Introducing ASP.NET Web Pages 2

Mike Pope

Step-by-Step

Microsoft®

Introducing ASP.NET Web Pages 2

Mike Pope

Summary: This set of tutorials gives you an overview of ASP.NET Web Pages (version 2) and Razor syntax, a lightweight framework for creating dynamic websites. It also introduces WebMatrix, a tool for creating pages and sites. The tutorials take you from novice programmer through seeing your site live on the Internet. Topics include how to install Microsoft WebMatrix (a set of tools for creating sites); how to work with forms; how to display, add, update, and delete data; how to create a consistent site layout; and how to publish to the Web.

Microsoft®

Copyright © 2012 by Microsoft Corporation

Contents

This set of tutorials gives you an overview of ASP.NET Web Pages (version 2) and Razor syntax, a lightweight framework for creating dynamic websites. It also introduces WebMatrix, a tool for creating pages and sites.

Level: New to ASP.NET Web Pages.
Skills assumed: HTML, basic cascading style sheets (CSS).
Prerequisites: Windows XP SP3 or later. (See What Do You Need? later for more details.)
Downloads: Completed website for the ASP.NET Web Pages introductory tutorial

> **Note**   This tutorial is based on the ASP.NET Web Pages version 2 RC and Microsoft WebMatrix 2 RC.

Tutorial 1: Getting Started

What you'll learn in the first tutorial of the set:

- What ASP.NET Web Pages technology is and what it's for.
- What WebMatrix is.
- How to install everything.
- How to create a website by using WebMatrix.

Features/technologies discussed:

- Microsoft Web Platform Installer.
- WebMatrix.
- *.cshtml* pages

What Should You Know?

We're assuming that you're familiar with:

- **HTML**. No in-depth expertise is required. We won't explain HTML, but we also don't use anything complex. We'll provide links to HTML tutorials where we think they're useful.

- **Cascading style sheets (CSS)**. Same as with HTML.

- **Basic database ideas**. If you've used a spreadsheet for data and sorted and filtered the data, that's the level of expertise we're generally assuming for this tutorial set.

We're also assuming that you're interested in learning basic programming. ASP.NET Web Pages use a programming language called C#. You don't have to have any background in programming, just an interest in it. If you've ever written any JavaScript in a web page before, you've got plenty of background.

Note that if you are familiar with programming, you might find that this tutorial set initially moves slowly while we bring new programmers up to speed. As we get past the first few tutorials, though, there will be less basic programming to explain and things will move along at a faster clip.

What Do You Need?

Here's what you'll need:

- A computer that is running Windows 7, Windows Vista SP2, Windows XP SP3, Windows Server 2003 SP2, Windows Server 2008, or Windows Server 2008 R2.
- A live internet connection.
- Administrator privileges (required for the installation process).

What Is ASP.NET Web Pages?

ASP.NET Web Pages is a framework that you can use to create dynamic web pages. A simple HTML web page is static; its content is determined by the fixed HTML markup that's in the page. Dynamic pages like those you create with ASP.NET Web Pages let you create the page content on the fly, by using code.

Dynamic pages let you do all sorts of things. You can ask a user for input by using a form and then change what the page displays or how it looks. You can take information from a user, save it in a database, and then list it later. You can send email from your site. You can interact with other services on the web (for example, a mapping service) and produce pages that integrate information from those sources.

What Is WebMatrix?

WebMatrix is a tool that integrates a web page editor, a database utility, a web server for testing pages, and features for publishing your website to the Internet. WebMatrix is free, and it's easy to install and easy to use. (It also works for just plain HTML pages, as well as for other technologies like PHP.)

You don't actually *have* to use WebMatrix to work with ASP.NET Web Pages. You can create pages by using a text editor, for example, and test pages by using a web server that you have access to. However, WebMatrix makes it all very easy, so these tutorials will use WebMatrix throughout.

About These Tutorials

This tutorial set is an introduction to how to use ASP.NET Web Pages. There are 9 tutorials total in this introductory tutorial set. It's part of a series of tutorial sets that takes you from ASP.NET Web Pages novice to creating real, professional-looking websites.

This first tutorial set concentrates on showing you the basics of how to work with ASP.NET Web Pages. When you're done, you can work with additional tutorial sets that pick up where this one ends and that explore Web Pages in more depth.

We deliberately go easy on the in-depth explanations. And whenever we show something, for this tutorial set we always chose the way that we think is easiest to understand. Later tutorial sets go into more depth and show you more efficient or more flexible approaches (also more fun). But those tutorials require you to understand the basics first.

The tutorial set you've just started covers these features of ASP.NET Web Pages:

- Introduction and getting everything installed. (That's in the tutorial you're reading.)
- The basics of ASP.NET Web Pages programming.
- Creating a database.
- Creating and processing a user input form.
- Adding, updating, and deleting data in the database.

At any point you can publish (deploy) your site to a hosting provider. We'll talk about that at the end of this tutorial set and link you to a tutorial on how to do that.

What Will You Create?

This tutorial set and subsequent ones revolve around a website where you can list movies that you like. You'll be able to enter movies, edit them, and list them. Here are a couple of the pages you'll create in this tutorial set. The first one shows the movie listing page that you'll create:
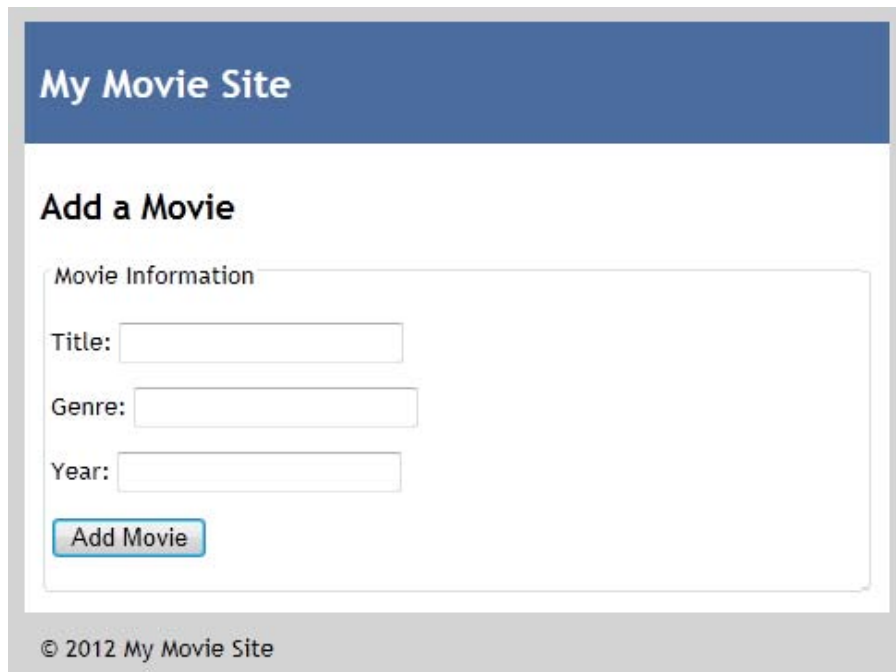
And here's the page that lets you add new movie information to your site:

Subsequent tutorial sets build on this set and add more functionality, like uploading pictures, letting people log in, sending email, and integrating with social media.

Ok, let's get started.

> **Note** You can [download](#) a finished version of the website that's described in these tutorials.

Installing Everything

You can install everything by using the Web Platform Installer from Microsoft. In effect, you install the installer, and then use it to install everything else.

To use Web Pages, you have to be have at least Windows XP with SP3 installed, or Windows Server 2008 or later.

On the [Web Pages page](#) of the ASP.NET website, click **Install WebMatrix**.
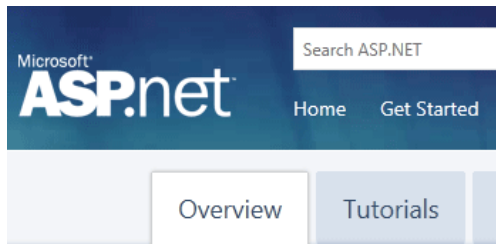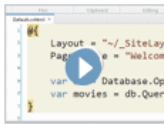
This button takes you to the [Web Platform Installer page](#) on the Microsoft.com site.

If the download doesn't start automatically, click the **Install Now** button. Then click **Run**. (If you want to save the installer, click **Save** and then run the installer from the folder where you saved it.)
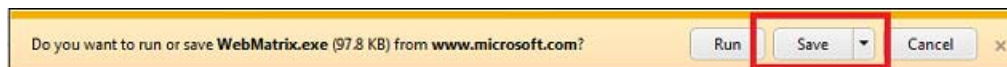
The Web Platform Installer appears, ready to install WebMatrix. Click **Install**.
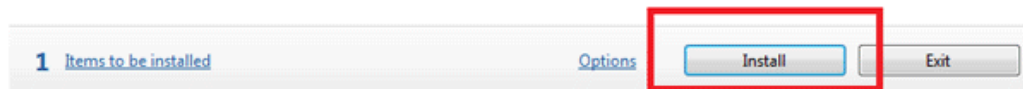
The installation process figures out what it has to install on your computer and starts the process. Depending on what exactly has to be installed, the process can take anywhere from a few moments to several minutes.

Hello, WebMatrix

When it's done, the installation process can launch WebMatrix automatically. If it doesn't, in Windows, from the **Start** menu, launch **Microsoft WebMatrix**.

In Windows, start **Microsoft WebMatrix**.
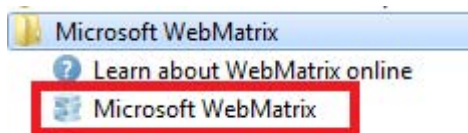
To begin, you'll create a blank website and add a page. In the next tutorial set, you'll use one of the built-in website templates.

In the start window, click **Templates**. Templates are prebuilt files and pages for different types of websites.
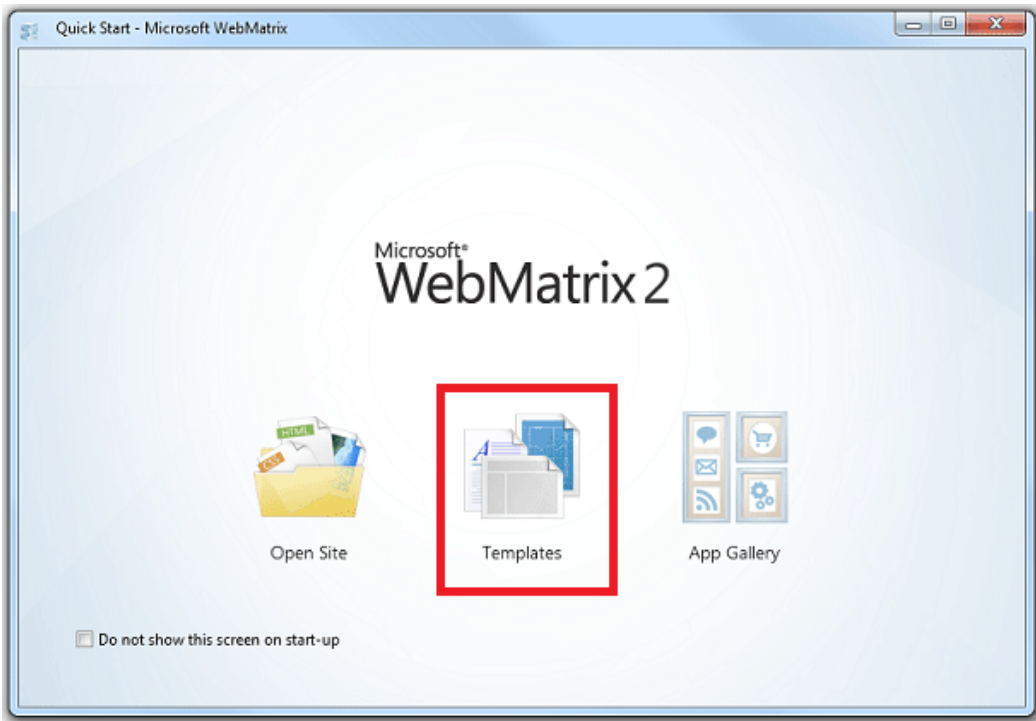
In the **Quick Start** window, select **Empty Site** and name the new site "WebPagesMovies".
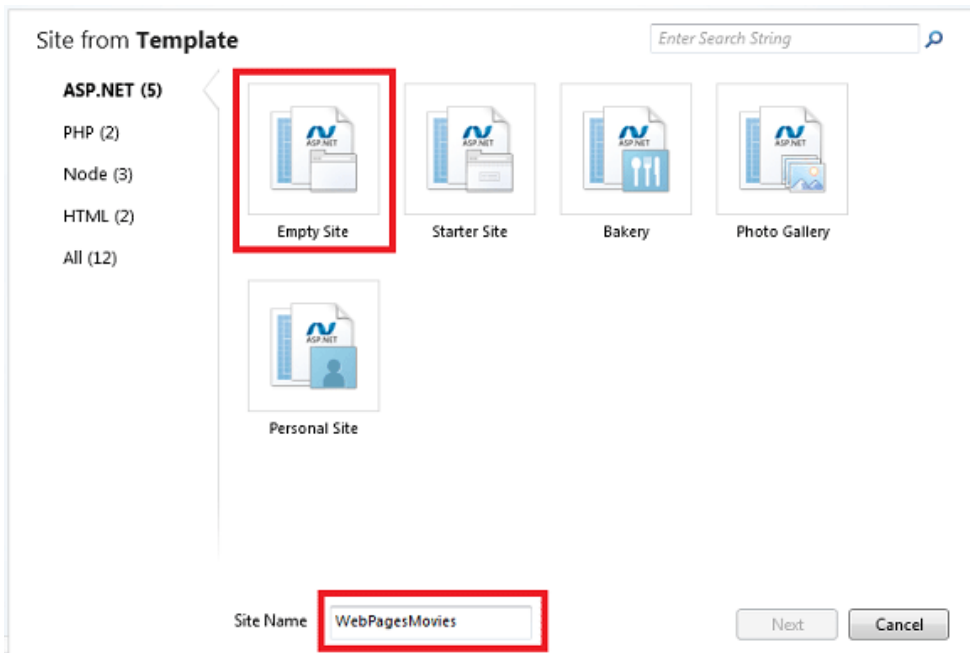
Click **Next**.

WebMatrix creates and opens the site:
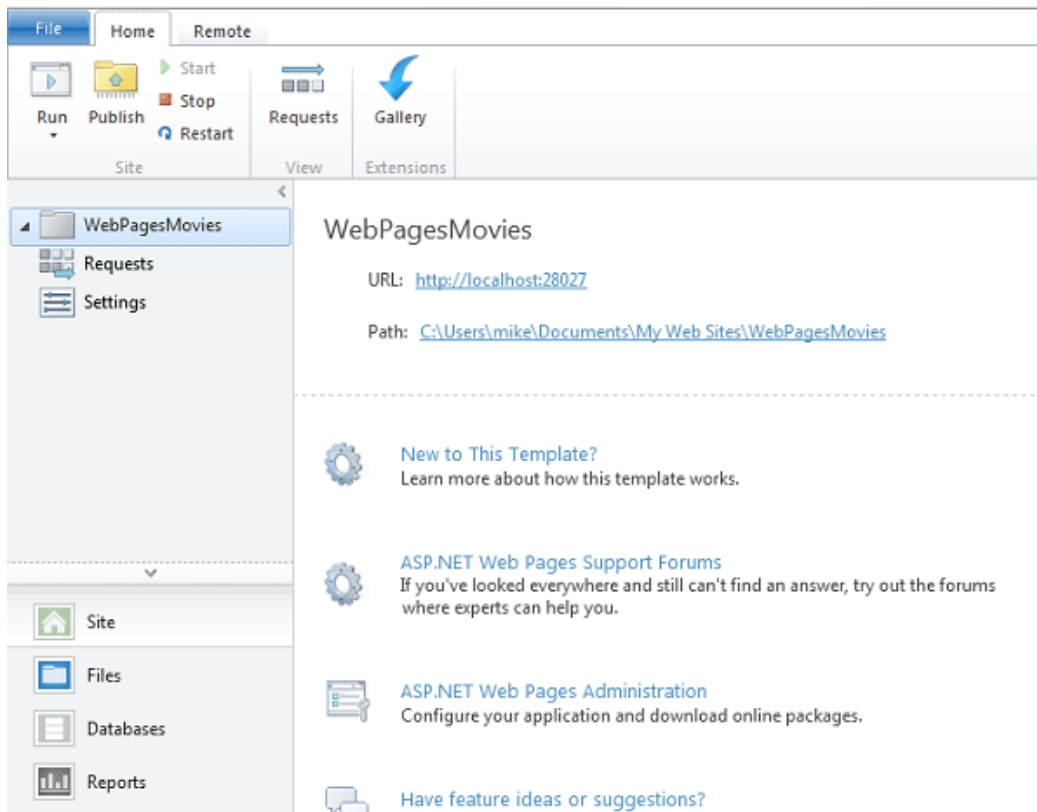
At the top, there's a Quick Access Toolbar and a ribbon, like in Microsoft Office 2010. At the bottom left, you see the workspace selector where you switch between tasks (**Site**, **Files**, **Databases**, **Reports**). On the right is the content pane for the editor and for reports. And across the bottom you'll occasionally see a notification bar for messages.

You'll learn more about WebMatrix and its features as you go through these tutorials.

Creating a Web Page

To become familiar with WebMatrix and ASP.NET Web Pages, you'll create a simple page.

In the workspace selector, select the **Files** workspace. This workspace lets you work with files and folders. The left pane shows the file structure of your site. The ribbon changes to show file-related tasks.
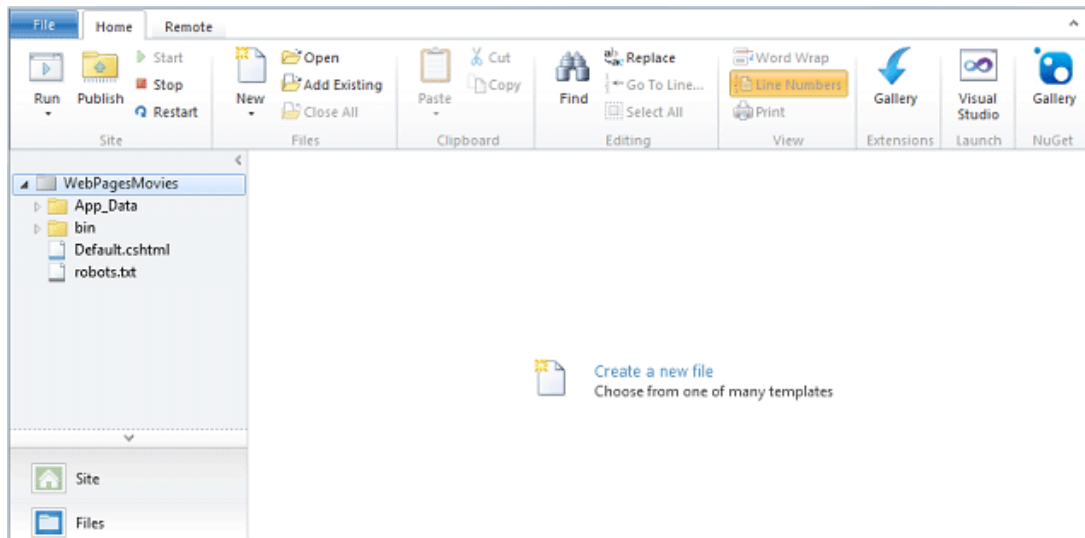
In the ribbon, click the arrow under **New** and then click **New File**.
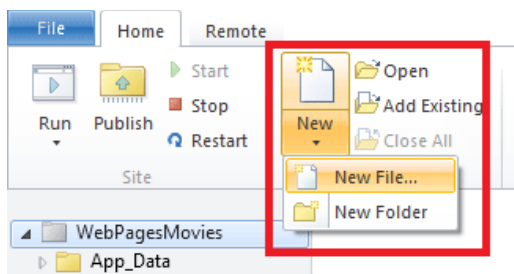
WebMatrix displays a list of file types. Select **CSHTML**, and in the **Name** box, type "HelloWorld". A CSHTML page is an ASP.NET Web Pages page.
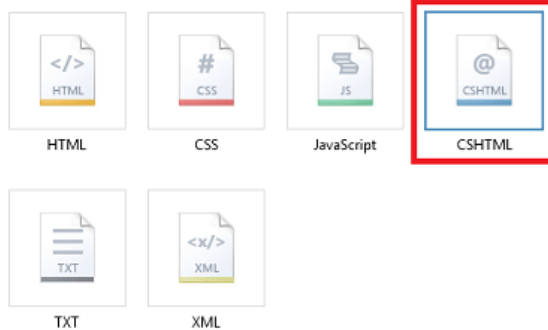
Click **OK**.

WebMatrix creates the page and opens it in the editor.
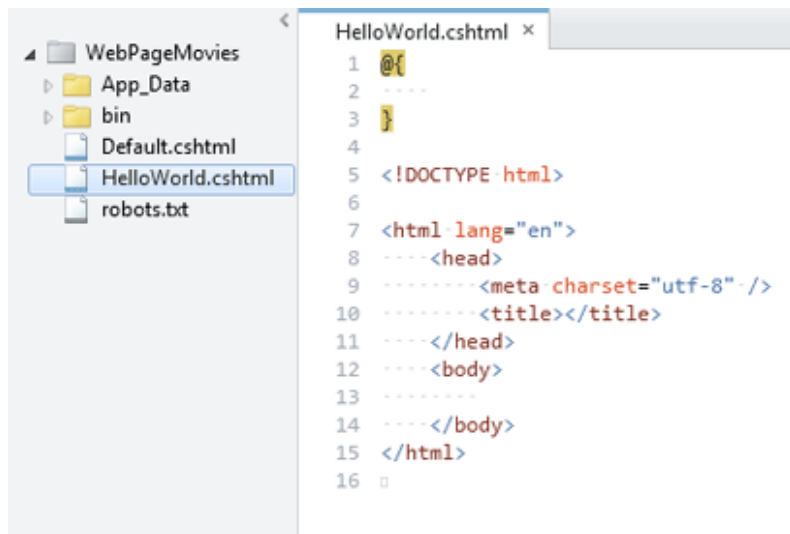
As you can see, the page contains mostly ordinary HTML markup, except for a block at the top that looks like this:

```
@{

}
```

That's for adding code, as you'll see shortly.

Notice that the different parts of the page — the element names, attributes, and text, plus the block at the top — are all in different colors. This is called *syntax highlighting*, and it makes it easier to keep everything clear. It's one of the features that makes it easy to work with web pages in WebMatrix.

Add content for the `<head>` and `<body>` elements like in the following example. (If you want, you can just copy the following block and replace the entire existing page with this code.)

```
@{

}

<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title>Hello World Page</title>
</head>
<body>
<h1>Hello World Page</h1>
<p>Hello World!</p>
</body>
</html>
```

In the Quick Access Toolbar or in the **File** menu, click **Save**.
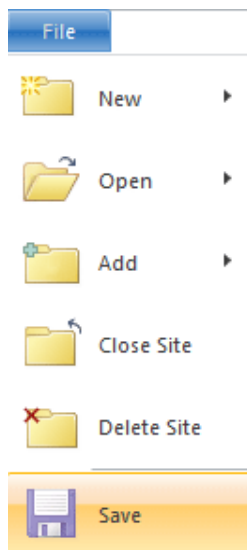
Testing the Page

In the **Files** workspace, right-click the *HelloWorld.cshtml* page and then click **Launch in browser**.
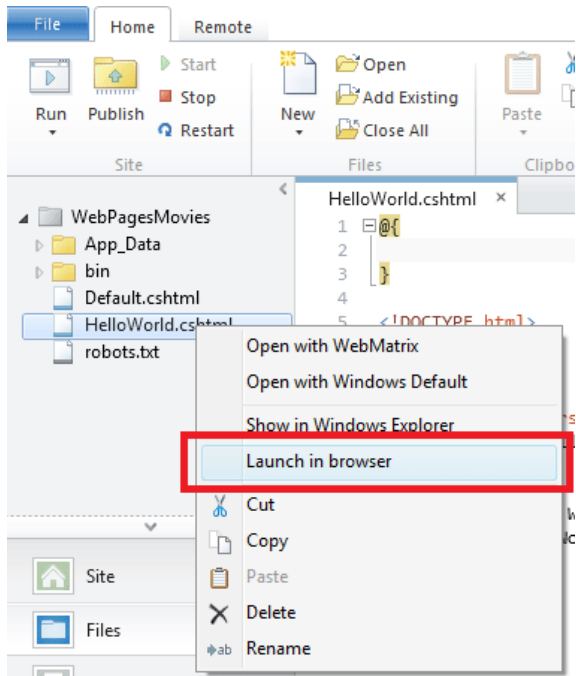
WebMatrix starts a built-in web server (IIS Express) that you can use to test pages on your computer. (Without IIS Express in WebMatrix, you'd have to publish your page to a web server somewhere before you could test it.) The page is displayed in your default browser.

Adding Some Server-Side Code

Close the browser and go back to the page in WebMatrix.

Add a line to the code block so that it looks like the following code:

```
@{
    var currentDateTime = DateTime.Now;
}
```

This is a little bit of Razor code. It's probably clear that it gets the current date and time and puts that value into a *variable* named `currentDateTime`. You'll read more about Razor syntax in the next tutorial.

In the body of the page, after the `<p>Hello World!</p>` element, add the following:

```
<p>Right now it's @currentDateTime</p>
```

This code gets the value that you put into the `currentDateTime` variable at the top and inserts it into the markup of the page. The `@` character marks the ASP.NET Web Pages code in the page.

Run the page again (WebMatrix saves the changes for you before it runs the page). This time you see the date and time in the page.

Wait a few moments and then refresh the page in the browser. The date and time display is updated.

In the browser, look at the page source. It looks like the following markup:

```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title>Hello World Page</title>
</head>
<body>
<h1>Hello World Page</h1>
<p>Hello World!</p>
<p>Right now it's 1/18/2012 2:49:50 PM</p>
</body>
</html>
```

Notice that the @{ } block at the top isn't there. Also notice that the date and time display shows an actual string of characters (1/18/2012 2:49:50 PM or whatever), not @currentDateTime like you had in the *.cshtml* page. What happened here is that when you ran the page, ASP.NET processed all the code (very little in this case) that was marked with @. The code produces output, and that output was inserted into the page.

