

C# .NET 4: Web Development and User Interface Design Using .NET

Lesson 1: **Introduction and Web Development Overview**

[Understanding the Learning Sandbox Environment](#)

[Code Snippets](#)

[The OST Plug-In](#)

[Web and Internet Overview](#)

[Hello World - A Simple Website](#)

[Hello World - Adding Web Controls](#)

[Debugging](#)

[Quiz 1 Project 1](#)

Lesson 2: **Introduction to Web Forms**

[Introduction to ASP.NET Web Forms](#)

[Web Calculator - Understanding the ASP.NET Architecture](#)

[Quiz 1 Project 1](#)

Lesson 3: **Web Forms**

[Web Forms](#)

[Creating the Project](#)

[Examining the Auto-generated Website](#)

[Creating A Site](#)

[Quiz 1 Project 1](#)

Lesson 4: **Security**

[Authentication and Authorization](#)

[Membership and Roles](#)

[Authorization](#)

[Quiz 1 Project 1](#)

Lesson 5: **Introduction to MVC and Razor**

[Defining MVC](#)

[Model](#)

[Controller](#)

[View](#)

[Model/View/Controller](#)

[MVC vs. Smart UI Design Pattern](#)

[Creating a New Project](#)

[Project Folders](#)

[The Controller](#)

[The View](#)

[Controller Actions](#)

[Introduction to Razor](#)

[URLs and Routing](#)

[Basics of Routing](#)

[Routing Constraints](#)

[Quiz 1 Project 1](#)

Lesson 6: **MVC**

[The Model](#)

[The Entity Framework](#)

[MVC and the Entity Framework](#)

[Models and Entities](#)

[Filters](#)

[Quiz 1 Project 1](#)

Lesson 7: **Unit Testing**

[Testing with Visual Studio](#)

[Creating the Project](#)

[Software Testing](#)

[Unit Testing](#)

[Unit and Integrated Testing](#)

[MVC Unit Testing](#)

[Additional Testing Templates](#)

[Final Thoughts](#)

[Quiz 1 Project 1](#)

Lesson 8: **Client Solutions: JavaScript, jQuery, Ajax, XML, and JSON**

[Client-Side and Server-Side Development](#)

[JavaScript](#)

[jQuery](#)

[JSON](#)

[XML](#)

[Ajax](#)

[Partial Views and Ajax](#)

[Quiz 1 Project 1](#)

Lesson 9: **ASP.NET and Databases**

[Adding a Database](#)

[Creating a Menu Action](#)

[Creating, Editing, Deleting, Searching](#)

[Quiz 1 Project 1](#)

Lesson 10: **Object Relational Mapping: Entity Framework and Data Abstraction**

[Entity Framework](#)

[Data Abstraction](#)

[Repository](#)

[Unit of Work](#)

[Quiz 1 Quiz 2](#)

Lesson 11: **Object Relational Mapping: CRUD and Automatic Code Generation, LINQ, and Entity Data Model**

[CRUD and Automatic Code Generation](#)

[Repository with CRUD](#)

[Unit of Work with CRUD](#)

[LINQ to Entities](#)

[Entity Data Model \(EDM\)](#)

[Quiz 1 Project 1](#)

Lesson 12: **Interfaces and Extensions**

[Interfaces and MVC](#)

[IQueryable Interface](#)

[Extension Methods](#)

[Filtering Using Extension Methods](#)

[Quiz 1 Project 1](#)

Lesson 13: **[Web Services](#)**

[What is a Web Service?](#)

[Our First Web Service](#)

[Web Services: SOAP and REST](#)

[Adding REST Methods](#)

[Accessing Public Web Services](#)

[Web Service Legacy and Windows Communication Foundation \(WCF\)](#)

[Quiz 1 Project 1](#)

Lesson 14: **[Final Project](#)**

[Project Functional Specifications](#)

[Planning the Project](#)

[Use Cases](#)

[UI Design](#)

[Data Modeling: Entity Data Model](#)

[Data Modeling: Database Tables](#)

[Software Development and Unit Test Development](#)

[Testing and Completion](#)

[Project 1](#)

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Introduction and Web Development Overview

Welcome to the O'Reilly School of Technology's C# .NET 4: Web Development and User Interface Design Using .NET course!

Course Objectives

When you complete this course, you will be able to:

- use web development terms and architecture.
- create dynamic websites using ASP.NET Web Forms.
- create dynamic websites using the ASP.NET MVC (Model-View-Controller) framework and the Razor programming syntax.
- employ JavaScript, jQuery, AJAX, JSON and XML in web applications.
- use ASP.NET security for authentication and authorization with web applications.
- connect databases and websites.
- use the .NET Entity Framework with objects and LINQ.
- develop multi-threaded code.
- work with and create web services.
- use the Silverlight framework.
- create user interfaces using .NET WPF (Windows Presentation Foundation) and XAML.

Lesson Objectives

When you complete this lesson, you will be able to:

- [use the Learning Sandbox Environment](#).
- [access an Overview of the Web and Internet](#).
- [create A Simple Website](#).
- [add Web Controls to Your Website](#).

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you

blew through all of the coursework too quickly.

- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add *looks like this*.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type *look like this*.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:




Gray "Observe" boxes like this contain **information** (usually code specifics) for you to *observe*.

The paragraph(s) that follow may provide additional details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

When you see the  icon, save your work. You can save whenever you like; we want you to get into the habit of saving your projects frequently. In fact, whenever you pause to think, click the  icon on the toolbar (or press **Ctrl+S**). When you see the  icon, click that icon on the toolbar to run the currently open program.

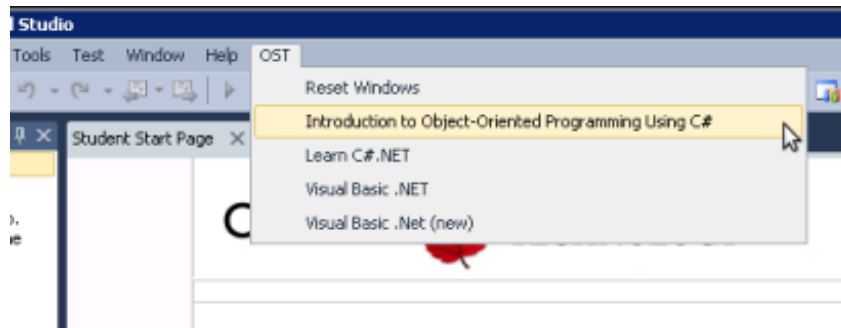
Understanding the Learning Sandbox Environment

Code Snippets

Whenever we present snippets of code, you'll want to create a sample project where you can enter that code. We'll prompt you to create test project(s) for each lesson. As we progress through those lessons, you'll create methods and call them from the Form constructor under the `InitializeComponent()` line of code. Eventually you'll create more topic-specific methods, and comment out the calls to the methods that you're not using.

The OST Plug-In

We've added a new menu item to the Visual Studio system that we think you'll like. Now you can use the **OST** menu to get to your syllabus for this course at any time. Your menu may display other courses you've enrolled in as well.



Tip

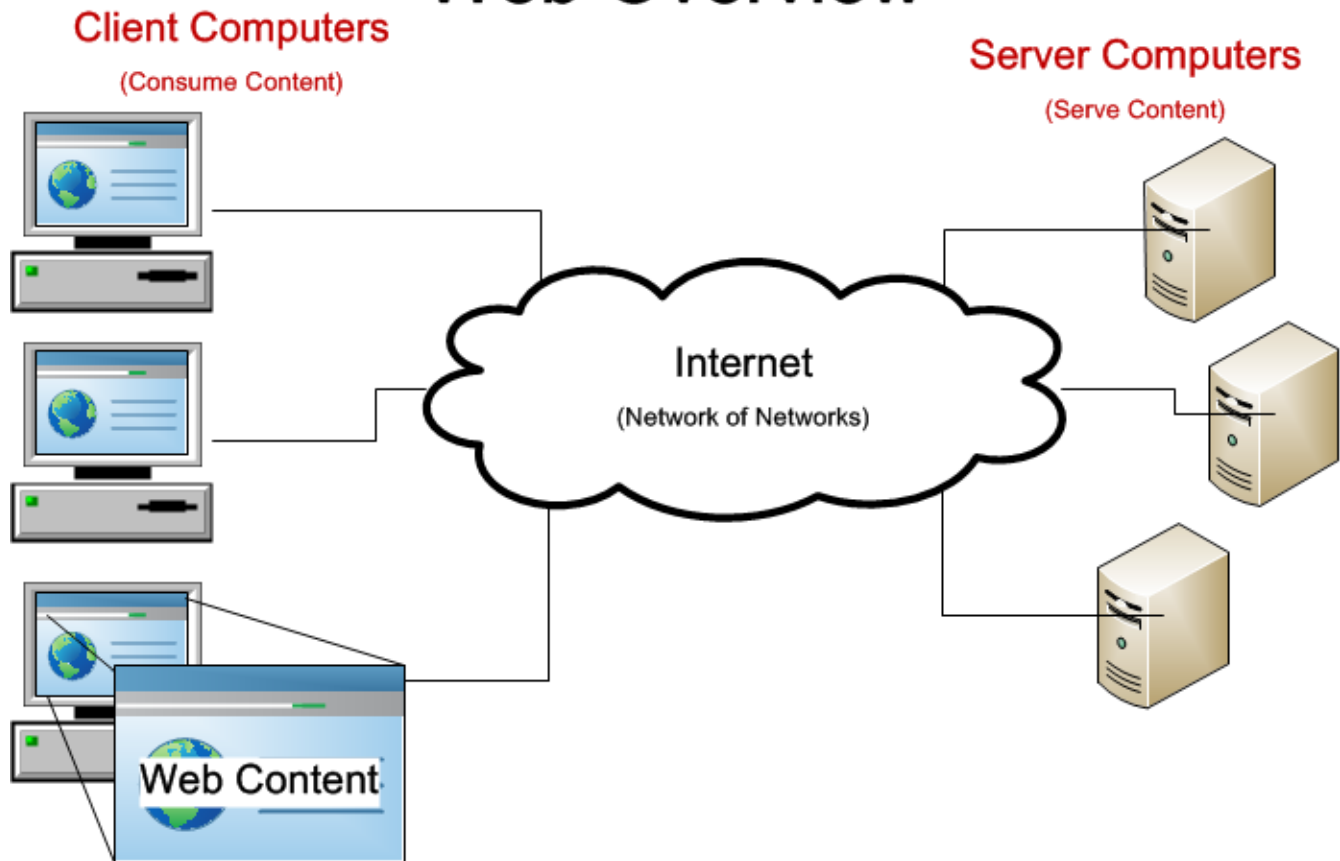
If your Visual Studio menus start to get confusing, you can always get back to the default view by selecting this course from the OST menu.

Web and Internet Overview

You'll be learning how to create web-based applications, so it's important that you understand the web.

The web (or World Wide Web, WWW, or W3) is the ever-expanding base of hypertext documents accessible via the internet, capable of delivering a wealth of information via numerous formats such as text, images, video, and so on. The internet is a network of communication networks; it allows access to web content, as well as a host of other possible resources. Frequently, the terms web and internet are used interchangeably, but their definitions refer to two distinct concepts. The image below portrays a typical overview of the web and internet.

Web Overview



We will be learning how to create web content using a number of .NET technologies. From the image above, web content is retrieved, or served, from server computers to client computers (consumers) via the Internet. Often, cloud-like images are used to portray the Internet because of the numerous networks and servers that serve a seemingly unending quantity of web content from everywhere imaginable. Web content is typically displayed using a web browser (such as Firefox, Internet Explorer, Chrome, Safari).

Here's a brief overview of how content is delivered to your web browser:

1. A web document address (or URL) is entered in the web browser.
2. The URL is resolved to a computer server (or web server) accessible through the internet.
3. A request is made through the internet for the web document from the web server.
4. The web server generates the web document and returns the web document content to the requesting computer (or client computer) through the internet.
5. The web browser on the client computer receives and renders the web document content.

