmer can use to check the state of the server. In the .NET Framework, these are stored in the *Request.ServerVariables* collection. An easy way to loop through the values in the *Request.ServerVariables* collection is to write a *foreach* loop that does something along the lines of, "For each item in the collection, assign that item to a variable, and then do something with it."

Here is what that would look like in code:

```
@foreach (var serverVariable in Request.ServerVariables)
{
    <p>@serverVariable</p>
}
```

This loop iterates over the values in *Request.ServerVariables*. For each value, it assigns that value to a variable named *serverVariable* and then displays that value. You can see the result of running that code in Figure A-2.

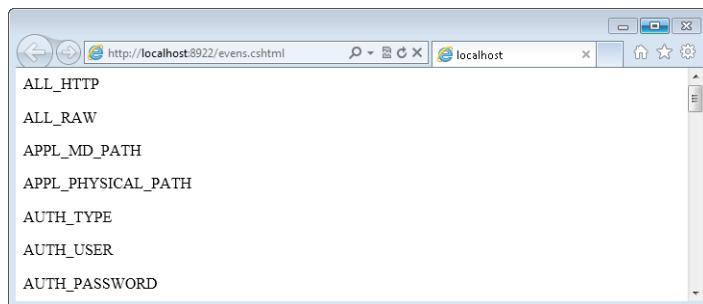


FIGURE A-2 Using the foreach loop.

As you work through this book, you're likely to see and use a lot of looping. Loops are probably the single most common programming construct. If you haven't done a lot of programming before, it might be a little confusing to decide which type of loop is most appropriate for what you want to accomplish, but in time such decisions will become second nature.

Summary

This appendix provided a little taste of server programming with WebMatrix. You examined a program that used dynamic server code to generate HTML that was returned to the user and displayed in a browser. You explored some of the programming basics involved in creating web applications, including understanding variables and data types, and using common programming terms and concepts such as conditions and loops.

Index

Symbols

& (ampersand), separating parameters in URLs 74
* (asterisk), wildcard character in SQL 127
@ (at sign)
 bold text in ASP.NET code 15
 @Facebook.ActivityFeed() method 223
 @Facebook.Comments() method 221
 @Facebook.Facepile() method 225
 @Facebook.GetInitializationScripts() method 221
 @Facebook.LiveStream() method 226
 @Facebook.Recommendations() method 224
 @foreach() method 194
 @functions syntax 261
 @helper.GetWidget() method 256
 @HelperName.FunctionName(params) syntax 264
preceding parameters in SQL INSERT 131
preceding server-side code 58
SQL parameters, @0, @1, etc. 206
@String.Format() method 203
@Translator.GetWidget() method 256
using with {} (code block) 193
@Video.Flash 96
@Video.MediaPlayer helper 93
@Video.Silverlight 96
{ } (curly braces)
 enclosing code blocks 62, 307
 enclosing CSS style properties 179
. (dot) syntax 179
= (equals sign)
 = (assignment) and == (equality) operator 308
(number sign)
 href placeholder for <a> tags 176
 preceding class ID in CSS 181
() (parentheses), surrounding conditions in if
 statements 307
% (percent sign), wildcard in SQL 127
? (question mark), preceding parameters in URL 74
~ (tilde), ~\ in relative paths 77

A

absolute paths 77
accept attribute, <form> tags 105
accept-charset attribute, <form> tags 105
action attribute, <form> tags 105
 email form 172
 leaving blank 106
Active Server Pages eXtended (ASPX) 39
activity feed, Facebook 223
adaptive payments interface, PayPal helper 248

add data page, creating 197–202
 adding data to the database 200–202
 creating form for user input 198
 handling submitted data from add form 199
Add Existing Files menu 72
Add New Page screen 293
Add To Cart button 238, 241
administration
 ASP.NET Web Pages Administration 213–216
 site administrator account for WordPress site 288
 WordPress administrator dashboard 290
Administrator Password setting 287
anchor tags. *See* <a> tags
API access, live (PayPal) 248
API Credentials, PayPal Sandbox account 234
API key, getting for Translator API 257
App configuration screen, Facebook 219
App_Data folder, _Password.config file 215
App ID and App secret, from Facebook 219
application/x-www-form-urlencoded 106
Applied Innovations hosting service 270
 signing up for WebMatrix site 271
 URL for site hosted by 276
arrays, looping through 311
ascending and descending order sorts in SQL 127
AsDateTime() method 307
AsDecimal() method 307
AsInt() method, string objects 306
aspect ratio of images 71
ASP.NET 2
 applications 18
 in web stack 3
 Web Helpers Library 147
ASP.NET Web Pages 15
 installed with WebMatrix 11
 in web stack 2, 3
 in WebMatrix stack 4
ASP.NET Web Pages Administration
 accessing 213–216
 installing Facebook helpers from NuGet 217–218
ASPx (Active Server Pages eXtended) 39
assigning values to variables 308
asterisk (*), wildcard character in SQL 127
<a> (anchor) tags 176
 defining appearance of, when mouse points to 182
 linking edit page to web page 203
 wrapping items in 176
at sign. *See* @ (at sign), under Symbols
attributes
 defining for HTML elements 176
 specifying appearance of HTML elements 178
 <video> tag 98

audio file format (WMA) 93
 authentication, role-based system 25
 autoplay attribute, <video> tag 98
 autoStart property, Video.MediaPlayer 93
 Auto (translation setting) 252