This Is What ASP.NET Web Pages Are About

When you read that ASP.NET Web Pages produces dynamic web content, what you've seen here is the idea. The page you just created contains the same HTML markup that you've seen before. It can also contain code that can perform all sorts of tasks. In this example, it did the trivial task of getting the current date and time. As you saw, you can intersperse code with the HTML to produce output in the page. When someone requests a *.cshtml* page in the browser, ASP.NET processes the page while it's still in the hands of the web server. ASP.NET inserts the output of the code (if any) into the page as HTML. When the code processing is done, ASP.NET sends the resulting page to the browser. All the browser ever gets is HTML. Here's a diagram:

Browser (Client)

URL: `http://.../HelloWorld`

```
http://...
Hello, World

Hello, Joe
```

Request

Web server

ASP.NET

```
@{
    var name="Joe";
}
```

```
<html>
<body>
    <h1>Hello, World</h1>
    <p>Hello, @name</p>
</body>
</html>
```

```
<html>
<body>
    <h1>Hello, World</h1>
    <p>Hello, Joe</p>
</body>
</html>
```

Response
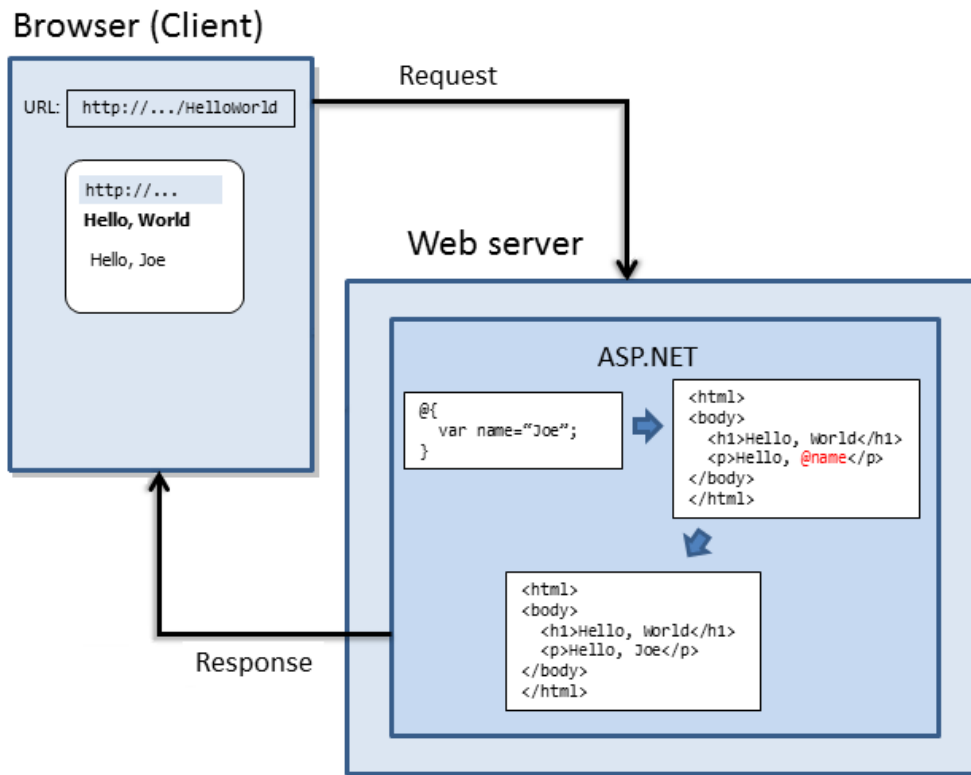
The idea is simple, but there are many interesting tasks that the code can perform, and there are many interesting ways in which you can dynamically add HTML content to the page. And ASP.NET *.cshtml* pages, like any HTML page, can also include code that runs in the browser itself (JavaScript and jQuery code). You'll explore all of these things in this tutorial set and in subsequent ones.

Coming Up Next

In the next tutorial in this series, you explore ASP.NET Web Pages programming a little more.

Additional Resources

- HTML Tutorial on the W3Schools site.

Tutorial 2: Programming Basics

This tutorial gives you an overview of how to program in ASP.NET Web Pages with Razor syntax.

What you'll learn:

- The basic "Razor" syntax that you use for programming in ASP.NET Web Pages.
- Some basic C#, which is the programming language you'll use.
- Some fundamental programming concepts for Web Pages.
- How to install packages (components that contain prebuilt code) to use with your site.
- How to use *helpers* to perform common programming tasks.

Features/technologies discussed:

- NuGet and the package manager.
- The `Twitter` helper.

This tutorial is primarily an exercise in introducing you to the programming syntax that you'll use for ASP.NET Web Pages. You'll learn about *Razor syntax* and code that's written in the C# programming language. You got a glimpse of this syntax in the previous tutorial; in this tutorial we'll explain the syntax more.

We promise that this tutorial involves the most programming that you'll see in a single tutorial, and that it's the only tutorial that is *only* about programming. In the remaining tutorials in this set, you'll actually create pages that do interesting things.

You'll also learn about *helpers*. A helper is a component — a packaged-up piece of code — that you can add to a page. The helper performs work for you that otherwise might be tedious or complex to do by hand.

Creating a Page to Play with Razor

In this section you'll play a bit with Razor so you can get a sense of the basic syntax.

Start WebMatrix if it's not already running. You'll use the website you created in the previous tutorial (Getting Started). To reopen it, click **Open Site** and choose **My Sites**:
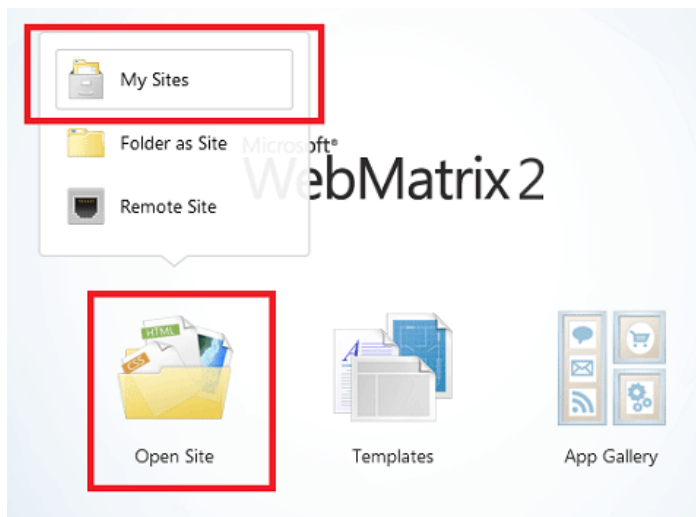
Choose the **WebPagesMovies** site, and then click **OK**.
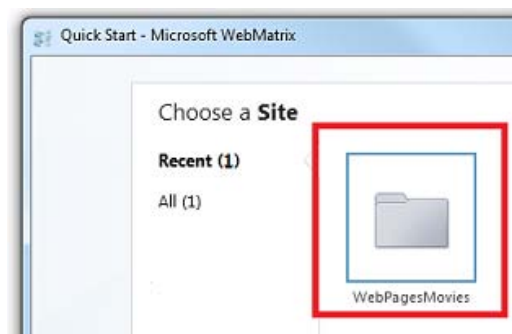
Select the **Files** workspace.

In the ribbon, click **New** to create a page. Select **CSHTML** and name the new page
TestRazor.cshtml.

Click **OK**.

Copy the following into the file, completely replacing what's there already.

> **Note**   When you copy code or markup from the examples into a page, the indentation and
> alignment might not be the same as in the tutorial. Indentation and alignment don't affect
> how the code runs, though.

```
@{
    // Working with numbers
    var a = 4;
    var b = 5;
    var theSum = a + b;

    // Working with characters (strings)
    var technology = "ASP.NET";
    var product ="Web Pages";

    // Working with objects
    var rightNow = DateTime.Now;
}

<!DOCTYPE html>
<html lang="en">
<head>
<title>Testing Razor Syntax</title>
<meta charset="utf-8" />
<style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    span.bright {color:red;}
</style>
</head>
<body>
<h1>Testing Razor Syntax</h1>
<form method="post">

<div>
<p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
<p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
<p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
</div>

<div>
<p>The technology is @technology, and the product is @product.</p>
<p>Together they are <span class="bright">@(technology + " " +
 product)</span></p>
</div>

<div>
<p>The current date and time is: @rightNow</p>
<p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
</div>

</form>
</body>
</html>
```

Examining the Example Page

Most of what you see is ordinary HTML. However, at the top there's this code block:

```
@{
    // Working with numbers.
    var a = 4;
    var b = 5;
    var theSum = a + b;

    // Working with characters (strings).
    var technology = "ASP.NET";
    var product ="Web Pages";

    // Working with objects.
    var rightNow = DateTime.Now;
}
```

Notice the following things about this code block:

- The `@` character tells ASP.NET that what follows is Razor code, not HTML. ASP.NET will treat everything after the `@` character as code until it runs into some HTML again. (In this case, that's the `<!DOCTYPE>` element.)

- The braces (`{` and `}`) enclose a block of Razor code if the code has more than one line. The braces tell ASP.NET where the code for that block starts and ends.

- The `//` characters mark a comment — that is, a part of the code that won't execute.

- Each statement has to end with a semicolon (`;`). (Not comments, though.)

- You can store values in *variables*, which you create (*declare*) with the keyword `var`.When you create a variable, you give it a name, which can include letters, numbers, and underscore (_). Variable names can't start with a number and can't use the name of a programming keyword (like `var`).

- You enclose character strings (like "ASP.NET" and "Web Pages") in quotation marks. (They must be double quotation marks.) Numbers are not in quotation marks.

- Whitespace outside of quotation marks doesn't matter. Line breaks mostly don't matter; the exception is that you can't split a string in quotation marks across lines. Indentation and alignment don't matter.

Something that's not obvious from this example is that all code is case sensitive. This means that the variable `TheSum` is a different variable than variables that might be named `theSum` or `thesum`. Similarly, `var` is a keyword, but `Var` is not.

Objects and properties and methods

Then there's the expression `DateTime.Now`. In simple terms, `DateTime` is an *object*. An object is a thing that you can program with—a page, a text box, a file, an image, a web request, an email message, a customer record, etc. Objects have one or more *properties* that describe their characteristics. A text box object has a `Text` property (among others), a request object has a

Url property (and others), an email message has a From property and a To property, and so on. Objects also have *methods* that are the "verbs" they can perform. You'll be working with objects a lot.

As you can see from the example, DateTime is an object that lets you program dates and times. It has a property named Now that returns the current date and time.

Using code to render markup in the page

In the body of the page, notice the following:

```
<div>
<p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
<p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
<p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
</div>

<div>
<p>The technology is @technology, and the product is @product.</p>
<p>Together they are <span class="bright">@(technology + " " +
product)</span></p>
</div>

<div>
<p>The current date and time is: @rightNow</p>
<p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
</div>
```

Again, the @ character tells ASP.NET that what follows is code, not HTML. In the markup you can add @ followed by a code expression, and ASP.NET will render the value of that expression right at that point. In the example, @a will render whatever the value is of the variable named a, @product renders whatever is in the variable named product, and so on.

You're not limited to variables, though. In a few instances here, the @ character precedes an expression:

- @(a*b) renders the product of whatever is in the variables a and b. (The * operator means multiplication.)
- @(technology + " " + product) renders the values in the variables technology and product after concatenating them and adding a space in between. The operator (+) for concatenating strings is the same as the operator for adding numbers. ASP.NET can usually tell whether you're working with numbers or with strings and does the right thing with the + operator.
- @Request.Url renders the Url property of the Request object. The Request object contains information about the current request from the browser, and of course the Url property contains the URL of that current request.

The example is also designed to show you that you can do work in different ways. You can do calculations in the code block at the top, put the results into a variable, and then render the

variable in markup. Or you can do calculations in an expression right in the markup. The approach you use depends on what you're doing and, to some extent, on your own preference.

Seeing the code in action

Right-click the name of the file and then choose **Launch in browser**. You see the page in the browser with all the values and expressions resolved in the page.

(Remember that the URL that you see in the browser might use a different port number than what you see in these screenshots. Instead of *localhost:56011*, you'll see *localhost* followed by a different number.)

Look at the source in the browser.

```
1
2
3   <!DOCTYPE html>
4   <html lang="en">
5     <head>
6       <title>Testing Razor Syntax</title>
7       <meta charset="utf-8" />
8       <style>
9       body {font-family:Verdana; margin-left:50px; margin-top:50px;}
10      div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
11      span.bright {color:red;}
12      </style>
13    </head>
14  <body>
15    <h1>Testing Razor Syntax</h1>
16    <form method="post">
17
18      <div>
19        <p>The value of <em>a</em> is 4.   The value of <em>b</em> is 5.
20        <p>The sum of <em>a</em> and <em>b</em> is <strong>9</strong></p>
21        <p>The product of <em>a</em> and <em>b</em> is <strong>20</strong></p>
22      </div>
23
24      <div>
25        <p>The technology is ASP.NET, and the product is Web Pages.</p>
26        <p>Together they are <span class="bright">ASP.NET Web Pages</span></p>
27      </div>
28
29      <div>
30        <p>The current date and time is: 4/29/2012 11:44:02 PM</p>
31        <p>The URL of the current page is <code>http://localhost:45661/TestRazor.cshtml</code></p>
32      </div>
33
34    </form>
35  </body>
36  </html>
```

As you expect from your experience in the previous tutorial, none of the Razor code is in the page.
All you see are the actual display values. When you run a page, you're actually making a request
to the web server that's built into WebMatrix. When the request is received, ASP.NET resolves all
the values and expressions and renders their values into the page. It then sends the page to the
browser.

Razor and C#

Up to now we've said that you're working with Razor syntax. That's true, but it's not the complete
story. The actual programming language you're using is called *C#*. C# was created by Microsoft
over a decade ago and has become one of the primary programming languages for creating
Windows apps. All the rules you've seen about how to name a variable and how to create
statements and so on are actually all rules of the C# language.

Razor refers more specifically to the small set of conventions for how you embed this code into a
page. For example, the convention of using @ to mark code in the page and using @{ } to embed a
code block is the Razor aspect of a page. Helpers are also considered to be part of Razor. Razor
syntax is used in more places than just in ASP.NET Web Pages. (For example, it's used in ASP.NET
MVC views as well.)

We mention this because if you look for information about programming ASP.NET Web Pages,
you'll find lots of references to Razor. However, a lot of those references don't apply to what
you're doing and might therefore be confusing. And in fact, many of your programming questions
are really going to be about either working with C# or working with ASP.NET. So if you look
specifically for information about Razor, you might not find the answers you need.

Adding Some Conditional Logic

One of the great features about using code in a page is that you can change what happens based on various conditions. In this part of the tutorial, you'll play around with some ways to change what's displayed in the page.

The example will be simple and somewhat contrived so that we can concentrate on the conditional logic. The page you'll create will do this:

- Show different text on the page depending on whether it's the first time the page is displayed or whether you've clicked a button to submit the page. That will be the first conditional test.
- Display the message only if a certain value is passed in the query string of the URL (http://...?show=true). That will be the second conditional test.

In WebMatrix, create a page and name it *TestRazorPart2.cshtml*. (In the ribbon, click **New**, choose **CSHTML**, name the file, and then click **OK**.)

Replace the contents of that page with the following:

```
@{
    var message = "This is the first time you've requested the page.";
}
<!DOCTYPE html>
<html lang="en">
<head>
<title>Testing Razor Syntax - Part 2</title>
<meta charset="utf-8" />
<style>
     body {font-family:Verdana; margin-left:50px; margin-top:50px;}
     div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
</style>
</head>
<body>
<h1>Testing Razor Syntax - Part 2</h1>
<form method="post">
<div>
<p>@message</p>
<p><input type="submit" value="Submit" /></p>
</div>
</form>
</body>
</html>
```

The code block at the top initializes a variable named `message` with some text. In the body of the page, the contents of the `message` variable are displayed inside a `<p>` element. The markup also contains an `<input>` element to create a **Submit** button.

Run the page to see how it works now. For now, it's basically a static page, even if you click the **Submit** button.

Go back to WebMatrix. Inside the code block, add the following code *after* the line that initializes message:

```
if(IsPost){
    message = "Now you've submitted the page.";
}
```

The if{ } block

What you just added was an `if` condition. In code, the `if` condition has a structure like this:

```
if(some condition){
One or more statements here that run if the condition is true;
}
```

The condition to test is in parentheses. It has to be a value or an expression that returns true or false. If the condition is true, ASP.NET runs the statement or statements that are inside the braces. (Those are the *then* part of the *if-then* logic.) If the condition is false, the block of code is skipped.

Here are a few examples of conditions you can test in an `if` statement:

```
if( currentValue > 12 ){ ... }

if( dueDate <= DateTime.Today ) { ... }

if( IsDone == true ) { ... }

if( IsPost ) { ... }

if( !IsPost ) { ... }

if(a != 0) { ... }

if( fileProcessingIsDone != true && displayMessage == false ) { ... }
```

You can test variables against values or against expressions by using a *logical operator* or *comparison operator*: equal to (`==`), greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`). The `!=` operator means not equal to — for example, `if(a != 0)` means *if a is not equal to 0*.

> **Note** Make sure you notice that the comparison operator for equals to (`==`) is not the same as `=`. The `=` operator is used only to assign values (`var a=2`). If you mix these operators up, you'll either get an error or you'll get some strange results.

To test whether something is true, the complete syntax is `if(IsDone == true)`. But you can also use the shortcut `if(IsDone)`. If there's no comparison operator, ASP.NET assumes that you're testing for true.

The `!` operator by itself means a logical NOT. For example, the condition `if(!IsPost)` means *if IsPost is not true*.

You can combine conditions by using a logical AND (`&&` operator) or logical OR (`||` operator). For example, the last of the `if` conditions in the preceding examples means *if FileProcessingIsDone is set to true AND displayMessage is set to false*.

The else block

One final thing about `if` blocks: an `if` block can be followed by an `else` block. An `else` block is useful is you have to execute different code when the condition is false. Here's a simple example:

```
var message = "";
if(errorOccurred == true)
{
    message = "Sorry, an error occurred.";
}
else
{
    message = "The process finished without errors!";
}
```

You'll see some examples in later tutorials in this series where using an `else` block is useful.

Testing whether the request is a submit (post)

There's more, but let's get back to the example, which has the condition `if(IsPost){ ... }`. `IsPost` is actually a property of the current page. The first time the page is requested, `IsPost` returns false. However, if you click a button or otherwise submit the page — that is, you post it — `IsPost` returns true. So `IsPost` lets you determine whether you're dealing with a form submission. (In terms of HTTP verbs, if the request is a GET operation, `IsPost` returns false. If the request is a POST operation, `IsPost` returns true.) In a later tutorial you'll work with input forms, where this test becomes particularly useful.

Run the page. Because this is the first time you're requested the page, you see "This is the first time you've requested the page". That string is the value that you initialized the `message` variable to. There's an `if(IsPost)` test, but that returns false at the moment, so the code inside the `if` block doesn't run.

Click the **Submit** button. The page is requested again. As before, the `message` variable is set to "This is the first time ...".But this time, the test `if(IsPost)` returns true, so the code inside the

`if`block runs. The code changes the value of the `message` variable to a different value, which is what's rendered in the markup.

Now add an `if` condition in the markup. Below the `<p>` element that contains the **Submit** button, add the following markup:

```
@if(IsPost){
<p>You submitted the page at @DateTime.Now</p>
}
```

You're adding code inside the markup, so you have to start with `@`. Then there's an `if` test similar to the one you added earlier up in the code block. Inside the braces, though, you're adding ordinary HTML — at least, it's ordinary until it gets to `@DateTime.Now`. This is another little bit of Razor code, so again you have to add `@` in front of it.

The point here is that you can add `if` conditions in both the code block at the top and in the markup. If you use an `if` condition in the body of the page, the lines inside the block can be markup or code. In that case, and as is true anytime you mix markup and code, you have to use `@` to make it clear to ASP.NET where the code is.

Run the page and click **Submit**.This time you not only see a different message when you submit ("Now you've submitted ..."), but you see a new message that lists the date and time.

Testing Razor Syntax - Part 2

Now you've submitted the page.

Submit

You submitted the page at 2/7/2012 7:51:24 PM
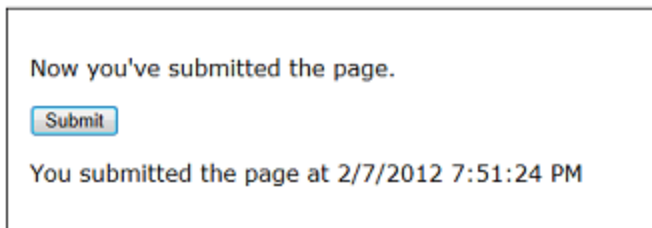
Testing the value of a query string

One more test. This time, you'll add an `if` block that tests a value named `show` that might be passed in the query string.(Like this: `http://localhost:43097/TestRazorPart2.cshtml?show=true`) You'll change the page so that the message you've been displaying ("This is the first time ...", etc.) is only displayed if the value of `show` is true.

At the bottom (but inside) the code block at the top of the page, add the following:

```
var showMessage = false;
if(Request.QueryString["show"].AsBool() == true){
    showMessage = true;
}
```

The complete code block now looks like the following example. (Remember that when you copy the code into your page, the indentation might look different. But that doesn't affect how the code runs.)

```
@{
    var message = "This is the first time you've requested the page.";

    if(IsPost){
        message = "Now you've submitted the page.";
    }

    var showMessage = false;
    if(Request.QueryString["show"].AsBool() == true){
      showMessage = true;
    }
}
```

The new code in the block initializes a variable named `showMessage` to false. It then does an `if` test to look for a value in the query string. When you first request the page, it has a URL like this one:

`http://localhost:43097/TestRazorPart2.cshtml`

The code determines whether the URL contains a variable named `show` in the query string, like this version of the URL:

`http://localhost:43097/TestRazorPart2.cshtml?show=true`

The test itself looks at the `QueryString` property of the `Request` object. If the query string contains an item named `show`, and if that item is set to true, the `if` block runs and sets the `showMessage` variable to true.

There's a trick here, as you can see. Like the name says, the query string is a string. However, you can only test for true and false if the value you're testing is a Boolean (true/false) value. Before you can test the value of the `show` variable in the query string, you have to convert it to a Boolean value. That's what the `AsBool` method does — it takes a string as input and converts it to a Boolean value. Clearly, if the string is "true", the `AsBool` method converts that value to true.If the value of the string is anything else, `AsBool` returns `false`.

Data Types and As( ) Methods

We've only said so far that when you create a variable, you use the keyword `var`. That's not the entire story, though. In order to manipulate values — to add numbers, or concatenate strings, or compare dates, or test for true/false — C# has to work with an appropriate internal representation of the value. C# can *usually* figure out what that representation should be (that is, what *type* the data is) based on what you're doing with the values. Now and then, though, it can't do that. If not, you have to help out by explicitly indicating how C# should represent the data. The `AsBool` method does that — it tells C# that a string value of `"true"` or `"false"` should be treated as a Boolean value. Similar methods exist to represent strings as other types as well, like `AsInt` (treat as an integer), `AsDateTime` (treat as a date/time), `AsFloat` (treat as a floating-point number), and so on. When you use these `As( )` methods, if C# can't represent the string value as requested, you'll see an error.

In the markup of the page, remove or comment out this element (here it's shown commented out):

```
<!-- <p>@message</p> -->
```

Right where you removed or commented out that text, add the following:

```
@if(showMessage){
<p>@message</p>
}
```

The `if` test says that if the `showMessage` variable is true, render a `<p>` element with the value of the `message` variable.

Summary of your conditional logic

In case you're not entirely sure of what you've just done, here's a summary.

- The `message` variable is initialized to a default string ("This is the first time ...").
- If the page request is the result of a submit (post), the value of `message` is changed to "Now you've submitted ..."
- The `showMessage` variable is initialized to false.
- If the query string contains `?show=true`, the `showMessage` variable is set to true.
- In the markup, if `showMessage` is true, a `<p>` element is rendered that shows the value of `message`. (If `showMessage` is false, nothing is rendered at that point in the markup.)
- In the markup, if the request is a post, a `<p>` element is rendered that displays the date and time.

Run the page. There's no message, because `showMessage` is false, so in the markup the `if(showMessage)` test returns false.

Click **Submit**. You see the date and time, but still no message.

In your browser, go to the URL box and add the following to the end of the URL: `?show=true` and then press Enter.
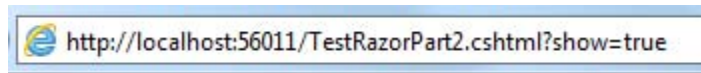
The page is displayed again. (Because you changed the URL, this is a new request, not a submit.) Click **Submit** again. The message is displayed again, as is the date and time.
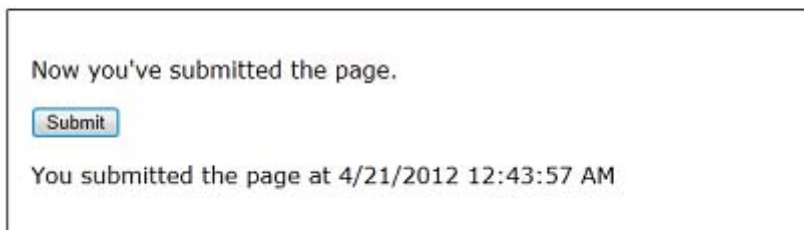
In the URL, change `?show=true` to `?show=false` and press Enter. Submit the page again. The page is back to how you started — no message.

As noted earlier, the logic of this example is a little contrived. However, `if` is going to come up in many of your pages, and it will take one or more of the forms you've seen here.

Installing a Helper (Displaying a Twitter Feed)

Some tasks that people often want to do on web pages require a lot of code or require extra knowledge. Examples: displaying a chart for data; putting a Facebook "Like" button on a page; sending email from your website; cropping or resizing images; using PayPal for your site. To make it easy to do these kinds of things, ASP.NET Web Pages lets you use *helpers*. Helpers are components that you install for a site and that let you perform typical tasks by using just a few lines of Razor code.

ASP.NET Web Pages has a few helpers built in. However, many helpers are available in packages (add-ins) that are provided using the NuGet package manager. NuGet lets you select a package to install and then it takes care of all the details of the installation.

In this part of the tutorial, you'll install a helper that lets you manage a Twitter feed. You'll learn two things. One is how to find and install a helper. You'll also learn how a helper makes it easy to do something you'd otherwise need to do by using a lot of code you'd have to write yourself.

In WebMatrix, click the **Gallery** button.
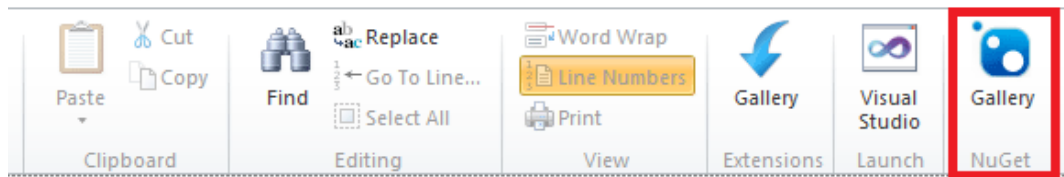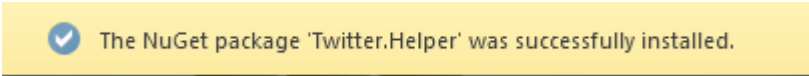
This launches the NuGet package manager and displays available packages. (Not all of the packages are helpers; some add functionality to WebMatrix itself, some are additional templates, and so on.)
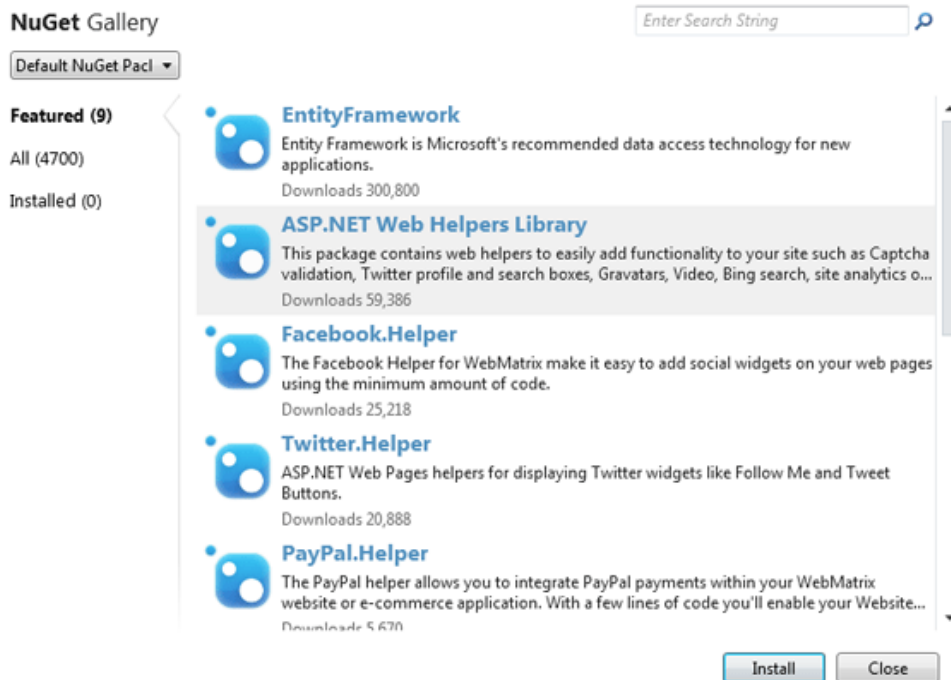
In the search box, enter "Twitter". NuGet shows the packages that have Twitter functionality. (The link underneath the package icon links to details about that package.)
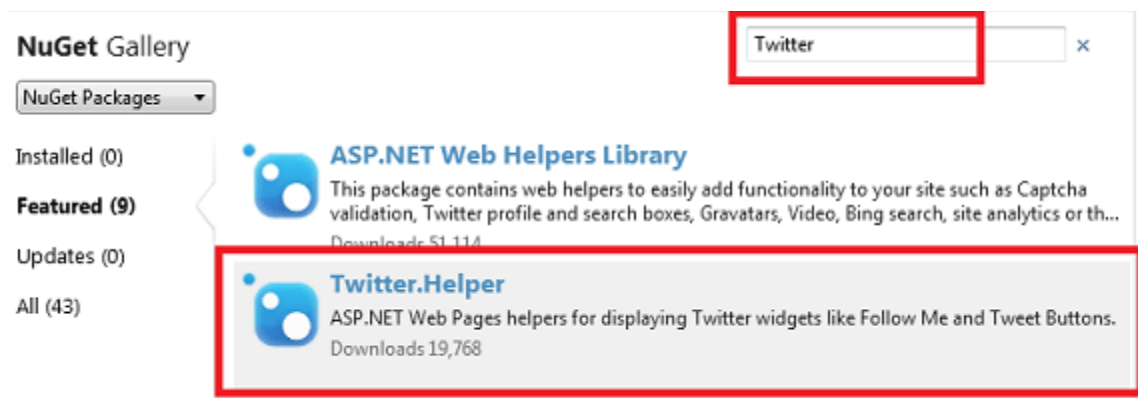
Select the **Twitter.Helper** package and then click **Install** to launch the installer. When it's done,you see a message in the notification area at the bottom of the screen.