Hello World - A Simple Website

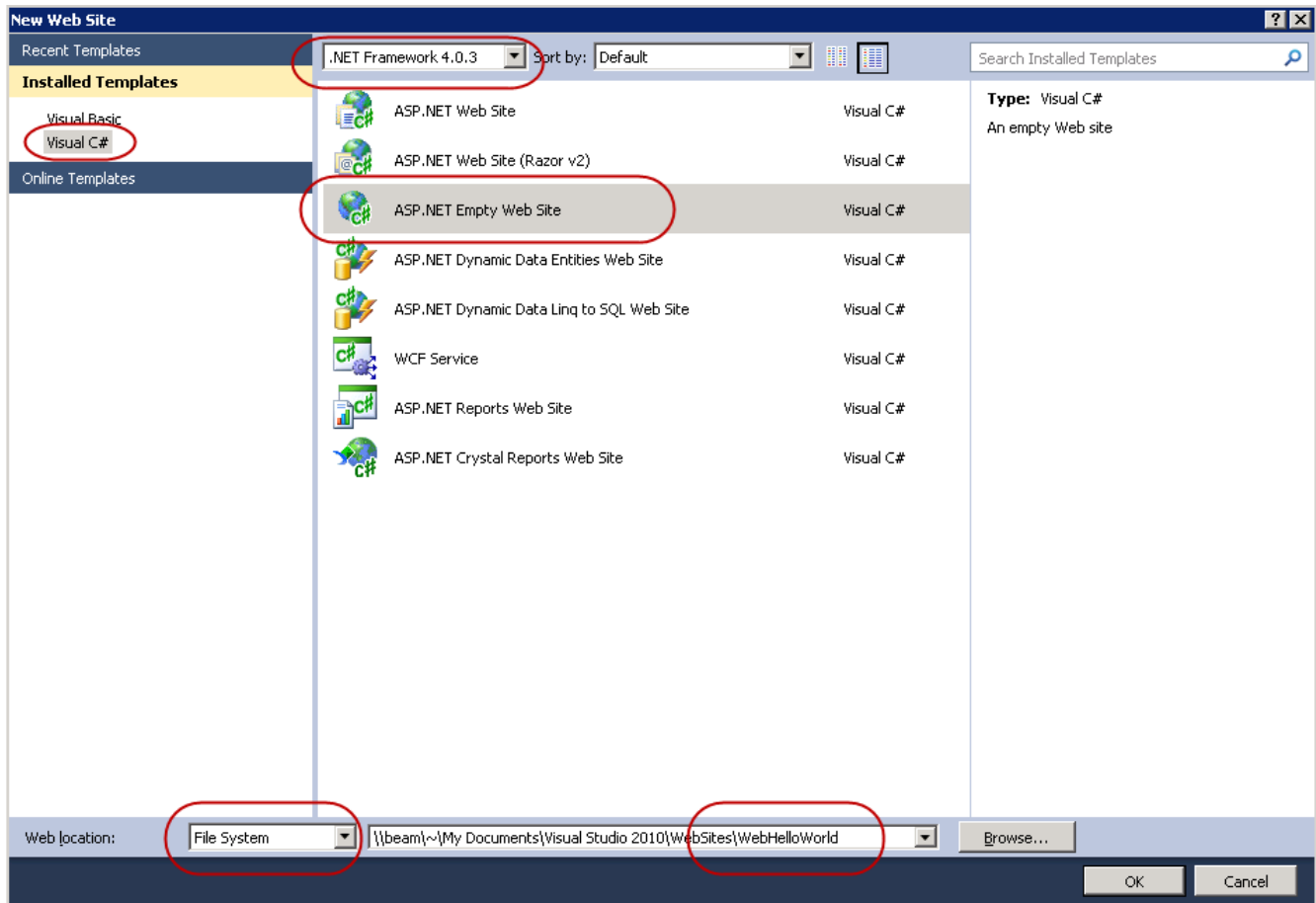
You can create a basic website fairly quickly. Let's create one that displays the static content, "Hello, World!"

Note

Microsoft Visual Studio includes a number of techniques for creating websites, such as ASP, Web Forms, and MVC. We'll start out learning Web Forms, then move on to more recent technologies. Also, Visual Studio supports two different types of web projects: web application projects and website projects. Initially, we'll use website projects, and then move on to web application projects. You can compare these project types, in this MSDN article: [Web Application Projects versus Web Site Projects in Visual Studio](#).

Create a test project using **File | New | Web Site....** In the New Web Site dialog box, select **Visual C#** under Installed Templates, select **.NET Framework 4.0.3** in the middle top dropdown, and select the **ASP.NET Empty Web Site** template. Select **File System** from the web location dropdown box. Modify the folder location from My Documents\Visual Studio 2010\WebSites\Website1 to My Documents\Visual Studio

2010\WebSites\WebHelloWorld, and click **OK**.



Because we selected an Empty Web Site, Visual Studio generated no web content. In the Solution Explorer, you'll see a single entry for a web.config file. We'll discuss this file in some detail in a later lesson. For now, let's add a web page for our website.

Before you do move on, note that the new document you create will replace this lesson text in the main Visual Studio window. To see the lesson content again, you'll need to right-click the tab with the filename (**Default.aspx**) and select **New Horizontal Tab Group**.

To add a website document, right-click in the Solution Explorer, select **Add New Item...**, select **Web Form**, make sure the Name is **Default.aspx**, and click **Add**.

A new web document (sometimes referred to as a "web page") is created and added to the website project, as shown:





If you're familiar with HTML syntax used with web documents, you may notice a number of HTML elements, but other elements are unique to .NET. We'll explore the .NET syntax of a Web Form web document in the next lesson, but for now, we'll just add HTML text that will display "Hello, World!" Modify **Default.aspx**. Modify your code as shown:

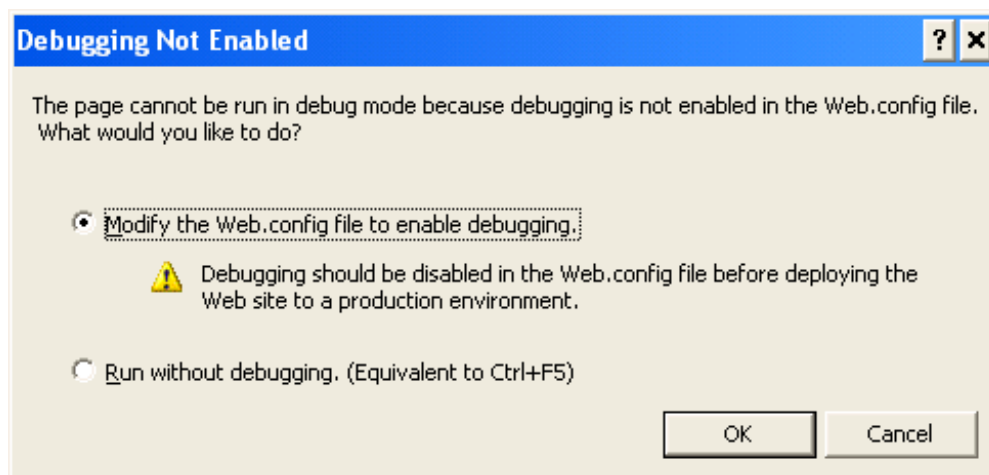
CODE TO TYPE:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head runat="server">
    <title></title>
  </head>
  <body>
    <form id="form1" runat="server">
      <div>
        <b>Hello, World!</b>
      </div>
    </form>
  </body>
</html>
```

 and  to run the program. A dialog box may prompt you to enable debugging for this project:



Select **Modify the Web.config file to enable debugging**, and click **OK**. A browser (Internet Explorer) window appears, displaying the Default.aspx webpage, including the "Hello, World!" text.

Before closing the browser, minimize it and look at Visual Studio. Visual Studio is in Debug mode, and the debug menu items and buttons are enabled. Our website code is "running" in a browser, rather than in an application. To exit debug mode and return to editing our project, we need to either close the browser or use Visual Studio to stop debugging. Close the browser by maximizing it and then clicking the **X** at the top right of the browser window.

Tip Go ahead and run the website project, and minimize and restore the Visual Studio and browser windows to practice switching between these two applications.

Congratulations, you've created your first website project!

Tip For this course, you need to have a working knowledge of HTML. If you don't, consider taking the OST HTML courses—for more information, [contact us](#). We'll explain some HTML concepts as they come up in the course, but ultimately you'll need to have a pretty good understanding of HTML. Web developers produce software that generates web content, and since web content is expressed in HTML, a working knowledge of HTML is essential.

Hello World - Adding Web Controls

Let's modify our, "Hello, World!" website to include web controls, controls you can add to your website. We'll also

learn how to toggle the Code Editor views, and get a quick introduction to the concept of a code-behind file. Let's go!

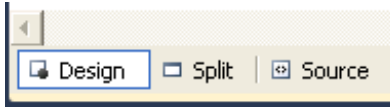
Note

If you closed the WebHelloWorld project, or closed Studio completely, you'll need to open the website project. When opening a website project, rather than use Open Project in Studio, use Open Web Site.

Note

As the functionality of a website grows and interaction with the user increases, the website is often referred to as a *web application*.

Open and examine **Default.aspx** in the Code Editor. Note the controls along the bottom left of the Code Editor Window: Design, Split, and Source:

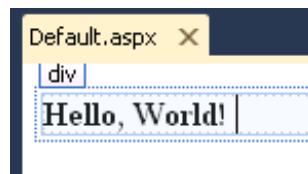


When we created the WebHelloWorld website project and added a web document, the code editor opened in Source view so we could edit the contents of the new web document. The web document source is not what appears when we run the website project. Instead, the web document source is processed by the web server, and the web server generates the HTML content that web browsers understand. When creating web documents, we want to be able to see what the rendered HTML content will be as we go. In the Code Editor, we can switch to the Design view to see what the content should look like, or we can use the Split view option to see both the web document source and rendered content.

Note

The rendered web content should match what you see in your web browser, but not all browsers render HTML content exactly the same way. Professional web developers test their website applications with multiple browsers to ensure a consistent rendering of their application code. Rendering the HTML in the Code Editor may take a while.

Switch to the Code Editor Design view by clicking the **Design** tab at the bottom of the Code Editor window. Now instead of document source code, you see the text, "Hello, World!":



You may prefer to add or edit web content to your web documents using the Design view, but often you may need to switch to the Source view to make sure that your new content is exactly what you want.

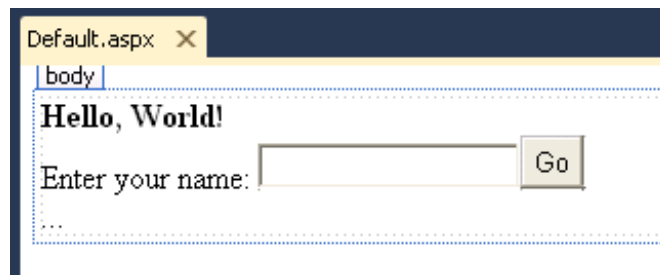
We're going to add a text box to our website to prompt for a name and a button. When you enter a name and click on the button, a greeting with the name on the web page using a label will be displayed. To add controls to your web document, use the controls on the Toolbox under the Standard category.



Tip

If you want to be able to add HTML controls using drag and drop, you can find the HTML non-ASP controls under the HTML section of the Toolbox.

In the Code Editor, click after the Hello, World! text, and press **Enter** to go to the next line. Type **Enter your name:**. Select the Toolbox window, and expand the Standard category. Click and drag a **TextBox** control, and drop it in after the "Enter your name:" text. Click and drag a **Button** control and drop it after the TextBox control. Press **Enter** again to go to the next line. Click and drag a **Label** control onto the new line.

As we've seen with other windows' controls, we can edit the control properties. Click on the TextBox control, and change the ID property to **nameTextBox**. Click on the Button control and change the ID property to **goButton**, and the **Text** property to **Go**. Click the Label control and change the ID property to **resultLabel**, and the Text property to Your webpage will look like this:



 and  to run the web application. Once the browser window appears, enter a name, and click the **Go** button. Did anything happen?

As with other .NET programs, we need to program the functionality of the Button.

Double-click the **Go** Button control. A new Code Editor window opens that we use to edit the **Default.aspx.cs** file. Notice that the name of the file is almost identical to the web document, except that the filename includes a **.cs** suffix. **.cs** stands for C#, and is called the "code-behind file." Before we edit, let's talk about the aspx and aspx.cs files.

We initially edited the aspx source file. An aspx, or the ASPX acronym, means an Active Server Page Extended file. An ASPX source file contains HTML and ASP.NET script code used to create web content. Each ASPX file has a code-behind file that contains code in a .NET language (such as C#, VB, and so on) and responds to and interacts with the ASPX file. The suffix used with the code-behind file should match the .NET language, so aspx.cs for C#, and aspx.vb for VB (Visual Basic). We'll continue to learn more about the architecture of website development using .NET. The aspx.cs file will become very familiar to you because it contains C# code.

Okay, let's get back to adding the necessary functionality for our web application.

By double-clicking on the Button control, we've added an event handler for this Button. Whenever this Button is clicked, the event handler code in the code-behind file will be called.

Modify the Default.aspx.cs file as shown:

CODE TO TYPE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void goButton_Click(object sender, EventArgs e)
    {
        resultLabel.Text = "Hi, " + nameTextBox.Text;
    }
}
```

Let's discuss this code.