B

Bakery template 23
 building a site from 9
 creating BakeryPP site (example) 235
 creating site with translation capabilities 251–255
 running PayPal-enabled Bakery site 238
 running the Bakery site 10
Bing
 API key for using Translator API 257
 search results, SEO efficiency 45
BlogEngine.NET 20
 downloading and installing 22
 blogs 19
<body> tags
 adding content 55
 adding <form> element 60
 adding tags 68
 adding upload form 76
 empty, in CSS file 184
 @Facebook.GetInitializationScripts() method 221
bookmarks
 adding using Delicious 149
 organizing, service provided by Delicious 148
 sharing, using Google Reader 153
bool data type 306
break statements 308
browsers. See **web browsers**
 building a simple web application 173–190
 creating and styling a site 173–187
 using layout pages and templates 187–190
bulleted lists 181
Button Manager, PayPal 241
 implementing AddToCart button 237
 implementing Buy Now button 243
 implementing Donate button 244
 implementing Subscribe button 247
Buy Now button 242
 implementing in PayPalOrder.cshtml page 243

C

C# 39, 255
 assignment (=) and equality (==) operators 308
 choosing for Translator API method 260
 code blocks 307
 Razor syntax 15
 capturing form input 120–122
 cascading style sheets. See **CSS**
 case statements 308
 certificates for running SSL 35

character sets used by browser for form data 105
 chat, Facebook Live Stream feed 226
 check boxes 114
 Choose a File Type dialog box 38
 Choose A File Type dialog box
 Web.config file 80
Chrome
 downloading 98
 viewing HTML5 H.264 video in 100
 class attribute, specifying for <h1> elements 179
classID
<object> tags bound to 92
 Silverlight using <object> tag without classID 97
 Windows Media Player 89
client programming 51
closing HTML tags 54
CMS (content management systems) 20
code
 editing in WordPress using WebMatrix 299–304
 editing with WordPress code editor 297
 preceding with @ (at sign) 58
 repeating with loops 309
collections, looping through 311
colors
 background color creating highlighted effect 182
 changing for fonts 178
columns, adding to database table 124
comments box, Facebook 220–223
 replacing built-in comments engine in
 WordPress 299–304
Compact SDF databases 40
completion status, publication progress 276
conditional statements
 if statement 307
 switch statement 308
conditions syntax in SQL statements 127
configuration file for websites (Web.config) 80–82
 adding to a site 80
 editing maximum request length 81–82
confirmation page for test payment (PayPal) 240
connections, database
 connection string not required for SQL Server
 Compact 274
 setting for MySQL-based application 277
content management systems (CMS) 20
controls attribute, <video> tag 98
controls (form) 109–119
 check boxes 114
 option buttons 112
 password boxes 110
 select control for lists 117
 TextArea 115
 text boxes 109
Create New Database dialog box 287
CRUD (Create, Read, Update, Delete) operations 173
CSHTML
 creating and editing a page 57–59

CSHTML files 39
CSS (cascading style sheets) 38
 code syntax 179
 defining appearance of <a> tags within tags 182
 defining appearance of <a> tag when mouse pointer
 is over it 182
getting page ready for CSS 176–178
setting style for specific elements within <div>
 tags 181
style “language” 179
styling websites 178
using CSS files 184–187
curly braces ({ })
 enclosing code blocks 62, 307
 enclosing CSS style properties 179
current date and time 58
Cytanium hosting provider 151

D

Database Administrator setting 287
Database.Open() method 193
databases 123–146
 adding data 130–133, 200–202
 in ASP.NET in web stack 5
 connection strings for 274
Create New Database dialog box 287
creating 191
 creating add data page for 197–202
 creating with WebMatrix 123–126
 deleting records 140–145
 deleting records from 207–212
 dynamic website data stored in 10
 editing records 134–140
 framework support of 3
 pointing data retrieval page to 193
 retrieving data from, using SQL 193
sites deployed using FTP 278
SQL Server and SQL Server Compact 4
 updating 206
used with PHP on Windows stack 6
using in code 126–129
in web stack 2
WebMatrix database designer 14
working with, in WebMatrix 15
Databases workspace 26, 40–48
 creating a database 40, 191
 creating and executing queries 127
 creating and using tables 41
 editing a table 42
 New Database button 123
 querying data 43
data entry view, database tables 125
data retrieval page, creating 192–197
 foreach loop 194
 SQL SELECT FROM command 193
 writing row details into list items 195