> ✓ The NuGet package 'Twitter.Helper' was successfully installed.

That's it. NuGet downloads and installs everything, including any additional components that might be required (*dependencies*). Since this is the first time you've installed a helper, NuGet also creates folders in your website for the code that makes up the helper.

If for some reason you have to uninstall a helper, the process is very similar. Click the **Gallery** button, click the **Installed** tab, and pick the package you want to uninstall.

Using a Helper in a Page

Now you'll use the Twitter helper that you just installed. The process for adding a helper to a page is similar for most helpers.

In WebMatrix, create a page and name it *TwitterTest.cshml*. (You're creating a special page to test the helper, but you can use helpers in any page in your site.)
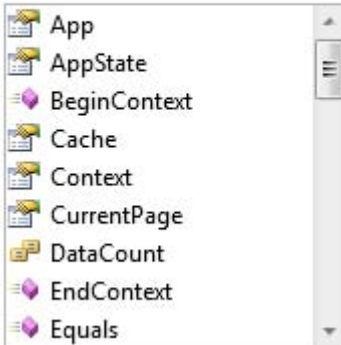
Inside the `<body>` element, add a `<div>` element. Inside the `<div>` element, type this:

```
@TwitterGoodies.
```

The `@` character is the same character you've been using to mark Razor code. `TwitterGoodies` is the helper object that you're working with.

As soon as you type the period (.), WebMatrix displays a list of *methods* (functions) that the `TwitterGoodies` helper makes available:

```
<body>
  <div>
      @TwitterGoodies.
  </div>

</body>
tml>
```

📑	App
📑	AppState
🔵	BeginContext
📑	Cache
📑	Context
📑	CurrentPage
🔲	DataCount
🔵	EndContext
🔵	Equals

This feature is known as *IntelliSense*. It helps you code by providing context-appropriate choices. IntelliSense works with HTML, CSS, ASP.NET code, JavaScript, and other languages that are supported in WebMatrix. It's another feature that makes it easier to develop web pages in WebMatrix.

Press S on the keyboard, and you see that IntelliSense finds the Search method:
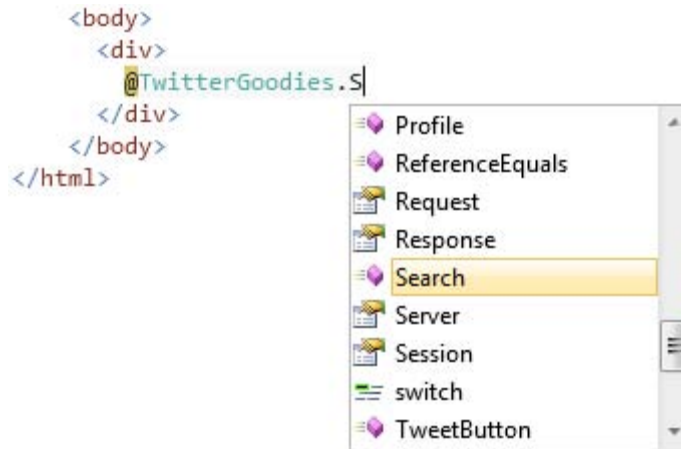
Press Tab. IntelliSense inserts the selected method (Search) for you. Type an open parenthesis (**(**), then the string "webmatrix" in quotation marks, then a closing parenthesis (**)**). When you're done, the line looks like this:

```
@TwitterGoodies.Search("webmatrix")
```

The Search method finds tweets that contain the string that you specify — in this case, it will look for tweets that mention "webmatrix". (Either in text or in hashtags.)

Run the page. You see a Twitter feed. (It might take a few moments for the feed to start populating.)

To get an idea of what the helper is doing for you, view the source of the page in the browser.Along with the HTML that you had in your page, you see a block of JavaScript code that looks roughly like the following block. (It might be all on one line or otherwise compressed.)

```
<script> new TWTR.Widget({ version: 2, type: 'search', search: 'webmatrix', interval:
6000, title: '', subject: '', width: 250, height: 300, theme: { shell: { background:
'#8ec1da', color: '#ffffff' }, tweets: { background: '#ffffff', color: '#444444', links:
'#1985b5' } }, features: { scrollbar: false, loop: true, live: true, hashtags: true,
timestamp: true, avatars: true, toptweets: true, behavior: 'all' } }).render().start();
</script>
```

This is code that the helper rendered into the page at the place where you had `@TwitterGoodies.Search`. (There's also some markup that's not shown here.) The helper took the information you provided and generated the code that talks directly to Twitter in order to get back the Twitter feed that you see. If you know the Twitter programming interface (API), you can create this code yourself. But because the helper can do it for you, you don't have to know the details of how to communicate with Twitter. And even if you are familiar with the Twitter API, it's a lot easier to include the `TwitterGoodies` helper on the page and let it do the work.

Return to the page. At the bottom, inside the `<body>` element, add the following code. Substitute your own Twitter account name if you have one.

```
<div>
    @TwitterGoodies.FollowButton("microsoft")
</div>
```

This code calls the `FollowButton` method of the `TwitterGoodies` helper. As you can guess, the method adds a **Follow Me on Twitter** button. You pass a Twitter name to this method to indicate who to follow.

Run the page and you see the **Follow Me** button:

Click it, and you go to the Twitter page for the user you specified.

As before, you can look at the source of the page in the browser to see what the `Twitter` helper generated for you. This time the code looks something like the following example:

```
<a href="http://www.twitter.com/microsoft"><img src="http://twitter-
badges.s3.amazonaws.com/follow_me-a.png" alt="Follow microsoft on Twitter"/></a>
```

Again, you could have written this code yourself, but the helper makes it much easier.

Server-Side (Razor) and Client-Side (JavaScript) Programming

How does Razor code in an ASP.NET Web Pages relate to JavaScript code that runs in the browser? If you've got experience with JavaScript, you might realize as you work with these tutorials that many of the tasks could also be done in JavaScript. That's true, especially with the simple examples you've seen so far.

A *.cshtml* page can contain both Razor code and JavaScript code. The traditional division of labor has been that server code handled tasks that it made sense to run on the server. This included accessing resources like a shared database and performing various types of business logic. In contrast, client code has typically been used to create a rich user experience. Pop-up calendars, sliders, animations, and many other UI effects are created by client code, and can all be done easily using JavaScript libraries (especially jQuery).

These days the distinction has blurred a little because client code libraries now let you communicate with the server in ways that formerly could only be done in server code. In general, though, it's still useful to think of server code (Razor and C#) as being for back-end work and client code (JavaScript) as useful for UI. In subsequent tutorial sets you'll learn how to integrate JavaScript into a *.cshtml* page for just this purpose, namely to create a lively user experience.

Coming Up Next

To keep this tutorial short, we had to focus on only a few basics. Naturally, there's a *lot* more to Razor and C#. You'll learn more as you go through these tutorials. If you're interested in learning

more about the programming aspects of Razor and C# right now, you can read a more thorough introduction here:Introduction to ASP.NET Web Programming Using the Razor Syntax.

The next tutorial introduces you to working with a database. In that tutorial, you'll begin creating the sample application that lets you list your favorite movies.

Additional Resources

- Complete Listing for Test Razor Page
- Complete Listing for TestRazorPart2 Page
- Complete Listing for TwitterTest Page
- Introduction to ASP.NET Web Programming Using the Razor Syntax

Tutorial 3: Displaying Data

This tutorial shows you how to create a database in WebMatrix and how to display database data in a page when you use ASP.NET Web Pages (Razor).

What you'll learn:

- How to use WebMatrix tools to create a database and database tables.
- How to use WebMatrix tools to add data to a database.
- How to display data from the database on a page.
- How to run SQL commands in ASP.NET Web Pages.
- How to customize the `WebGrid` helper to change the data display and to add paging and sorting.

Features/technologies discussed:

- WebMatrix database tools.
- `WebGrid` helper.

What You'll Build

In the previous tutorial, you were introduced to ASP.NET Web Pages (*.cshtml* files), to the basics of Razor syntax, and to helpers. In this tutorial, you'll begin creating the actual web application that you'll use for the rest of the series. The app is a simple movie application that lets you view, add, change, and delete information about movies.

When you're done with this tutorial, you'll be able to view a movie listing that looks like this page:

Movies

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984
Fantasia	Children	1939
Clueless	Comedy	1995
Silverado	Western	1985
The Three Musketeers	Adventure	1973
Ronin	Action	1998

But to begin, you have to create a database.

A Very Brief Introduction to Databases

This tutorial will provide only the briefest introduction to databases. If you have database experience, you can skip this short section.

A database contains one or more tables that contain information — for example, tables for customers, orders, and vendors,or for students, teachers, classes, and grades. Structurally, a database table is like a spreadsheet. Imagine a typical address book. For each entry in the address book (that is, for each person) you have several pieces of information such as first name, last name, address, email address, and phone number.

ID	FirstName	LastName	Address	Email	Phone
1	Lisa	Miller	4567 Main St	lisa@adatum.com	555-0199
2	Patrick	Hines	123 Elm St	patrickh@tailspintoys.com	555-0111
3	Julia	Ilyina	89 Oak Ave	julia@fourthcoffee.com	555-0166
4	Mark	Alexieff	678 8th St	marka@proseware.com	555-0129
5	Jim	Daly	345 Broad Blvd	jdaly@contoso.com	555-0188

(Rows are sometimes referred to as *records*, and columns are sometimes referred to as *fields*.)

For most database tables, the table has to have a column that contains a unique value, like a customer number, account number, and so on. This value is known as the table's *primary key*, and you use it to identify each row in the table. In the example, the ID column is the primary key for the address book shown in the previous example.

Much of the work you do in web applications consists of reading information out of the database and displaying it on a page. You'll also often gather information from users and add it to a database, or you'll modify records that are already in the database. (We'll cover all of these operations in the course of this tutorial set.)

Database work can be enormously complex and can require specialized knowledge. For this tutorial set, though, you have to understand only basic concepts, which will all be explained as you go.

Creating a Database

WebMatrix includes tools that make it easy to create (or *define*) a database and to create tables in the database. For this tutorial set, you'll create a database that has only one table in it — Movies.

Open WebMatrix if you haven't already done so, and open the WebPagesMovies site that you created in the previous tutorial.

In the left pane, click the **Database** workspace.
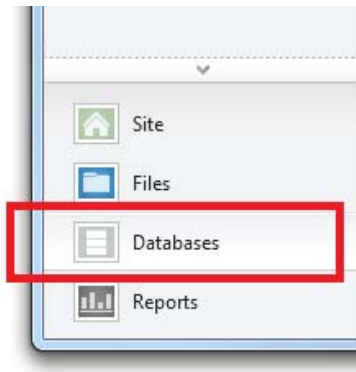
The ribbon changes to show database-related tasks. In the ribbon, click **New Database**.
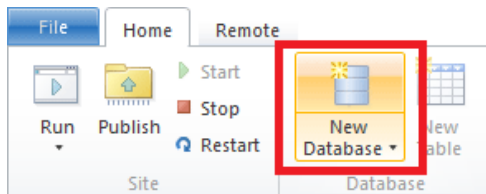
WebMatrix creates a database (an *.sdf* file) that has the same name as your site —
WebPagesMovies.sdf. (You won't do this here, but you can rename the file to anything you like, as
long as it has an *.sdf* extension.)

Creating a Table

In the ribbon, click **New Table**. WebMatrix opens the table designer in a new tab. (If the New
Table option isn't available, make sure that the new database is selected in the tree view on the
left.)
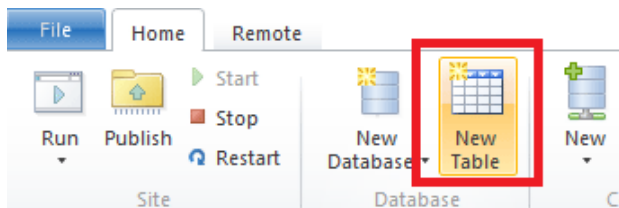
In the text box at the top (where the watermark says "Enter table name"), enter "Movies".
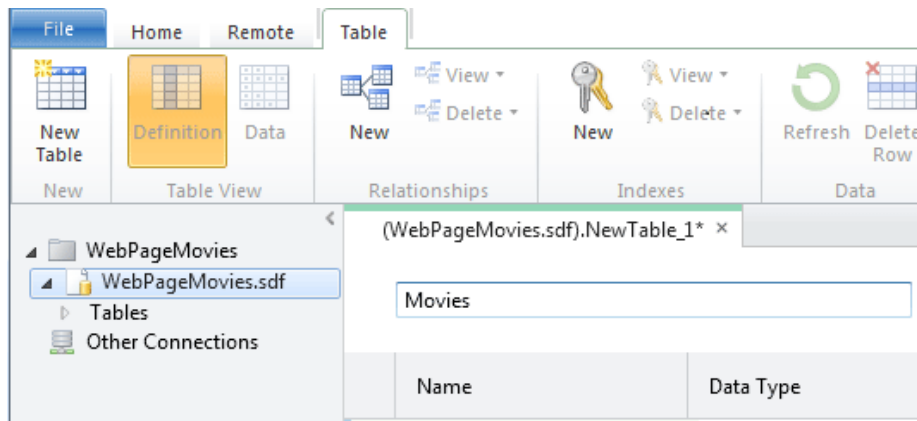
The pane underneath the table name is where you define individual columns. For the *Movies* table in this tutorial, you'll create only a few columns: *ID*, *Title*, *Genre*, and *Year*.

In the **Name** box, enter "ID". Entering a value here activates all the controls for the new column.

Tab to the **Data Type** list and choose **int**. This value specifies that the ID column will contain integer (number) data.

> **Note** We won't call it out any more here (much), but you can use standard Windows keyboard gestures to navigate in this grid. For example, you can tab between fields, you can just start typing in order to select an item in a list, and so on.

Tab past the **Default Value** box (that is, leave it blank). Tab to the **Is Primary Key** check box and select it. This option tells the database that the *ID* column will contain the data that identifies individual rows. (That is, each row will have a unique value in the ID column that you can use to find that row.)

Choose the **Is Identity** option. This option tells the database that it should automatically generate the next sequential number for each new row. (The **Is Identity** option works only if you've also selected the **Is Primary Key** option.)

Click in the next grid row, or press Tab twice to finish the current row. Either gesture saves the current row and starts the next one. Notice that the **Default Value** column now says **Null**. (Null is the default value for the default value, so to speak.)

When you've finished defining the new **ID** column, the designer will look like this illustration:
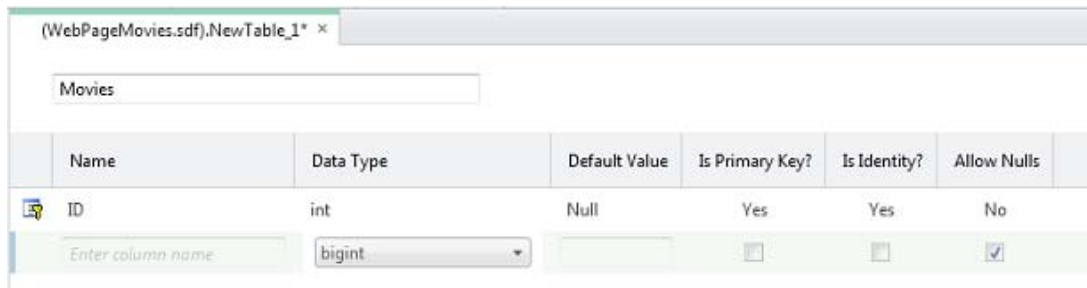
To create the next column, click in the box in the **Name** column. Enter "Title" for the column and then select **nvarchar** for the **Data Type** value. The "var" part of **nvarchar** tells the database that the data for this column will be a string whose size might vary from record to record. (The "n" prefix represents "national," which indicates that the field can hold character data for any alphabet or writing system — that is, the field holds Unicode data.)

When you choose **nvarchar**, another box appears where you can enter the maximum length for the field. Enter 50, on the assumption that no movie title that you'll work with in this tutorial will be longer than 50 characters.

Skip **Default Value** and clear the **Allow Nulls** option. You don't want the database to allow any movies to be entered into the database that don't have a title.

When you're done and move to the next row, the designer looks like this illustration:
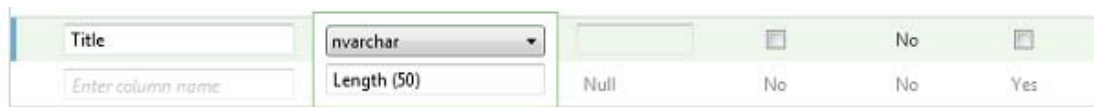
Repeat these steps to create a column named "Genre", except for the length, set it to just 30.

Create another column named "Year." For the data type, choose **nchar** (not **nvarchar**) and set the length to 4. For the year, you're going to use a 4-digit number like "1995" or "2010", so you don't require a variable-sized column.

Here's what the finished design looks like:
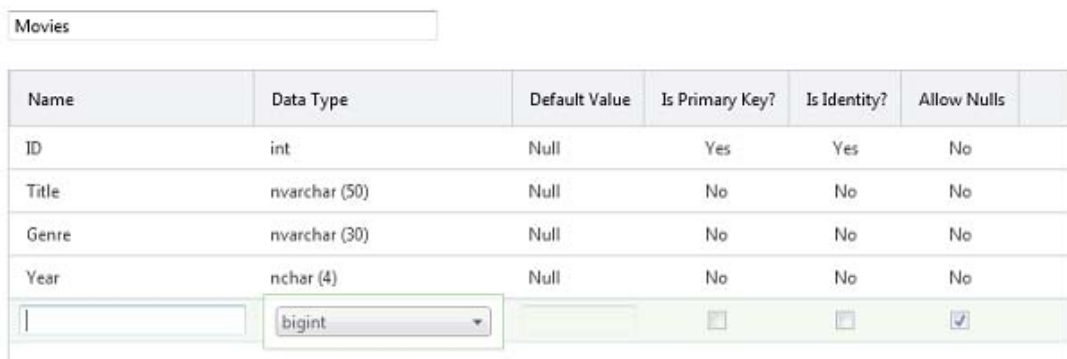
Press Ctrl+S or click the **Save** button in the Quick Access toolbar. Close the database designer by closing the tab.
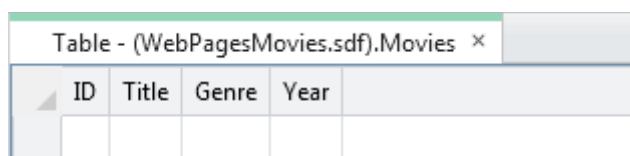
Adding Some Example Data

Later in this tutorial series you'll create a page where you can enter new movies in a form. For now, however, you can add some example data that you can then display on a page.

In the **Database** workspace in WebMatrix, notice that there's a tree that shows you the *.sdf* file you created earlier. Open the node for your new *.sdf* file, and then open the **Tables** node.
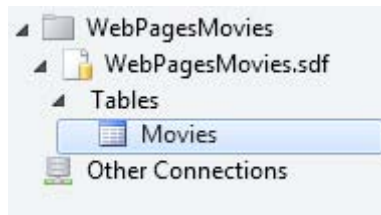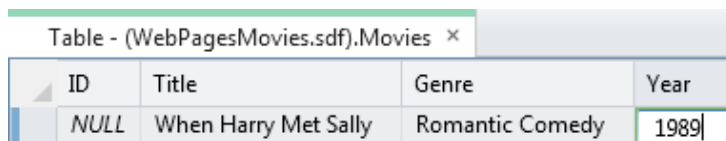
Right-click the **Movies** node and then choose **Data**. WebMatrix opens a grid where you can enter data for the *Movies* table:
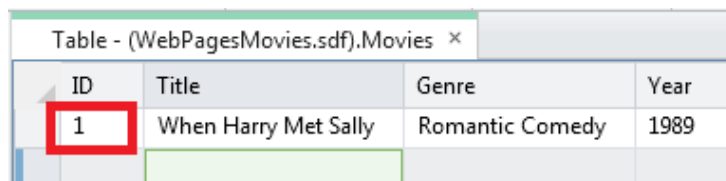
Click the **Title** column and enter "When Harry Met Sally". Move to the **Genre** column (you can use the Tab key) and enter "Romantic Comedy". Move to the **Year** column and enter "1989":

Press Enter, and WebMatrix saves the new movie. Notice that the **ID** column has been filled in.

Enter another movie (for example, "Gone with the Wind", "Drama", "1939"). The ID column is filled in again:

Enter a third movie (for example, "Ghostbusters", "Comedy"). As an experiment, leave the **Year** column blank and then press Enter. Because you unselected the **Allow Nulls** option, the database shows an error:
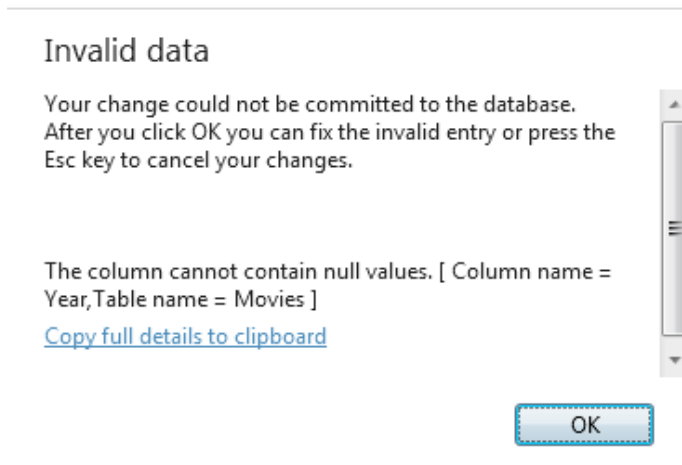
Click **OK** to go back and fix the entry (the year for "Ghostbusters" is 1984), and then press Enter.

Fill in several movies until you have 8 or so. (Entering 8 makes it easier to work with paging later. But if that's too many, enter just a few for now.) The actual data doesn't matter.

If you entered all the movies without any errors, the ID values are sequential. If you tried to save an incomplete movie record, the ID numbers might not be sequential. If so, that's okay. The numbers don't have any inherent meaning, and the only thing that's important is that they're unique in the *Movies* table.

Close the tab that contains the database designer.

Now you can turn to displaying this data on a web page.

Adding the Web Data Package

You access data in ASP.NET Web Pages sites using helpers that are available in the WebData package. Because you used the **Empty Site** template in WebMatrix to create your Movies site, the WebData package isn't automatically included. Therefore, you have to add the package to your site.

In the left pane, click the **Files** workspace.
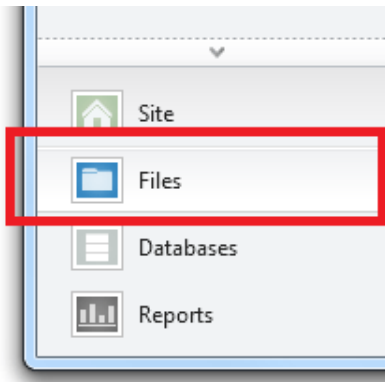
In the ribbon, click the **Gallery** button.
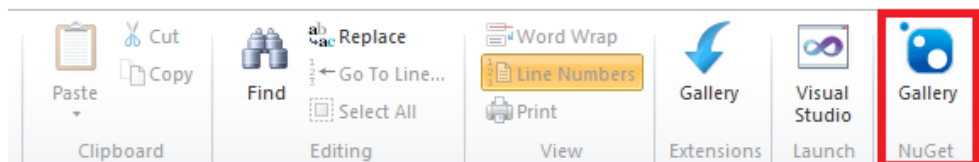
The NuGet Gallery is displayed.

In the search box, enter "Microsoft.AspNet.WebPages.WebData" to narrow down the list of packages that are displayed.
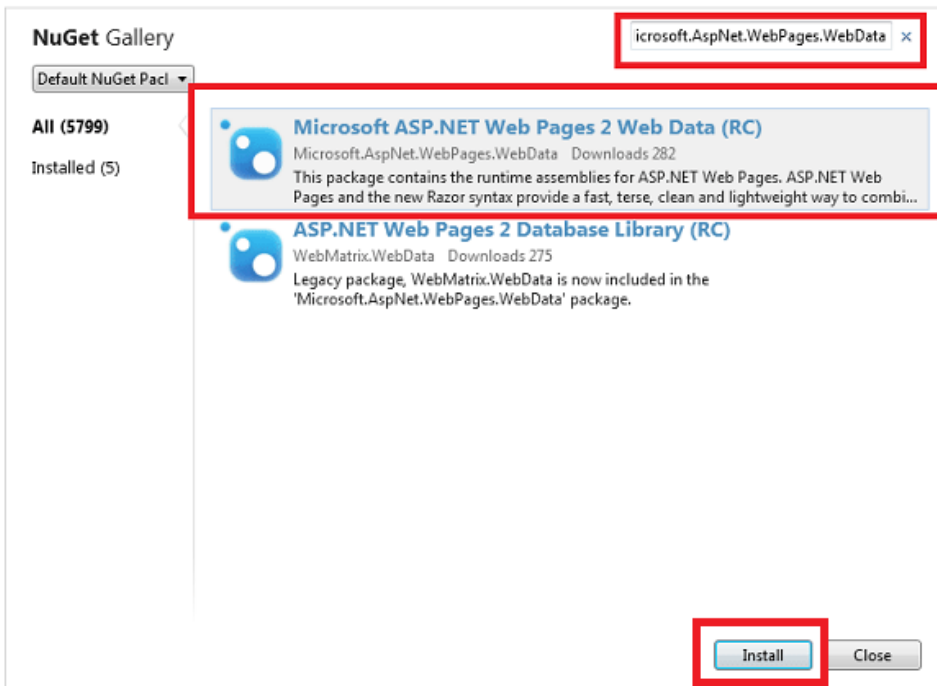
Select the **Microsoft ASP.NET Web Pages 2 Web Data** package and then click **Install**.