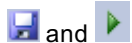
OBSERVE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void goButton_Click(object sender, EventArgs e)
    {
        resultLabel.Text = "Hi, " + nameTextBox.Text;
    }
}
```

The Default.aspx.cs file follows the pattern we've seen with other C# source code files; it has namespaces to be used, and a class definition for the webpage that inherits from **System.Web.UI.Page**. This webpage class includes two event handlers: a **Page_Load** event, provided when the webpage was created and called whenever a webpage is loaded; and the **goButton_Click** event that was added when we double-clicked the Button control.

In the **goButton_Click** event handler, we can use the names of controls we've added to the webpage, such as **resultLabel**, and Intellisense which will help us to select the **Text** property. We can use the **nameTextBox.Text** property to retrieve the entered name. Using these controls, we can respond to the Button click and produce an appropriate greeting.



and to run the web application. When the browser window appears, enter a name and click **Go**. This time, you see the greeting!

Debugging

Before you leave this first lesson, re-run the lesson project code using debugging tools. Just as with C# applications, we can place a debugging breakpoint by clicking in the shaded area to the left of a line of code, or by right-clicking on any source code line and selecting **Breakpoint | Insert Breakpoint**. Add a breakpoint to the line of code that sets the Label control Text property. When you run the website project and click on the Button control, the IDE will break on the breakpoint. Highlight or hover the mouse over different variables or properties to observe their values. Debugging in an ASP.NET website project works just as you've seen with a C# application project.

To close this project, select **File | Close Solution**. To open a project later, select **File | Open | Web Site...**, select **File System**, and navigate to **Computer**, your **\\beam\\winusers** folder, **My Documents\\Visual Studio 2010\\WebSites**, select the website you want to open and click **Open**.

Before you move on to the next lesson, make sure to do any projects and quizzes for this lesson.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Introduction to Web Forms

Lesson Objectives

In this lesson you will:

- use ASP.NET Web Forms and Web Controls
- create a web calculator project using ASP.NET

The latest versions of Microsoft ASP.NET, .NET, and Visual Studio, include numerous enhancements to support Rapid Application Development (RAD). We will cover these newer technologies, but first we want to cover ASP.NET Web Forms, a technology still used by many web developers, and often mixed with the newer techniques. In this lesson, we'll cover the basics of Web Forms and Web Controls, including exploring the mechanisms behind Web Forms. Let's get started.

Introduction to ASP.NET Web Forms

Web Calculator - Understanding the ASP.NET Architecture

For this lesson, we will create a web calculator as we learn more about Web Forms and Web Controls.

Create a new web project using **File | New | Web Site**. In the New Web Site dialog box, make sure **Visual C#** is selected under Installed Templates, and select the **ASP.NET Empty Web Site** template. Select **File System** from the web location dropdown box, and change the Name of the Web Project to **WebCalculator**. Click **OK**.

To add a website document, right-click in the Solution Explorer, select **Add New Item...**, select **Web Form**, make sure the Name is **Default.aspx**, and click **Add**.

Dynamic Code Blocks and Script Directives

After adding the new website document, the Code Editor opens to the Default.aspx file. Let's take a closer look at the generated code:

OBSERVE:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <div>

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

You see specific sections and syntax. The first section begins with **<%** and ends with **%>**. These delimiters mark a section of ASP.NET code that may contain either dynamic code to be executed on the server, or script directives that control and direct how ASP.NET should process the contents of the ASPX file. Script directives add an at (@) symbol after the **<%** markup code. This specific section is the Page directive used to define page-specific attributes used by the ASP.NET page parser and compiler, and it can only be included in ASPX files. Note that directives may have attributes; we'll discuss that shortly.

Note

ASPX source file content includes HTML code consisting of tags delimited by the angle brackets `<` and `>`. HTML code is commonly referred to as *markup language* or *markup code*. Many markup tags also include attributes. For example, in the code generated when we added a new website document, the **form** HTML markup tag includes two attributes: **id** and **runat**.

In previous versions of .NET, these dynamic code blocks contained dynamic code, but since .NET 2.0, .NET added and encouraged the use of code-behind files rather than embedded dynamic code. From previous lessons and experience with object-oriented programming, you might be able to guess why we would want the dynamic code separate from the user interface (UI) code: encapsulation, ease of development, and so on. We'll revisit this separation of code and UI later when we cover MVC.

When you create an ASPX web document, typically you may accept the default settings for the Page directive. In future lessons we'll explore adding other attributes and modifying the default values. Here is a list of the current attributes specified with the Page directive and their current values:

- **Language (C#)**: Specifies the language used when compiling all inline rendering (`<% %>` and `<%= %>`) and code declaration blocks within the page. Values can represent any .NET Framework-supported language, including Visual Basic, C#, or JScript. Only one language can be used and specified per page.
- **AutoEventWireup (true)**: Indicates whether the page's events are autowired. If event autowiring is enabled, this will be true; otherwise, false. The default is true.
- **CodeFile**: Specifies a path to the referenced code-behind file for the page. This attribute is used together with the **Inherits** attribute to associate a code-behind source file with a Web page. The attribute is valid only for compiled pages.
- **Inherits**: Defines a code-behind class for the page to inherit. This can be any class derived from the Page class. This attribute is used with the **CodeFile** attribute, which contains the path to the source file for the code-behind class. The **Inherits** attribute is case-sensitive when using C# as the page language, and case-insensitive when using Visual Basic as the page language.

The description for each of these attributes comes directly from Microsoft's MSDN website, at [Directives for ASP.NET Web Pages](#).

Tip

As we work through ASP.NET, you may want to review the Wikipedia article devoted to this topic at [ASP.NET](#).

Two of the static HTML statements also include **runat="server"**. The **runat** attribute specifies that a specific tag is to be handled in some way on the server computer. In the supplied code, both the `<head>` HTML tag and the `<form>` HTML tag will be handled by ASP.NET and processed on the server. ASP.NET requires these attributes; we'll see the reasons for that as we work through the code.

ASP Controls and View State

The last highlighted component is the HTML **div** tag, which is a standard HTML tag. The **div** tag is used frequently with ASP.NET web pages, essentially to create blocks. Let's add a Panel control and then examine the resulting changes to the code.

Click the **Panel** control in the ToolBox, and drag it onto the Code Editor, dropping it in after the ending `/div` tag. After adding the Panel control, modify the code as shown (you don't need to change the Code Editor to the Design view; you can add controls in either Source View or Design View):

CODE TO TYPE:

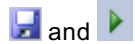
```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            Standard div tag
        </div>
        <asp:Panel ID="Panel1" runat="server">
            ASP div tag
        </asp:Panel>
    </form>
</body>
</html>
```

ASP Controls are referred to as "markup code" and are prefixed with **asp:**. Like HTML markup tags, ASP markup tags include attributes. With the ASP Panel control, we see an ID tag to identify the name of the ASP Panel control instance, and the ubiquitous runat attribute that must be present in order for the ASP Control to be processed by the server.

We used a Panel because it creates an HTML **div** element, which our web application can use to display any controls we want to add. Our web application generates web content that is used by a browser, so let's run our application and examine the rendered HTML content.



and to run the web project. If you see the "Debugging Not Enabled" dialog box, enable debugging for this project.

Once the web project is running in the browser, we need to examine the generated HTML code. You're likely using Internet Explorer, so to view the webpage source code, right-click on the webpage window and select **View source**. You see rendered content:

OBSERVE:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>

  </title></head>
  <body>
    <form method="post" action="Default.aspx" id="form1">
      <div class="aspNetHidden">
        <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="
/wEPDwUJNTA3NjE3NzI4ZGRQOgH5u9xxH62juA47QdfpREReZFQ6fUMatiVnKsjNag==" />
      </div>

      <div>
        Standard div tag
      </div>
      <div id="Panel1">

        ASP div tag

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

In the rendered content, we can see the **div** tag that resulted from the ASP Panel control. Why would we want to differentiate between a standard HTML div tag and an ASP Panel control rendered as a div tag? Even though the only rendered difference is the presence of the id attribute, ASP Controls such as the Panel control may be referenced in our code-behind file. Because a div control and a Panel control are used to contain other controls, you can use either a standard HTML div or an ASP Panel control—the next concept of view state we go over will help you to decide which to use.

Part of the rendered HTML content includes a hidden form element **__VIEWSTATE**, with a somewhat cryptic value consisting of a sequence of alphanumeric characters. When coding using ASP.NET, you want to understand the ASP.NET webpage life cycle, which includes the generation and use of the view state concept. You can read a good explanation of the life cycle and view state in the Microsoft MSDN article [Understanding ASP.NET View State](#). We encourage you to look over the article, paying particular attention to the image representing the life cycle, and perhaps review the descriptions of each of the steps.

ASP Controls are web form controls, and as such need to send data to the web server for each of the form controls. When web pages are returned, form element data will need to be restored based on the previous state and entered user data. The MSDN article we mentioned earlier explains the purpose of view state: "to persist state across postbacks." A postback is the name given whenever data is sent (posted) back to the server from the client, typically as part of a form control. State is the prior and/or current state of all of the controls. We need state because the web browser and client-server architecture are stateless, unless some type of state is maintained.

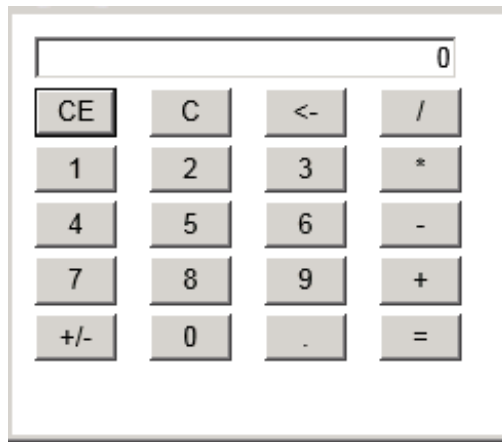
Note Because ASP controls are form controls, always add ASP controls within a form markup block.

Finally, note that the form **action** attribute value is the name of our ASPX web page: Default.aspx.

So, now that we have a better understanding of web controls and the ASP web page life cycle, let's create our web calculator.

Designing a Web User Interface

To create a web calculator, we'll add controls that will represent our web user interface. Our interface will eventually look like this:



We emphasize that our user interface is a *web user interface*, because a web browser is the target component that will render our user interface, and provide the framework for all interaction with our web calculator application. When designing our user interface, we design for the abilities of a web browser.

When we look over the web user interface in the image above, we've used three primary controls: buttons, text box, and a table. The buttons and text box are straightforward enough, but the table was implied because of the layout. Keep in mind that this course doesn't provide instruction on ways to create incredible websites; we'll just use whichever HTML controls we need to facilitate our discussion of the ASP.NET material. There are multiple ways to represent web user interfaces though. For example, you may prefer to use cascading style sheets, or HTML 5. We'll discuss those topics throughout the course, but you might want to consider taking additional HTML development courses to enhance your web-development abilities.

Let's add the controls we need to our web project to create our UI.

We won't need the generated div tag, or the ASP Panel control we added, so delete those lines from the source code. Then, find the Table control in the HTML section of the ToolBox, and drag it onto the Default.aspx Code Editor window (Source View) between the form tags as shown:

CODE TO TYPE:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
Standard div tag
        </div>
<asp:Panel ID="Panel1" runat="server">
ASP div tag
</asp:Panel>
        <table style="width: 100%;">
            <tr>
                <td>

                </td>
                <td>

                </td>
            </tr>
            <tr>
                <td>

                </td>
                <td>

                </td>
            </tr>
            <tr>
                <td>

                </td>
                <td>

                </td>
            </tr>
        </table>
    </form>
</body>
</html>
```

When you add a Table control from the ToolBox, Visual Studio generates a 3x3 table by default, and the Table markup code is automatically formatted. Adding HTML controls from the HTML section is useful, especially if you can't remember the HTML syntax. However, often a developer will modify the default HTML code significantly; typing the HTML code directly is sometimes more efficient. We need a 4x6 table (four rows, six columns), where each row has similar syntax because it will contain an ASP Button control. Let's modify our table so that it has only two rows, insert an ASP TextBox control into the first row, and an ASP Button control into the second row, first column. Change the TextBox ID to resultTextBox, add a ReadOnly attribute and set it to true, then add a Style attribute to right-align the text. Change the Button control ID to clearErrorButton,

OBSERVE:

```
<body>
  <form id="form1" runat="server">
    <table style="width: 100%;">
      <tr><td colspan="4"><asp:TextBox ID="resultTextBox" runat="server" ReadOnly=
"true" Style="text-align: right" Width="200px" ></asp:TextBox></td></tr>
      <tr>
        <td><asp:Button ID="clearErrorButton" runat="server" Text="CE" Width="40
" /></td>
      </tr>
    </table>
  </form>
</body>
```

Let's address the layout and reformatting of the table tags first. Like source code, HTML markup language can be a challenge to follow, so we'll try to use formatting tools like indentation and consolidate logical blocks on the same line; this will make the code easier to read and edit. We used a single line with the TextBox control because the entire row is one line, but we've used multiple lines and indentation with the second row because it will have multiple Button controls.

We add two ASP Controls: **TextBox** and **Button**. The markup code for ASP Controls begins with <asp:, and uses the same rules as HTML markup: you need a beginning and ending tag, or a self-closed tag (the Button is a self-closed tag, ending in />). Also, as with HTML components, ASP Controls include several attributes; you can use Intellisense to examine these attributes by clicking within the ASP control markup and pressing the space bar to bring up a context-sensitive list of attributes. For **TextBox**, we added a **ReadOnly** property to block a user from directly entering text, and a **Style** (CSS Style) to force the TextBox text to be right-aligned just like a calculator.

We also add a **Width** attribute to control the size of each **Button** control.



We set the **colspan** attribute to allow the **TextBox** control to span all four columns of our table.

Let's add the rest of the Button controls.

Highlight the Button line in the source code, and make three copies of it within the same table row. Then, copy each table row block four more times to create the twenty rows we need for our calculator table. Change the **ID** and **Text** attributes for each Button control according to this table:

ID	Text
clearErrorButton	CE
clearButton	C
backspaceButton	<-
divideButton	/
oneButton	1
twoButton	2
threeButton	3
multiplyButton	*
fourButton	4
fiveButton	5
sixButton	6
subtractButton	-
sevenButton	7
eightButton	8
nineButton	9
addButton	+
signButton	+/-
zeroButton	0

pointButton	.
equalButton	=

 and  to run the program. The UI is similar to the UI presented earlier (though sometimes you get different results from different browsers). Here's the code listing:

CODE TO TYPE:

```
.
.
.
<body>
<form id="form1" runat="server">
    <table style="width: 100%;">
        <tr><td colspan="4"><asp:TextBox ID="resultTextBox" runat="server" ReadOnly=
"true" Style="text-align: right"Width="200px" ></asp:TextBox></td></tr>
        <tr>
            <td><asp:Button ID="clearErrorButton" runat="server" Text="CE" Width="40
" /></td>
            <td><asp:Button ID="clearButton" runat="server" Text="C" Width="40" /></
td>
            <td><asp:Button ID="backspaceButton" runat="server" Text="<-" Width="40"
/></td>
            <td><asp:Button ID="divideButton" runat="server" Text="/" Width="40" /><
/td>
        </tr>
        <tr>
            <td><asp:Button ID="oneButton" runat="server" Text="1" Width="40" /></td
>
            <td><asp:Button ID="twoButton" runat="server" Text="2" Width="40" /></td
>
            <td><asp:Button ID="threeButton" runat="server" Text="3" Width="40" /></
td>
            <td><asp:Button ID="multiplyButton" runat="server" Text="*" Width="40" /
></td>
        </tr>
        <tr>
            <td><asp:Button ID="fourButton" runat="server" Text="4" Width="40" /></t
d>
            <td><asp:Button ID="fiveButton" runat="server" Text="5" Width="40" /></t
d>
            <td><asp:Button ID="sixButton" runat="server" Text="6" Width="40" /></td
>
            <td><asp:Button ID="subtractButton" runat="server" Text="-" Width="40" /
></td>
        </tr>
        <tr>
            <td><asp:Button ID="sevenButton" runat="server" Text="7" Width="40" /></
td>
            <td><asp:Button ID="eightButton" runat="server" Text="8" Width="40" /></
td>
            <td><asp:Button ID="nineButton" runat="server" Text="9" Width="40" /></t
d>
            <td><asp:Button ID="addButton" runat="server" Text="+" Width="40" /></td
>
        </tr>
        <tr>
            <td><asp:Button ID="signButton" runat="server" Text="+/-" Width="40" /><
/td>
            <td><asp:Button ID="zeroButton" runat="server" Text="0" Width="40" /></t
d>
            <td><asp:Button ID="pointButton" runat="server" Text="." Width="40" /></
td>
            <td><asp:Button ID="equalButton" runat="server" Text="=" Width="40" /></
td>
        </tr>
    </table>
</form>
</body>
.
.
.
```

That completes our web calculator interface! We'll add a few more elements to our UI shortly, but first let's add functionality to our web project.