data types 305–307
 converting variable types 306
 specifying for variables 305
Data View, opening database table in 42
dates and times
 converting value to DateTime object 307
 current, creating dynamic HTML page to
 display 57–59
 current, creating static HTML page to display 52–57
 DateTime object 306
 entering dates in database table 192
DateTime.Now.ToString() function 58
db variable 129
decimal data type 307
dedicated hosting 268
Default.cshtml page 15
default pages 36
DELETE command (SQL) 209
 examples 142
delete data page, creating 207–212
 full code 209
 viewing delete page and deleting an item 211
 view to-do list page with delete links (example) 210
deleting database records 140–145
Delicious 148–150
 bookmark added to 150
 Save Bookmark page 149
denial-of-service attacks, maximum request length
 and 82
deploying a site 267–280
 creating WordPress-based site 277–279
 finding web hosting provider 267–272
 using Publish Settings dialog box 272–277
descending order sorts in SQL 127
Digg 151–153
 adding a site to 152
<div> tags 15
 dividing page into logical blocks 175
 styling with CSS 179, 181
documentation
 Facebook helpers 224
 MSDN, for Translator API 259
 PayPal helper 248
Donate button 242, 244–246
dot () syntax 179
dynamic websites 10

E

eCommerce applications 20
 Bakery template 23
editing data in database table 42
edit links in index.cshtml file 135
edit page 202–207
 creating edit.cshtml page for database edits 135–140
 handling submitted data from edit form 202–206
 updating the database 206
elements, HTML 54

else statements 307
 email 163–172
 address for PayPal account 229
 address for PayPal Sandbox account 232
 creating simple email application 167–172
 publish settings attachment 272
 using SMTP (Simple Mail Transfer Protocol) 163
 using WebMail helper 164–166
 eMailSent variable 170
 eMailSubject, eMailMessage and eMailAdditional
 variables 171
 <embed> tags 91
 deprecated in favor of <object> tags 93
 Empty Site template
 creating a site with 52
 selecting for a site 173
 enableContextMenu property, Video.MediaPlayer 93
 encryption
 password box content 111
 enctype attribute, <form> tags 106
 End User License Agreement (EULA) 21
 equality, testing values for 308
 eRightSoft, SUPER file conversion tool 95
 EULA (End User License Agreement) 21
 Execute() method
 database updates 206
 inserting data into database 201
 parameterized queries 132
 eXtensible Markup Language. *See XML*

F

Facebook 154, 213–228
 activity feed, using 223
 comments box, adding to WordPress site 302–304
 comments box, using 220–223
 content shared with StumbleUpon 158
 creating an application 299–301
 Facepile feed, using 225
 Live Stream feed 226
 recommendations, using 224
 setting up a Facebook application 219
 sharing your site on 155
 Social Plugins page 301
 Facebook helpers
 documentation 224
 getting started with 218
 installing 213
 installing helpers from NuGet 217
 Facebook Markup Language (FBML), namespaces 220
 Facepile feed, using 225
 fallback HTML 89
 favicon.ico files 33
 Fiddler, exploring HTTP headers with 107–109
 inspecting GET request and response 108
 inspecting web traffic 107
 POST request and response 109

file formats
 images 68
 video 87
 files
 Add Existing Files menu 72
 adding to a site 68
 image file name versus path 78
 using external CSS file 184–187
 Files workspace 26, 37–39
 adding a file 53
 creating new files 37
 creating todo.cshtml file (example) 174
 deleting a file 52
 file types 38
 refreshing 218
 ribbon 37
 Web.config files 80
 file types
 Choose a File Type dialog box 38
 HTML file type 53
 list of common types 38
 filtering monitored requests in Site workspace 31
 Firefox
 localhost address, access by Fiddler 107
 preference for <embed> tag instead of <object>
 tag 92
 running WebMatrix site in 107
 Flash videos
 file format 87
 Video.Flash helper 95
 float data type 306
 FLV files 87, 95
 folders
 creating 37
 Site From Folder menu option 18
 fonts
 changing font and color for <h1> elements 178
 specifying font face and size with CSS file 38
 footers
 adding to web pages 177
 editing in WordPress site 297
 styling with CSS file 186
 foreach loops 311
 using in database queries 129
 foreach() method 194
 for loops 309–311
 writing out even numbers between 0 and 100 309
 formatting dates and times 58
 forms 60, 103–122
 adding data to a database 130–133
 capturing input 120–122
 check box controls 114
 creating delete item form 207–211
 creating for add data page 198
 creating form allowing user input 60
 creating upload form 76
 deleting database records 142
 editing database records 135–140
 email 167–172

exploring HTTP headers with Fiddler 106–109
 handling submitted data from add form 199
 handling submitted data from edit form 202–206
 how they work 103
 option buttons 112
 password box controls 110
 posting to a server with HTML 62–65
 select control for lists 117–122
 simple forms-based application example 104
 TextArea controls 115
 text box controls 109
 <form> tags 60, 104
 attributes 105
 <input> controls 61
 forums 20
 frameworks 2
 ASP.NET Web Pages 15
 French, translating English text to, using Translator helper 264
 FTP, difficulty of deploying site data over 278
 functions, defining for Translator helper 261