The Gallery page displays details about the package.
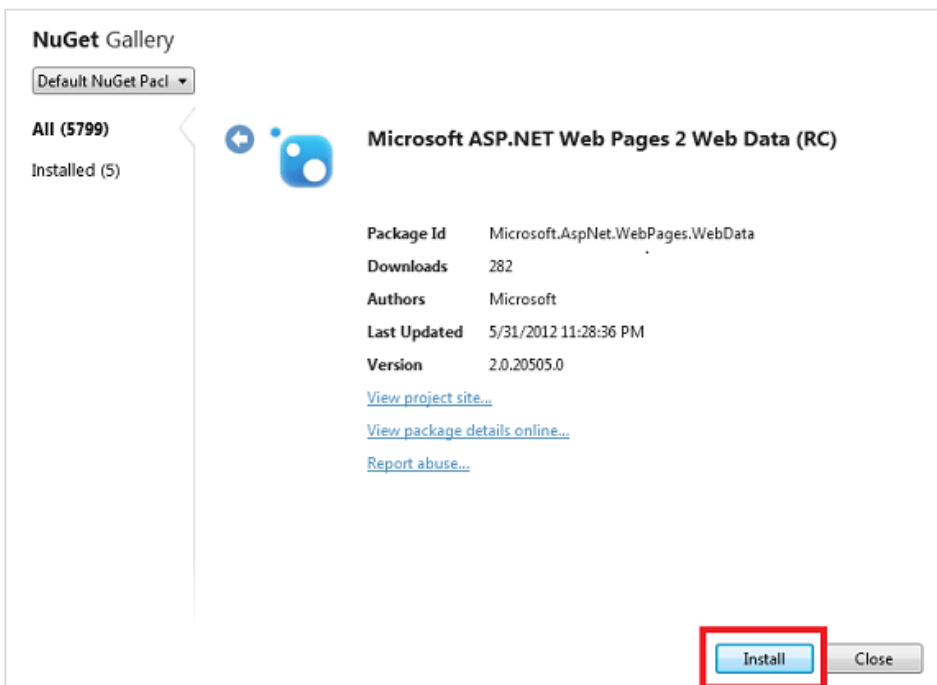
Click **Install**. After you accept the license, WebMatrix installs the Web Data package.

Now you're ready to display data on a page.

Displaying Data in a Page by Using the WebGrid Helper

To display data in a page, you're going to use the `WebGrid` helper. This helper produces a display in a grid or table (rows and columns). As you'll see, you'll be able refine the grid with formatting and other features.

To run the grid, you'll have to write a few lines of code. These few lines will serve as a kind of pattern for almost all of the data access that you do in this tutorial.

> **Note** You actually have many options for displaying data on a page; the `WebGrid` helper is just one. We chose it for this tutorial because it's the easiest way to display data and because it's reasonably flexible. In the next tutorial set, you'll see how to use a more "manual" way to work with data in the page, which gives you more direct control over how to display the data.

In the left pane in WebMatrix, click the **Files** workspace. In the tree view, select the root of the website.

The new database you created is in the *App_Data* folder. If the folder didn't already exist, WebMatrix created it for your new database. (The folder might have existed if you'd previously installed helpers.)

In the ribbon, click **New**. In the **Choose a File Type** box, choose **CSHTML**.

In the **Name** box, name the new page "Movies.cshtml":
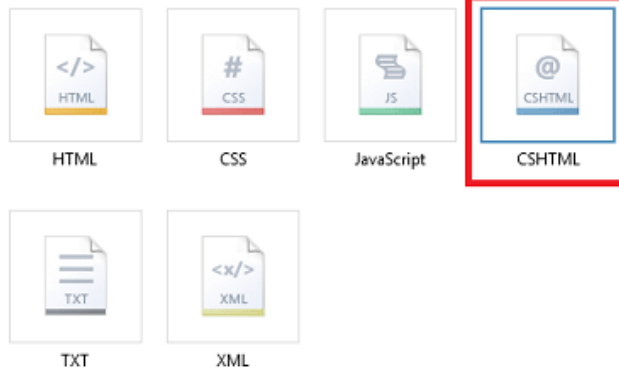
Click the **OK** button. WebMatrix opens a new file with some skeleton elements in it. First you'll write some code to go get the data from the database. Then you'll add markup to the page to actually display the data.

Writing the Data Query Code

At the top of the page, between the `@{` and `}` characters, enter the following code. (Make sure that you enter this code between the opening and closing braces.)

```
var db = Database.Open("WebPagesMovies");
var selectedData = db.Query("SELECT * FROM Movies");
var grid = new WebGrid(source: selectedData);
```

The first line opens the database that you created earlier, which is always the first step before doing something with the database. You tell the `Database.Open` method name of the database to open. Notice that you don't include *.sdf* in the name. The `Open` method assumes that it's looking for an *.sdf* file (that is, *WebPagesMovies.sdf*) and that the *.sdf* file is in the *App_Data* folder. (Earlier we noted that the *App_Data* folder is reserved; this scenario is one of the places where ASP.NET makes assumptions about that name.)

When the database is opened, a reference to it is put into the variable named `db`. (Which could be named anything.) The `db` variable is how you'll end up interacting with the database.

The second line actually fetches the database data by using the `Query` method. Notice how this code works: the `db` variable has a reference to the opened database, and you invoke the `Query` method by using the `db` variable (`db.Query`).

Thequery itself is a SQL `Select` statement. (For a little background about SQL, see the explanation later.) In the statement, `Movies` identifies the table to query. The `*` character specifies that the query should return all the columns from the table. (You could also list columns individually, separated by commas.)

The results of the query, if any, are returned and made available in the `selectedData` variable. Again, the variable could be named anything.

Finally, the third line tells ASP.NET that you want to use an instance of the `WebGrid` helper. You create (*instantiate*) the helper object by using the `new` keyword and pass it the query results via the `selectedData` variable. The new `WebGrid` object, along with the results of the database query, are made available in the `grid` variable. You'll need that result in a moment to actually display the data in the page.

At this stage, the database has been opened, you've gotten the data you want, and you've prepared the `WebGrid` helper with that data. Next is to create the markup in the page.

Structured Query Language (SQL)

SQL is a language that's used in most relational databases for managing data in a database. It includes commands that let you retrieve data and update it, and that let you create, modify, and manage data in database tables. SQL is different than a programming language (like C#). With SQL, you tell the database what you want, and it's the database's job to figure out how to get the data or perform the task. Here are examples of some SQL commands and what they do:

```
Select * From Movies
```

```
SELECT ID, Name, Price FROM Product WHERE Price > 10.00 ORDER BY Name
```

The first `Select` statement gets all the columns (specified by `*`) from the *Movies* table.

The second `Select` statement fetches the ID, Name, and Price columns from records in the *Product* table whose Price column value is more than 10. The command returns the results in alphabetical order based on the values of the Name column. If no records match the price criteria, the command returns an empty set.

```
INSERT INTO Product (Name, Description, Price) VALUES ('Croissant', 'A flaky delight', 1.99)
```

This command inserts a new record into the *Product* table, setting the Name column to "Croissant", the Description column to "A flaky delight", and the price to 1.99.

Notice that when you're specifying a non-numeric value, the value is enclosed in single quotation marks (not double quotation marks, as in C#). You use these quotation marks around text or date values, but not around numbers.

```
DELETE FROM Product WHERE ExpirationDate < '01/01/2008'
```

This command deletes records in the *Product* table whose expiration date column is earlier than January 1, 2008. (The command assumes that the *Product* table has such a column, of course.) The date is entered here in MM/DD/YYYY format, but it should be entered in the format that's used for your locale.

The `Insert` and `Delete` commands don't return result sets. Instead, they return a number that tells you how many records were affected by the command.

For some of these operations (like inserting and deleting records), the process that's requesting the operation has to have appropriate permissions in the database. That's why for production databases you often have to supply a user name and password when you connect to the database.

There are dozens of SQL commands, but they all follow a pattern like the commands you see here. You can use SQL commands to create database tables, count the number of records in a table, calculate prices, and perform many more operations.

Adding Markup to Display the Data

Inside the `<head>` element, set contents of the `<title>` element to "Movies":

```
<head>
<meta charset="utf-8" />
<title>Movies</title>
</head>
```

Inside the `<body>` element of the page, add the following:

```
<h1>Movies</h1>
<div>
    @grid.GetHtml()
</div>
```

That's it. The `grid` variable is the value you created when you created the `WebGrid` object in code earlier.

In the WebMatrix tree view, right-click the page and select **Launch in browser**.You'll see something like this page:

Movies

ID	Title	Genre	Year
1	When Harry Met Sally	Romantic Comedy	1989
2	Gone With The Wind	Drama	1939
4	Ghostbusters	Comedy	1984
5	Fantasia	Children	1939
8	Clueless	Comedy	1995
9	Silverado	Western	1985
10	The Three Musketeers	Adventure	1973
11	Ronin	Action	1998

(Remember that the URL that you see in the browser might use a different port number than what you see in these screenshots — instead of *locahost:56011*, you'll see *localhost* followed by a different number.)

Click a column heading link to sort by that column. Being able to sort by clicking a heading is a feature that's built into the **WebGrid** helper.

The `GetHtml` method, as its name suggests, generates markup that displays the data. By default, the `GetHtml` method generates an HTML `<table>` element. (If you want, you can verify the rendering by looking at the source of the page in the browser.)

Modifying the Look of the Grid

Using the `WebGrid` helper like you just did is easy, but the resulting display is plain. The `WebGrid` helper has all sorts of options that let you control how the data is displayed. There are far too many to explore in this tutorial, but this section will give you an idea of some of those options. A few additional options will be covered in later tutorials in this series.

Specifying Individual Columns to Display

To start, you can specify that you want to display only certain columns. By default, as you've seen, the grid shows all four of the columns from the *Movies* table.

In the *Movies.cshtml* file, replace the `@grid.GetHtml()` markup that you just added with the following:

```
@grid.GetHtml(
    columns: grid.Columns(
        grid.Column("Title"),
        grid.Column("Genre"),
        grid.Column("Year")
    )
)
```

To tell the helper which columns to display, you include a `columns` parameter for the `GetHtml` method and pass in a collection of columns. In the collection, you specify each column to include. You specify an individual column to display by including a `grid.Column` object, and pass in the name of the data column you want. (These columns must be included in the SQL query results — the `WebGrid` helper cannot display columns that were not returned by the query.)

Launch the *Movies.cshtml* page in the browser again, and this time you get a display like the following one (notice that no ID column is displayed):

Movies

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984
Fantasia	Children	1939
Clueless	Comedy	1995
Silverado	Western	1985
The Three Musketeers	Adventure	1973
Ronin	Action	1998

Changing the Look of the Grid

There are quite a few more options for displaying columns, some of which will be explored in later tutorials in this set. For now, this section will introduce you to ways in which you can style the grid as a whole.

Inside the `<head>` section of the page, just before the closing `</head>` tag, add the following `<style>` element:

```
<style type="text/css">
  .grid { margin: 4px; border-collapse: collapse; width: 600px; }
  .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
  .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
  .alt { background-color: #E8E8E8; color: #000; }
</style>
```

This CSS markup defines classes named `grid`, `head`, and so on. You could also put these style definitions in a separate *.css* file and link that to the page. (In fact, you'll do that later in this tutorial set.) But to make things easy for this tutorial, they're inside the same page that displays the data.

Now you can get the `WebGrid` helper to use these style classes. The helper has a number of properties (for example, `tableStyle`) for just this purpose — you assign a CSS style class name to them, and that class name is rendered as part of the markup that's rendered by the helper.

Change the `grid.GetHtml` markup so that it now looks like this code:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    columns: grid.Columns(
        grid.Column("Title"),
        grid.Column("Genre"),
        grid.Column("Year")
    )
)
```

What's new here is that you've added `tableStyle`, `headerStyle`, and `alternatingRowStyle` parameters to the `GetHtml` method. These parameters have been set to the names of the CSS styles that you added a moment ago.

Run the page, and this time you see a grid that looks much less plain than before:

Movies

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984
Fantasia	Children	1939
Clueless	Comedy	1995
Silverado	Western	1985
The Three Musketeers	Adventure	1973
Ronin	Action	1998

To see what the `GetHtml` method generated, you can look at the source of the page in the browser. We won't go into detail here, but the important point is that by specifying parameters like `tableStyle`, you caused the grid to generate HTML tags like the following:

```
<table class="grid">
```

The `<table>` tag has had a `class` attribute added to it that references one of the CSS rules that you added earlier. This code shows you the basic pattern — different parameters for the `GetHtml` method let you reference CSS classes that the method then generates along with the markup. What you do with the CSS classes is up to you.

Adding Paging

As the last task for this tutorial, you'll add paging to the grid. Right now it's no problem to display all your movies at once. But if you added hundreds of movies, the page display would get long.

In the page code, change the line that creates the `WebGrid` object to the following code:

```
var grid = new WebGrid(source: selectedData, rowsPerPage: 3);
```

The only difference from before is that you've added a `rowsPerPage` parameter that's set to 3.

Run the page. The grid displays 3 rows at a time, plus navigation links that let you page through the movies in your database:

Movies

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984
1 2 3 >		

Coming Up Next

In the next tutorial, you'll learn how to use Razor and C# code to get user input in a form. You'll add a search box to the Movies page so that you can find movies by title or genre.

Additional Resources

- Complete Listing for Movies Page
- Introduction to ASP.NET Web Programming Using the Razor Syntax

Tutorial 4: HTML Form Basics

This tutorial shows you the basics of how to create an input form and how to handle the user's input when you use ASP.NET Web Pages (Razor). And now that you've got a database, you'll use your form skills to let users find specific movies in the database.

What you'll learn:

- How to create a form by using standard HTML elements.
- How to read the user's input in a form.
- How to create a SQL query that selectively gets data by using a search term that the user supplies.
- How to have fields in the page "remember" what the user entered.

Features/technologies discussed:

- The `Request` object.
- The SQL `Where` clause.

What You'll Build

In the previous tutorial, you created a database, added data to it, and then used the `WebGrid` helper to display the data. In this tutorial,you'll add a search box that lets you find movies of a specific genre or whose title contains whatever word you enter. (For example, you'll be able to find all movies whose genre is "Action" or whose title contains "Harry" or "Adventure.")

When you're done with this tutorial, you'll have a page like this one:

Movies

Title	Genre	Year
Ronin	Action	1998
The Three Musketeers	Adventure	1973
Fantasia	Children	1939

1 2 3 ≥

The listing part of the page is the same as in the last tutorial — a grid. The difference will be that the grid will show only the movies that you searched for.

About HTML Forms

(If you've got experience with creating HTML forms and with the difference between GET and POST, you can skip this section.)

A form has user input elements — text boxes, buttons, radio buttons, check boxes, drop-down lists, and so on. Users fill in these controls or make selections and then submit the form by clicking a button.

The basic HTML syntax of a form is illustrated by this example:

```
<form method="post">
<input type="text" name="name" value="" />
<br/>
<input type="submit" name="submit" value="Submit" />
</form>
```

When this markup runs in a page, it creates a simple form that looks like this illustration:

The `<form>` element encloses HTML elements to be submitted. (An easy mistake to make is to add elements to the page but then forget to put them inside a `<form>` element. In that case, nothing is submitted.) The `method` attribute tells the browser how to submit the user input. You set this to `post` if you're performing an update on the server or to `get` if you're just fetching data from the server.

GET, POST, and HTTP Verb Safety

HTTP, the protocol that browsers and servers use to exchange information, is remarkably simple in its basic operations. Browsers use only a few verbs to make requests to servers. When you write code for the web, it's helpful to understand these verbs and how the browser and server use them. Far and away the most commonly used verbs are these:

GET. The browser uses this verb to fetch something from the server. For example, when you type a URL into your browser, the browser performs a GET operation to request the page you want. If the page includes graphics, the browser performs additional GET operations to get the images. If the GET operation has to pass information to the server, the information is passed as part of the URL in the query string.

POST. The browser sends a POST request in order to submit data to be added or changed on the server. For example, the POST verb is used to create records in a database or change existing ones.

Most of the time, when you fill in a form and click the submit button, the browser performs a POST operation. In a POST operation, the data being passed to the server is in the body of the page.

An important distinction between these verbs is that a GET operation is not supposed to change anything on the server — or to put it in a slightly more abstract way, a GET operation does not result in a change in state on the server. You can perform a GET operation on the same resources as many times as you like, and those resources don't change. (A GET operation is often said to be "safe," or to use a technical term, is *idempotent*.) In contrast, of course, a POST request changes something on the server each time you perform the operation.

Two examples will help illustrate this distinction. When you perform a search using an engine like Bing or Google, you fill in a form that consists of one text box, and then you click the search button. The browser performs a GET operation, with the value you entered into the box passed as part of the URL. Using a GET operation for this type of form is fine, because a search operation doesn't change any resources on the server, it just fetches information.

Now consider the process of ordering something online. You fill in the order details and then click the submit button. This operation will be a POST request, because the operation will result in changes on the server, such as a new order record, a change in your account information, and perhaps many other changes. Unlike the GET operation, you cannot repeat your POST request — if you did, each time you resubmitted the request, you'd generate a new order on the server. (In cases like this, websites will often warn you not to click a submit button more than once, or will disable the submit button so that you don't resubmit the form accidentally.)

In the course of this tutorial, you'll use both a GET operation and a POST operation to work with HTML forms. We'll explain in each case why the verb you use is the appropriate one.

(To learn more about HTTP verbs, see the [Method Definitions](#) article on the W3C site.)

Most user input elements are HTML <input> elements. They look like `<input type="type" name="name">`, where *type* indicates the kind of user input control you want. These elements are the common ones:

- Text box: `<input type="text">`
- Check box: `<input type="check">`
- Radio button: `<input type="radio">`
- Button: `<input type="button">`
- Submit button: `<input type="submit">`

You can also use the `<textarea>` element to create a multiline text box and the `<select>` element to create a drop-down list or scrollable list. (For more about HTML form elements, see [HTML Forms and Input](#) on the W3Schools site.)

The name attribute is very important, because the name is how you'll get the value of the element later, as you'll see shortly.

The interesting part is what you, the page developer, do with the user's input. There's no built-in behavior associated with these elements. Instead, you have to get the values that the user has entered or selected and do something with them. That's what you'll learn in this tutorial.

HTML5 and Input Forms

As you might know, HTML is in transition and the latest version (HTML5) includes support for more intuitive ways for users to enter information. For example, in HTML5, you (the page developer) can tell the page that you want the user to enter a date. The browser can then automatically display a calendar rather than requiring the user to enter a date manually. However, HTML5 is new and is not supported in all browsers yet.

ASP.NET Web Pages supports HTML5 input to the extent that the user's browser does. For an idea of the new attributes for the `<input>` element in HTML5, see [HTML <input> type Attribute](#) on the W3Schools site.

Creating the Form

In WebMatrix, in the **Files** workspace, open the *Movies.cshtml* page.

After the closing `</h1>` tag and before the opening `<div>` tag of the `grid.GetHtml` call, add the following markup:

```
<form method="get">
<div>
<label for="searchGenre">Genre to look for:</label>
<input type="text" name="searchGenre" value="" />
<input type="Submit" value="Search Genre" /><br/>
    (Leave blank to list all movies.)<br/>
</div>
</form>
```

This markup creates a form that has a text box named `searchGenre` and a submit button. The text box and submit button are enclosed in a `<form>` element whose `method` attribute is set to `get`. (Remember that if you don't put the text box and submit button inside a `<form>` element, nothing will be submitted when you click the button.) You use the `GET` verb here because you're creating a form that does not make any changes on the server — it just results in a search. (In the previous tutorial, you used a `post` method, which is how you submit changes to the server. You'll see that in the next tutorial again.)

Run the page. Although you haven't defined any behavior for the form, you can see what it looks like:

Movies

Genre to look for: [_____] [Search Genre]
(Leave blank to list all movies.)

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984

1 2 3 >

Enter a value into the text box, like "Comedy." Then click **Search Genre**.

Take note of the URL of the page. Because you set the `<form>` element's `method` attribute to `get`, the value you entered is now part of the query string in the URL, like this:

```
http://localhost:45661/Movies.cshtml?searchGenre=Comedy
```

Reading Form Values

The page already contains some code that gets database data and displays the results in a grid. Now you have to add some code that reads the value of the text box so you can run a SQL query that includes the search term.

Because you set the form's method to `get`, you can read the value that was entered into the text box by using code like the following:

```
var searchTerm = Request.QueryString["searchGenre"];
```

The `Request.QueryString` object (the `QueryString` property of the `Request` object) includes the values of elements that were submitted as part of the `GET` operation. The `Request.QueryString` property contains a *collection* (a list) of the values that are submitted in the form. To get any individual value, you specify the name of the element that you want. That's why you have to have a `name` attribute on the `<input>` element (`searchTerm`) that creates the text box. (For more about the `Request` object, see the [sidebar](#) later.)

It's simple enough to read the value of the text box. But if the user didn't enter anything at all in the text box but clicked **Search** anyway, you can ignore that click, since there's nothing to search.

The following code is an example that shows how to implement these conditions. (You don't have to add this code yet; you'll do that in a moment.)

```
if(!Request.QueryString["searchGenre"].IsEmpty() ) {
    // Do something here
}
```

The test breaks down in this way:

- Get the value of `Request.QueryString["searchGenre"]`, namely the value that was entered into the `<input>` element named `searchGenre`.
- Find out if it's empty by using the `IsEmpty` method.This method is the standard way to determine whether something (for example, a form element) contains a value. But really, you care only if it's *not* empty, therefore ...
- Add the `!` operator in front of the `IsEmpty` test. (The `!` operator means logical NOT).

In plain English, the entire `if` condition translates into the following: *If the form's searchGenre element is not empty, then ...*

This block sets the stage for creating a query that uses the search term. You'll do that in the next section.

The Request Object

The `Request` object contains all the information that the browser sends to your application when a page is requested or submitted. This object includes any information that the user provides, like text box values or a file to upload. It also includes all sorts of additional information, like cookies, values in the URL query string (if any), the file path of the page that is running, the type of browser that the user is using, the list of languages that are set in the browser, and much more.

The `Request` object is a *collection* (list) of values. You get an individual value out of the collection by specifying its name:

```
var someValue = Request["name"];
```

The `Request` object actually exposes several subsets. For example:

`Request.Form` gives you values from elements inside the submitted `<form>` element if the request is a `POST` request.

`Request.QueryString` gives you just the values in the URL's query string. (In a URL like `http://mysite/myapp/page?searchGenre=action&page=2`, the `?searchGenre=action&page=2` section of the URL is the query string.)

`Request.Cookies` collection gives you access to cookies that the browser has sent.

To get a value that you know is in the submitted form, you can use `Request["`*`name`*`"]`. Alternatively, you can use the more specific versions `Request.Form["`*`name`*`"]` (for `POST` requests) or `Request.QueryString["`*`name`*`"]` (for `GET` requests). Of course, *name* is the name of the item to get.

The name of the item you want to get has to be unique within the collection you're using. That's why the `Request` object provides the subsets like `Request.Form` and `Request.QueryString`. Suppose that your page contains a form element named `userName` and *also* contains a cookie named `userName`. If you get `Request["userName"]`, it's ambiguous whether you want the form value or the cookie. However, if you get `Request.Form["userName"]` or `Request.Cookie["userName"]`, you're being explicit about which value to get.

It's a good practice to be specific and use the subset of `Request` that you're interested in, like `Request.Form` or `Request.QueryString`. For the simple pages that you're creating in this tutorial, it probably doesn't really make any difference. However, as you create more complex pages, using the explicit version `Request.Form` or `Request.QueryString` can help you avoid problems that can arise when the page contains a form (or multiple forms), cookies, query string values, and so on.

Creating a Query by Using a Search Term

Now that you know how to get the search term that the user entered, you can create a query that uses it. Remember that to get all the movie items out of the database, you're using a SQL query that looks like this statement:

```
SELECT * FROM Movies
```

To get only certain movies, you have to use a query that includes a `Where` clause. This clause lets you set a condition on which rows are returned by the query. Here's an example:

```
SELECT * FROM Movies WHERE Genre = 'Action'
```

The basic format is `WHERE` *`column`* `=` *`value`*. You can use different operators besides just `=`, like `>` (greater than), `<` (less than), `<>` (not equal to), `<=` (less than or equal to), etc., depending on what you're looking for.

In case you're wondering, SQL statements are not case sensitive — `SELECT` is the same as `Select` (or even `select`). However, people often capitalize keywords in a SQL statement, like `SELECT` and `WHERE`, to make it easier to read.

Passing the search term as a parameter

Searching for a specific genre is easy enough (`WHERE Genre = 'Action'`), but you want to be able to search for any genre that the user enters. To do that, you create as SQL query that includes a placeholder for the value to search. It will look like this command:

```
SELECT * FROM Movies WHERE Genre = @0
```

The placeholder is the `@` character followed by zero. As you might guess, a query can contain multiple placeholders, and they'd be named `@0`, `@1`, `@2`, etc.

To set up the query and actually pass it the value, you use the code like the following:

```
selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
selectedData = db.Query(selectCommand, Request.QueryString["searchGenre"]);
```

This code is similar to what you've already done to display data in the grid. The only differences are:

- The query contains a placeholder (`WHERE Genre = @0"`).
- The query is put into a variable (`selectCommand`); before, you passed the query directly to the `db.Query` method.
- When you call the `db.Query` method, you pass both the query and the value to use for the placeholder. (If the query had multiple placeholders, you'd pass them all as separate values to the method.)

If you put all these elements together, you get the following code:

```
if(!Request.QueryString["searchGenre"].IsEmpty() ) {
    searchTerm = Request.QueryString["searchGenre"];
    selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
    selectedData = db.Query(selectCommand, searchTerm);
}
```

> **Important!** Using placeholders (like `@0`) to pass values to a SQL command is *extremely important* for security. The way you see it here, with placeholders for variable data, is the only way you should construct SQL commands.
>
> Never construct a SQL statement by putting together (concatenating) literal text and values you get from the user. Concatenating user input into a SQL statement opens your site to a *SQL injection attack* where a malicious user submits values to your page that hack your database. (You can read more in the article SQL Injection the MSDN website.)

Updating the Movies Page with Search Code

Now you can update the code in the *Movies.cshtml* file. To begin, replace the code in the code block at the top of the page with this code:

```
var db = Database.Open("WebPagesMovies");
var selectCommand = "SELECT * FROM Movies";
var searchTerm = "";
```

The difference here is that you've put the query into the `selectCommand` variable, which you'll pass to `db.Query` later. Putting the SQL statement into a variable lets you change the statement, which is what you'll do to perform the search.

You've also removed these two lines, which you'll put back in later:

```
var selectedData = db.Query("SELECT * FROM Movies");
var grid = new WebGrid(source: selectedData, rowsPerPage: 3);
```

You don't want to run the query yet (that is, call `db.Query`) and you don't want to initialize the `WebGrid` helper yet either. You'll do those things after you've figured out which SQL statement has to run.

After this rewritten block, you can add the new logic for handling the search. The completed code will look like the following. Update the code in your page so it matches this example:

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre",
rowsPerPage:3);
}
```

The page now works like this. Every time the page runs, the code opens the database and the `selectCommand` variable is set to the SQL statement that gets all the records from the `Movies` table. The code also initializes the `searchTerm` variable.

However, if the current request includes a value for the `searchGenre` element, the code sets `selectCommand` to a different query — namely, to one that includes the `Where` clause to search for a genre. It also sets `searchTerm` to whatever was passed for the search box (which might be nothing).