Adding Functionality

Now that our UI is complete, it's time to edit our code-behind file.

In the Solution Explorer, select and expand the **Default.aspx** entry, then double-click the **Default.aspx.cs** entry to open the Code Editor. Alternatively, you could switch to the Design View and double-click the webpage away from the Table control. You see this code in the Code Editor:

OBSERVE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
}
```

When you add a Web Form webpage to a web project, the class in the code-behind file has the same name as the webpage, prepended with an underscore, in this case **_Default**, and includes a single method for the **Page_Load** event. This event is called whenever the page is loaded, whether it's being called for the first time, or by clicking a form submit element like a Button. ASP.NET provides a property of the Page object which is the base class for our **_Default** class, called **IsPostBack**. We need it in order to be able to determine if a page is being loaded for the first time, or a form submit element has been clicked. Let's add that code now:

Modify **Default.aspx.cs** as shown:

CODE TO TYPE:

```
.
.
.
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            // Initialize our calculator window
            resultTextBox.Text = "0";
        }
        else
        {
            // Respond to postback
        }
    }
}
.
.
.
```



and to run the program. The TextBox now contains 0. Let's discuss this code.

OBSERVE:

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            // Initialize our calculator window
            resultTextBox.Text = "0";
        }
        else
        {
            // Respond to postback
        }
    }
}
```

We check the **IsPostBack** boolean property to make sure we're **not** posting back, so we can **initialize our calculator window**.

Next, let's add code to handle the number Button click events. We'll also add a couple of class variables that handle the calculator window, and code to populate these variables during a postback action. Modify **Default.aspx.cs** as show:

CODE TO TYPE:

```
.
.
.
public partial class _Default : System.Web.UI.Page
{
    // Private class variables
    private string _currentValueString = "";
    private double _currentValueDouble = 0.0;

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            // Initialize our calculator window
            resultTextBox.Text = "0";
        }
        else
        {
            // Respond to postback
            // Grab calculator window, convert and save values as necessary
            _currentValueString = resultTextBox.Text;
            if (_currentValueString.Length > 0)
                _currentValueDouble = double.Parse(_currentValueString);
        }

        // Add event delegates to number buttons
        zeroButton.Click += new EventHandler(numberButton_Click);
        oneButton.Click += new EventHandler(numberButton_Click);
        twoButton.Click += new EventHandler(numberButton_Click);
        threeButton.Click += new EventHandler(numberButton_Click);
        fourButton.Click += new EventHandler(numberButton_Click);
        fiveButton.Click += new EventHandler(numberButton_Click);
        sixButton.Click += new EventHandler(numberButton_Click);
        sevenButton.Click += new EventHandler(numberButton_Click);
        eightButton.Click += new EventHandler(numberButton_Click);
        nineButton.Click += new EventHandler(numberButton_Click);
    }

    void numberButton_Click(object sender, EventArgs e)
    {
        // Determine number by extracting number from Button Text property
        int number = Int32.Parse((sender as Button).Text);

        // Append number to calculator window
        resultTextBox.Text = (_currentValueString+number.ToString());
    }
}
.
.
.
```



and to run the program. You can now click on any of the number Button controls, and have that number added to the calculator result window. Notice that the default 0 value in the calculator window is kept as part of the number. We'll fix that soon, but first let's discuss this code:

OBSERVE:

```

public partial class _Default : System.Web.UI.Page
{
    // Private class variables
    private string _currentValueString = "";
    private double _currentValueDouble = 0.0;

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            // Initialize our calculator window
            resultTextBox.Text = "0";
        }
        else
        {
            // Respond to postback
            // Grab calculator window, convert and save values as necessary
            _currentValueString = resultTextBox.Text;
            if (_currentValueString.Length > 0)
                _currentValueDouble = double.Parse(_currentValueString);
        }

        // Add event delegates to number buttons
        zeroButton.Click += new EventHandler(numberButton_Click);
        oneButton.Click += new EventHandler(numberButton_Click);
        twoButton.Click += new EventHandler(numberButton_Click);
        threeButton.Click += new EventHandler(numberButton_Click);
        fourButton.Click += new EventHandler(numberButton_Click);
        fiveButton.Click += new EventHandler(numberButton_Click);
        sixButton.Click += new EventHandler(numberButton_Click);
        sevenButton.Click += new EventHandler(numberButton_Click);
        eightButton.Click += new EventHandler(numberButton_Click);
        nineButton.Click += new EventHandler(numberButton_Click);
    }

    void numberButton_Click(object sender, EventArgs e)
    {
        // Determine number by extracting number from Button Text property
        int number = Int32.Parse((sender as Button).Text);

        // Append number to calculator window
        resultTextBox.Text = (_currentValueString+number.ToString());
    }
}

```

We add two class variables, `_currentValueString` and `_currentValueDouble`, that we'll use to hold the string value and a numerical double equivalent of the calculator window. Earlier, we added the code to set the calculator window when we're not doing a postback; now we add code for a page load resulting from a postback. When a Button is clicked and the Page_Load event is called, we extract the `resultTextBox.Text` value and store the result in `_currentValueString`. Then we test the length of `_currentValueString` to ensure we have something to convert, convert it to a double and storing the converted value in `_currentValueDouble`.

In order to handle the Button click events, we create delegates that use a single method `numberButton_Click`. The `numberButton_Click` method determines which number was pressed and saves the numerical value in `number`. We append this number to the calculator window `resultTextBox.Text` property to update our display. This solution works well to test our code for now, but we're going to have to change this method shortly.

Tip

With Visual Studio we can generate the delegate event handler automatically by typing in the object (such as `zeroButton`, entering the method (such as `Click`), and then the `+=` operator, and then pressing the **Tab** key.

Next, we need to respond to the "action" events of our web calculator. In order to respond, though, we will

need a way to "remember" information from previous "states" of our web page. A webpage is a stateless environment though, so each time the user submits a webpage to a web server, the only information we have is that which is sent to us from the client's web browser. If we want to preserve "state" information, we need to preserve that information by sending it back to the client's computer when we generate the web page, otherwise we would need to preserve that information on the web server and associate the server-saved information with a specific client browser session. We'll discuss the latter technique involving sessions later; for now, we'll return information and store it in "hidden" fields on our web page.

Earlier, we examined how the ASP.NET uses View State to act essentially as memory for any ASP Control, persisting information across page refreshes. We'll use the ASP HiddenField controls to create persistent web page variables. For our web calculator, we'll need a hidden field variable for each of these pieces of state information:

- First operation (firstOperationField): Indicates whether the next operation is the first operation. We'll use that information to determine if we're appending a number, or starting a new number; the default will be true.
- Previous value (pendingValueField): Provides storage for any pending value; the default will be an empty string.
- Current action (pendingActionField): Provides storage for any pending actions; the default will be an empty string.

Use the ToolBox to add the three ASP **HiddenField** controls to our markup code in **Default.aspx**, as shown:

CODE TO TYPE:

```
.  
.   
.   
<body>  
    <form id="form1" runat="server">  
        <asp:HiddenField ID="firstOperationField" runat="server" Value="true" />  
        <asp:HiddenField ID="pendingValueField" runat="server" Value="" />  
        <asp:HiddenField ID="pendingActionField" runat="server" Value="" />  
        <table style="width: 100px;">  
.  
.  
.
```



Let's discuss this code.

OBSERVE:

```
<body>  
    <form id="form1" runat="server">  
        <asp:HiddenField ID="firstOperationField" runat="server" Value="true" />  
        <asp:HiddenField ID="pendingValueField" runat="server" Value="" />  
        <asp:HiddenField ID="pendingActionField" runat="server" Value="" />  
    <table style="width: 100px;">
```

These hidden fields, **asp:HiddenField**, do not display in the client browser, but if you examine the rendered HTML source, you'll see that they do render as HTML hidden fields. For two of the fields, we set the **Value** attribute to empty, but for **firstOperationField**, we set an initial value (or state) of **true**. The **firstOperationField** will act as a boolean variable, so we'll be converting (parsing) the value of **firstOperationField** in our code-behind file.

Next, let's add the delegates and method to handle the action Button controls. Even though we're going to add the action handling code incrementally, let's add all of the delegates now. Also, let's add a few more private class variables for our hidden fields, add code to store and convert the hidden field values, modify the number event method handler to deal with adding a 0 to the beginning of a number, and add a test to determine if we need to reset the calculator window. Modify **Default.aspx.cs** as shown:

CODE TO TYPE:

```
.
.
.
public partial class _Default : System.Web.UI.Page
{
    // Private class variables
    private string _currentValueString = "";
    private double _currentValueDouble = 0;
    private string _pendingAction = "";
    private string _pendingValueString = "";
    private double _pendingValueDouble = 0.0;
    private bool _firstOperationField = true;

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            // Initialize our calculator window
            resultTextBox.Text = "0";
        }
        else
        {
            // Respond to postback
            // Grab calculator window, convert and save values as necessary
            _currentValueString = resultTextBox.Text;
            if (_currentValueString.Length > 0)
                _currentValueDouble = double.Parse(_currentValueString);

            // Save and convert hidden field values
            _firstOperationField = bool.Parse(firstOperationField.Value);
            _pendingAction = pendingActionField.Value;
            _pendingValueString = pendingValueField.Value;
            if (_pendingValueString.Length > 0)
                _pendingValueDouble = double.Parse(_pendingValueString);
        }

        // Add event delegates to number buttons
        zeroButton.Click += new EventHandler(numberButton_Click);
        oneButton.Click += new EventHandler(numberButton_Click);
        twoButton.Click += new EventHandler(numberButton_Click);
        threeButton.Click += new EventHandler(numberButton_Click);
        fourButton.Click += new EventHandler(numberButton_Click);
        fiveButton.Click += new EventHandler(numberButton_Click);
        sixButton.Click += new EventHandler(numberButton_Click);
        sevenButton.Click += new EventHandler(numberButton_Click);
        eightButton.Click += new EventHandler(numberButton_Click);
        nineButton.Click += new EventHandler(numberButton_Click);

        // Add event delegates for action buttons
        divideButton.Click += new EventHandler(actionButton_Click);
        multiplyButton.Click += new EventHandler(actionButton_Click);
        subtractButton.Click += new EventHandler(actionButton_Click);
        addButton.Click += new EventHandler(actionButton_Click);
        equalButton.Click += new EventHandler(actionButton_Click);
        clearErrorButton.Click += new EventHandler(actionButton_Click);
        clearButton.Click += new EventHandler(actionButton_Click);
        backspaceButton.Click += new EventHandler(actionButton_Click);
        signButton.Click += new EventHandler(actionButton_Click);
        pointButton.Click += new EventHandler(actionButton_Click);
    }

    void numberButton_Click(object sender, EventArgs e)
    {
        // Determine number by extracting number from Button Text property
        int number = Int32.Parse((sender as Button).Text);
    }
}
```



```

        // Set/append number to calculator window
        if (_firstOperationField)
            resultTextBox.Text = number.ToString();
        else
        {
            // Prevent prepending a 0 to a number
            if (_currentValueString == "0") _currentValueString = "";
            resultTextBox.Text = (_currentValueString + number.ToString());
        }

        // Indicate no longer first operation
        firstOperationField.Value = "false";
    }

    void actionButton_Click(object sender, EventArgs e)
    {
        // Determine action
        string currentAction = (sender as Button).Text;
    }
}
.
.
.

```

 and  to run the program. The leading 0 concatenation problem is fixed, but we still can't execute any actions. Let's discuss this code:

OBSERVE:

```
// Private class variables
private string _currentValueString = "";
private double _currentValueDouble = 0;
private string _pendingAction = "";
private string _pendingValueString = "";
private double _pendingValueDouble = 0.0;
private bool _firstOperationField = true;

protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        // Initialize our calculator window
        resultTextBox.Text = "0";
    }
    else
    {
        // Respond to postback
        // Grab calculator window, convert and save values as necessary
        _currentValueString = resultTextBox.Text;
        if (_currentValueString.Length > 0)
            _currentValueDouble = double.Parse(_currentValueString);

        // Save and convert hidden field values
        _firstOperationField = bool.Parse(firstOperationField.Value);
        _pendingAction = pendingActionField.Value;
        _pendingValueString = pendingValueField.Value;
        if (_pendingValueString.Length > 0)
            _pendingValueDouble = double.Parse(_pendingValueString);
    }

    // Add event delegates to number buttons
    zeroButton.Click += new EventHandler(numberButton_Click);
    oneButton.Click += new EventHandler(numberButton_Click);
    twoButton.Click += new EventHandler(numberButton_Click);
    threeButton.Click += new EventHandler(numberButton_Click);
    fourButton.Click += new EventHandler(numberButton_Click);
    fiveButton.Click += new EventHandler(numberButton_Click);
    sixButton.Click += new EventHandler(numberButton_Click);
    sevenButton.Click += new EventHandler(numberButton_Click);
    eightButton.Click += new EventHandler(numberButton_Click);
    nineButton.Click += new EventHandler(numberButton_Click);

    // Add event delegates for action buttons
    divideButton.Click += new EventHandler(actionButton_Click);
    multiplyButton.Click += new EventHandler(actionButton_Click);
    subtractButton.Click += new EventHandler(actionButton_Click);
    addButton.Click += new EventHandler(actionButton_Click);
    equalButton.Click += new EventHandler(actionButton_Click);
    clearErrorButton.Click += new EventHandler(actionButton_Click);
    clearButton.Click += new EventHandler(actionButton_Click);
    backspaceButton.Click += new EventHandler(actionButton_Click);
    signButton.Click += new EventHandler(actionButton_Click);
    pointButton.Click += new EventHandler(actionButton_Click);
}

void numberButton_Click(object sender, EventArgs e)
{
    // Determine number by extracting number from Button Text property
    int number = Int32.Parse((sender as Button).Text);

    // Set/append number to calculator window
    if (_firstOperationField)
        resultTextBox.Text = number.ToString();
    else
    {

```

```

        // Prevent prepending a 0 to a number
        if (_currentValueString == "0") _currentValueString = "";
        resultTextBox.Text = (_currentValueString + number.ToString());
    }

    // Indicate no longer first operation
    firstOperationField.Value = "false";
}

void actionButton_Click(object sender, EventArgs e)
{
    // Determine action
    string currentAction = (sender as Button).Text;

}
.
.
.