G

galleries 20
 Photo Gallery template 23
 Gamercards, Xbox, rendering 161
 gamers, Xbox Live social network 161
 GetImageFromRequest method 78
 GET method, HTTP 31, 61, 103
 determining whether page is retrieved by using GET or POST 200
 inspecting request and response with Fiddler 108
 results of GET request 63
 submitting forms 120
 GetTranslation() function 261
 using 264
 globally unique identifier (GUID) 89
 Google Chrome. *See* Chrome
 Google Reader 153

H

<h> (heading) elements 55
 <h1> (heading) tags
 setting CSS style for 185
 styling using CSS 179
 styling using inline markup 178
 H.264 video format 98
 web browsers' support of 99–101
 <head></head> element tags 54
 headers
 adding to web pages 177
 setting CSS style for <header> and <h1> tags 185
 headers, HTTP
 checkbox form control name and values 114
 inspecting with Fiddler 106–109

request header with values of selected options 113
 TextArea form control data in 116
 headings. *See also* <h1> tags
 creating in HTML page 55
 height attribute
 tags 69
 <video> tags 98
 height property
 Video.MediaPlayer 93
 helpers
 creating for Translator widget 255
 creating using Translator API 257–265
 initializing PayPal helper 236
 installing 147
 LinkShare 147–162
 PayPal helpers package, downloading 235
 Video helper 93–97
 WebMail 164–166
 Hosting Gallery website 267
 Narrow Results tool 269
 spotlight sites 272
 hosting providers 151
 hosting services 267–272
 dedicated hosting 268
 shared hosting 267
 signing up with 271
 virtual hosting 268
 Hotmail 163
 settings for email account 164
 href attribute
 <a> tags 176
 # (number sign) placeholder for 176
 HTML
 creating files 38, 53
 editing code 54
 generated on the fly for dynamic websites 15
 headings (<h>) tags 55
 tags 31
 source code, as browser sees it 56
 unordered list () 174
 HTML5
 <header> and <footer> tags 177
 MP4 video format 87
 <video> tag 93, 98–101
 HTML forms. *See* forms
 <html></html> tags 54
 HtmlString object 242
 HTTP
 determining if page is retrieved by GET or POST method 200
 GET and POST methods 31, 103
 header containing check boxes' names and values 115
 headers 106
 POSTing form data to the server 60–65
 request header with values of selected options 113
 status messages 32
 use with helper created using Translator API 259
 http://localhost:<NUMBER> URLs 34

HTTPS 35
 SSL (Secure Sockets Layer) encryption 111
 hyperlinks. *See links*
 hypertext reference (HREF) 176

I

icons, Show Hidden Icons button 13
 identity field (SQL Server) 124
 IDs, naming elements by 181
 if(IsPost) check 62, 120
 code storing data in database 131
 using for submitted form data 200
 if statements 307
 IIS Express server 2, 12
 controlling from Site workspace ribbon 29
 enabling SSL 35
 installed with WebMatrix 11
 IIS (Internet Information Services) 2
 imagename parameter 75
 image requests, filtering 31
 images 67–86
 creating page that uses an image 67–70
 creating thumbnail and adding link 70–73
 file formats 68
 programming an image tag 73–76
 thumbnails and links 70–73
 WebImage helper 76–85
 tags 31, 67
 height and width attributes 69
 programming 73–76
 src attribute 68, 78
 for thumbnail image 72
 Import Publish Settings option, Publish Settings dialog box 273
 index.cshtml file
 adding hyperlink to video file 88
 editing to add tag 68
 editing to add tag pointing to thumbnail 72
 edit links 135
 inline markup 179
 input
 capturing form input 120–122
 textarea input control, for email body 170
 users adding data to database 130–133
 <input> tags 61
 check boxes 114
 creating input controls for add data form 199
 option buttons 112
 password boxes 110
 text box form control 109
 value attribute 64
 INSERT command (SQL) 200
 examples 131
 storing data from input form in database 132
 installing
 WebMatrix 6–8
 WebMatrix helpers 147

integers, converting strings to int type 306
 Internet Explorer
 choosing to run starter site in 28
 IE 9, supporting HTML5 98
 response to SSL site with no certificate present 35
 scaling images to current browser size 73
 using Fiddler with 107
 using HTML5 in IE 8 99
 viewing HTML5 H.264 video in IE 9 101
 viewing source in IE 9 203
 View Source command 56
 Internet Information Services. *See IIS; IIS Express server*
 Invalid Data message box 42
 IP address in email header 166
 IsBool() method 306
 IsDateTime() method 307
 IsDecimal() method 307
 IsFloat() method 306