Regardless of which SQL statement is in `selectCommand`, the code then calls `db.Query` to run the query, passing it whatever is in `searchTerm`. If there's nothing in `searchTerm`, it doesn't matter, because in that case there's no parameter to pass the value to `selectCommand` anyway.

Finally, the code initializes the `WebGrid` helper by using the query results, just like before.

You can see that by putting the SQL statement and the search term into variables, you've added flexibility to the code. As you'll see later in this tutorial, you can use this basic framework and keep adding logic for different types of searches.

Testing the Search-by-Genre Feature

In WebMatrix, run the *Movies.cshtml* page. You see the page with the text box for genre.

Enter a genre that you've entered for one of your test records, then click **Search**. This time you see a listing of just the movies that match that genre:

Movies

Genre to look for: [_____] [Search Genre]
(Leave blank to list all movies.)

Title	Genre	Year
Ghostbusters	Comedy	1984
Clueless	Comedy	1995

Enter a different genre and search again. Try entering the genre by using all lowercase or all uppercase letters so that you can see that the search is not case sensitive.

"Remembering" What the User Entered

You might have noticed that after you entered a genre and clicked **Search Genre**, you saw a listing for that genre. However, the search text box was empty — in other words, it didn't remember what you'd entered.

It's important to understand why this behavior occurs. When you submit a page, the browser sends a request to the web server. When ASP.NET gets the request, it creates a brand-new instance of the page, runs the code in it, and then renders the page to the browser again. In effect, though, the page doesn't know that you were just working with a previous version of itself. All it knows is that it got a request that had some form data in it.

Every time you request a page — whether for the first time or by submitting it — you're getting a new page. The web server has no memory of your last request. Neither does ASP.NET, and neither does the browser. The only connection between these separate instances of the page is any data that you transmit between them. If you submit a page, for example, the new page instance can get the form data that was sent by the earlier instance. (Another way to pass data between pages is to use cookies.)

A formal way to describe this situation is to say that web pages are *stateless*. Web servers and the pages themselves and the elements in the page do not maintain any information about the previous state of a page. The web was designed this way because maintaining state for individual requests would quickly exhaust the resources of web servers, which often handle thousands, maybe even hundreds of thousands, of requests per second.

So that's why the text box was empty. After you submitted the page, ASP.NET created a new instance of the page and ran through the code and markup. There was nothing in that code that told ASP.NET to put a value into the text box. So ASP.NET didn't do anything, and the text box was rendered without a value in it.

There's actually an easy way to get around this issue. The genre that you entered into the text box *is* available to you in code — it's in `Request.QueryString["searchGenre"]`.

Update the markup for the text box so that the `value` attribute gets its value from `searchTerm`, like this example:

```
<input type="text" name="searchGenre" value="@Request.QueryString["searchGenre"]"
/>
```

In this page, you could have also set the `value` attribute to the `searchTerm` variable, since that variable also contains the genre you entered. But using the `Request` object to set the `value` attribute as shown here is the standard way to accomplish this task. (Assuming you even want to do this — in some situations, you might want to render the page *without* values in the fields. It all depends on what's going on with your app.)

> **Note** You can't "remember" the value of a text box that's used for passwords. It would be a security hole to allow people to fill in a password field by using code.

Run the page again, enter a genre, and click **Search Genre**. This time not only do you see the results of the search, but the text box remembers what you entered last time:

Searching for Any Word in the Title

You can now search for any genre, but you might also want to search for a title. It's hard to get a title exactly right when you search, so instead you can search for a word that appears anywhere inside a title. To do that in SQL, you use the `LIKE` operator and syntax like the following:

```
SELECT * FROM Movies WHERE Title LIKE '%adventure%'
```

This command gets all the movies whose titles contain "adventure". When you use the `LIKE` operator, you include the wildcard character `%` as part of the search term. The search `LIKE 'adventure%'` means "starting with 'adventure'". (Technically, it means "The string 'adventure' followed by anything.") Similarly, the search term `LIKE '%adventure'` means "anything followed by the string 'adventure'", which is another way to say "ending with 'adventure'".

The search term `LIKE '%adventure%'` therefore means "with 'adventure' anywhere in the title." (Technically, "anything in the title, followed by 'adventure', followed by anything.")

Inside the `<form>` element, add the following markup right under the closing `</div>` tag for the genre search (just before the closing `</form>` element):

```
<div>
<label for="SearchTitle">Movie title contains the following:</label>
<input type="text" name="searchTitle"
value="@Request.QueryString["searchTitle"]" />
<input type="Submit" value="Search Title" /><br/>
</div>
```

The code to handle this search is similar to the code for the genre search, except that you have to assemble the `LIKE` search. Inside the code block at the top of the page, add this `if` block just after the `if` block for the genre search:

```
if(!Request.QueryString["searchTitle"].IsEmpty() ) {
    selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
    searchTerm = "%" + Request["searchTitle"] + "%";
}
```

This code uses the same logic you saw earlier, except that the search uses a `LIKE` operator and the code puts "%" before and after the search term.

Notice how it was easy to add another search to the page. All you had to do was:

- Create an `if` block that tested to see whether the relevant search box had a value.
- Set the `selectCommand` variable to a new SQL statement.
- Set the `searchTerm` variable to the value to pass to the query.

Here's the complete code block, which contains the new logic for a title search:

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

  if(!Request.QueryString["searchTitle"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
        searchTerm = "%" + Request["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre",
rowsPerPage:8);
}
```

Here's a summary of what this code does:

- The variables `searchTerm` and `selectCommand` are initialized at the top. You're going to set these variables to the appropriate search term (if any) and appropriate SQL command based on what the user does in the page. The default search is the simple case of getting all the movies from the database.
- In the tests for `searchGenre` and `searchTitle`, the code sets `searchTerm` to the value you want to search for. Those code blocks also set `selectCommand` to an appropriate SQL command for that search.
- The `db.Query` method is invoked only once, using whatever SQL command is in `selectedCommand` and whatever value is in `searchTerm`. If there is no search term (no genre and no title word), the value of `searchTerm` is an empty string. However, that doesn't matter, because in that case the query doesn't require a parameter.

Testing the Title Search Feature

Now you can test your completed search page. Run *Movies.cshtml*.

Enter a genre and click **Search Genre**. The grid displays movies of that genre, like before.

Enter a title word and click **Search Title**. The grid displays movies that have that word in the title.

Movies

Genre to look for: Action   [Search Genre]
(Leave blank to list all movies.)
Movie title contains the following: The   [Search Title]

Title	Genre	Year
The Three Musketeers	Adventure	1973
Gone With The Wind	Drama	1939

Leave both text boxes blank and click either button. The grid displays all the movies.

Combining the Queries

You might notice that the searches you can perform are exclusive. You can't search the title and the genre at the same time, even if both search boxes have values in them. For example, you can't search for all action movies whose title contains "Adventure". (As the page is coded now, if you enter values for both genre and title, the title search gets precedence.) To create a search that combines the conditions,you would have to create a SQL query that has syntax like the following:

```
SELECT * FROM Movies WHERE Genre = @0 AND Title LIKE @1
```

And you'd have to run the query by using a statement like the following (roughly speaking):

```
var selectedData = db.Query(selectCommand, searchGenre, searchTitle);
```

Creating logic to allow many permutations of search criteria can get a bit involved, as you can see. Therefore, we'll stop here.

Coming Up Next

In the next tutorial, you'll create a page that uses a form to let users add movies to the database.

Additional Resources

- Introduction to ASP.NET Web Programming Using the Razor Syntax
- SQL WHERE Clause on the W3Schools site
- Method Definitions article on the W3C site

Tutorial 5: Entering Database Data by Using Forms

This tutorial shows you how to create an entry form and then enter the data that you get from the form into a database table when you use ASP.NET Web Pages (Razor).

What you'll learn:

- More about how to process entry forms.
- How to add (insert) data in a database.
- How to make sure that users have entered a required value in a form (how to validate user input).
- How to display validation errors.
- How to jump to another page from the current page.

Features/technologies discussed:

- The `Database.Execute` method.
- The SQL `Insert Into` statement
- The `Validation` helper.
- The `Response.Redirect` method.

What You'll Build

In the tutorial earlier that showed you how to create a database, you entered database data by editing the database directly in WebMatrix, working in the **Database** workspace. In most apps, that's not a practical way to put data into the database, though. So in this tutorial, you'll create a web-based interface that lets you or anyone enter data and save it to the database.

You'll create a page where you can enter new movies. The page will contain an entry form that has fields (text boxes) where you can enter a movie title, genre, and year.The page will look like this page:

The text boxes will be HTML `<input>` elements that will look like this markup:

```
<input type="text" name="genre" value="" />
```

Creating the Basic Entry Form

Create a page named *AddMovie.cshtml*.

Replace what's in the file with the following markup. Overwrite everything; you'll add a code block at the top shortly.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Add a Movie</title>
</head>
<body>
<h1>Add a Movie</h1>
<form method="post">
<fieldset>
<legend>Movie Information</legend>
<p><label for="title">Title:</label>
<input type="text" name="title" value="@Request.Form["title"]" />

<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@Request.Form["genre"]" />

<p><label for="year">Year:</label>
<input type="text" name="year" value="@Request.Form["year"]" />

<p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
</fieldset>
</form>
</body>
</html>
```

This example shows is typical HTML for creating a form. It uses `<input>` elements for the text boxes and for the submit button. The captions for the text boxes are created by using standard `<label>` elements. The `<fieldset>` and `<legend>` elements put a nice box around the form.

Notice that in this page, the `<form>` element uses `post` as the value for the `method` attribute. In the previous tutorial, you created a form that used the `get` method. That was correct, because although the form submitted values to the server, the request did not make any changes. All it did was fetch data in different ways. However, in this page you *will* make changes—you're going to add new database records. Therefore, this form should use the `post` method. (For more about the difference between GET and POST operations, see the [GET, POST, and HTTP Verb Safety](#) sidebar in the previous tutorial.)

Note that each text box has a `name` element (`title`, `genre`, `year`). As you saw in the previous tutorial, these names are important because you must have those names so you can get the user's input later. You can use any names. It's helpful to use meaningful names that help you remember what data you're working with.

The `value` attribute of each `<input>` element contains a bit of Razor code (for example, `Request.Form["title"]`). You learned a version of this trick in the previous tutorial to preserve the value entered into the text box (if any) after the form has been submitted.

Getting the Form Values

Next, you add code that processes the form. In outline, you'll do the following:

1. Check whether the page is being posted (was submitted). You want to your code to run only when users have clicked the button, not when the page first runs.
2. Get the values that the user entered into the text boxes. In this case, because the form is using the `POST` verb, you get the form values from the `Request.Form` collection.
3. Insert the values as a new record in the *Movies* database table.

At the top of the file, add the following code:

```
@{
    var title = "";
    var genre = "";
    var year = "";

    if(IsPost){
        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];
    }
}
```

The first few lines create variables (`title`, `genre`, and `year`) to hold the values from the text boxes. The line `if(IsPost)` makes sure that the variables are set *only* when users click the **Add Movie** button — that is, when the form has been posted.

As you saw in an earlier tutorial, you get the value of a text box by using an expression like `Request.Form["`*name*`"]`, where *name* is the name of the `<input>` element.

The names of the variables (`title`, `genre`, and `year`) are arbitrary. Like the names that you assign to `<input>` elements, you can call them anything you like. (The names of the variables don't have to match the name attributes of `<input>` elements on the form.) But as with the `<input>` elements, it's a good idea to use variable names that reflect the data that they contain. When you write code, consistent names make it easier for you to remember what data you're working with.

Adding Data to the Database

In the code block you just added, just *inside* the closing brace (}) of the `if` block (not just inside the code block), add the following code:

```
var db = Database.Open("WebPagesMovies");
var insertCommand = "INSERT INTO Movies (Title, Genre, Year) VALUES(@0, @1, @2)";
db.Execute(insertCommand, title, genre, year);
```

This example is similar to the code you used in a previous tutorial to fetch and display data. The line that starts with `db =` opens the database, like before, and the next line defines a SQL statement, again as you saw before. However, this time it defines a SQL `Insert Into` statement. The following example shows the general syntax of the `Insert Into` statement:

```
INSERT INTO table (column1, column2, column3, ...) VALUES (value1, value2, value3, ...)
```

In other words, you specify the table to insert into, then list the columns to insert into, and then list the values to insert.(As noted before, SQL is not case sensitive but some people capitalize the keywords to make it easier to read the command.)

The columns that you're inserting into are already listed in the command — `(Title, Genre, Year)`. The interesting part is how you get the values from the text boxes into the `VALUES` part of the command. Instead of actual values, you see `@0`, `@1`, and `@2`, which are of course placeholders. When you run the command (on the `db.Execute` line), you pass the values that you got from the text boxes.

> **Important!**Remember that the only way you should ever include data entered online by a user in a SQL statement is to use placeholders, as you see here (`VALUES(@0, @1, @2)`). If you concatenate user input into a SQL statement, you open yourself to a SQL injection attack, as explained in [HTML Form Basics](#) (the previous tutorial).

Still inside the `if` block, add the following line after the `db.Execute` line:

```
Response.Redirect("~/Movies");
```

After the new movie has been inserted into the database, this line jumps you (redirects) to the *Movies* page so you can see the movie you just entered. The ~ operator means "root of the website." (The ~ operator works only in ASP.NET pages, not in HTML generally.)

The complete code block looks like this example:

```
@{
    var title = "";
    var genre = "";
    var year = "";

    if(IsPost){
        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];

        var db = Database.Open("WebPagesMovies");
        var insertCommand = "INSERT INTO Movies (Title, Genre, Year) " +
"Values(@0, @1, @2)";
        db.Execute(insertCommand, title, genre, year);
        Response.Redirect("~/Movies");
    }
}
```

Testing the Insert Command (So Far)

You're not done yet, but now is a good time to test.

In the tree view of files in WebMatrix, right-click the *AddMovie.cshtml* page and then click **Launch in browser**.

Add a Movie

Movie Information

Title:

Genre:

Year:

Add Movie

(If you end up with a different page in the browser, make sure that the URL is
`http://localhost:nnnnn/AddMovie`), where *nnnnn* is the port number that you're using.)

Did you get an error page? If so, read it carefully and make sure that the code looks exactly what was listed earlier.

Enter a movie in the form — for example, use "Citizen Kane", "Drama", and "1941".(Or whatever.)
Then click **Add Movie**.

If all goes well, you're redirected to the *Movies* page. Make sure that your new movie is listed.

Validating User Input

Go back to the *AddMovie* page, or run it again. Enter another movie, but this time, enter only the title — for example, enter "Singin' in the Rain". Then click **Add Movie**.

You're redirected to the *Movies* page again. You can find the new movie, but it's incomplete.

When you created the *Movies* table, you explicitly said that none of the fields could be null. Here you have an entry form for new movies, and you're leaving fields blank. That's an error.

In this case, the database didn't actually raise (or *throw*) an error. You didn't supply a genre or year, so the code in the *AddMovie* page treated those values as so-called *empty strings*. When the SQL `Insert Into` command ran, the genre and year fields didn't have useful data in them, but they weren't null.

Obviously, you don't want to let users enter half-empty movie information into the database. The solution is to validate the user's input. Initially, the validation will simply make sure that the user has entered a value for all of the fields (that is, that none of them contains an empty string).

Null and Empty Strings

In programming, there's a distinction between different notions of "no value." In general, a value is *null* if it has never been set or initialized in any way. In contrast, a variable that expects character data (strings) can be set to an *empty string*. In that case, the value is not null; it's just been explicitly set to a string of characters whose length is zero. These two statements show the difference:

```
var firstName;        // Not set, so its value is null
var firstName = "";   // Explicitly set to an empty string -- not null
```

It's a little more complicated than that, but the important point is that `null` represents a sort of undetermined state.

Now and then it's important to understand exactly when a value is null and when it's just an empty string. In the code for the *AddMovie* page, you get the values of the text boxes by using `Request.Form["title"]` and so on. When the page first runs (before you click the button), the value of `Request.Form["title"]` is null. But when you submit the form, `Request.Form["title"]` gets the value of the `title` text box. It's not obvious, but an empty text box is not null; it just has an empty string in it. So when the code runs in response to the button click, `Request.Form["title"]` has an empty string in it.

Why is this distinction important? When you created the *Movies* table, you explicitly said that none of the fields could be null. But here you have an entry form for new movies, and you're leaving fields blank. You would reasonably expect the database to complain when you tried to save new movies that didn't have values for genre or year. But that's the point — even if you leave those text boxes blank, the values aren't null; they're empty strings. As a result, you're able to save new movies to the database with these columns empty — but not null! — values. Therefore, you have to make sure that users don't submit an empty string, which you can do by validating the user's input.

The Validation Helper

ASP.NET Web Pages includes a helper — the `Validation` helper — that you can use to make sure that users enter data that meets your requirements. The `Validation` helper is one of the helpers that's built in to ASP.NET Web Pages, so you don't have to install it as a package by using NuGet, the way you installed the Twitter helper in an earlier tutorial.

To validate the user's input, you'll do the following:

- Use code to specify that you want to require values in the text boxes on the page.
- Put a test into the code so that the movie information is added to the database only if everything validates properly.
- Add code into the markup to display error messages.

In the code block in the *AddMovie* page, right up at the top before the variable declarations, add the following code:

```
Validation.RequireField("title", "You must enter a title");
Validation.RequireField("genre", "Genre is required");
Validation.RequireField("year", "You haven't entered a year");
```

You call `Validation.RequireField` once for each field (`<input>` element) where you want to require an entry. You can also add a custom error message for each call, like you see here. (We varied the messages just to show that you can put anything you like there.)

If there's a problem, you want to prevent the new movie information from being inserted into the database. In the `if(IsPost)` block, use `&&` (logical AND) to add another condition that tests `Validation.IsValid()`. When you're done, the whole `if(IsPost)` block looks like this code:

```
if(IsPost && Validation.IsValid()){
    title = Request.Form["title"];
    genre = Request.Form["genre"];
    year = Request.Form["year"];

    var db = Database.Open("WebPagesMovies");
    var insertCommand = "INSERT INTO Movies (Title, Genre, Year)" +
        " Values(@0, @1, @2)";
    db.Execute(insertCommand, title, genre, year);
    Response.Redirect("~/Movies");
}
```

If there's a validation error with any of the fields that you registered by using the `Validation` helper, the `Validation.IsValid` method returns false. And in that case, none of the code in that block will run, so no invalid movie entries will be inserted into the database. And of course you're not redirected to the *Movies* page.

The complete code block, including the validation code, now looks like this example:

```
@{
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost && Validation.IsValid()){
        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];

        var db = Database.Open("WebPagesMovies");
        var insertCommand = "INSERT INTO Movies (Title, Genre, Year) " +
            " Values(@0, @1, @2)";
```

```
        db.Execute(insertCommand, title, genre, year);
        Response.Redirect("~/Movies");
    }
}
```

Displaying Validation Errors

The last step is to display any error messages. You can display individual messages for each validation error, or you can display a summary, or both. For this tutorial, you'll do both so that you can see how it works.

Next to each `<input>` element that you're validating, call the `Html.ValidationMessage` method and pass it the name of the `<input>` element you're validating. You put the `Html.ValidationMessage` method right where you want the error message to appear. When the page runs, the `Html.ValidationMessage` method renders a `<span>` element where the validation error will go. (If there's no error, the `<span>` element is rendered, but there's no text in it.)

Change the markup in the page so that it includes an `Html.ValidationMessage` method for each of the three `<input>` elements on the page, like this example:

```
<p><label for="title">Title:</label>
<input type="text" name="title" value="@Request.Form["title"]" />
    @Html.ValidationMessage("title")
</p>

<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@Request.Form["genre"]" />
    @Html.ValidationMessage("genre")
</p>

<p><label for="year">Year:</label>
<input type="text" name="year" value="@Request.Form["year"]" />
    @Html.ValidationMessage("year")
</p>
```

To see how the summary works, also add the following markup and code right after the `<h1>Add a Movie</h1>` element on the page:

```
@Html.ValidationSummary()
```

By default, the `Html.ValidationSummary` method displays all the validation messages in a list (a `<ul>` element that's inside a `<div>` element). As with the `Html.ValidationMessage` method, the markup for the validation summary is always rendered; if there are no errors, no list items are rendered.

The summary can be an alternative way to display validation messages instead of by using the `Html.ValidationMessage` method to display each field-specific error. Or you can use both a

summary and the details. Or you can use the `Html.ValidationSummary` method to display a generic error and then use individual `Html.ValidationMessage` calls to display details.

The complete page now looks like this example:

```
@{
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost && Validation.IsValid()){
        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];

        var db = Database.Open("WebPagesMovies");
        var insertCommand = "INSERT INTO Movies (Title, Genre, Year) " +
            " Values(@0, @1, @2)";
        db.Execute(insertCommand, title, genre, year);
        Response.Redirect("~/Movies");
    }
}

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Add a Movie</title>
</head>
<body>
<h1>Add a Movie</h1>
  @Html.ValidationSummary()
<form method="post">
<fieldset>
<legend>Movie Information</legend>
<p><label for="title">Title:</label>
<input type="text" name="title" value="@Request.Form["title"]" />
        @Html.ValidationMessage("title")
</p>

<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@Request.Form["genre"]" />
        @Html.ValidationMessage("genre")
</p>

<p><label for="year">Year:</label>
<input type="text" name="year" value="@Request.Form["year"]" />
        @Html.ValidationMessage("year")
</p>

<p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
</fieldset>
</form>
</body>
```

```
    </html>
```

That's it. You can now test the page by adding a movie but leaving out one or more of the fields.
When you do, you see the following error display:

Add a Movie

- Genre is required
- You haven't entered a year

Movie Information

Title: Lawrence of Arabia

Genre: [   ] Genre is required

Year: [   ] You haven't entered a year

[Add Movie]

Styling the Validation Error Messages

You can see that there are error messages, but they don't really stand out very well. There's an
easy way to style the error messages, though.

To style the individual error messages that are displayed by `Html.ValidationMessage`, create a CSS
style class named `field-validation-error`. To define the look for the validation summary, create
a CSS style class named `validation-summary-errors`.

To see how this technique works, add a `<style>` element inside the `<head>` section of the page.
Then define style classes named `field-validation-error` and `validation-summary-errors` that
contain the following rules:

```
<head>
<meta charset="utf-8" />
<title>Add a Movie</title>
<style type="text/css">
    .field-validation-error {
      font-weight:bold;
      color:red;
      background-color:yellow;
     }
    .validation-summary-errors{
      border:2px dashed red;
      color:red;
      background-color:yellow;
      font-weight:bold;
      margin:12px;
    }
```

```
        </style>
    </head>
```

Normally you'd probably put style information into a separate *.css* file, but for simplicity you can put them in the page for now. (Later in this tutorial set, you'll move the CSS rules to a separate *.css* file.)

If there's a validation error, the `Html.ValidationMessage` method renders a `<span>` element that includes `class="field-validation-error"`. By adding a style definition for that class, you can configure what the message looks like. If there are errors, the `ValidationSummary` method likewise dynamically renders the attribute `class="validation-summary-errors"`.

Run the page again and deliberately leave out a couple of the fields. The errors are now more noticeable. (In fact, they're overdone, but that's just to show what you can do.)

Add a Movie

> • **Genre is required**
> • **You haven't entered a year**

Movie Information

Title: Lawrence of Arabia

Genre: **Genre is required**

Year: **You haven't entered a year**

[Add Movie]

Adding a Link to the Movies Page

One final step is to make it convenient to get to the *AddMovie* page from the original movie listing.

Open the *Movies* page again. After the closing `</div>` tag that follows the `WebGrid` helper, add the following markup:

```
<p>
<a href="~/AddMovie">Add a movie</a>
</p>
```

As you saw before, ASP.NET interprets the `~` operator as the root of the website. You don't have to use the `~` operator; you could use the markup `<a href="./AddMovie">Add a movie</a>` or some

other way to define the path that HTML understands. But the ~ operator is a good general approach when you create links for Razor pages, because it makes the site more flexible — if you move the current page to a subfolder, the link will still go to the *AddMovie* page. (Remember that the ~ operator only works in *.cshtml* pages. ASP.NET understands it, but it's not standard HTML.)

When you're done, run the *Movies* page. It will look like this page:

Click the **Add a movie** link to make sure that it goes to the *AddMovie* page.

Coming Up Next

In the next tutorial, you'll learn how to let users edit data that's already in the database.

Additional Resources

- Complete Listing for AddMovie Page
- Introduction to ASP.NET Web Programming Using the Razor Syntax
- SQL INSERT INTO Statement on the W3Schools site
- Validating User Input in ASP.NET Web Pages Sites. More information about working with the `Validation` helper.

Tutorial 6: Updating Database Data

This tutorial shows you how to update (change) an existing database entry when you use ASP.NET Web Pages (Razor).

What you'll learn:

- How to select an individual record in the `WebGrid` helper.
- How to read a single record from a database.
- How to preload a form with values from the database record.
- How to update an existing record in a database.
- How to store information in the page without displaying it.
- How to use a hidden field to store information.

Features/technologies discussed:

- The `WebGrid` helper.
- The SQL `Update` command.
- The `Database.Execute` method.
- Hidden fields (`<input type="hidden">`).

What You'll Build

In the previous tutorial, you learned how to add a record to a database. Here, you'll learn how to display a record for editing.In the *Movies* page, you'll update the `WebGrid` helper so that it displays an **Edit** link next to each movie:

When you click the **Edit** link, it takes you to a different page, where the movie information is already in a form:

Edit a Movie

Movie Information

Title: The Three Musketeers

Genre: Adventure

Year: 1973

Submit Changes

You can change any of the values. When you submit the changes, the code in the page updates the database and takes you back to the movie listing.

This part of the process works almost exactly like the *AddMovie.cshtml* page you created in the previous tutorial, so much of this tutorial will be familiar.

There are several ways you could implement a way to edit an individual movie. The approach shown was chosen because it's easy to implement and easy to understand.

Adding an Edit Link to the Movie Listing

To begin, you'll update the *Movies* page so that each movie listing also contains an **Edit** link.

Open the *Movies.cshtml* file.