```

Note

The comments in the code should also help you to understand the functionality of the code. Use the comments as a guideline for your own code comments.

We added more class variables to represent the ASP HiddenField variables, and we set these property variables in the Page_Load event.

We also added more event delegates, as well as a single method, `actionButton_Click`, to handle any action events. The number event handler method `numberButton_Click` now includes code to test to determine whether the calculator window contains only 0, and uses the class variable `_firstOperationField` to determine whether we're executing a first operation, which means we can overwrite the calculator window. Once we've processed a number action, we set the ASP HiddenField variable `firstOperationField` to false.

You may have noticed that we're saving true and false in `firstOperationField` as a string rather than a boolean, so we need to convert the string value to a boolean when we retrieve the value of `firstOperationField`, and we need to use a string representation of true or false when we set `firstOperationField`.

Let's begin implementing the action event method. Modify the Default.aspx.cs as shown:

CODE TO TYPE:

```
.
.
.
void actionButton_Click(object sender, EventArgs e)
{
    // Determine action
    string currentAction = (sender as Button).Text;

    // Determine if there is a pending operation
    bool pendingOperation = (_pendingValueString.Length > 0 &&
        _currentValueString.Length > 0 && _pendingAction.Length > 0);

    // Perform previous action
    if ("/*--+=" .IndexOf(currentAction) >= 0)
    {
        // Handle arithmetic operation actions
        if (pendingOperation)
            resultTextBox.Text = doCalculation(_pendingAction, _pendingValueDouble, _currentValueDouble).ToString();


        // Update first operation hidden field
        firstOperationField.Value = "true";

        // Update previous value and action hidden fields
        if (currentAction != "=")
        {
            // Support chaining calculations
            pendingActionField.Value = currentAction;
            pendingValueField.Value = resultTextBox.Text;
        }
        else
        {
            pendingActionField.Value = "";
            pendingValueField.Value = "";
        }
    }
}

private double doCalculation(string action, double left, double right)
{
    // Perform arithmetic calculations
    double result = 0.0;
    switch (action)
    {
        case "/":
            // Prevent divide by zero
            if (right != 0)
                result = left / right;
            else
            {
                // TODO: Handle divide by zero error
            }
            break;
        case "*":
            result = left * right;
            break;
        case "-":
            result = left - right;
            break;
        case "+":
            result = left + right;
            break;
    }
    return result;
}
```


.



and  to run the program. Now, you can perform the basic arithmetic calculations. Not only can you use the equals sign to get a result, you can also string calculations together, and the calculator works! Let's discuss this code.

OBSERVE:

```
void actionButton_Click(object sender, EventArgs e)
{
    // Determine action
    string currentAction = (sender as Button).Text;

    // Determine if there is a pending operation
    bool pendingOperation = (_pendingValueString.Length > 0 &&
        _currentValueString.Length > 0 && _pendingAction.Length > 0);

    // Perform previous action
    if ("/*-+=".IndexOf(currentAction) >= 0)
    {
        // Handle arithmetic operation actions
        if (pendingOperation)
            resultTextBox.Text = doCalculation(_pendingAction, _pendingValueDouble,
                _currentValueDouble).ToString();

        // Update first operation hidden field
        firstOperationField.Value = "true";

        // Update previous value and action hidden fields
        if (currentAction != "=")
        {
            // Support chaining calculations
            pendingActionField.Value = currentAction;
            pendingValueField.Value = resultTextBox.Text;
        }
        else
        {
            pendingActionField.Value = "";
            pendingValueField.Value = "";
        }
    }
}

private double doCalculation(string action, double left, double right)
{
    // Perform arithmetic calculations
    double result = 0.0;
    switch (action)
    {
        case "/":
            // Prevent divide by zero
            if (right != 0)
                result = left / right;
            else
            {
                // TODO: Handle divide by zero error
            }
            break;
        case "*":
            result = left * right;
            break;
        case "-":
            result = left - right;
            break;
        case "+":
            result = left + right;
            break;
    }
    return result;
}
```

The action event handler method sets **pendingOperation** as a boolean flag to indicate whether an arithmetic calculation is pending. Then this method checks to see whether the **currentAction** variable

contains one of the arithmetic action buttons which would indicate that the calculator needs to complete a calculation using a string literal `"/*-+="` and the `IndexOf` string method. If we do have a pending operation, we perform the calculation by calling the `doCalculation` class helper method, resetting the `firstOperationField`. The next conditional statement supports calculation chaining by testing to see if the equals sign was clicked, setting the ASP HiddenField pending action and pending value fields.

The `doCalculation` methods works as you would expect for these arithmetic functions; note that we prevent division by zero, but we didn't add any code to alert the user.

How might we have handled the division by zero? We could have used a try/catch block, and then alerted the user by setting an ASP Label control.

Let's add the remaining action code to complete the web calculator. Modify `Default.aspx.cs` as shown:

CODE TO TYPE:

```
.
.
.
void actionButton_Click(object sender, EventArgs e)
{
    // Determine action
    string currentAction = (sender as Button).Text;

    // Determine if there is a pending operation
    bool pendingOperation = (_pendingValueString.Length > 0 &&
        _currentValueString.Length > 0 && _pendingAction.Length > 0);

    // Perform previous action
    if ("/*--+=" .IndexOf(currentAction) >= 0)
    {
        // Handle arithmetic operation actions
        if (pendingOperation)
            resultTextBox.Text = doCalculation(_pendingAction, _pendingValueDouble, _currentValueDouble).ToString();

        // Update first operation hidden field
        firstOperationField.Value = "true";



        // Update previous value and action hidden fields
        if (currentAction != "=")
        {
            // Support chaining calculations
            pendingActionField.Value = currentAction;
            pendingValueField.Value = resultTextBox.Text;
        }
        else
        {
            pendingActionField.Value = "";
            pendingValueField.Value = "";
        }
    }
    else
    {
        // Handle remaining actions
        switch (currentAction)
        {
            case "C":
                // Reset calculator window and hidden fields
                resultTextBox.Text = "0";
                pendingActionField.Value = "";
                pendingValueField.Value = "";
                firstOperationField.Value = "true";
                break;
            case "CE":
                // Clear error
                resultTextBox.Text = "0";
                firstOperationField.Value = "true";
                break;
            case "<-":
                // Backspace - prevent leaving "bad" data in calculator window
                string newResult = _currentValueString.Substring(0, _currentValueString.Length - 1);
                if (newResult.Length == 0 || newResult == "-")
                    resultTextBox.Text = "0";
                else
                    resultTextBox.Text = newResult;
                break;
            case ".":
                // Decimal point
                if (_currentValueString.IndexOf(".") < 0) resultTextBox.Text = _currentValueString + ".";
        }
    }
}
```

```

        break;
    case "+/-":
        // Sign
        resultTextBox.Text = (_currentValueDouble * -1).ToString();
        break;
    }
}

private double doCalculation(string action, double left, double right)
{
    .
    .
    .
}

```

 and  to run the program. Test the remaining action buttons. "C" (Clear) will reset the calculator. "<-" (Backspace) will delete the last typed number from the calculator window, or set the value to 0 if there is only a single number. "CE" (Clear Error) resets the first operation and the calculator window to 0. "." sets a decimal in the number. "+/-" will toggle the sign of the number. Let's discuss this code:

OBSERVE:

```

void actionButton_Click(object sender, EventArgs e)
{
    .
    .
    .
    else
    {
        // Handle remaining actions
        switch (currentAction)
        {
            case "C":
                // Reset calculator window and hidden fields
                resultTextBox.Text = "0";
                pendingActionField.Value = "";
                pendingValueField.Value = "";
                firstOperationField.Value = "true";
                break;
            case "CE":
                // Clear error
                resultTextBox.Text = "0";
                firstOperationField.Value = "true";
                break;
            case "<-":
                // Backspace - prevent leaving "bad" data in calculator window
                string newResult = _currentValueString.Substring(0, _currentValueString.Length - 1);
                if (newResult.Length == 0 || newResult == "-")
                    resultTextBox.Text = "0";
                else
                    resultTextBox.Text = newResult;
                break;
            case ".":
                // Decimal point
                if (_currentValueString.IndexOf(".") < 0) resultTextBox.Text = _currentValueString + ".";
                break;
            case "+/-":
                // Sign
                resultTextBox.Text = (_currentValueDouble * -1).ToString();
                break;
        }
    }
}

```

Most of this action code will make sense to you; the inline comments should help. Note that the hidden fields

and calculator window are reset with the Clear selection. For Clear Error, we reset the calculator window, and reset the **firstOperationField** variable. To handle the backspace, we use the **Substring** string method to trim off the rightmost digit (or character). We do have to test to make sure we have something left over after removing the number, and that we aren't left with just the negative sign. We also have to make sure we don't have a second decimal point. Finally, to change the sign, we multiply by -1.

So, that's it! You've completed the web calculator project using ASP.NET! (Applause.)

Before you move on to the next lesson, make sure to do your homework! (Not that you need reminding.) See you in the next lesson...

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Web Forms

Lesson Objectives

In this lesson you will:

- create an ASP.NET web form project.
- use a template that creates a basic ASP.NET website.
- customize the website with pages and ASP controls.

Web Forms

Visual Studio includes several templates that are used when creating a new project. In this lesson, we'll create an ASP.NET web form project using a template that creates a basic ASP.NET website. We will customize the website by adding pages and ASP controls, including validation controls.

Creating the Project

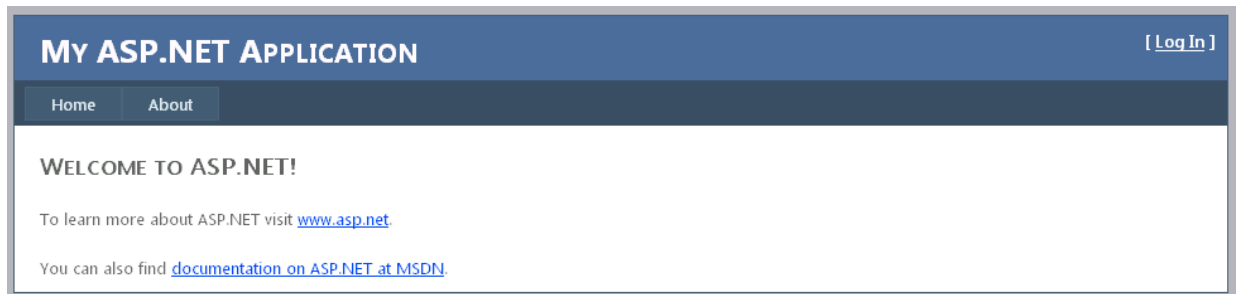
First, create the ASP.NET website to use with this project by selecting **File | New | Web Site**. In the New Web Site dialog box, make sure **Visual C#** is selected under Installed Templates, and select the **ASP.NET Web Site** template. Select **File System** from the Web location dropdown box, and change the Name of the Web Project to **Biography**. Click **OK**.

Note

We are still creating websites rather than web projects, although we could have selected a New Project and then selected the ASP.NET Web Application template. The underlying code would have been the same, except that our website would have been part of a Studio Solution and Project rather than just a website.



and  to run the web project. You see a website like this:



Without any coding on your part, Visual Studio has generated the beginnings of a basic dynamic website. As you navigate the site, you'll find these features:

- Page navigation using tabs to switch between a Home page and an About page
- User registration page
- Login and logout of registered users
- Password change for registered users

Note

The term *dynamic website* is used frequently to refer to a website with content that is not fixed. Currently, the default website dynamic content only displays whether a registered user is logged in and, if so, which registered user that is.

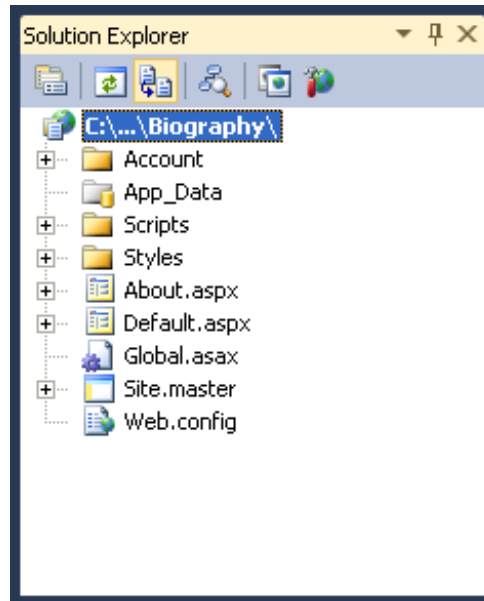
Before modifying the content, let's review what was generated, and how it works.

Examining the Auto-generated Website

We'll start by examining and describing the website contents in the Solution Explorer. Included with each

description is an indication of whether the item is accessible from the web. What does that mean? When creating an ASP.NET website, users will access your website content through a web browser. When we run our website using Visual Studio, a web browser launches and our website is displayed just as if it were deployed to a web server, including using a URL. Developing a website requires that we begin to understand a web address, how a user enters a web address to access website content we develop, and what information is accessible.