J

Japanese, web page translated to 254
 JPEG (or JPG) images 68
 JScript files 38

L

languages
 choosing from Translator widget 254
 codes for, in translation helper functions 264
 launching WebMatrix 17
 layout pages
 creating 187–190
 _SiteLayout.cshtml page, SocialBakery site 218, 221
 using for data retrieval page 192
 Learn About WebMatrix Online menu option 17
 Like button (Facebook), adding to your page 218
 LIKE keyword, in SQL statements 127
 Link Directory template 23
 links
 adding to thumbnail image 73
 adding to video file 88
 creating using <a> tags 176
 LinkShare helper 147–159
 adding your page to Digg 151–153
 adding your site to Delicious 148–150
 creating share link using Facebook 154
 enabling users to add site to Google Reader 155
 sharing your site on Reddit 156
 sharing your site via StumbleUpon 157
 sharing your site via Twitter 158
 using 148
 <link> tags 184, 186
 Linux web platform 1
 lists
 creating HTML unordered list () 174
 default rendering of items in lists 181
 select control in forms 117–119

 tags 15
 default rendering of objects in lists 181
 linking edit page to list items 203
 styling appearance of <a> tags within 182
 turning items into hyperlinks 176
Live Stream feed, Facebook 226
loop attribute, <video> tag 98
loop counter 309
loops 309
 foreach loop 311
 for loop 309–311
 while loop 311
writing foreach loop to query database 194

M

mail services, free 163
Manual (translation setting) 252
Maximum Request Length Exceeded error 80
media files, conversions 95
media players
 contained in SWF files 95
 embedding using <object> tag 89–93
 embedding Windows Media Player on web page 90
 Silverlight 97
 using Video.MediaPlayer helper 93–97
message control, email form 170
method attribute, <form> tags 106
Microsoft
 ASP.NET Web Pages Administration page 213
 Hosting Gallery website 267
 Translator Developer Offerings page 259
 Translator widget 251
 WebMatrix site hosting providers 151
Microsoft.com website 6
Microsoft Visual Web Developer 2010 Express 37
Microsoft WebMatrix, on Windows Start menu 17
Microsoft web platform 1
 applications running on 18
MIME types
 official list of 92
 specifying in <embed> tags 92
"missing images" error 31
mouse, hovering over an element 182
MOV files 98
MP4 video format 87
.MSG files 166
multipart/form-data type 77, 106
multiple-selection lists 118
mute property, Video.MediaPlayer 93
My Sites button, Site workspace ribbon 27
My Sites dialog box 27
My Sites menu option 18
MySQL Connector/Net component 285
MySQL databases 40
 installing MySQL 283
 root password management 284
 use in WordPress-based site 277–279
 use with PHP web stack 6

N

name attribute, <form> tags 106
namespaces, Facebook Markup Language (FBML) 220
Narrow Results tool, Hosting Gallery 269
.NET Framework
 foreach loop 311
 object types used as variable types 306
.NET Framework
 DateTime object 58
 Request variable 63
New Database button 123
New Files dialog box 53
 creating CSS file 184
New Query button 127
New Report dialog box 45
New Table button 41, 123
Notify (translation setting) 252
Now property, DateTime object 58
NuGet feed 147, 213
 available packages 216
 Facebook helpers from, installing 217
 packaging and publishing your helper 265
numbers
 converting strings to 306
 converting text values to and summing 63
number sign (#)
 href placeholder for hyperlinks 176
 preceding class ID in CSS 181

O

<object> tags 89–93
 <embed> tags deprecated in favor of 93
 Silverlight using, without classID 97
 using <embed> tags instead of 92
 using to embed media player into webpage 89
opening and closing HTML tags 54
open source web applications 1
 PHP 3
Opera
 attempt to view HTML5 H.264 video in 100
 running starter site in 28
 running video with <object> tag 91
 using <embed> tag fallback for Windows Media
 Player 92
operating systems
 classNames specific to 92
 open source web applications on 1
 Windows, in web stack 2
option buttons in forms 112
 email form 170
<option> tags 117
ORDER BY clause, in SQL statements 127
Outlook, creating .MSG files from email messages 166

P

Package Manager 216
 Paint program, resizing an image with 71
 <param> tags, within <object> tags 89
 parameterized GET requests 103
 parameterized queries 131
 parameters in URLs 74
 parameters in WebMatrix pages 73
 creating for image and thumbnail 74
 parentheses (()), surrounding conditions in if statements 308
 password boxes 110
 Password.config file 215
 passwords
 managing MySQL root password 284
 required for WordPress site 289
 root password for MySQL 283
 for SMTP mail server 164
 path property, Video.MediaPlayer 93
 paths
 absolute and relative 77
 image path on server 78
 pattern matching, using LIKE keyword and wildcards in SQL 127
 payments, test business with PayPal account 239
 PayPal 229–250
 creating a Sandbox account 231–235
 creating a shopping cart 237
 creating PayPal-enabled Bakery site 235
 custom buttons and functionality 248
 Donate button 244–246
 exploring PayPalOrder.cshtml page 241
 going live with your site 248
 initializing PayPal helper 236
 running PayPal-enabled Bakery site 238
 setting up other types of payments 242
 signing up for an account 229–231
 single-item purchases, Buy Now button 243
 Subscribe button 246
 PayPalButton object 242
 PayPal helpers package, downloading 235
 PayPal.Profile.Initialize() method 249
 percent sign (%), wildcard in SQL 127
 Personal, Premier, or Business account, PayPal 229
 photo galleries 20
 Photo Gallery template 23
 Photo Viewer 71
 PHP
 applications running on Microsoft Web Platform 18
 editing theme file in WebMatrix 302
 PHP on Windows web stack 3, 5
 programming framework in web stack 2
 use in WordPress-based site 277–279
 using with WebMatrix 281
 placeholder parameters in queries 131
 playCount property, Video.MediaPlayer 93
 port numbers, in site settings 34