In the body of the page, change the `WebGrid` markup by adding a column. Here's the modified markup:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    columns: grid.Columns(
        grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
        grid.Column("Title"),
        grid.Column("Genre"),
        grid.Column("Year")
    )
)
```

The new column is this one:

```
grid.Column(format: @<a href="~/EditMovie?id=@item.ID)">Edit</a>)
```

The point of this column is to show a link (`<a>` element) whose text says "Edit". What we're after is to create a link that looks like the following when the page runs, with the `id` value different for each movie:

```
http://localhost:43097/EditMovie?id=7
```

This link will invoke a page named *EditMovie*, and it will pass the query string `?id=7` to that page.

The syntax for the new column might look a bit complex, but that's only because it puts together several elements. Each individual element is straightforward. If you concentrate on just the `<a>` element, you see this markup:

```
<a href="~/EditMovie?id=@item.ID)">Edit</a>
```

Some background about how the grid works: the grid displays rows, one for each database record, and it displays columns for each field in the database record. While each grid row is being constructed,the `item` object contains the database record (item) for that row. This arrangement gives you a way in code to get at the data for that row. That's what you see here: the expression `item.ID` is getting the ID value of the current database item. You could get any of the database values (title, genre, or year) the same way by using `item.Title`, `item.Genre`, or `item.Year`.

The expression "`~/EditMovie?id=@item.ID` combines the hard-coded part of the target URL (`~/EditMovie?id=`) with this dynamically derived ID. (You saw the `~` operator in the previous tutorial; it's an ASP.NET operator that reprethe current website root.)

The result is that this part of the markup in the column simply produces something like the following markup at run time:

```
href="/EditMovie?id=2"
```

Naturally, the actual value of `id` will be different for each row.

Creating a Custom Display for a Grid Column

Now back to the grid column. The three columns you originally had in the grid displayed only data values (title, genre, and year). You specified this display by passing the name of the database column — for example, `grid.Column("Title")`.

This new **Edit** link column is different. Instead of specifying a column name, you're passing a `format` parameter. This parameter lets you define markup that the `WebGrid` helper will render along with the `item` valueto display the column data as bold or green or in whatever format that you want. For example, if you wanted the title to appear bold, you could create a column like this example:

```
grid.Column(format:@<strong>@item.Title</strong>)
```

(The various @ characters you see in the `format` property mark the transition between markup and a code value.)

Once you know about the `format` property, it's easier to understand how the new **Edit** link column is put together:

```
grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
```

The column consists *only* of the markup that renders the link, plus some information (the ID) that's extracted from the database record for the row.

Named Parameters and Positional Parameters for a Method

Many times when you've called a method and passed parameters to it, you've simply listed the parameter values separated by commas. Here are a couple of examples:

```
db.Execute(insertCommand, title, genre, year)
```

```
Validation.RequireField("title", "You must enter a title")
```

We didn't mention the issue when you first saw this code, but in each case, you're passing parameters to the methods in a specific order — namely, the order in which the parameters are defined in that method. For `db.Execute` and `Validation.RequireFields`, if you mixed up the order of the values you pass, you'd get an error message when the page runs, or at least some strange results. Clearly, you have to know the order to pass the parameters in. (In WebMatrix, IntelliSense can help you learn figure out the name, type, and order of the parameters.)

As an alternative to passing values in order, you can use *named parameters*. (Passing parameters in order is known as using *positional parameters*.) For named parameters, you explicitly include the name of the parameter when passing its value. You've used named parameters already a number of times in these tutorials. For example:

```
var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:3)
```

and

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    columns: grid.Columns(
```

```
        grid.Column("Title"),
        grid.Column("Genre"),
        grid.Column("Year")
    )
)
```

Named parameters are handy for a couple of situations, especially when a method takes many parameters. One is when you want to pass only one or two parameters, but the values you want to pass are not among the first positions in the parameter list. Another situation is when you want to make your code more readable by passing the parameters in the order that makes the most sense to you.

Obviously, to use named parameters, you have to know the names of the parameters. WebMatrix IntelliSense can *show* you the names, but it cannot currently fill them in for you.

Creating the Edit Page

Now you can create the *EditMovie* page. When users click the **Edit** link, they'll end up on this page.

Create a page named *EditMovie.cshtml* and replace what's in the file with the following markup:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Edit a Movie</title>
<style>
    .validation-summary-errors{
      border:2px dashed red;
      color:red;
      font-weight:bold;
      margin:12px;
    }
</style>
</head>
</head>
<body>
<h1>Edit a Movie</h1>
    @Html.ValidationSummary()
<form method="post">
<fieldset>
<legend>Movie Information</legend>

<p><label for="title">Title:</label>
<input type="text" name="title" value="@title" /></p>

<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@genre" /></p>

<p><label for="year">Year:</label>
<input type="text" name="year" value="@year" /></p>
```

```
<input type="hidden" name="movieid" value="@movieId" />

<p><input type="submit" name="buttonSubmit" value="Submit Changes" /></p>
</fieldset>
</form>
</body>
</html>
```

This markup and code is similar to what you have in the *AddMovie* page. There's a small difference in the text for the submit button. As with the *AddMovie* page, there's an `Html.ValidationSummary` call that will display validation errors if there are any. This time we're leaving out calls to `Validation.Message`, since errors will be displayed in the validation summary. As noted in the previous tutorial, you can use the validation summary and the individual error messages in various combinations.

Notice again that the `method` attribute of the `<form>` element is set to `post`. As with the *AddMovie.cshtml* page, this page makes changes to the database. Therefore, this form should perform a `POST` operation. (For more about the difference between `GET` and `POST` operations, see the [GET, POST, and HTTP Verb Safety](#) sidebar in the tutorial on HTML forms.)

As you saw in an earlier tutorial, the `value` attributes of the text boxes are being set with Razor code in order to preload them. This time, though, you're using variables like `title` and `genre` for that task instead of `Request.Form["title"]`:

```
<input type="text" name="title" value="@title" />
```

As before, this markup will preload the text box values with the movie values. You'll see in a moment why it's handy to use variables this time instead of using the `Request` object.

There's also a `<input type="hidden">` element on this page. This element stores the movie ID without making it visible on the page. The ID is initially passed to the page by using a query string value (`?id=7` or similar in the URL).By putting the ID value into a hidden field, you can make sure that it's available when the form is submitted, even if you no longer have access to the original URL that the page was invoked with.

Unlike the *AddMovie* page, the code for the *EditMovie* page has two distinct functions. The first function is that when the page is displayed for the first time (and *only* then), the code gets the movie ID from the query string. The code then uses the ID to read the corresponding movie out of the database and display (preload) it in the text boxes.

The second function is that when the user clicks the **Submit Changes** button, the code has to read the values of the text boxes and validate them. The code also has to update the database item with the new values. This technique is similar to adding a record, as you saw in *AddMovie*.

Adding Code to Read a Single Movie

To perform the first function, add this code to the top of the page:

```
@{
    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty()){
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);
            title = row.Title;
            genre = row.Genre;
            year = row.Year;
        }
        else{
            Validation.AddFormError("No movie was selected.");
            // If you are using a version of ASP.NET Web Pages 2 that's
            // earlier than the RC release, comment out the preceding
            // statement and uncomment the following one.
            //ModelState.AddFormError("No movie was selected.");
        }
    }
}
```

Most of this code is inside a block that starts `if(!IsPost)`. The `!` operator means "not,"so the expression means *if this request is not a post submission*, which is an indirect way of saying *if this request is the first time that this page has been run*. As noted earlier, this code should run *only* the first time the page runs. If you didn't enclose the code in `if(!IsPost)`, it would run every time the page is invoked, whether the first time or in response to a button click.

Notice that the code includes an `else` block this time. As we said when we introduced `if` blocks, sometimes you want to run alternative code if the condition you're testing isn't true. That's the case here. If the condition passes (that is, if the ID passed to the page is ok), you read a row from the database. However, if the condition doesn't pass, the `else` block runs and the code sets an error message.

Validating a Value Passed to the Page

The code uses `Request.QueryString["id"]` to get the ID that's passed to the page. The code makes sure that a value was actually passed for the ID. If no value was passed, the code sets a validation error.

This code shows a different way to validate information. In the previous tutorial, you worked with the `Validation` helper. You registered fields to validate, and ASP.NET automatically did the

validation and displayed errors by using `Html.ValidationMessage` and `Html.ValidationSummary`. In this case, however, you're not really validating user input. Instead,you're validating a value that was passed to the page from elsewhere. The `Validation` helper doesn't do that for you.

Therefore, you check the value yourself, by testing it with`if(!Request.QueryString["ID"].IsEmpty())`. If there's a problem, you can display the error by using `Html.ValidationSummary`, as you did with the `Validation` helper. To do that, you call `Validation.AddFormError` and pass it a message to display. `Validation.AddFormError` is a built-in method that lets you define custom messages that tie in with the validation system you're already familiar with. (Later in this tutorial we'll talk about how to make this validation process a little more robust.)

> **Note** If you're using a version of ASP.NET Web Pages 2 earlier than the RC release (for example, if you're using the Beta refresh release from February 2012), use `ModelState.AddFormError` instead of `Validation.AddFormError`. The `AddFormError` method was added to the `Validation` helper in the RC release.

After making sure that there's an ID for the movie, the code reads the database, looking for only a single database item. (You probably have noticed the general pattern for database operations: open the database, define a SQL statement, and run the statement.) This time, the SQL `Select` statement includes `WHERE ID = @0`. Because the ID is unique, only one record can be returned.

The query is performed by using `db.QuerySingle` (not `db.Query`, as you used for the movie listing), and the code puts the result into the `row` variable. The name `row` is arbitrary; you can name the variables anything you like.The variables initialized at the top are then filled with the movie details so that these values can be displayed in the text boxes.

Testing the Edit Page (So Far)

If you'd like to test your page, run the *Movies* page now and click an **Edit** link next to any movie. You'll see the *EditMovie* page with the details filled in for the movie you selected:

Edit a Movie

Movie Information

Title: The Three Musketeers

Genre: Adventure

Year: 1973

[Submit Changes]

Notice that the URL of the page includes something like `?id=10` (or some other number). So far you've tested that **Edit** links in the *Movie* page work, that your page is reading the ID from the query string, and that the database query to get a single movie record is working.

You can change the movie information, but nothing happens when you click **Submit Changes**.

Adding Code to Update the Movie with the User's Changes

In the *EditMovie.cshtml* file, to implement the second function (saving changes), add the following code just inside the closing bracket of the `@` block. (If you're sure exactly where to put the code, you can look at the complete page listing for the Edit Movie page that appears at the end of this tutorial.)

```
if(IsPost){
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");
    Validation.RequireField("movieid", "No movie ID was submitted!");

    title = Request.Form["title"];
    genre = Request.Form["genre"];
    year = Request.Form["year"];
    movieId = Request.Form["movieId"];

    if(Validation.IsValid()){
        var db = Database.Open("WebPagesMovies");
        var updateCommand = "UPDATE Movies " +
                "SET Title=@0, Genre=@1, Year=@2 WHERE Id=@3";
        db.Execute(updateCommand, title, genre, year, movieId);
        Response.Redirect("~/Movies");
    }
}
```

Again, this markup and code is similar to the code in *AddMovie*. The code is in an `if(IsPost)` block, because this code runs only when the user clicks the **Submit Changes** button — that is, when (and only when) the form has been posted. In this case, you're not using a test like `if(IsPost && Validation.IsValid())` — that is, you're not combining both tests by using AND. In this page, you first determine whether there's a form submission (`if(IsPost)`), and only then register the fields for validation. Then you can test the validation results (`if(Validation.IsValid())`). The flow is slightly different than in the *AddMovie.cshtml* page, but the effect is the same.

You get the values of the text boxes by using `Request.Form["title"]` and similar code for the other `<input>` elements. Notice that this time, the code gets the movie ID out of the hidden field (`<input type="hidden">`). When the page ran the first time, the code got the ID out of the query string. You get the value from the hidden field to make sure that you're getting the ID of the movie that was originally displayed, in case the query string was somehow altered since then.

The really important difference between the *AddMovie* code and this code is that in this code you use the SQL `Update` statement instead of the `Insert Into` statement. The following example shows the syntax of the SQL `Update` statement:

```
UPDATE table SET col1="value", col2="value", col3="value" ... WHERE ID = value
```

You can specify any columns in any order, and you don't necessarily have to update every column during an `Update` operation. (You cannot update the ID itself, because that would in effect save the record as a new record, and that's not allowed for an `Update` operation.)

> **Important** The `Where` clause with the ID is very important, because that's how the database knows which database record you want to update. If you left off the `Where` clause, the database would update *every* record in the database. In most cases, that would be a disaster.

In the code, the values to update are passed to the SQL statement by using placeholders. To repeat what we've said before: for security reasons, *only* use placeholders to pass values to a SQL statement.

After the code uses `db.Execute` to run the `Update` statement,it redirects back to the listing page, where you can see the changes.

Different SQL Statements, Different Methods

You might have noticed that you use slightly different methods to run different SQL statements. To run a `Select` query that potentially returns multiple records, you use the `Query` method. To run a `Select` query that you know will return only one database item, you use the `QuerySingle` method. To run commands that make changes but that don't return database items, you use the `Execute` method.

You have to have different methods because each of them returns different results, as you saw already in the difference between `Query` and `QuerySingle`. (The `Execute` method actually returns a value also — namely, the number of database rows that were affected by the command — but you've been ignoring that so far.)

Of course, the `Query` method might return only one database row. However, ASP.NET always treats the results of the `Query` method as a collection. Even if the method returns just one row, you have to extract that single row from the collection. Therefore, in situations where you *know* you'll get back only one row, it's a bit more convenient to use `QuerySingle`.

There are a few other methods that perform specific types of database operations. You can find a listing of database methods in the [ASP.NET Web Pages API Quick Reference](#).

Making Validation for the ID More Robust

The first time that the page runs, you get the movie ID from the query string so that you can go get that movie from the database. You made sure that there actually was a value to go look for, which you did by using this code:

```
if(!IsPost){
    if(!Request.QueryString["ID"].IsEmpty()){
        // Etc.
    }
}
```

You used this code to make sure that if a user gets to the *EditMovies* page without first selecting a movie in the *Movies* page, the page would display a user-friendly error message. (Otherwise, users would see an error that would probably just confuse them.)

However, this validation isn't very robust. The page might also be invoked with these errors:

- The ID isn't a number. For example, the page could be invoked with a URL like `http://localhost:nnnnn/EditMovie?id=abc`.

- The ID is a number, but it references a movie that doesn't exist (for example, `http://localhost:nnnnn/EditMovie?id=100934`).

If you're curious to see the errors that result from these URLs, run the *Movies* page. Select a movie to edit, and then change the URL of the *EditMovie* page to a URL that contains an alphabetic ID or the ID of a non-existent movie.

So what should you do? The first fix is to make sure that not only is an ID passed to the page, but that the ID is an integer. Change the code for the `!IsPost` test to look like this example:

```
if(!IsPost){
    if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].IsInt())
    {
        // Etc.
```

You've added a second condition to the `IsEmpty` test, linked with `&&` (logical AND):

```
Request.QueryString["ID"].IsInt()
```

You might remember from the [Programming Basics](#) tutorial that methods like `AsBool` an `AsInt` convert a character string to some other data type. The `IsInt` method (and others, like `IsBool` and `IsDateTime`) are similar. However, they test only whether you *can* convert the string, without actually performing the conversion. So here you're essentially saying *If the query string value can be converted to an integer ...* .

The other potential problem is looking for a movie that doesn't exist. The code to get a movie looks like this code:

```
var row = db.QuerySingle(dbCommand, movieId);
```

If you pass a `movieId` value to the `QuerySingle` method that doesn't correspond to an actual movie, nothing is returned and the statements that follow (for example, `title=row.Title`) result in errors.

Again there's an easy fix. If the `db.QuerySingle` method returns no results, the `row` variable will be null. So you can check whether the `row` variable is null before you try to get values from it. The following code adds an `if` block around the statements that get the values out of the `row` object:

```
if(row != null) {
    title = row.Title;
    genre = row.Genre;
    year = row.Year;
}
else{
    Validation.AddFormError("No movie was found for that ID.");
    // Use the following line instead for versions of ASP.NET Web Pages 2 earlier
    // than the RC release.
    //ModelState.AddFormError("No movie was found for that ID.");
}
```

With these two additional validation tests, the page becomes more bullet-proof. The complete code for the `!IsPost` branch now looks like this example:

```
if(!IsPost){
    if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].IsInt())
{
        movieId = Request.QueryString["ID"];
        var db = Database.Open("WebPagesMovies");
        var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
        var row = db.QuerySingle(dbCommand, movieId);

        if(row != null) {
            title = row.Title;
            genre = row.Genre;
             year = row.Year;
        }
        else{
                Validation.AddFormError("No movie was found for that ID.");
                // Use the following line instead for versions of ASP.NET
                // Web Pages 2 earlierthan the RC release.
                //ModelState.AddFormError("No movie was found for that ID.");
        }
    }
    else{
        Validation.AddFormError("No movie was selected.")
        // Use the following line instead for versions of ASP.NET
        // Web Pages 2 earlierthan the RC release.
```

```
            //ModelState.AddFormError("No movie was selected.");
        }
    }
```

We'll note once more that this task is a good use for an `else` block. If the tests don't pass, the `else` blocks set error messages.

Adding a Link to Return to the Movies Page

A final and helpful detail is to add a link back to the *Movies* page. In the ordinary flow of events, users will start at the *Movies* page and click an **Edit** link. That brings them to the *EditMovie* page, where they can edit the movie and click the button. After the code processes the change, it redirects back to the *Movies* page.

However:

- The user might decide not to change anything.
- The user might have gotten to this page without first clicking an **Edit** link in the *Movies* page.

Either way, you want to make it easy for them to return to the main listing. It's an easy fix — add the following markup just after the closing `</form>` tag in the markup:

```
<p><a href="~/Movies">Return to movie listing</a></p>
```

This markup uses the same syntax for an `<a>` element that you've seen elsewhere. The URL includes `~` to mean "root of the website."

Testing the Movie Update Process

Now you can test. Run the *Movies* page,and click **Edit** next to a movie. When the *EditMovie* page appears, make changes to the movie and click**Submit Changes**. When the movie listing appears, make sure that your changes are shown.

To make sure that validation is working, click **Edit** for another movie. When you get to the *EditMovie* page, clear the **Genre** field (or **Year** field, or both) and try to submit your changes. You'll see an error, as you'd expect:

Edit a Movie

---
* **Genre is required**
* **You haven't entered a year**

Movie Information

Title: Fantasia

Genre:

Year:

[Submit Changes]

Return to movie listing

Click the **Return to movie listing** link to abandon your changes and return to the *Movies* page.

Coming Up Next

In the next tutorial,you'll see how to delete a movie record.

Additional Resources

* Complete Listing for Movie Page (Updated with Edit Links)
* Complete Page Listing for Edit Movie Page
* SQL UPDATE Statement on the W3Schools site

Tutorial 7: Deleting Database Data

This tutorial shows you how to delete an individual database entry.

What you'll learn:

- How to select an individual record from a listing of records.
- How to delete a single record from a database.
- How to check that a specific button was clicked in a form.

Features/technologies discussed:

- The `WebGrid` helper.
- The SQL `Delete` command.
- The `Database.Execute` method to run a SQL `Delete` command.

What You'll Build

In the previous tutorial, you learned how to update an existing database record.This tutorial is similar, except that instead of updating the record,you'll delete it. The processes are much the same, except that deleting issimpler, so this tutorial will be short.

In the *Movies* page,you'll update the `WebGrid` helper so that it displays a **Delete** link next to each movie to accompany the **Edit**link you added earlier.

As with editing, when you click the **Delete** link, it takes you to a differentpage, where the movie information is already in a form:

Delete a Movie

Return to movie listing

Movie Information

Title: Fantasia

Genre: Children

Year: 1939

[Delete Movie]

You can then click the button to delete the record permanently.

Adding a Delete Link to the Movie Listing

You'll start by adding a **Delete** link to the `WebGrid` helper. This link is similar to the **Edit** link you added in aprevious tutorial.

Open the *Movies.cshtml* file.

Change the `WebGrid` markup in the body of the page by adding acolumn. Here's the modified markup:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    columns: grid.Columns(
      grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
      grid.Column("Title"),
      grid.Column("Genre"),
      grid.Column("Year"),
      grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
    )
)
```

The new column is this one:

```
grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
```

The way the grid is configured, the **Edit** column is leftmostin the grid and the **Delete** column is rightmost. (There's acomma after the `Year` column now, in case you didn't notice that.)There's nothing special about where these link columns go, and you could aseasily put them next to each other. In this case, they're separate to make them harder to get mixed up.

The new column shows a link (`<a>` element)whose text says "Delete". The target of the link (its `href` attribute)is code that ultimately resolves to something like this URL, with the `id`value different for each movie:

```
http://localhost:43097/DeleteMovie?id=7
```

This link will invoke a page named *DeleteMovie* and pass it the ID of themovie you've selected.

This tutorial won't go into detail about how this link is constructed,because it's almost identical to the **Edit** link from theprevious tutorial ([Updating Database Data](#)).

Creating the Delete Page

Now you can create the page that will be the target for the **Delete**link in the grid.

> **Important** The technique of firstselecting a record to delete and then using a separate page and button to confirm the process is extremely important for security. As you've read in previous tutorials, making *any* sort of change toyour website should *always*be done using a form — that is, using an HTTP POST operation. If you made itpossible to change the site just by clicking a link (that is, using a GEToperation), people could make simple requests to your site and delete your data. Even a search-engine crawler that's indexing your site could inadvertently delete data just by following links.

When your app lets people change a record, you have to present the record to the user for editing anyway. Butyou might be tempted to skip this step for deleting a record. Don't skip that step, though. (It's also helpful for users to see the record and confirm that they're deleting the record that they intended.)

In a subsequent tutorial set, you'll see how to add loginfunctionality so a user would have to log in before deleting a record.

Create a page named *DeleteMovie.cshtml* and replace what's in the file with the following markup:

```
<html>
<head>
  <title>Delete a Movie</title>
</head>
<body>
     <h1>Delete a Movie</h1>
       @Html.ValidationSummary()
     <p><a href="~/Movies">Return to movie listing</a></p>

     <form method="post">
       <fieldset>
       <legend>Movie Information</legend>

       <p><span>Title:</span>
        <span>@title</span></p>

       <p><span>Genre:</span>
        <span>@genre</span></p>

       <p><span>Year:</span>
         <span>@year</span></p>

       <input type="hidden" name="movieid" value="@movieId" />
       <p><input type="submit" name="buttonDelete" value="Delete Movie" /></p>
       </fieldset>
     </form>
   </body>
</html>
```

This markup is like the *EditMovie* pages, except that instead of using textboxes
(`<input type="text">`), the markup includes `<span>`elements. There's nothing here to edit. All you
have to do is display the moviedetails so that users can make sure that they're deleting the right
movie.

The markup already contains a link that lets the user return to the movielisting page.

As in the *EditMovie* page, the ID of the selected movie is stored ina hidden field. (It's passed into
the page in the first place as a query stringvalue.)There's an `Html.ValidationSummary` call thatwill
display validation errors. In this case, the errormight be that no movie ID was passed to the page
or that the movie ID is invalid. This situation could occur if someoneran this page without first
selecting a movie in the *Movies* page.

The button caption is **Delete Movie**, and its name attributeis set to `buttonDelete`. The `name`
attribute will be usedin the code to identify the button that submitted the form.

You'll have to write code to 1) read the movie details when the page is firstdisplayed and 2)
actually delete the movie when the user clicks the button.

Adding Code to Read a Single Movie

At the top of the *DeleteMovie.cshtml* page, add the following codeblock:

```
@{
    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty() &&
Request.QueryString["ID"].IsInt()){
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);
            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was found for that ID.");
                // If you are using a version of ASP.NET Web Pages 2 that's
                // earlier than the RC release, comment out the preceding
                // statement and uncomment the following one.
                //ModelState.AddFormError("No movie was found for that ID.");
            }
        }
        else{
            Validation.AddFormError("No movie was found for that ID.");
            // If you are using a version of ASP.NET Web Pages 2 that's
            // earlier than the RC release, comment out the preceding
            // statement and uncomment the following one.
            //ModelState.AddFormError("No movie was found for that ID.");
        }
    }
}
```

This markup is the same as the corresponding code in the *EditMovie* page. It gets the movie ID out of the query string and uses the ID to read a recordfrom the database. The code includes the validation test (`IsInt()`and `row != null`) to make sure that the movie ID being passed to thepage is valid.

Remember that this code should only run the first time the page runs. You don't want to re-read the movie record from the database when the user clicksthe **Delete Movie** button.Therefore, code to read the movie is inside a test that says `if(!IsPost)` — that is, *if the request is not a post operation (form submission).*

Adding Code to Delete the Selected Movie

To delete the movie when the user clicks the button, add the following code just inside the closing bracket of the @ block:

```
if(IsPost && !Request["buttonDelete"].IsEmpty()){
    movieId = Request.Form["movieId"];
    var db = Database.Open("WebPagesMovies");
    var deleteCommand = "DELETE FROM Movies WHERE ID = @0";
    db.Execute(deleteCommand, movieId);
    Response.Redirect("~/Movies");
}
```

This code is similar to the code for updating an existing record, but simpler. The code basically runs a SQL `Delete` statement.

As in the *EditMovie* page, the code is in an `if(IsPost)` block. This time, the `if()` condition is a little more complicated:

```
if(IsPost && !Request["buttonDelete"].IsEmpty())
```

There are two conditions here. The first is that the page is being submitted, as you've seen before — `if(IsPost)`.

The second condition is `!Request["buttonDelete"].IsEmpty()`, meaning that the request has an object named `buttonDelete`. Admittedly, it's an indirect way of testing which button submitted the form. If a form contains multiple submit buttons, only the name of the button that was clicked appears in the request. Therefore, logically, if the name of a particular button appears in the request — or as stated in the code, if that button isn't empty — that's the button that submitted the form.

The `&&` operator means "and" (logical AND). Therefore the entire `if` condition is ...

This request is a post (not a first-time request)

AND

The `buttonDelete` button was the button that submitted the form.

This form (in fact, this page) contains only one button, so the additional test for `buttonDelete` is technically not required. Still, you're about to perform an operation that will permanently remove data. So you want to be as sure as possible that you're performing the operation only when the user has explicitly requested it. For example, suppose that you expanded this page later and added other buttons to it. Even then, the code that deletes the movie will run only if the `buttonDelete` button was clicked.

As in the *EditMovie* page, you get the ID from the hidden field andthen run the SQL command. The syntax for the `Delete` statement is:

```
DELETE FROM table WHERE ID = value
```

It's vital to include the `WHERE` clause and the ID. If you leaveout the WHERE clause, *all the records in the table will be deleted*. As youhave seen, you pass the ID value to the SQL command by using a placeholder.

Testing the Movie Delete Process

Now you can test. Run the *Movies* page, and click **Delete**next to a movie. When the *DeleteMovie* page appears, click **Delete Movie**.

Delete a Movie

Return to movie listing

Movie Information

Title: Fantasia

Genre: Children

Year: 1939

Delete Movie

When you click the button, the code deletes the movies and returns to themovie listing. There you can search for the deleted movie and confirm that it's been deleted.

Coming Up Next

The next tutorial shows you how to give all the pages on your site a commonlook and layout.

Additional Resources

- Complete Listing for Movie Page (Updated with Delete Links)
- Complete Listing for DeleteMovie Page
- Introduction to ASP.NET Web Programming by Using the Razor Syntax
- SQL DELETE Statement on the W3Schools site

Tutorial 8: Creating a Consistent Layout

This tutorial shows you how to use *layouts* to create a consistent look for the pages on a site that uses ASP.NET Web Pages.

What you'll learn:

- What a layout page is.
- How to combine layout pages with dynamic content.
- How to pass values to a layout page.

About Layouts

The pages you've created so far have all been complete, standalone pages. They all belong to the same site, but they don't have any common elements or a standard look.

Most sites do have a consistent look and layout. For example, if you go to the Microsoft.com/web site and look around, you see that the pages all adhere to an overall layout and to a visual theme:

An *inefficient* way to create this layout would be to define a header, navigation bar, and footer separately on each of your pages. You'd be duplicating the same markup each time. If you wanted to change something (for example, update the footer), you'd have to change each page separately.

That's where *layout pages* come in. In ASP.NET Web Pages, you can define a layout page that provides an overall container for pages on your site. For example, the layout page can contain the header, navigation area, and footer. The layout page includes a placeholder where the main content goes.