Tip For an excellent brief explanation of URLs, see [URL Definition - What Is a URL?](#)



Account Folder	This folder contains registration, login, and password-related markup and code-behind files. Pages that reside in the account folder are accessible from the website. Users will access these webpages if they attempt to login or register. Note that registration is not required to access any content of the website, but if a user is registered, our website can deliver user-specific content. Register so you can see that the user's registered name will appear at the top-right of every page once they've logged in. We'll learn more about this folder in a later lesson.
App_Data Folder	This folder contains database files used by the application, and is inaccessible from the web. The folder is empty unless a user registers. With the first attempt to register, a SQL Express database file ASPNETDB.MDF is created in this folder and linked to your website. The database is created as a User Instance database. User instances require that SQL Express be installed and running. User instances are great for development, but usually they're not appropriate for production web servers. As we stated earlier, our server set up precludes using the normal database access features for user accounts. Visual Studio 2010 ships with SQL Server 2008, which does not handle network access of its database file well. Normally, when you create a website or web application, it just works.
Scripts Folder	This folder contains script files such as JavaScript, and is accessible from the web. We will discuss this folder and scripts in more detail later in the course.
Styles Folder	This folder contains CSS (style sheet) files, and is accessible from the web.
About.aspx Webpage	The About markup and code-behind file, and the rendered web page is accessible from web. We'll modify this file to provide more information about our site. This file is also a good place to mention any third-party technologies you might use in your site and to link to open-source licenses.
Default.aspx Webpage	This is the default page for an ASP.NET website. When the browser connects to the website, this page will start automatically. On most other technologies, this would be analogous to an index.html or index.php file. It contains the markup and code-behind file; the rendered web page is accessible from the web.
Global.asax Source File	This file declares and handles application-level and session-level events and objects. It is inaccessible from the web. Typically it's managed by Visual Studio; you will rarely need to edit it directly.

Site.master Webpage Template	This is the website layout markup template and code-behind file that is inaccessible from the web. It is used when rendering pages that include the MasterPageFile attribute of the Page directive. This is a cool feature of an ASP.NET website where we can manage features that we want to show up across all of our site pages.
web.config File	This file contains XML settings and the configuration file that is inaccessible from the web. In some cases we will edit this file to turn on features and to control access to all or parts of the site.

The Solution Explorer for this website includes a number of folders. For example, the Account folder is accessible; if you attempt to access it while running your website, you'll see this in your browser:

Directory Listing -- /Biography/Account/

[\[To Parent Directory\]](#)

Tuesday, December 11, 2012 08:50 AM	4,464	ChangePassword.aspx
Tuesday, December 11, 2012 08:50 AM	296	ChangePassword.aspx.cs
Tuesday, December 11, 2012 08:50 AM	514	ChangePasswordSuccess.aspx
Tuesday, December 11, 2012 08:50 AM	303	ChangePasswordSuccess.aspx.cs
Tuesday, December 11, 2012 08:50 AM	3,052	Login.aspx
Tuesday, December 11, 2012 08:50 AM	412	Login.aspx.cs
Tuesday, December 11, 2012 08:50 AM	5,658	Register.aspx
Tuesday, December 11, 2012 08:50 AM	814	Register.aspx.cs
Tuesday, December 11, 2012 08:50 AM	330	Web.config

Version Information: ASP.NET Development Server 10.0.0.0

For security reasons, a public web server typically will not allow directory listings, but for ease of development, the ASP.NET Development Server allows them. How did we get this listing? We modified the URL that Studio provided when the browser was launched. Initially, the URL was **http://localhost:1863/Biography/** (or something similar), but we can modify the URL directly by adding the Account folder name. The URL becomes **http://localhost:1863/Biography/Account**, and will then generate the page shown above.

You should use the Solution Explorer to open any folders and examine any source files. Let's discuss how the different web pages are created, and how the navigation works.

Layout and Navigation

The generated ASP.NET website includes a master website page. This page forms the background for all of the webpages. Locate the **Site.Master** page in the Solution Explorer and open it, switching to Source view if necessary. As you examine the markup source, you'll see that:

- rather than a Page directive, the Site.Master has a Master directive.
- the markup source includes a complete set of HTML tags, such as <html>, <head>, and <body>.
- site styles are referenced and loaded in the Site.Master <head> section.
- the Site.Master includes a number of ASP Controls: ContentPlaceHolder, LoginView, and Menu.
- the ContentPlaceHolder ASP controls have unique IDs that will be exposed as content targets in webpages.
- the Site.Master includes a code-behind file.

Let's take a look at the **Site.master** markup as it is created by Visual Studio:

OBSERVE:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="Site.master.cs" Inherits="SiteMaster" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head runat="server">
    <title></title>
    <link href="~/Styles/Site.css" rel="stylesheet" type="text/css" />
    <asp:ContentPlaceholder ID="HeadContent" runat="server">
        </asp:ContentPlaceholder>
</head>
<body>
    <form runat="server">
        <div class="page">
            <div class="header">
                <div class="title">
                    <h1>
                        My ASP.NET Application
                    </h1>
                </div>
                <div class="loginDisplay">
                    <asp:LoginView ID="HeadLoginView" runat="server" EnableViewState="false">
                        <AnonymousTemplate>
                            [ <a href="~/Account/Login.aspx" ID="HeadLoginStatus" runat="server">Log In</a> ]
                        </AnonymousTemplate>
                        <LoggedInTemplate>
                            Welcome <span class="bold"><asp:LoginName ID="HeadLoginName" runat="server" /></span>!
                            [ <asp:LoginStatus ID="HeadLoginStatus" runat="server" LogoutAction="Redirect" LogoutText="Log Out" LogoutPageUrl="~/"/> ]
                        </LoggedInTemplate>
                    </asp:LoginView>
                </div>
                <div class="clear hideSkiplink">
                    <asp:Menu ID="NavigationMenu" runat="server" CssClass="menu" EnableViewState="false" IncludeStyleBlock="false" Orientation="Horizontal">
                        <Items>
                            <asp:MenuItem NavigateUrl="~/Default.aspx" Text="Home"/>
                            <asp:MenuItem NavigateUrl="~/About.aspx" Text="About"/>
                        </Items>
                    </asp:Menu>
                </div>
            </div>
            <div class="main">
                <asp:ContentPlaceholder ID="MainContent" runat="server"/>
            </div>
            <div class="clear">
            </div>
        </div>
        <div class="footer">
        </div>
    </form>
</body>
</html>
```

The **Site.master** file is the "template" for all of the pages in your site. It contains the basic HTML markup for the entire site. It also contains the reference to the **CSS style sheet** that the site will use, which is the **/Styles/Site.css** file. Notice that, in the entire file, we don't specifically indicate any colors or positioning of our HTML tags or ASP.NET controls. Everything will be controlled by the **Site.css** file.

Also notice that the **Site.master** page does not use the **Page** directive at the top of the file. Instead, the **Site.master** uses the **Master** directive to tell the server that this is a master page and should be used as a

template.

Note

You can have multiple master pages. Master pages can also extend other master files. For details, see the MSDN article on [Nested ASP.NET Master Pages](#)

The ASP.NET **LoginView** control sits in the upper right hand corner of every page, since it is in the **Site.master** file. Anonymous users will see the text in the **AnonymousTemplate** section of the control; logged-in users will see the text in the **LoggedInTemplate**.

The last control in the header of the page is the **Menu** control. Inside the **Items** element, you can add as many **MenuItem**s as you want. Keep in mind that this is the **Site.master** page, so this menu will be on all pages of the site.

The ASP.NET control, `<asp:ContentPlaceHolder ID="MainContent" runat="server"/>`, is the real heart of the **Site.master** file. All of the other pages in the site will place their content inside this control.

Note

You can have multiple **ContentPlaceHolder** controls in your **Site.master** file, just make sure they have different **IDs** so you can add or remove content in your pages just based on which placeholder they use. Most pages will only need to use the default placeholder.

OBSERVE: Default.aspx

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <h2>
        Welcome to ASP.NET!
    </h2>
    <p>
        To learn more about ASP.NET visit <a href="http://www.asp.net" title="ASP.NET Website">www.asp.net</a>.
    </p>
    <p>
        You can also find <a href="http://go.microsoft.com/fwlink/?LinkID=152368&clcid=0x409"
            title="MSDN ASP.NET Docs">documentation on ASP.NET at MSDN</a>.
    </p>
</asp:Content>
```

The **Content** control in the **Default.aspx** file is linked to the **ContentPlaceHolderID** from the **Site.master** file. When the **Default.aspx** file is loaded, the **Site.master** file is used as its template. The content inside this control is placed within that template and the page is loaded as **Default.aspx**.

Creating A Site

Let's take the site that Visual Studio give us and modify it for our purposes.

Note



Our focus here is on using different navigation controls on the site and to utilize the **Site.master** page to our advantage. Currently we are not focused on the **Site.css** file. We aren't really creating a full-fledged working site right now, but just using the Biography site as a tool to showcase the navigation controls available in ASP.NET.

Edit the **Default.aspx** file as shown:

CODE TO TYPE:

```
<%@ Page Title="Home Page" Language= : MasterPageFile="~/Site.master" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>

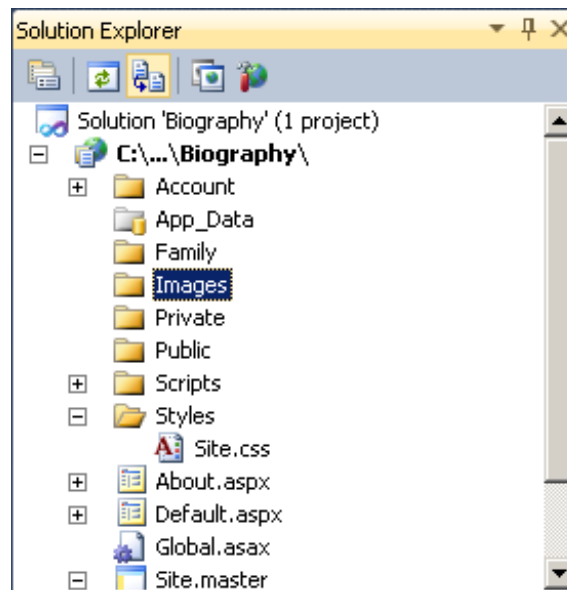
<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <h2>
        Welcome to ASP.NET!
    </h2>
    <p>
        To learn more about ASP.NET visit <a href="http://www.asp.net" title="ASP.NET Website">
            www.asp.net</a>.
        </p>
        <p>
            You can also find <a href="http://go.microsoft.com/fwlink/?LinkID=152368&eclid=0x409"
            title="MSDN ASP.NET Docs">documentation on ASP.NET at MSDN</a>.
        </p>
        <asp:Panel ID="IntroductionPanel" runat="server">
            <h1>
                Introduction</h1>
            <p>
                Put some text here about yourself, or some historic figure, as an introduction.
                Notice that you can use <i>any</i> HTML that you want and, since many
                elements are styled by the <b>Site.css</b> file, they will pick up that style.</p>
        </asp:Panel>
    </asp:Content>
```

 and . If you have some knowledge of CSS, you can edit the **Site.css** file to change the styling of the site, such as colors, fonts, and such. If you don't edit the CSS file, it's fine too; this course is not focused on CSS.

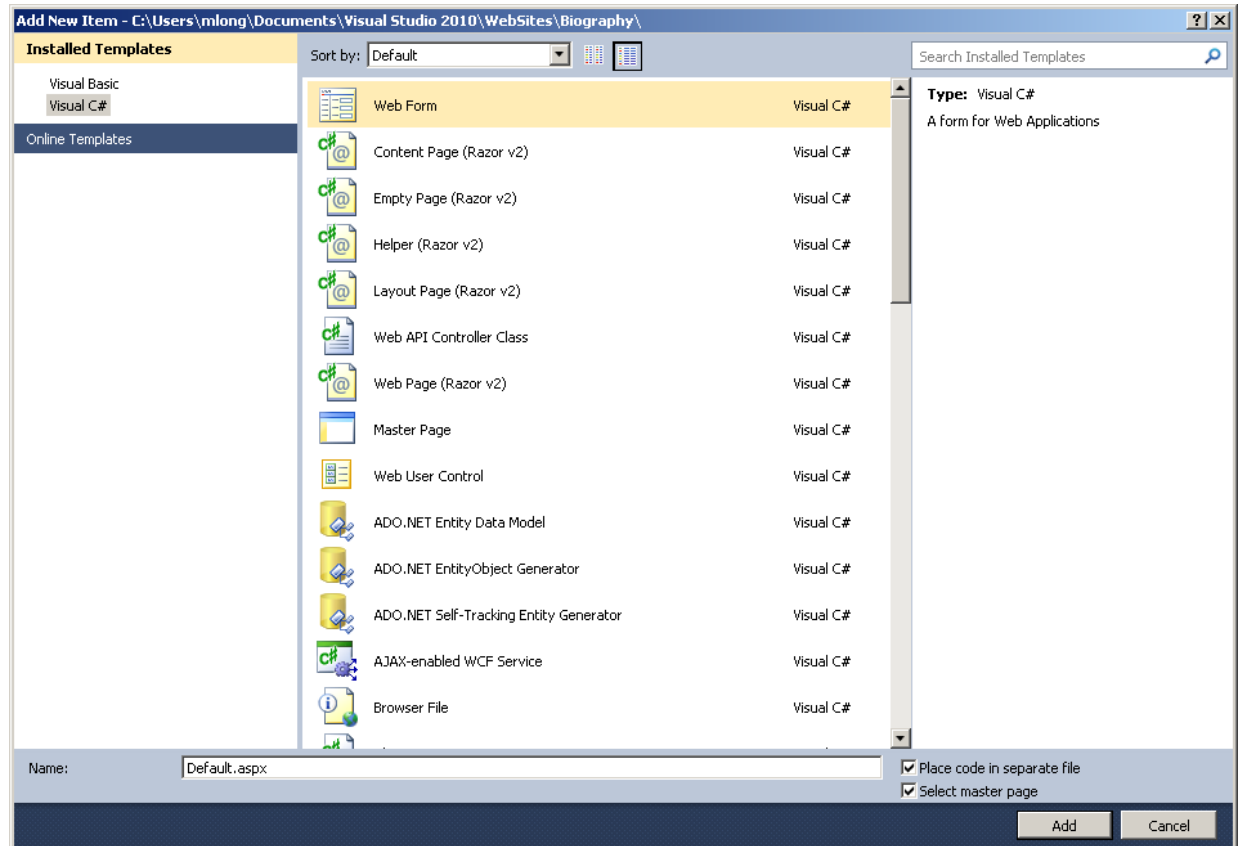
Let's create some other files now.