POST method, HTTP 31, 103
 browser sending form data to server 60
 determining whether page is retrieved by using GET or POST 200
 form data posted back to originating page 106
 handling POST requests 63–65
 inspecting request and response with Fiddler 108
 submitting forms 120
 posts
 adding Facebook comments functionality 304
 in WordPress site 291–293
 <p> (paragraph) elements 55
 practice files for this book 70
 preload attribute, <video> tag 98
 PrimePress theme 294, 302
 profiles
 Facebook, accessing 222
 Twitter, displaying 159
 ProgramData directory, deleting MySQL subdirectory 284
 programming frameworks 2
 programming with WebMatrix 305–312
 converting variable types 306
 first programmed page 52–57
 foreach loops 311
 for loops 309–311
 further information on 307
 making a page dynamic 57–59
 sending data to the server 60–65
 server programming 51
 specifying data type for variables 305
 testing conditions with if and switch 307
 while loops 311
 Publish button 28, 274
 publish compatibility, checking 275
 publishing your site to the Internet 28
 Publish Preview screen 275
 publish settings attachment to hosting service email 272
 Publish Settings dialog box 273–277
 database connections 274
 settings for PHP and MySQL-based application 277

Q

querying data in databases 43
 Query() method 193
 example 129
 QuerySingle() method 137, 205
 query variable 129
 QuickTime MOV files 98

R

radio buttons 112
 Razor syntax 15, 58
 recommendations, Facebook 224
 Reddit 156

- Redirect() method 201
 - calling after database updates 206
 - relative paths 77
 - RenderBody() command 187
 - Reports workspace 26, 44–48
 - creating a new report 45
 - details view of SEO violations report 47
 - example SEO violations report 46
 - exploring and fixing SEO violations 47
 - saving reports automatically 48
 - Request collection
 - maximum length of items placed in 80
 - parameters in WebMatrix pages 74
 - Request object, using to find value of a parameter 204
 - Requests button, Site workspace ribbon 30
 - Request.ServerVariables collection 311
 - Requests management, Site workspace 30–33
 - getting detailed information about a request 32
 - recommendations for solving request problems 32
 - viewing incoming requests 31
 - Request variable 63
 - Resize And Skew dialog box 72
 - resizing images 69
 - using WebImage helper 83–85
 - in Windows Photo Viewer, Paint 71
 - Restart button, Site template ribbon 29
 - ribbon
 - Databases workspace 40
 - Files workspace 37
 - Reports workspace 45
 - Site workspace 27–30
 - rich Internet applications (RIAs) 37, 51
 - role-based authentication 25
 - root password for MySQL 283
 - entering into Create New Database dialog box
 - settings 287
 - managing 284
 - Run button
 - running the Bakery site 10
 - on Site template ribbon 28
- S**
- Sandbox account, PayPal 231–235
 - API credentials 234
 - signing into, for test Bakery site 239
 - Test Accounts 233
 - Save Table dialog box 125
 - SDF (SQL Database File) 39
 - creating 40
 - Search Engine Optimization (SEO) reports 44–48
 - automatically saved by Reports workspace 48
 - creating 45
 - example of 46
 - exploring and fixing SEO violations 47
 - search results (Twitter), displaying on your site 160
- Secure Sockets Layer. *See SSL*
 - secure sockets via Telnet 164
 - security
 - ASP.NET Web Pages Administration 214
 - denial-of-service attacks, maximum request length and 82
 - preventing SQL injection attacks 131
 - SELECT command (SQL) 193
 - most common use 126
 - selecting database record to edit 205
 - select control for lists 117–119
 - changing list from drop-down to fully rendered list 118
 - users selecting multiple values 118
 - <select multiple> tags 118
 - <select>tags 117
 - Seller account, PayPal Sandbox test account 234
 - sending email 163
 - server address (SMTP) 164
 - server programming 51–66
 - creating a web page 52–57
 - making a page dynamic 57–59
 - sending data to the server 60–65
 - servers 2
 - starting, stopping, or restarting IIS Express server 29
 - server-side code
 - examining Default.cshtml page 15
 - running in Bakery website 10
 - server status notifications 29
 - shared hosting 267
 - shopping carts
 - creating 237
 - exploring PayPalOrder.cshtml page 241
 - running PayPal-enabled Bakery site 238–240
 - Show Hidden Icons button 13
 - Silverlight
 - rich Internet applications (RIAs) based on 37
 - using Silverlight video 96
 - single-item purchases, payment for 243
 - site administrator account for WordPress site 288
 - Site From Folder menu option 18
 - Site From Template dialog box 23
 - Site From Template menu 9
 - Site From Template menu option 18
 - Site From Web Gallery menu option 18
 - Site From Web Gallery option 281
 - _SiteLayout.cshtml page (SocialBakery site) 218
 - full code 221
 - sites
 - adding a file to 68
 - creating and styling 173–187
 - creating using a template 23–25
 - creating using Web Application Gallery 20–23
 - deploying. *See deploying a site*
 - live site with social links 151
 - new site options in WebMatrix 282