You can then define individual content pages that contain the markup and the code for only that page. Content pages don't have to be complete HTML pages; they don't even have to have a `<body>` element. They also have a line of code that tells ASP.NET what layout page you want to display the content in. Here's a picture that shows roughly how this relationship works:

Movies page

```
@{
  Layout = "_Layout.cshtml";
  // more code ...
}
<h1>Movies</h1>

@grid.GetHtml()
```

Layout page (_Layout.cshtml)

```
<html>
<body>
 <div id="header">
   Welcome to My Site!
 </div>
 <div id="main">
  @RenderBody()
 </div>
 <div id="footer">
   &copy; 2012
 </div>
</body>
</html>
```

Finished Movies page

```
<html>
<body>
 <div id="header">
   Welcome to My Site!
 </div>
 <div id="main">
 <h1>Movies</h1>
 <div id="main">
  <table>
    <tr>
     <td>Title</td>
     <td>Genre</td>
     <td>Year</td>
     ... etc
  </table>
 </div>
 <div id="footer">
   &copy; 2012
 </div>
</body>
</html>
```
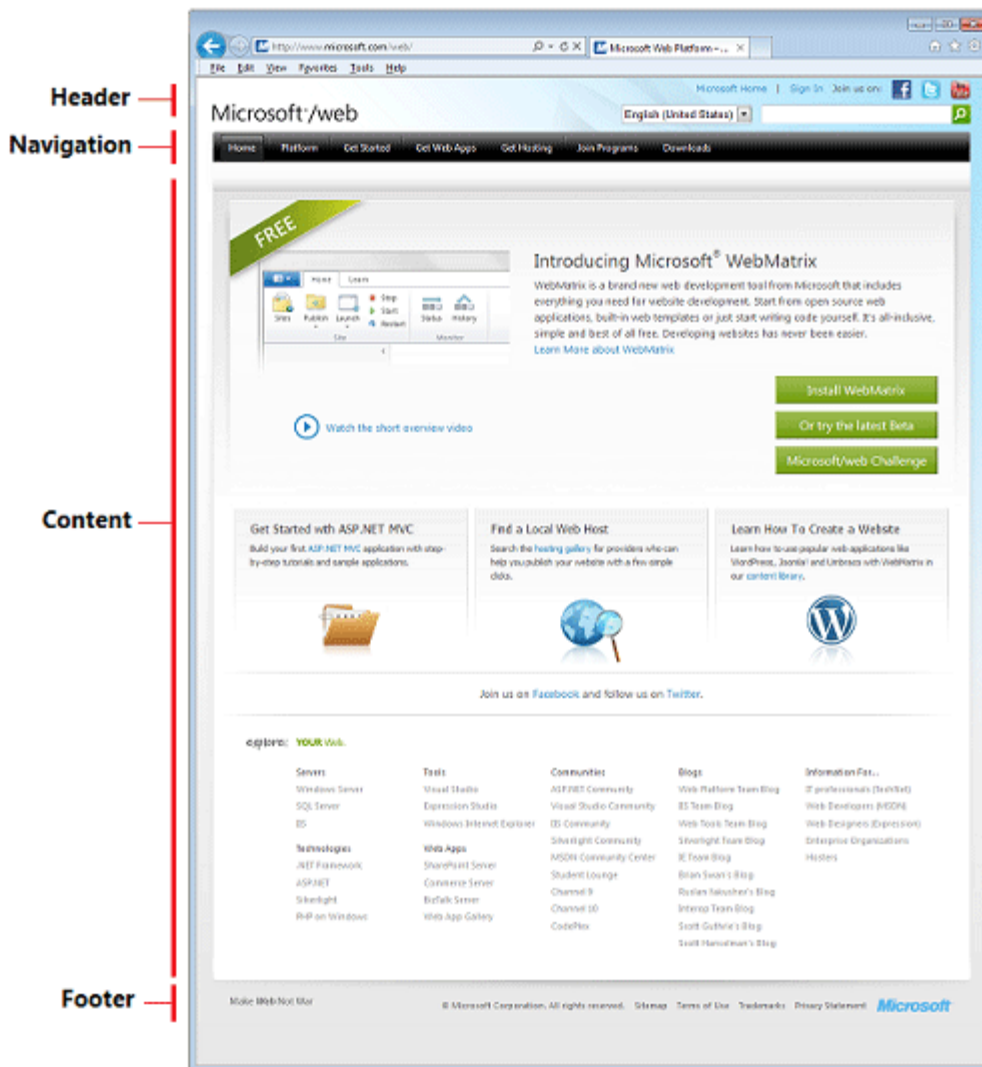
This interaction is easy to understand when you see it in action. In this tutorial, you'll change your movies pages to use a layout.

Adding a Layout Page

You'll start by creating a layout page that defines a typical page layout with a header, footer, and an area for the main content. In the WebPagesMovies site, add a CSHTML page named _Layout.cshtml.

The leading underscore (_) character is significant. If a page's name starts with an underscore, ASP.NET won't directly send that page to the browser. This convention lets you define pages that are required for your site but that users shouldn't be able to request directly.

Replace the content in the page with the following:

```
<!DOCTYPE html>
<html>
<head>
<title>@Page.Title</title>
<link href="~/Styles/Movies.css" rel="stylesheet" type="text/css" />
</head>
<body>
<div id="container">
<div id="header">
<h1>My Movie Site</h1>
</div>
<div id="main">
        @RenderBody()
</div>
<div id="footer">
&copy; @DateTime.Now.Year My Movie Site
</div>
</div>
</body>
</html>
```

As you can see, this markup is just HTML that uses `<div>` elements to define three sections in the page plus one more `<div>` element to hold the three sections. The footer contains a bit of Razor code: `@DateTime.Now.Year`, which will render the current year at that location in the page.

Notice that there's a link to a style sheet named *Movies.css*. The style sheet is where the details of the physical layout of the elements will be defined. You'll create that in a moment.

The only unusual feature in this *_Layout.cshtml* page is the `@Render.Body()` line. That's the placeholder where the content will go when this layout is merged with another page.

Adding a .css File

The preferred way to define the actual arrangement (that is, appearance) of elements on the page is to use cascading style sheet (CSS) rules. So you'll create a *.css* file that has the rules for your new layout.

In WebMatrix, select the root of your site. Then in the **Files** tab of the ribbon, click the arrow under the **New** button and then click **New Folder**.
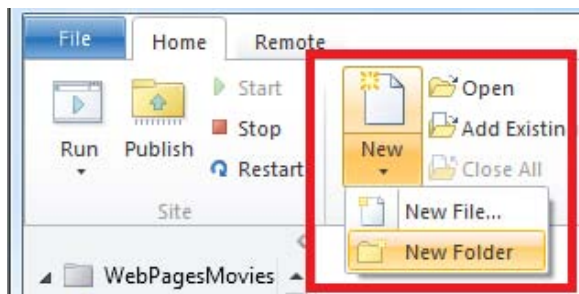
Name the new folder *Styles*.
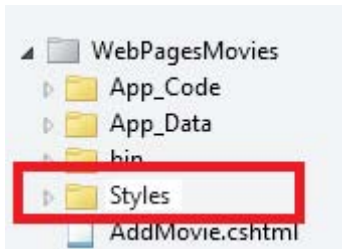
Inside the new *Styles* folder, create a file named *Movies.css*.
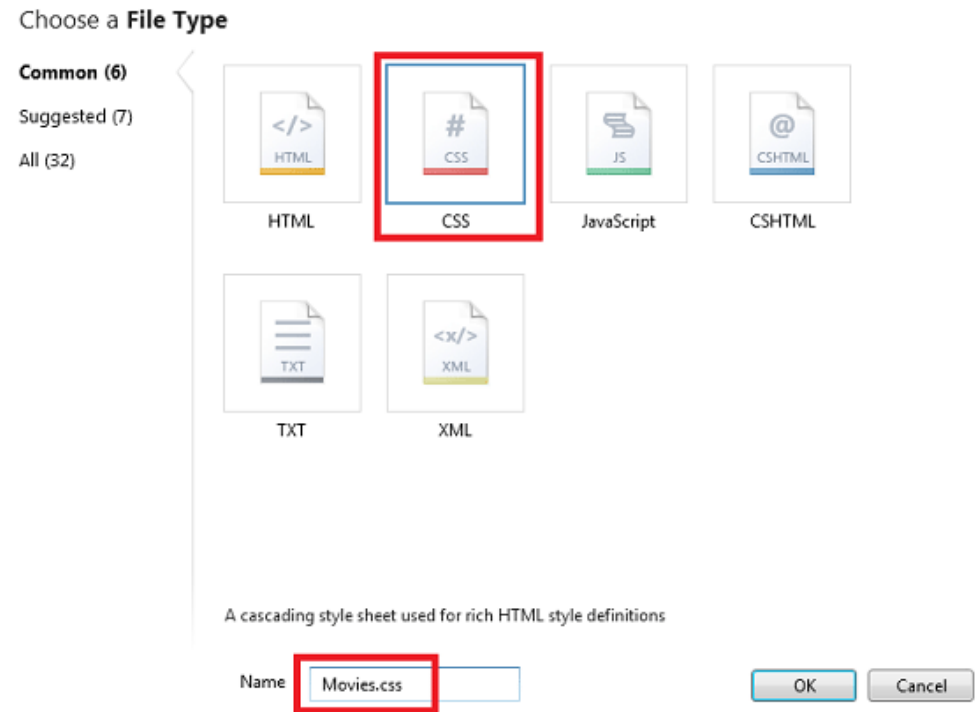
Replace the contents of the new *.css* file with the following:

```
html{ height:100%; margin:0; padding:0; }

body {
  height:60%;
  font-family:'Trebuchet MS',  'Arial', 'Helvetica', 'sans-serif';
  font-size:10pt;
  background-color: LightGray;
  line-height:1.6em;
}

h1{ font-size:1.6em; }
h2{ font-size:1.4em; }

#container{
    min-height:100%;
```

```
    position:relative;
    left:10%;
}

#header{
   padding:8px;
   width:80%;
   background-color:#4b6c9e;
   color:White;
}

#main{
   width:80%;
   padding: 8px;
   padding-bottom:4em;
   background-color:White;
}

#footer{
   position:absolute;
   bottom:0;
   width:80%;
   height:2em;
   padding:8px;
   margin-top:-2em;
   background-color:LightGray;
}

.head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
.grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
.alt { background-color: #E8E8E8; color: #000; }
.selected {background-color:Yellow;}
span.caption {width:100px;}
span.dataDisplay {font-weight:bold;}
```

We won't say much about these CSS rules, except to note two things. One is that in addition to setting fonts and sizes, the rules use absolute positioning to establish the location of the header, footer, and main content area. If you're new to positioning in CSS, you can read the CSS Positioning tutorial at the W3Schools site.

The other thing to note is that at the bottom, we've copied the style rules that were originally defined individually in the *Movies.cshtml* file. These rules were used in the Displaying Data tutorial to make the `WebGrid` helper render markup that added stripes to the table. (If you're going to use a *.css* file for style definitions, you might as well put the style rules for the whole site in it.)

Updating the Movies File to Use the Layout

Now you can update the existing files in your site to use the new layout. Open the *Movies.cshtml* file. At the top, as the first line of code, add the following:

```
Layout = "~/_Layout.cshtml";
```

The page now starts out this way:

```
@{
    Layout = "~/_Layout.cshtml";

    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    // Etc.
```

This one line of code tells ASP.NET that when the *Movies* page runs, it should be merged with the *_Layout.cshtml* file.

Since the *Movies.cshtml* file now uses a layout page, you can remove the markup from the *Movies.cshtml* page that's taken care of by the *_Layout.cshtml* file. Take out the `<!DOCTYPE>`, `<html>`, and `<body>` opening and closing tags. Take out the entire `<head>` element and its contents, which includes the style rules for the grid, since you've now got those rules in a *.css* file. While you're at it, change the existing `<h1>` element to an `<h2>` element; you have an `<h1>` element in the layout page already. Change the `<h2>` text to "List Movies".

Normally you wouldn't have to make these sorts of changes in a content page. When you start your site out with a layout page, you create content pages without all these elements to begin with. In this case, though, you're converting a standalone page to one that uses a layout, so there's a bit of cleanup.

When you're finished, the *Movies.cshtml* page will look like the following:

```
@{
    Layout = "~/_Layout.cshtml";

    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
      selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
      searchTerm = "%" + Request.QueryString["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre",
rowsPerPage:3);
}
<h2>List Movies</h2>
<form method="get">
<div>
```

```
<label for="searchGenre">Genre to look for:</label>
<input type="text" name="searchGenre"
         value="@Request.QueryString["searchGenre"]" />
<input type="Submit" value="Search Genre" /><br/>
     (Leave blank to list all movies.)<br/>
</div>
<div>
<label for="SearchTitle">Movie title contains the following:</label>
<input type="text" name="searchTitle"
value="@Request.QueryString["searchTitle"]" />
<input type="Submit" value="Search Title" /><br/>
</div>
</form>
<div>
    @grid.GetHtml(
        tableStyle: "grid",
        headerStyle: "head",
        alternatingRowStyle: "alt",
        columns: grid.Columns(
            grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
            grid.Column("Title"),
            grid.Column("Genre"),
            grid.Column("Year"),
            grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
        )
    )
</div>
<p><a href="~/AddMovie">Add a movie</a></p>
```

Testing the Layout

Now you can see what the layout looks like. In WebMatrix, right-click the *Movies.cshtml* page and select **Launch in browser**. When the browser displays the page, it looks like this page:
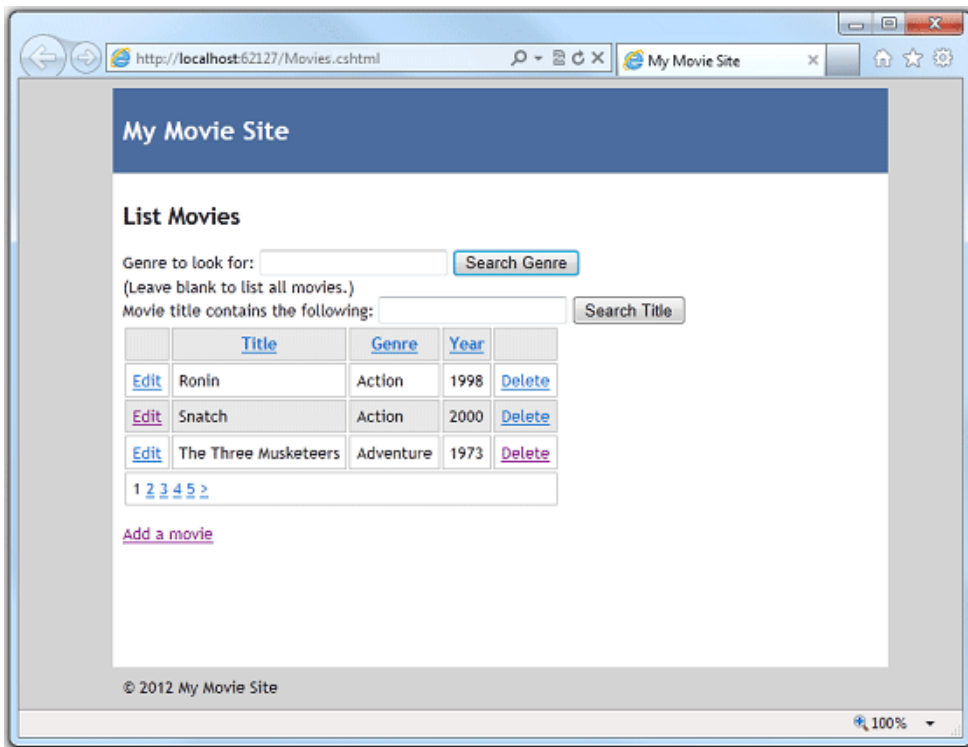
ASP.NET has merged the content of the Movies.cshtml page into the _Layout.cshtml_ page right where the `RenderBody` method is. And of course the _Layout.cshtml_ page references a *.css* file that defines the look of the page.

Updating the AddMovie Page to Use the Layout

The real benefit of layouts is that you can use them for all the pages in your site. Open the *AddMovie.cshtml* page.

You might remember that the *AddMovie.cshtml* page originally had some CSS rules in it to define the look of validation error messages. Since you have a *.css* file for your site now, you can move those rules to the *.css* file. Remove them from the *AddMovie.cshtml* file and add them to the bottom of the *Movies.css* file. You are moving the following rules:

```
.field-validation-error {
  font-weight:bold;
  color:red;
  background-color:yellow;
 }
.validation-summary-errors{
  border:2px dashed red;
  color:red;
  background-color:yellow;
  font-weight:bold;
  margin:12px;
}
```

Now make the same sorts of changes in *AddMovie.cshtml* that you did for *Movies.cshtml* — add `Layout="~/_Layout.cshtml;` and remove the HTML markup that's now extraneous. Change the `<h1>` element to `<h2>`. When you're done, the page will look like this example:

```
@{
    Layout = "~/_Layout.cshtml";
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost){
        if(Validation.IsValid()){
            title = Request.Form["title"];
            genre = Request.Form["genre"];
            year = Request.Form["year"];

            var db = Database.Open("WebPagesMovies");
            var insertCommand =
                "INSERT INTO Movies (Title, Genre, Year) Values(@0, @1, @2)";
            db.Execute(insertCommand, title, genre, year);
            Response.Redirect("~/Movies");
        }
    }
}
<h2>Add a Movie</h2>
    @Html.ValidationSummary()
<form method="post">
<fieldset>
<legend>Movie Information</legend>
<p><label for="title">Title:</label>
<input type="text" name="title" value="@Request.Form["title"]" />
      @Html.ValidationMessage("title")
</p>

<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@Request.Form["genre"]" />
      @Html.ValidationMessage("genre")
</p>

<p><label for="year">Year:</label>
<input type="text" name="year" value="@Request.Form["year"]" />
      @Html.ValidationMessage("year")
</p>

<p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
</fieldset>
</form>
```

Run the page. Now it looks like this illustration:
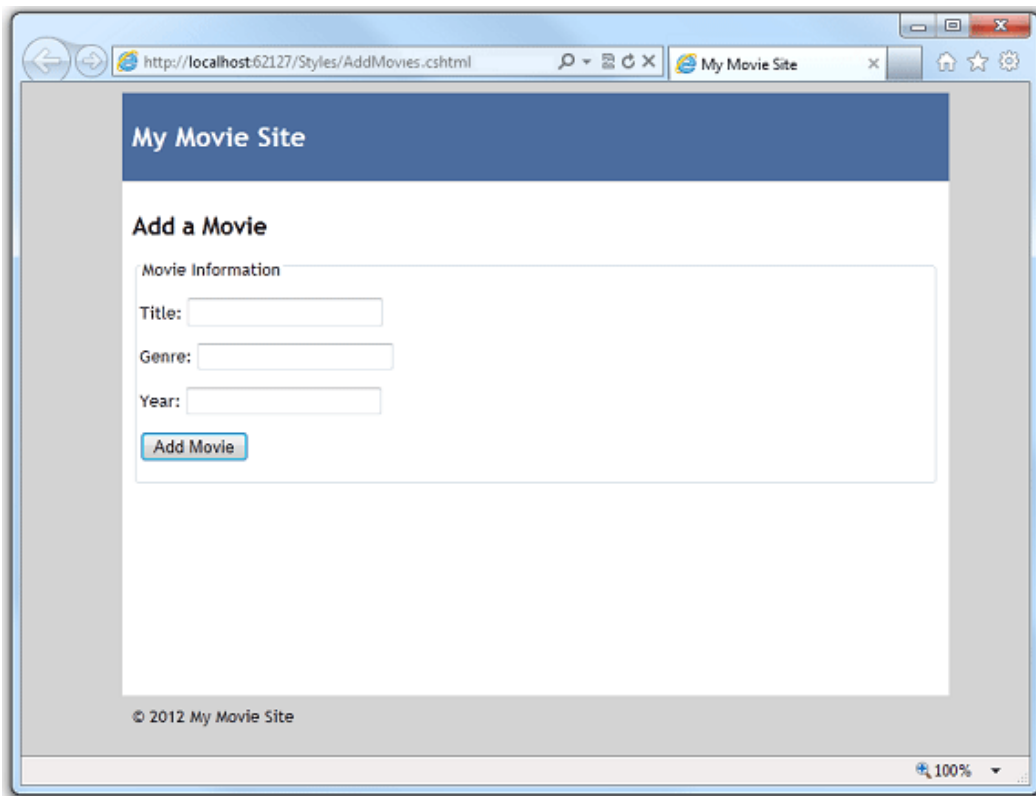
You want to make similar changes to the pages in the site — *EditMovie.cshtml*and *DeleteMovie.cshtml*. However, before you do, you can make another change to the layout that makes it a little more flexible.

Passing Title Information to the Layout Page

The *_Layout.cshtml* page that you created has a `<title>` element that's set to "My Movie Site". Most browsers display the content of this element as the text on a tab:

This title information is generic. Suppose that you want the title text to be more specific to the current page. (The title text is also used by search engines to determine what your page is about.) You can pass information from a content page like *Movies.cshtml* or *AddMovie.cshtml* to the layout page, and then use that information to customize what the layout page renders.

Open the *Movies.cshtml* page again. In the code at the top, add the following line:

```
Page.Title = "List Movies";
```

The `Page` object is available on all *.cshtml* pages and is for this purpose, namely to share information between a page and its layout.

Open the *_Layout.cshtml* page. Change the `<title>` element so that it looks like this markup:

```
<title>@Page.Title</title>
```

This code renders whatever is in the `Page.Title` property right at that location in the page.

Run the *Movies.cshtml* page. This time the browser tab shows what you passed as the value of `Page.Title`:
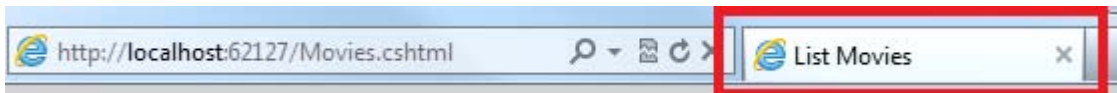
If you want, view the page source in the browser. You can see that the `<title>` element is rendered as `<title>List Movies</title>`.

The Page Object

A useful feature of `Page` is that it's a dynamic object — the `Title` propertyis not a fixed or reserved name. You can use *any* name for a value of the `Page` object. For example, you could as easily have passed the title by using a property named `Page.CurrentName` or `Page.MyPage`. The only restriction is that the name has to follow the normal rules for what properties can be named. (For example, the name can't contain a space.)

You can pass any number of values by using the `Page` object. If you wanted to pass movie information to the layout page, you could pass values by using something like `Page.MovieTitle` and `Page.Genre` and `Page.MovieYear`. (Or any other names that you invented to store the information.) The only requirement — which is probably obvious — is that you have to use the same names in the content page and the layout page.

The information you pass by using the `Page` object isn't limited to just text to display on the layout page. You can pass a value to the layout page, and then code in the layout page can use the value to decide whether to display a section of the page, what *.css* file to use, and so on. The values you pass in the `Page` object are like any other values that you use in code. It's just that the values originate in the content page and are passed to the layout page.

Open the *AddMovie.cshtml* page and add a line to the top of the code that provides a title for the *AddMovie.cshtml* page:

```
Page.Title = "Add a Movie";
```

Run the *AddMovie.cshtml* page. You see the new title there:

Updating the Remaining Pages to Use the Layout

Now you can finish the remaining pages in your site so that they use the new layout. Open *EditMovie.cshtml* and *DeleteMovie.cshtml* in turn and make the same changes in each.

Add the line of code that links to the layout page:

```
Layout = "~/_Layout.cshtml";
```

Add a line to set the title of the page:

```
Page.Title = "Edit a Movie";
```

or:

```
Page.Title = "Delete a Movie";
```

Remove all the extraneous HTML markup — basically, leave only the bits that are inside the `<body>` element (plus the code block at the top).

Change the `<h1>` element to be an `<h2>` element.

When you've made these changes, test each and make sure thatit's displaying properly and that the title is correct.

Parting Thoughts About Layout Pages

In this tutorial you created a *_Layout.cshtml* page and used the `RenderBody` method to merge content from another page. That's the basic pattern for using layouts in Web Pages.

Layout pages have additional features that we didn't cover here. For example, you can nest layout pages — one layout page can in turn reference another. Nested layouts can be useful if you're working with subsections of a site that require different layouts. You can also use additional methods (for example, `RenderSection`) to set up named sections in the layout page.

The combination of layout pages and *.css* files is powerful. As you'll see in the next tutorial series, in WebMatrix you can create a site based on a *template*, which gives you a site that has prebuilt

functionality in it. The templates make good use of layout pages and CSS to create sites that look great and that have features like menus. Here's a screenshot of the home page from a site based on a template, showing features that use layout pages and CSS:
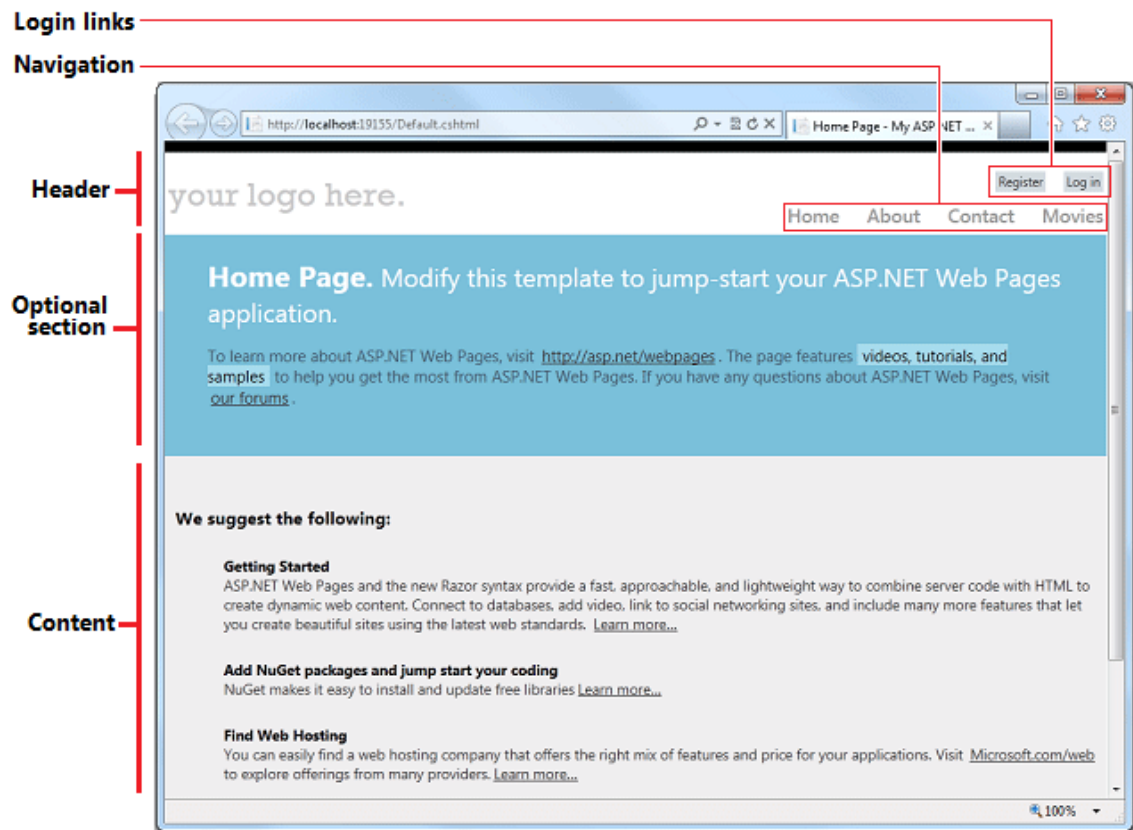
Coming Up Next

In the next tutorial, you'll learn how to publish your site to the Internet so everyone can see it.

Additional Resources

- [Complete Listing for Movie Page (Updated to Use a Layout Page)](#)
- [Complete Page Listing for Add Movie Page (Updated for Layout)](#)
- [Complete Page Listing for Delete Movie Page (Updated for Layout)](#)
- [Complete Page Listing for Edit Movie Page (Updated for Layout)](#)
- [Creating a Consistent Look](#) — An article that provides some more detail on working with layouts. It also describes how to pass a value to a layout page that shows or hides some of the content.
- [Nested Layout Pages with Razor](#) — Mike Brind blogs an example of how to nest layout pages. (Includes a download of the pages.)

Tutorial 9: Publishing a Site by Using WebMatrix

What you'll learn:

- How to publish your site to the Internet.
- What's coming up in the next tutorial set.