Tip One of the best CSS resources on the web is the W3C's [CSS Tutorial](#).

Add some new folders to your website and name them: **Public**, **Private**, **Family**, and **Images**.



Right-click on the **Public** directory and select **Add New Item**. Select a **Web Form** from the Visual C# list. Check the **Select Master Page** check box. Leave the name as **Default.aspx**. (In the next dialog, click **OK** to select the **Site.master** as the master file.)



Now, create the same file in the **Family** and **Private** directories.


We've created these files so that we can illustrate one of the Navigation controls available. Open the **Site.master** file and add the control as shown:

CODE TO TYPE:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="Site.master.cs" Inherits="SiteMaster" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head runat="server">
    <title></title>
    <link href="~/Styles/Site.css" rel="stylesheet" type="text/css" />
    <asp:ContentPlaceholder ID="HeadContent" runat="server">
        </asp:ContentPlaceholder>
</head>
<body>
    <form runat="server">
        <div class="page">
            <div class="header">
                <div class="title">
                    <h1>
                        My ASP.NET Application
                    </h1>
                </div>
                <div class="loginDisplay">
                    <asp:LoginView ID="HeadLoginView" runat="server" EnableViewState="false">
                        <AnonymousTemplate>
                            [ <a href="~/Account/Login.aspx" ID="HeadLoginStatus" runat="server">Log In</a> ]
                        </AnonymousTemplate>
                        <LoggedInTemplate>
                            Welcome <span class="bold"><asp:LoginName ID="HeadLoginName" runat="server" /></span>!
                            [ <asp:LoginStatus ID="HeadLoginStatus" runat="server" LogoutAction="Redirect" LogoutText="Log Out" LogoutPageUrl="~/"/> ]
                        </LoggedInTemplate>
                    </asp:LoginView>
                </div>
                <div class="clear hideSkiplink">
                    <asp:Menu ID="NavigationMenu" runat="server" CssClass="menu" EnableViewState="false" IncludeStyleBlock="false" Orientation="Horizontal">
                        <Items>
                            <asp:MenuItem NavigateUrl="~/Default.aspx" Text="Home"/>
                            <asp:MenuItem NavigateUrl="~/About.aspx" Text="About"/>
                        </Items>
                    </asp:Menu>
                </div>
                <asp:SiteMapPath ID="SiteMapPath1" runat="server">
                    </asp:SiteMapPath>
            </div>
            <div class="main">
                <asp:ContentPlaceholder ID="MainContent" runat="server"/>
            </div>
            <div class="clear">
            </div>
        </div>
        <div class="footer">
        </div>
    </form>
</body>
</html>
```



and . You get an error message:

Server Error in '/Biography' Application.

The file web.sitemap required by XmlSiteMapProvider does not exist.

That's because we haven't added the **Web.sitemap** file yet. Let's do that now.

Right-click on the project and select **Add New Item**. In the dialog, select **XML File**, and change the name in the text box to **Web.sitemap**. Modify the code as shown below:

CODE TO TYPE:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap>
  <siteMapNode title="Home" description="Home" url="~/Default.aspx" >
    <siteMapNode title="Public" description="Public Information"
      url="~/Public/Default.aspx" >
    </siteMapNode>
    <siteMapNode title="Private" description="Private Information"
      url="~/Private/Default.aspx">
    </siteMapNode>
    <siteMapNode title="Family" description="Information For Family"
      url="~/Family/Default.aspx">
    </siteMapNode>
    <siteMapNode title="About" description="About us"
      url="~/About.aspx">
    </siteMapNode>
  </siteMapNode>
</siteMap>
```

There is an outer **Home** site map node and the other nodes are children of that node. This ensures that the **Home** link will always be shown in the "breadcrumb" (for our purposes, "breadcrumbs" are links to pages you've visited within the site), since logically, everything in the site is within the Home page of the site. The `<siteMapNode>`s can be nested as needed:

OBSERVE:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap>
  <siteMapNode title="Home" description="Home" url="~/Default.aspx" >
    <siteMapNode title="Public" description="Public Information"
      url="~/Public/Default.aspx" >
    </siteMapNode>
    <siteMapNode title="Private" description="Private Information"
      url="~/Private/Default.aspx">
    </siteMapNode>
    <siteMapNode title="Family" description="Information For Family"
      url="~/Family/Default.aspx">
    </siteMapNode>
    <siteMapNode title="About" description="About us"
      url="~/About.aspx">
    </siteMapNode>
  </siteMapNode>
</siteMap>
```

Tip For more information on site maps, see the MSDN article on [ASP.NET Site Maps](#)

Now, let's update our menu so that we can actually get to our new pages. Edit your **Site.master** file as shown:



CODE TO TYPE:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="Site.master.cs" Inherits="SiteMaster" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head runat="server">
    <title></title>
    <link href="~/Styles/Site.css" rel="stylesheet" type="text/css" />
    <asp:ContentPlaceholder ID="HeadContent" runat="server">
        </asp:ContentPlaceholder>
</head>
<body>
    <form runat="server">
        <div class="page">
            <div class="header">
                <div class="title">
                    <h1>
                        My ASP.NET Application
                    </h1>
                </div>
                <div class="loginDisplay">
                    <asp:LoginView ID="HeadLoginView" runat="server" EnableViewState="false">
                        <AnonymousTemplate>
                            [ <a href="~/Account/Login.aspx" id="HeadLoginStatus" runat="server">Log In</a>
                        </AnonymousTemplate>
                        <LoggedInTemplate>
                            Welcome <span class="bold">
                                <asp:LoginName ID="HeadLoginName" runat="server" />
                            </span>! [
                                <asp:LoginStatus ID="HeadLoginStatus" runat="server" LogoutAction="Redirect" LogoutText="Log Out"
                                    LogoutPageUrl="~/ " />
                            ]
                        </LoggedInTemplate>
                    </asp:LoginView>
                </div>
                <div class="clear hideSkiplink">
                    <asp:Menu ID="NavigationMenu" runat="server" CssClass="menu" EnableViewState="false"
                        IncludeStyleBlock="false" Orientation="Horizontal">
                        <Items>
                            <asp:MenuItem NavigateUrl="~/Default.aspx" Text="Home" />
                            <asp:MenuItem NavigateUrl="~/About.aspx" Text="About" />
                            <asp:MenuItem NavigateUrl="~/Public/Default.aspx" Text="Public" />
                            <asp:MenuItem NavigateUrl="~/Private/Default.aspx" Text="Private" />
                            <asp:MenuItem NavigateUrl="~/Family/Default.aspx" Text="Family" />
                        </Items>
                    </asp:Menu>
                </div>
                <div class="main">
                    <asp:SiteMapPath ID="SiteMapPath1" runat="server">
                    </asp:SiteMapPath>
                    <asp:ContentPlaceholder ID="MainContent" runat="server" />
                </div>
                <div class="clear">
                </div>
            </div>
        </div>
    </form>
</body>
```



```
<div class="footer">
</div>
</form>
</body>
</html>
```

 and . Notice that the menu has been updated so that we can navigate to any page and the breadcrumb shows where we are in the site.

Note


When you add or remove pages from your site, don't forget to update the sitemap and menu so that they reflect the changes.

Now, let's explore another control that can aid in navigation, the **TreeView** control. In your **Family** directory, create two new files the same way you created the **Default.aspx** files. Name them **Mother.aspx** and **Father.aspx**.

Edit your **Web.sitemap** file to reflect the new pages, as shown:

CODE TO TYPE:

```
<?xml version="1.0" encoding="utf-8" ?>
<sitemap>
  <siteMapNode title="Home" description="Home" url="~/Default.aspx" >
    <siteMapNode title="Public" description="Public Information"
      url="~/Public/Default.aspx" >
    </siteMapNode>
    <siteMapNode title="Private" description="Private Information"
      url="~/Private/Default.aspx">
    </siteMapNode>
    <siteMapNode title="Family" description="Information For Family"
      url="~/Family/Default.aspx">
      <siteMapNode title="Mother" description="Mother"
        url="~/Family/Mother.aspx" >
      </siteMapNode>
      <siteMapNode title="Father" description="Father"
        url="~/Family/Father.aspx" >
      </siteMapNode>
    </siteMapNode>
    <siteMapNode title="About" description="About us"
      url="~/About.aspx">
    </siteMapNode>
  </siteMapNode>
</sitemap>
```


 Save your file but don't run it. We still need to add to our **Default.aspx** page:

CODE TO TYPE:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <asp:Panel ID="TreeViewSiteMapPanel" runat="server">
        <asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1">
        </asp:TreeView>
        <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
    </asp:Panel>
    <asp:Panel ID="IntroductionPanel" runat="server">
        <h1>
            Introduction</h1>
        <p>
            Put some text here about yourself, or some historic figure, as an introduction.
            Notice that we can use any HTML that we want and since many elements are styled
            by the <b>Site.css</b> file, they will pick up that style.</p>
    </asp:Panel>
</asp:Content>
```



and . Now, we have a **TreeView** control on the Home page that reflects our site structure, according to the **Web.sitemap** file:

OBSERVE:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <asp:Panel ID="TreeViewSiteMapPanel" runat="server">
        <asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMapDataSource1">
        </asp:TreeView>
        <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
    </asp:Panel>
    <asp:Panel ID="IntroductionPanel" runat="server">
        <h1>
            Introduction</h1>
        <p>
            Put some text here about yourself, or some historic figure, as an introduction.
            Notice that we can use any HTML that we want and since many elements are styled
            by the <b>Site.css</b> file, they will pick up that style.</p>
    </asp:Panel>
</asp:Content>
```

You just added a fairly complex control to your site. All you need to do is add a **TreeView** control and a **SiteMapDataSource** control and link the two by using the **SiteMapDataSource**'s ID as the **DataSourceID** of the **TreeView**.

Before you move on to the next lesson, be sure to do your homework!



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Security

Lesson Objectives

In this lesson, you will:

- use ASP.NET components to add security elements to your website.
- use authentication and authorization.

Authentication and Authorization

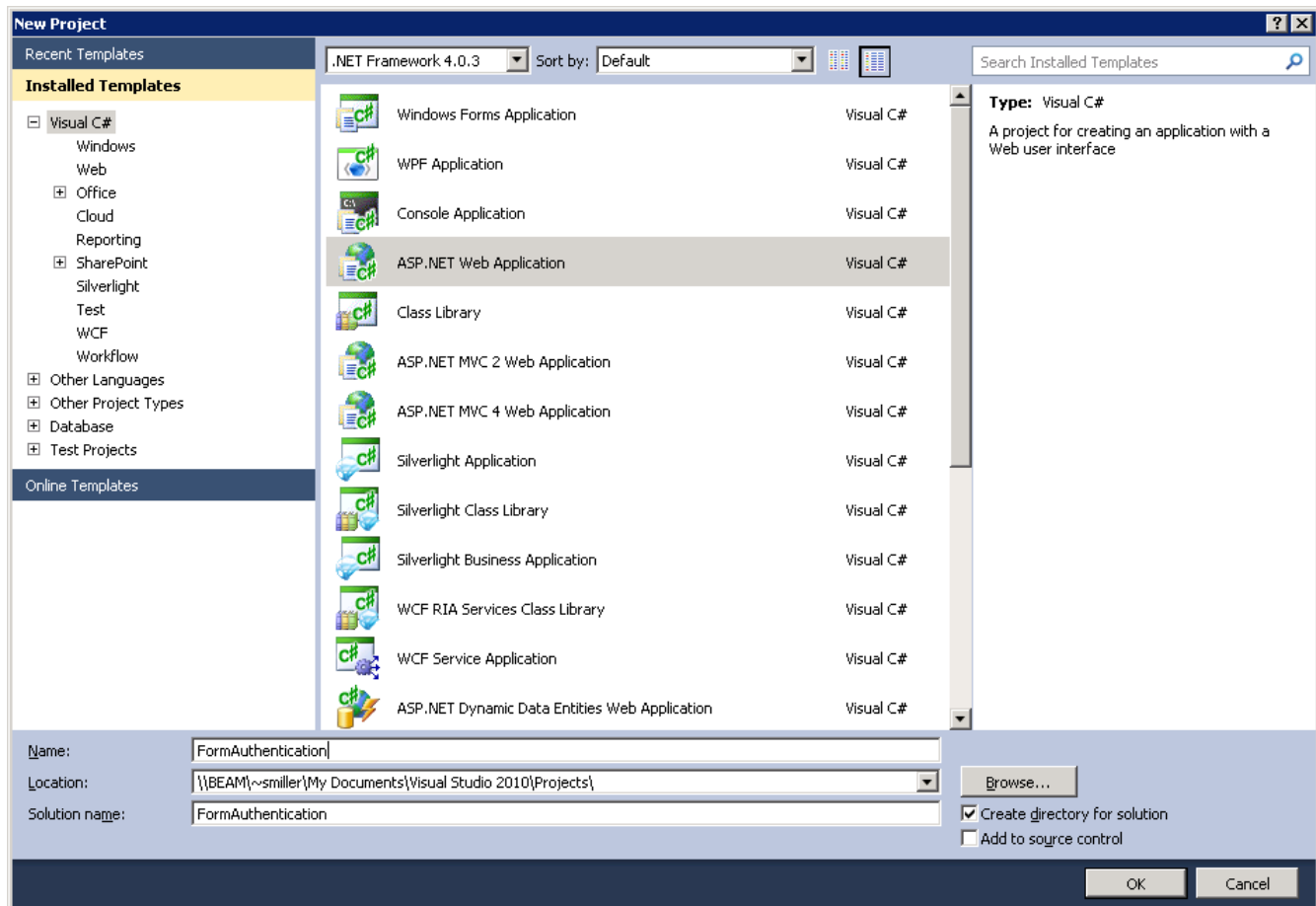
To create a website that has authorized users, we want a system that "authenticates" users' identities. The system also needs to grant authorization for users to enter certain areas of the site, or see certain information. We may also want to control a user's access based on their *role*.