site settings 34–36
 configuring site to use SSL 35
 managing default pages 36
 URL string and port numbers 34
 Site workspace 25, 26–36
 finding web hosting for your site 267
 managing site settings 34–36
 remote publishing settings for your site 273
 Requests management 30–33
 ribbon 27–30
 SMTP (Simple Mail Transfer Protocol) 163
 server address and port 164
 social networking 147–162
 adding Twitter content to your site 159–161
 rendering Xbox gamecards 161
 using Delicious 148–150
 using Digg 151–153
 using Facebook 154
 using Google Reader 153
 using Reddit 156
 using StumbleUpon 157
 using Twitter 158
 Social Plugins page (Facebook) 301
 sorting SQL query results 127
 spotlight sites in Hosting Gallery 272
 SQL injection attacks 131
 SQL Server 4
 Compact SDF databases 40
 use with ASP.NET web stack 5
 SQL Server Compact
 ASP.NET Web Pages with 4
 creating a database using WebMatrix 123–126
 installed with WebMatrix 11
 no database connection string required for 274
 SQL (Structured Query Language) 43, 126
 DELETE command 142, 209
 INSERT command 131, 200
 query constructed and passed to `QuerySingle()`
 method 137
 retrieving data from a database 193
 testing queries with WebMatrix 127
 UPDATE command 138, 206
 src attribute
 tags 31, 68
 set to `srcPath` variable 75
 set to `stringPath` variable 78
 <video> tags 98
`srcPath` variable, creating for image tags 75
 SSL (Secure Sockets Layer) 35
 use by SMTP server 164
 stacks. *See* web stacks
 Starter Site template
 creating a site with 23–25
 creating database file for a site 40
 Start menu, launching WebMatrix 17
 Start, Stop, and Restart buttons, Site template ribbon 29
 static HTML 57
`stretchToFit` property, `Video.MediaHelper` 93
`strImgPath` variable, creating for an image 78
`String.Format()` command 203, 242
 strings, converting to numbers 306
 Structured Query Language. *See* SQL
 StumbleUpon 157
 <style> tags 180
 styling web sites 178
 subject and message controls, email form 170
 Submit button, forms 61
 submitted data from forms, handling
 add data form 199
 edit form data 202
 submit type, input controls 199
 Subscribe button 242, 246
 SUPER media file conversion tool 95
 SWF files 87
 converting other media file types to 95
 switch statements 308
 system tray, WebMatrix running IIS Express 13
<system.web> setting in Web.config file 81

T

tables, database
 creating 41, 191
 creating with WebMatrix 123
 data entry view 125
 editing 42
 entering data using WebMatrix 126
 Save Table dialog box 125
 tags, HTML 38, 54
 target attribute, <form> tags 106
 TCP/IP ports
 running web pages using HTTP protocol 34
 SMTP (Simple Mail Transfer Protocol) 164
 Telnet, secure sockets via 164
 templates 1
 creating a layout page 187–190
 creating a site with 23–25
 creating site using Empty Site template 52
 Empty Site template 173
 Site From Template menu 9, 18
 use of ASP.NET Web Pages stack with 3
 Test Accounts, PayPal Sandbox 233
 TextArea controls 115
 <textarea> tags 115
 text boxes 109
 on email form 170
 password box 110
 text, converting to numbers 63
 text files 39
 text input controls 61
 themes
 configuring WordPress site theme 294–296
 editing using WordPress code editor 296–299
 integrating Facebook comments into WordPress using
 WordMatrix 299–304
 WordPress, editing in WebMatrix 302

thumbnails 70
adding parameters to `` tag 74–76
creating and linking 71–73
creating using `WebImage` helper 83–85
titles
 setting title of a web page in HTML 54
 using the `<title>` tags 55
`todo.cshtml` file (example) 174
Translator API, creating a helper using 257
 choosing C# for translate method 260
 creating the helper 261
 full code for helper and widget 263
 getting API key 257
 using HTTP interface with 259
 using the helper 264
Translator Developer Offerings page 259
Translator widget 251
 adding to `About.cshtml` page 253
 choosing language for page translation 254
 creating a helper for 255
 specifying translation settings 252
troubleshooting, recommendations for solving request problems 32
Twitter 158
 adding Twitter content to your site 159
 displaying a Twitter profile 159
 displaying Twitter search results 160
 translation of stream on web page 255
`Twitter.Profile` web helper 159
`TXT` (text) files 39