Publishing Your Site

Up to now, you've done all your work on a local computer, including testing your pages. To run your *.cshtml* pages, you've used the web server that's built into WebMatrix, namely IIS Express. But of course no one can see the site you've created except you. To let others work with your site, you have to publish it to the Internet.

Unless you have access to a public web server already, publishing means that you have to have an account with a *hosting provider*. A hosting provider is a company that owns publicly accessible web servers and that will rent you space for your site. Hosting plans run from a few dollars a month (or even free) for small sites to many hundreds of dollars a month for high-volume commercial websites.

> **Note** You might have access to a public web server via the internet service provider (ISP) that you use to get internet service at home. However, your hosting provider must support ASP.NET Web Pages. Many ISPs don't, but it's always worth checking.

In this tutorial, we'll give you an overview of how to publish. It's not practical to provide exact details for everything, because the process differs a bit for every hosting provider. But you'll get a good idea of how the process works.

Selecting a hosting provider

The first step is to find a hosting provider. You can look for one by searching the web or right from within WebMatrix.

In the WebMatrix ribbon, click the **Publish** button.
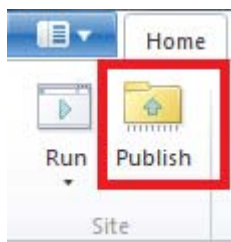
The **Publish Settings** dialog box is displayed.

Click the **Find web hosting** link.

You go to a page on the Microsoft site that lists hosting providers that support ASP.NET.
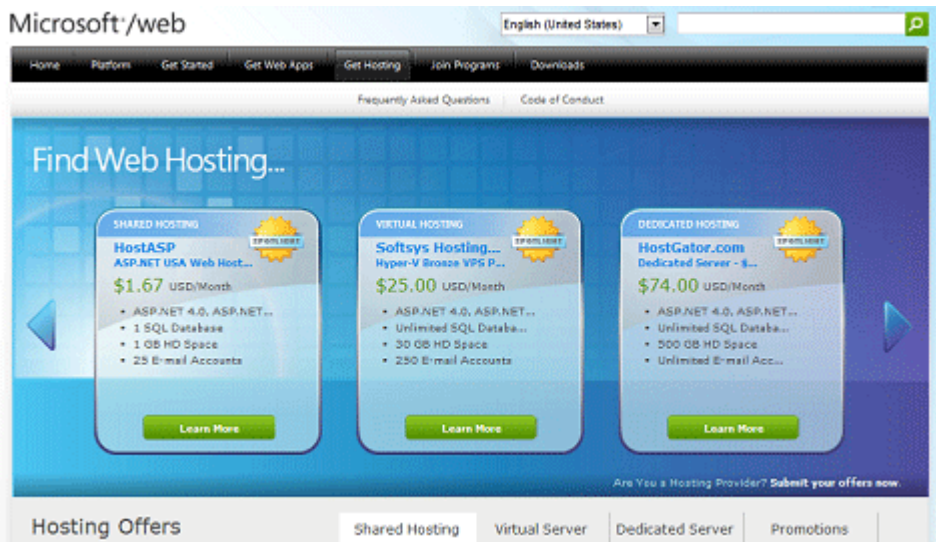
Obviously, it can be difficult to know now exactly what hosting features you might require over the long term. Here are a couple of things to consider:

- For purposes of the WebPagesMovies site, you don't have to have a separate add-on for SQL Server, which often costs extra. In your site, you're using SQL Server Compact Edition, which is self-contained. However, you might need SQL Server access for some future website work you do. If you think you might, make sure that you can add SQL Server capability later.
- Check whether the hosting provider supports the Web Deploy publishing protocol. You can publish by using FTP protocol, but it's more convenient to use Web Deploy.

Some sites offer a free trial period. A free trial is a good way to try publishing and hosting while you're still experimenting with WebMatrix and ASP.NET Web Pages.

Pick one that you like. For this tutorial, we selected DiscountASP.NET, because while we were creating the tutorial, that company had a promotion that let people host a site free for a few months.

> **Note** Our choice of a hosting provider for this tutorial shouldn't be interpreted as an endorsement of that company over any other. But we had to pick one for illustration, and DiscountASP.NET is one of the many companies that supports ASP.NET Web Pages and the Web Deploy protocol for publishing.

Typically, after you've signed up with the hosting provider, the company sends you an email that contains a user name and password, the URL of the web server, and so on. If the hosting company supports Web Deploy protocol, they might send you a file that contains publish settings, or let you download one. A publish settings file simplifies the process for you.

Publishing the site

When you've signed up and are ready to publish, click the **Publish** button in the WebMatrix ribbon. The **Publish Settings** dialog box is displayed.

If the hosting provider sent you a publish settings file, click the **Import publish settings** link and import the file. If you don't have a publish settings file, fill in the fields by using the values that the hosting company sent you in email. Here's what the **Publish Settings** dialog box might look like when you're done:

Click **Validate Connection**. If everything is ok, the dialog box reports **Connected successfully**, which means it can communicate with the hosting provider's server.

If there's a problem, WebMatrix does its best to tell you what the problem is:

Click **Save** to save your settings. WebMatrix offers to perform a test to make sure that it can communicate correctly with the hosting site:

Publish Compatibility

WebMatrix will upload a few files to
webmatrix2.discountasp.net to test compatibility with the site.
These files will be automatically deleted after the test.
Click Yes to proceed, or No to skip the test.

Note: WebMatrix will automatically adjust .NET settings on
the remote server if necessary.

Copy full details to clipboard

[Yes] [No]

Click **Yes**. WebMatrix uploads some sample files to the hosting provider. When the compatibility test is done, WebMatrix reports the results:

Publish Compatibility

Here are the results of checking your site's compatibility with **webmatrix2.discountasp.net**

| ✓ | ASP.NET version | Available |
| ✓ | Simple HTML page | Available |

Learn more about these tests

[Continue] [Cancel]

If you're ready to go, go ahead and click **Continue** to start the publish process for real. WebMatrix figures out what files are in your site and are already on the host server (right now, none) and gives you a preview of the publish process:

The list of files to publish includes the web pages that you've created like *Movies.cshtml*. The list also includes files for helpers that you've installed, the files to run SQL Server Compact Edition for your database, and so on. As a result, the initial publish process can be substantial.

Click **Continue**. WebMatrix copies your files to the hosting provider's server. When it's done, the results are reported in the status bar:

To see your live site, click the link in the status bar. Add *Movies* to the URL, and you'll see the *Movies.cshtml* file that you created:

Updating the live site: Republishing

Once you've published your site, there are two copies of it — the version on your computer and the version on the hosting provider's server. You'll probably want to continue developing the site (if nothing else, as part of the next tutorial set).When you do, you have to republish your site in order to copy changes from your computer to the hosting provider's server. The publish process in WebMatrix can determine what files have changed on your site and publish just those files.

To see how republishing works, open the *Movies.cshtml* site, make some small change, and then save the file. For example, change the title to `Movies - Updated`.

Click the **Publish** button in the ribbon. WebMatrix determines what's changed and shows you a preview of the files it will publish.

Publish Preview

	Name	Action	Date modified	Size	
☑ Changed Files (1)					Total: 2 KB
☑	Movies.cshtml	Update	April 12 10:04 PM	2 KB	

☐ Delete files on the remote server that are not on my computer.

Databases (1)

☐ WebPagesMovies.sdf   Copy as file

Publishing will overwrite any remote databases   ✕

[Continue]   [Cancel]

> **Important!**By default, WebMatrix publishes your database (*.sdf* file) only the first time you publish the site. Once your site is published and people are interacting with the website, the database on the live site typically has the site's real data. You have to be very careful not to overwrite the live database with the *.sdf* file that's on your computer, which usually contains only test data. That's why you see the warning **Publishing will overwrite any remote databases**, and why the check box for *WebPagesMovies.sdf* is cleared by default.

Click **Continue**. WebMatrix publishes the changed files and shows you a success message, like it did the first time you published.

Goto the live site (you can click the link in the success message if it's still showing) and verify that your change has been published.

> **Editing files remotely**
>
> As an alternative to changing your site and then republishing, you can edit remote files directly in WebMatrix. In this scenario, you open a file that's on the hosting provider's server, and WebMatrix downloads a copy of it for you to edit. Every time you save the file, WebMatrix sends the changes to the hosting site.
>
> Remote editing is an easy way to make changes to your live site. However, the changes you make this way aren't synchronized with the files in your local site. To synchronize the local files with the remote site, you can download the remote files. This process works much like publishing, except in reverse.
>
> We won't describe more about the remote-editing and remote-download facilities of WebMatrix here. They're quite useful if multiple people have to work on the same site on different computers. For more information, see Publish and Edit a Remote Site with WebMatrix 2.

Preview of the Next Tutorial Set

In this tutorial you've created a site that's functional and that illustrates much of the functionality of ASP.NET Web Pages. But there's plenty more to learn. The next tutorial set continues with your Movies site and teaches you the following:

- How to start a site from a template, which instantly gives you a professional layout plus built-in functionality for common tasks.
- More about how to work with forms — radio buttons, checkboxes, drop-down lists, etc.
- How to perform more sophisticated database searches.
- How to add social media to your site — for example, adding a Facebook "Like" button.
- How to begin adding client-side functionality to your site (by using jQuery), which gives the user a very responsive experience.
- How to add images to the site, including images uploaded by users.
- How to send email from your site.
- How to add login security to your site, so that some functions (like deleting a movie) are available only to users who've been authorized.

Additional Resources

- ASP.NET WebMatrix ASP.NET Web Pages forum, a great place to post questions and get answers.
- How to publish a web application using WebMatrix, an end-to-end tutorial about how to find a hosting provider and publish to the Internet.

This page intentionally left blank

Appendix: Code Listings

This appendix lists the finished versions of all the pages that you create in this tutorial set.

Complete Listing for TestRazor Page

```
@{
    // Working with numbers
    var a = 4;
    var b = 5;
    var theSum = a + b;

    // Working with characters (strings)
    var technology = "ASP.NET";
    var product ="Web Pages";

    // Working with objects
    var rightNow = DateTime.Now;
}

<!DOCTYPE html>
<html lang="en">
<head>
<title>Testing Razor Syntax</title>
<meta charset="utf-8" />
<style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    span.bright {color:red;}
</style>
</head>
<body>
<h1>Testing Razor Syntax</h1>
<form method="post">

<div>
<p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
<p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
<p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
</div>

<div>
<p>The technology is @technology, and the product is @product.</p>
<p>Together they are <span class="bright">@(technology + " " + product)</span></p>
```

```
</div>

<div>
<p>The current date and time is: @rightNow</p>
<p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
</div>

</form>
</body>
</html>
```

Complete Listing for TestRazorPart2 Page

```
@{
    var message = "This is the first time you've requested the page.";

    if(IsPost){
        message = "Now you've submitted the page.";
    }

    var showMessage = false;
    if(Request.QueryString["show"].AsBool() == true){
      showMessage = true;
    }
}

<!DOCTYPE html>
<html lang="en">
<head>
<title>Testing Razor Syntax - Part 2</title>
<meta charset="utf-8" />
<style>
      body {font-family:Verdana; margin-left:50px; margin-top:50px;}
      div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
</style>
</head>
<body>
<h1>Testing Razor Syntax - Part 2</h1>
<form method="post">
<div>
<!--<p>@message</p>-->
      @if(showMessage){
<p>@message</p>
```

```
        }
<p><input type="submit" value="Submit" /></p>
        @if(IsPost){
<p>You submitted the page at @DateTime.Now</p>
        }
</div>
</form>
</body>
</html>
```

Complete Listing for TwitterTest Page

```
@{
}
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title></title>
</head>
<body>
<div>
        @TwitterGoodies.Search("webmatrix")
</div>


<div>
        @TwitterGoodies.FollowButton("microsoft")
</div>
</body>
</html>
```

Complete Listing for Movies Page

```
@{
    var db = Database.Open("WebPagesMovies");
    var selectedData = db.Query("SELECT * FROM Movies");
    var grid = new WebGrid(source: selectedData, rowsPerPage: 3);
}


<!DOCTYPE html>


<html lang="en">
<head>
```

```
<meta charset="utf-8" />
<title>Movies</title>
<style type="text/css">
        .grid { margin: 4px; border-collapse: collapse; width: 600px; }
        .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
        .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
        .alt { background-color: #E8E8E8; color: #000; }
</style>
</head>
<body>
<h1>Movies</h1>
<div>
        @grid.GetHtml(
            tableStyle: "grid",
            headerStyle: "head",
            alternatingRowStyle: "alt",
            columns: grid.Columns(
                grid.Column("Title"),
                grid.Column("Genre"),
                grid.Column("Year")
            )
        )
</div>
</body>
</html>
```

Complete Listing for Movie Page (Updated with Search)

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
        searchTerm = "%" + Request["searchTitle"] + "%";
    }
```

```
    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre",
        rowsPerPage:3);
}

<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title>Movies</title>
<style type="text/css">
    .grid { margin: 4px; border-collapse: collapse; width: 600px; }
    .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
    .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
    .alt { background-color: #E8E8E8; color: #000; }
</style>
</head>
<body>
<h1>Movies</h1>
<form method="get">
<div>
<label for="searchGenre">Genre to look for:</label>
<input type="text" name="searchGenre"
            value="@Request.QueryString["searchGenre"]" />
<input type="Submit" value="Search Genre" /><br/>
        (Leave blank to list all movies.)<br/>
</div>

<div>
<label for="SearchTitle">Movie title contains the following:</label>
<input type="text" name="searchTitle"
            value="@Request.QueryString["searchTitle"]" />
<input type="Submit" value="Search Title" /><br/>
</div>
</form>

<div>
    @grid.GetHtml(
        tableStyle: "grid",
        headerStyle: "head",
        alternatingRowStyle: "alt",
        columns: grid.Columns(
            grid.Column("Title"),
```

```
            grid.Column("Genre"),
            grid.Column("Year")
          )
        )
</div>
</body>
</html>
```

Complete Listing for AddMovie Page

```
@{

    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost && Validation.IsValid()){
        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];

        var db = Database.Open("WebPagesMovies");
        var insertCommand = "INSERT INTO Movies (Title, Genre, Year) " +
            " Values(@0, @1, @2)";
        db.Execute(insertCommand, title, genre, year);
        Response.Redirect("~/Movies");
    }
}

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Add a Movie</title>
<style type="text/css">
    .field-validation-error {
      font-weight:bold;
      color:red;
      background-color:yellow;
```

```
      }
    .validation-summary-errors{
      border:2px dashed red;
      color:red;
      background-color:yellow;
      font-weight:bold;
      margin:12px;
    }
</style>
</head>
<body>
<h1>Add a Movie</h1>
  @Html.ValidationSummary()
<form method="post">
<fieldset>
<legend>Movie Information</legend>
<p><label for="title">Title:</label>
<input type="text" name="title" value="@Request.Form["title"]" />
          @Html.ValidationMessage("title")

<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@Request.Form["genre"]" />
        @Html.ValidationMessage("genre")

<p><label for="year">Year:</label>
<input type="text" name="year" value="@Request.Form["year"]" />
          @Html.ValidationMessage("year")

<p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
</fieldset>
</form>
</body>
</html>
```

Complete Listing for Movie Page (Updated with Edit Links)

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
```

```
            searchTerm = Request.QueryString["searchGenre"];
    }

    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
        searchTerm = "%" + Request.QueryString["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre",
        rowsPerPage:3);
}

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>Movies</title>
<style type="text/css">
        .grid { margin: 4px; border-collapse: collapse; width: 600px; }
        .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
        .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
        .alt { background-color: #E8E8E8; color: #000; }
</style>
</head>
<body>
<h1>Movies</h1>
<form method="get">
<div>
<label for="searchGenre">Genre to look for:</label>
<input type="text" name="searchGenre"
value="@Request.QueryString["searchGenre"]" />
<input type="Submit" value="Search Genre" /><br/>
                (Leave blank to list all movies.)<br/>
</div>

<div>
<label for="SearchTitle">Movie title contains the
following:</label>
<input type="text" name="searchTitle"
value="@Request.QueryString["searchTitle"]" />
<input type="Submit" value="Search Title" /><br/>
</div>
</form>
```

```
<div>
            @grid.GetHtml(
                tableStyle: "grid",
                headerStyle: "head",
                alternatingRowStyle: "alt",
                columns: grid.Columns(
                    grid.Column(format:
@<a href="~/EditMovie?id=@item.ID">Edit</a>),
                    grid.Column("Title"),
                    grid.Column("Genre"),
                    grid.Column("Year")
                )
            )
</div>
<p>
<a href="~/AddMovie">Add a movie</a>
</p>
</body>
</html>
```

Complete Page Listing for Edit Movie Page

```
@{
    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        //if(!Request.QueryString["ID"].IsEmpty()){
        if(!Request.QueryString["ID"].IsEmpty() &&
                Request.QueryString["ID"].IsInt()) {
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);

            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
```

```
                else{
                    Validation.AddFormError("No movie was selected.");
                    // Use the following line instead for versions of ASP.NET
                    // Web Pages 2 earlier than the RC release.
                    //ModelState.AddFormError("No movie was selected.");
                }
            }
            else{
                Validation.AddFormError("No movie was selected.");
                // Use the following line instead for versions of ASP.NET
                // Web Pages 2 earlierthan the RC release.
                //ModelState.AddFormError("No movie was selected.");
            }
        }

        if(IsPost){
            Validation.RequireField("title", "You must enter a title");
            Validation.RequireField("genre", "Genre is required");
            Validation.RequireField("year", "You haven't entered a year");
            Validation.RequireField("movieid", "No movie ID was submitted!");

            title = Request.Form["title"];
            genre = Request.Form["genre"];
            year = Request.Form["year"];
            movieId = Request.Form["movieId"];

            if(Validation.IsValid()){
                var db = Database.Open("WebPagesMovies");
                var updateCommand = "UPDATE Movies " +
                    " SET Title=@0, Genre=@1, Year=@2 WHERE Id=@3";
                db.Execute(updateCommand, title, genre, year, movieId);
                Response.Redirect("~/Movies");
            }
        }
}

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Edit a Movie</title>
<style>
    .validation-summary-errors{
        border:2px dashed red;
```

```
            color:red;
            font-weight:bold;
            margin:12px;
        }
</style>
</head>
</head>
<body>
<h1>Edit a Movie</h1>
        @Html.ValidationSummary()
<form method="post">
<fieldset>
<legend>Movie Information</legend>

<p><label for="title">Title:</label>
<input type="text" name="title" value="@title" /></p>

<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@genre" /></p>

<p><label for="year">Year:</label>
<input type="text" name="year" value="@year" /></p>

<input type="hidden" name="movieid" value="@movieId" />

<p><input type="submit" name="buttonSubmit" value="Submit Changes" /></p>
</fieldset>
</form>
<p><a href="~/Movies">Return to movie listing</a></p>
</body>
</html>
```

Complete Listing for Movie Page (Updated with Delete Links)

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }
```

```
    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
      selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
      searchTerm = "%" + Request.QueryString["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre",
        rowsPerPage:3);
}
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>Movies</title>
<style type="text/css">
        .grid { margin: 4px; border-collapse: collapse; width: 600px; }
        .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
        .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
        .alt { background-color: #E8E8E8; color: #000; }
</style>
</head>
<body>
<h1>Movies</h1>
<form method="get">
<div>
<label for="searchGenre">Genre to look for:</label>
<input type="text" name="searchGenre"
               value="@Request.QueryString["searchGenre"]" />
<input type="Submit" value="Search Genre" /><br/>
          (Leave blank to list all movies.)<br/>
</div>

<div>
<label for="SearchTitle">Movie title contains the following:</label>
<input type="text" name="searchTitle"
                value="@Request.QueryString["searchTitle"]" />
<input type="Submit" value="Search Title" /><br/>
</div>

</form>
<div>
          @grid.GetHtml(
            tableStyle: "grid",
```

```
            headerStyle: "head",
            alternatingRowStyle: "alt",
            columns: grid.Columns(
                grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
                grid.Column("Title"),
                grid.Column("Genre"),
                grid.Column("Year"),
                grid.Column(format:
                    @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
            )
        )
</div>
<p>
<a href="~/AddMovie">Add a movie</a>
</p>
</body>
</html>
```

Complete Listing for DeleteMovie Page

```
@{
    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty() &&
                Request.QueryString["ID"].IsInt()){
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);
            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was found for that ID.");
                // If you are using a version of ASP.NET Web Pages 2 that's
                // earlier than the RC release, comment out the preceding
                // statement and uncomment the following one.
```

```
                    //ModelState.AddFormError("No movie was found for that ID.");
            }
        }
        else{
            Validation.AddFormError("No movie was found for that ID.");
            // If you are using a version of ASP.NET Web Pages 2 that's
            // earlier than the RC release, comment out the preceding
            // statement and uncomment the following one.
            //ModelState.AddFormError("No movie was found for that ID.");
        }
    }

    if(IsPost && !Request["buttonDelete"].IsEmpty()){
        movieId = Request.Form["movieId"];
        var db = Database.Open("WebPagesMovies");
        var deleteCommand = "DELETE FROM Movies WHERE ID = @0";
        db.Execute(deleteCommand, movieId);
        Response.Redirect("~/Movies");
    }
}
<html>
<head>
<title>Delete a Movie</title>
</head>
<body>
<h1>Delete a Movie</h1>
        @Html.ValidationSummary()
<p><a href="~/Movies">Return to movie listing</a></p>

<form method="post">
<fieldset>
<legend>Movie Information</legend>

<p><span>Title:</span>
        <span>@title</span></p>

<p><span>Genre:</span>
        <span>@genre</span></p>

<p><span>Year:</span>
<span>@year</span></p>

<input type="hidden" name="movieid" value="@movieId" />
<p><input type="submit" name="buttonDelete" value="Delete Movie" /></p>
```

```
</fieldset>
<p><a href="~/Movies">Return to movie listing</a></p>
</form>
</body>
</html>
```

Complete Listing for Movie Page (Updated to Use a Layout Page)

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "List Movies";

    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }


    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
        searchTerm = "%" + Request.QueryString["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre",
        rowsPerPage:3);
}

<h2>List Movies</h2>
<form method="get">
<div>
<label for="searchGenre">Genre to look for:</label>
<input type="text" name="searchGenre"
            value="@Request.QueryString["searchGenre"]" />
<input type="Submit" value="Search Genre" /><br/>
        (Leave blank to list all movies.)<br/>
</div>

<div>
```

```
<label for="SearchTitle">Movie title contains the following:</label>
<input type="text" name="searchTitle"
            value="@Request.QueryString["searchTitle"]" />
<input type="Submit" value="Search Title" /><br/>
</div>
</form>

<div>
    @grid.GetHtml(
        tableStyle: "grid",
        headerStyle: "head",
        alternatingRowStyle: "alt",
        columns: grid.Columns(
          grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
          grid.Column("Title"),
          grid.Column("Genre"),
          grid.Column("Year"),
          grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
        )
    )
</div>
<p><a href="~/AddMovie">Add a movie</a></p>
```

Complete Page Listing for Add Movie Page (Updated for Layout)

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "Add a Movie";

    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost){
        if(Validation.IsValid()){
            title = Request.Form["title"];
            genre = Request.Form["genre"];
            year = Request.Form["year"];
```

```
            var db = Database.Open("WebPagesMovies");
            var insertCommand = "INSERT INTO Movies (Title, Genre, Year) " +
                " VALUES(@0, @1, @2)";
            db.Execute(insertCommand, title, genre, year);
            Response.Redirect("~/Movies");
        }
    }
}


<h2>Add a Movie</h2>
@Html.ValidationSummary()
<form method="post">
<fieldset>
<legend>Movie Information</legend>
<p><label for="title">Title:</label>
<input type="text" name="title" value="@Request.Form["title"]" />
        @Html.ValidationMessage("title")


<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@Request.Form["genre"]" />
        @Html.ValidationMessage("genre")


<p><label for="year">Year:</label>
<input type="text" name="year" value="@Request.Form["year"]" />
        @Html.ValidationMessage("year")


<p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
</fieldset>
</form>
```

Complete Page Listing for Delete Movie Page (Updated for Layout)

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "Delete a Movie";

    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty() &&
```

```
                    Request.QueryString["ID"].AsInt() > 0){
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);
            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was found for that ID.");
                // If you are using a version of ASP.NET Web Pages 2 that's
                // earlier than the RC release, comment out the preceding
                // statement and uncomment the following one.
                //ModelState.AddFormError("No movie was found for that ID.");
            }
        }
        else{
            Validation.AddFormError("No movie was found for that ID.");
            // If you are using a version of ASP.NET Web Pages 2 that's
            // earlier than the RC release, comment out the preceding
            // statement and uncomment the following one.
            //ModelState.AddFormError("No movie was found for that ID.");
        }
    }

    if(IsPost && !Request["buttonDelete"].IsEmpty()){
        movieId = Request.Form["movieId"];
        var db = Database.Open("WebPagesMovies");
        var deleteCommand = "DELETE FROM Movies WHERE ID = @0";
        db.Execute(deleteCommand, movieId);
        Response.Redirect("~/Movies");
    }
}
<h2>Delete a Movie</h2>
@Html.ValidationSummary()
<p><a href="~/Movies">Return to movie listing</a></p>

<form method="post">
<fieldset>
<legend>Movie Information</legend>

<p><span>Title:</span>
```

```
<span>@title</span></p>


<p><span>Genre:</span>
<span>@genre</span></p>


<p><span>Year:</span>
<span>@year</span></p>


<input type="hidden" name="movieid" value="@movieId" />
<p><input type="submit" name="buttonDelete" value="Delete Movie" /></p>
</fieldset>
</form>
```

Complete Page Listing for Edit Movie Page (Updated for Layout)

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "Edit a Movie";


    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";


    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty() &&
Request.QueryString["ID"].IsInt()) {
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);


            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was selected.");
                // If you are using a version of ASP.NET Web Pages 2 that's
                // earlier than the RC release, comment out the preceding
                // statement and uncomment the following one.
                //ModelState.AddFormError("No movie was selected.");
```

```
                }
            }
            else{
                Validation.AddFormError("No movie was selected.");
                // If you are using a version of ASP.NET Web Pages 2 that's
                // earlier than the RC release, comment out the preceding
                // statement and uncomment the following one.
                //ModelState.AddFormError("No movie was selected.");
            }
        }

        if(IsPost){
            Validation.RequireField("title", "You must enter a title");
            Validation.RequireField("genre", "Genre is required");
            Validation.RequireField("year", "You haven't entered a year");
            Validation.RequireField("movieid", "No movie ID was submitted!");

            title = Request.Form["title"];
            genre = Request.Form["genre"];
            year = Request.Form["year"];
            movieId = Request.Form["movieId"];

            if(Validation.IsValid()){
                var db = Database.Open("WebPagesMovies");
                var updateCommand = "UPDATE Movies " +
                     " SET Title=@0, Genre=@1, Year=@2 WHERE Id=@3";
                db.Execute(updateCommand, title, genre, year, movieId);
                Response.Redirect("~/Movies");
            }
        }
}

<h2>Edit a Movie</h2>
@Html.ValidationSummary()
<form method="post">
<fieldset>
<legend>Movie Information</legend>

<p><label for="title">Title:</label>
<input type="text" name="title" value="@title" /></p>

<p><label for="genre">Genre:</label>
<input type="text" name="genre" value="@genre" /></p>
```

```
<p><label for="year">Year:</label>
<input type="text" name="year" value="@year" /></p>


<input type="hidden" name="movieid" value="@movieId" />


<p><input type="submit" name="buttonSubmit" value="Submit Changes" /></p>
</fieldset>
</form>
<p><a href="~/Movies">Return to movie listing</a></p>
```