U

`uiMode` property, `Video.MediaPlayer` 93
`` tags 15
 creating unordered to-do list 174
 default rendering of `` objects in 181
unique ID values, adding to database field 124
unordered lists. *See* `` tags
`UPDATE` command (SQL) 206
 example 138
upload form, creating 76
`URIs`, Microsoft Translator service 262
URL encoding, `TextArea` form control data 116
URLs
 dynamic web pages 11
 entering your site URL for Facebook app 220
 GET request parameters added to 103
 image path 78
 parameters in 74
 `PayPalOrder.cshtml` page 241
 setting to run site on your computer 34
 site hosted by Applied Innovations 276
username and password for SMTP mail server 164
users (required), for WordPress site 289

V

Validate Connection button, Publish Settings dialog box 274
value attribute
 form elements 205
 `<input>` elements 64
variables 305–307
 converting types 306
 creating 62
 initializing form variables 205
 specifying data type stored in 305
VBHTML (Visual Basic HTML) 57
video 87–102
 creating simple video site in WebMatrix 88
 embedding media player using `<object>` tag 89–93
 formats 87
 HTML5 `<video>` tag 98–101
 Video helper 93–97
Video helper
 Flash video 95
 Silverlight video 96
 `Video.MediaPlayer` helper 93–97
`<video>` tags 93, 98–101
View Source command, Internet Explorer 56, 203
Violation Details dialog box 48
virtual hosting 268
Visual C#. *See* C#
Visual Studio 37
Visual Web Developer 2010 Express 37
volume property, `Video.MediaPlayer` 93

W

Web Application Gallery
 categories of applications using 19
 creating a site 20–23
 opening 282
web applications, building. *See* building a simple web application
web browsers
 HTML5 and H.264 video support 98–101
 playing video 90
 selecting for running a site 28
 using `<embed>` tag instead of `<object>` tag 92
`Web.config` file
 adding to a site 80
 editing maximum request length 81–85
Web Deploy 267
 deployment of PHP/MySQL-based sites 278
web helpers 251–266. *See also* helpers
 creating helper for Translator widget 255
 creating using Translator API 257–265
 using Microsoft Translator widget 251–255
Web Helpers Library 147
 `Twitter.Profile` 159
 `Twitter.Search` 160

- web hosting, finding for your site 267–272
- WeblImage helper 76–85
 - resizing an image 83–85
 - using Web.config to change allowed image size 80–82
- web logs. *See blogs*
- WebMail helper 164–166
- WebMatrix
 - building your first application 8–11
 - creating site using Web Application Gallery 20–23
 - installing 6–8
 - installing helpers 147
 - launching 17
 - purpose and goals of 1
 - stack 11–16
 - Web Application Gallery 19
 - WebMatrix Beta 3 6
- web pages
 - creating and adding to WordPress site 292
 - in WordPress sites 291
- Web Pages Administration, ASP.NET 213–216
 - installing Facebook helpers from NuGet 217
 - sign-in screen 215
- Web Platform Installer (Web PI) 6
 - installing 7
- web servers 2
 - IIS 3
 - IIS Express 3, 12
- WebSiteCode property, PayPalButton object 242
- Web Site Packages administration page 213
- web sites. *See sites*
- web stacks 1–6
 - ASP.NET 5
 - ASP.NET Web Pages 4
 - combination of components used in building applications 3
 - components of 2
 - defined 1
 - PHP on Windows stack 5
 - WebMatrix stack 11
- Welcome screen, Microsoft WebMatrix 18
- WHERE clause, in SQL statements 127
- while loops 311
- width attribute, <video> tag 98
- width property, Video.MediaPlayer 93
- Wikipedia
 - information on HTML5 <video> tag and formats support 101
- wikis 20
- wildcards in SQL queries 127
- Windows Live Hotmail 163
- Windows Media files
 - WMV and WMA files 93
 - WMV files 87
- Windows Media Player
 - embedding in webpage 89–95
 - running using Video.MediaPlayer helper 94
- Windows operating system
 - open source web applications on 1
 - PHP on 3, 5
 - in web stack 2
- Windows Photo Viewer 71
- WMA files 93
- WMV files 87, 93
- WordPress-based site, creating 277–279, 281–291
 - administrator dashboard 290
 - configuring site theme 294–296
 - editing theme using code editor 296–299
 - integrating Facebook-driven comments engine 299–304
 - PHP and MySQL dependencies 283
 - posts and pages 291–293
 - setting up WordPress 286
 - users and passwords 289
- workbench 9, 22, 25–48
 - contents of 25
 - Databases workspace 40–48
 - Files workspace 37–39
 - Reports workspace 44–48
 - Site workspace 26–36
 - with starter site loaded 24
- workspaces. *See workbench*

X

- XAP files 97
- Xbox, rendering Xbox gamecards 161
- XML
 - files 39
 - returned by Translator service 262
 - Web.config file 81
- X-Originating-IP property (email) 166

Laurence Moroney



Laurence is a Senior Technology Evangelist at Microsoft, focusing on Silverlight and the user experience. He has more than a decade of experience in software development and implementation, and has written dozens of books and articles on Windows Presentation Foundation, Web development, security, and interoperability.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft®
Press

Stay in touch!

To subscribe to the *Microsoft Press® Book Connection Newsletter*—for news on upcoming books, events, and special offers—please visit:

microsoft.com/learning/books/newsletter