In order to accomplish these goals, we'll maintain a database of user accounts and roles associated with those accounts. Visual Studio has a set of built-in tools to help us manage user accounts, as well as their roles.

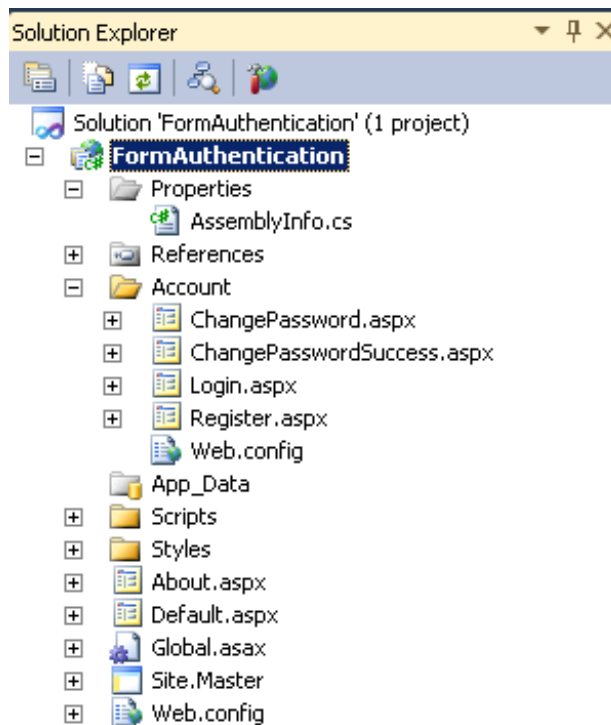
Create a new project named **FormAuthentication**.

WARNING


Do *not* name your project **FormsAuthentication** because there is already a C# class with that name. You can read more about this class, in the MSDN article, [FormsAuthentication Class](#).



Visual Studio creates a bunch of files and directories in the Solution Explorer:



We are interested primarily in the **Account** directory. We'll also be using other files like the various **Web.config** files.

Click on  to see this website. Click the **Log In** link in the upper-right corner to see the login process in action. Even if we aren't logged in, we can still see all pages within the site for now, because we haven't set up any pages or URLs that are protected. Let's see how to determine whether we are logged in.

Note The login, change password, and register pages are all located in the **Account** directory. Also, each directory has a **Web.config** file that controls how that directory is configured.

Before we go any further, let's enable roles for our application. Change the **roleManager enabled** setting from false to **true** in the **Web.config** file as shown:

CODE TO TYPE:

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

<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=169433
-->

<configuration>
  <connectionStrings>
    <add name="ApplicationServices"
      connectionString="data source=.\SQLEXPRESS;Integrated Security=SSPI;AttachDBFi
lename=|DataDirectory|\aspnetdb.mdf;User Instance=true"
      providerName="System.Data.SqlClient" />
    </connectionStrings>

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

      <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx" timeout="2880" />
      </authentication>

      <membership>
        <providers>
          <clear/>
          <add name="AspNetSqlMembershipProvider" type="System.Web.Security.SqlMembership
Provider" connectionStringName="ApplicationServices"
            enablePasswordRetrieval="false" enablePasswordReset="true" requiresQuestio
nAndAnswer="false" requiresUniqueEmail="false"
            maxInvalidPasswordAttempts="5" minRequiredPasswordLength="6" minRequiredNo
nalphanumericCharacters="0" passwordAttemptWindow="10"
            applicationName="/" />
          </providers>
        </membership>

        <profile>
          <providers>
            <clear/>
            <add name="AspNetSqlProfileProvider" type="System.Web.Profile.SqlProfileProvide
r" connectionStringName="ApplicationServices" applicationName="/" />
          </providers>
        </profile>

        <roleManager enabled="falsetrue">
          <providers>
            <clear/>
            <add name="AspNetSqlRoleProvider" type="System.Web.Security.SqlRoleProvider" co
nnectionStringName="ApplicationServices" applicationName="/" />
            <add name="AspNetWindowsTokenRoleProvider" type="System.Web.Security.WindowsTok
enRoleProvider" applicationName="/" />
          </providers>
        </roleManager>

      </system.web>

      <system.webServer>
        <modules runAllManagedModulesForAllRequests="true"/>
      </system.webServer>
    </configuration>
```



and close the **Web.config** file. Modify the website's **Default.aspx** page as shown:

CODE TO TYPE:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="FormAuthentication1._Default" >

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <asp:Panel runat="server" ID="LoggedInMessagePanel">
        <asp:Label runat="server" ID="WelcomeBackMessage"></asp:Label>
    </asp:Panel>
    <asp:Panel runat="Server" ID="NotLoggedInMessagePanel">
        <asp:HyperLink runat="server" ID="lnkLogin" Text="Log In" NavigateUrl="~/Account/Login.aspx"></asp:HyperLink>
    </asp:Panel>
    <h2>
        Welcome to ASP.NET!
    </h2>
    <p>
        To learn more about ASP.NET visit <a href="http://www.asp.net" title="ASP.NET Website">
            www.asp.net</a>.
        </p>
    <p>
        You can also find <a href="http://go.microsoft.com/fwlink/?LinkID=152368&clcid=0x409"
            title="MSDN ASP.NET Docs">documentation on ASP.NET at MSDN</a>.
        </p>
</asp:Content>
```



Save, but don't run it yet. We still need to do some work on the code-behind file:

OBSERVE: Default.aspx

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="FormAuthentication1._Default" >

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <asp:Panel runat="server" ID="LoggedInMessagePanel">
        <asp:Label runat="server" ID="WelcomeBackMessage"></asp:Label>
    </asp:Panel>
    <asp:Panel runat="Server" ID="NotLoggedInMessagePanel">
        <asp:HyperLink runat="server" ID="lnkLogin" Text="Log In" NavigateUrl="~/Account/Login.aspx"></asp:HyperLink>
    </asp:Panel>
    <h2>
        Welcome to ASP.NET!
    </h2>
    <p>
        To learn more about ASP.NET visit <a href="http://www.asp.net" title="ASP.NET Website">
            www.asp.net</a>.
        </p>
    <p>
        You can also find <a href="http://go.microsoft.com/fwlink/?LinkID=152368&clcid=0x409"
            title="MSDN ASP.NET Docs">documentation on ASP.NET at MSDN</a>.
        </p>
</asp:Content>
```

We added two ASP Panel objects to our page. They will be processed by the server before being delivered to the web browser. We'll look at the code-behind file in just a minute to see how they get processed. Initially, the

LoggedInMessagePanel will be set to be invisible; the **NotLoggedInMessagePanel** panel will be set to be visible. We can see that the **NotLoggedInMessagePanel**'s Label item has a predefined **Text="Log In"** attribute that displays "Log In" on the screen when the panel is visible. The **LoggedInMessagePanel**'s Label item will have text assigned to it later. We can access this panel via its **ID** attribute.

Either right-click on the **Default.aspx** file and select **View Code**, or expand its tree and double-click on the **Default.aspx.cs** file. Edit the file as shown:

CODE TO TYPE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace FormAuthentication1
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (Request.IsAuthenticated)
            {
                WelcomeBackMessage.Text = "You are Logged In, " + User.Identity.Name;
                LoggedInMessagePanel.Visible = true;
                NotLoggedInMessagePanel.Visible = false;
            }
            else
            {
                LoggedInMessagePanel.Visible = false;
                NotLoggedInMessagePanel.Visible = true;
            }
        }
    }
}
```



Log in using the link on the page, not at the upper-right corner of the app. You'll have to register from the login page first. After registering and logging in, you'll be returned to the main page. When the **Log In** link is gone and you get a welcome message, stop debugging:

OBSERVE: Default.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

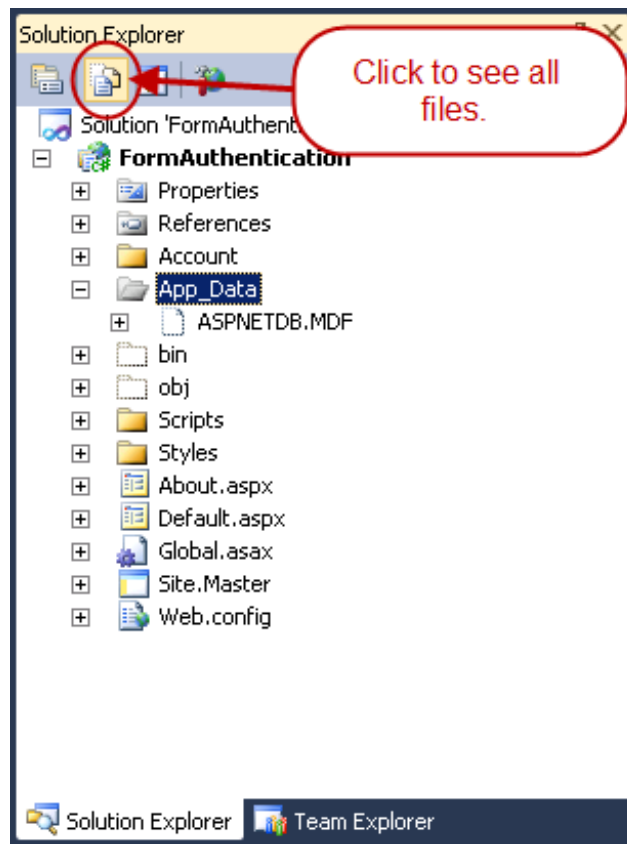
namespace FormAuthentication1
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (Request.IsAuthenticated)
            {
                WelcomeBackMessage.Text = "You are Logged In, " + User.Identity.Name;
                LoggedInMessagePanel.Visible = true;
                NotLoggedInMessagePanel.Visible = false;
            }
            else
            {
                LoggedInMessagePanel.Visible = false;
                NotLoggedInMessagePanel.Visible = true;
            }
        }
    }
}
```

As you saw when you ran the application initially, the **LoggedInMessagePanel** was set to be invisible by the code-behind file because our **Request** object was not authenticated. After you logged in, the page was reloaded and we got our welcome message, "**You are Logged In, username**". When the page initially loads, or reloads, the code-behind file's **Page_Load()** method runs its logic, and checks to determine whether we are logged in. Then it sets the panels' **Visible** properties to the correct values.

Note The **User** class can be used to get information about the currently logged-in user.

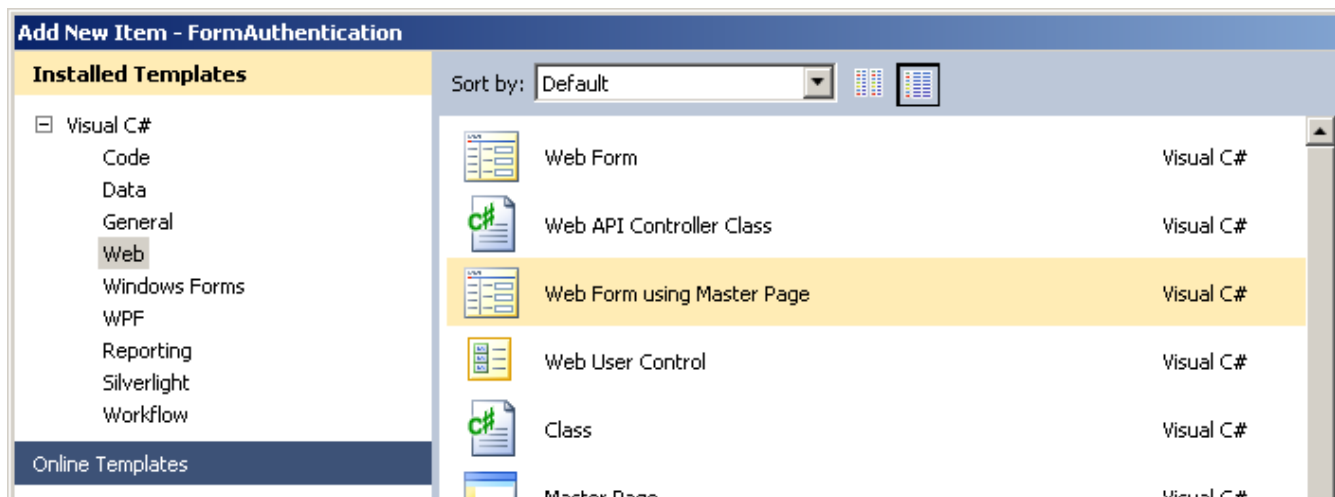
Membership and Roles

ASP.NET will create a database for **Membership** (User Accounts) in the **App_Data** directory. The contents of this directory are hidden by default. However, by clicking on the **Solution Explorer**'s **Show All Files** button, you can reveal the hidden files.



Now let's talk about Membership, User Accounts, and Roles.

The data for user accounts and roles is stored in the **/App_Data/aspnetdb.mdf** file. Let's create a folder named **Admin** in your project. Inside the **Admin** folder, create three files named **CreateUsers.aspx**, **ManageRoles.aspx**, and **ManageUsersWithRoles.aspx**. Use the **Web Form using Master Page** template to create the files. We'll use these files to create users and roles for our web app.



Modify your **ManageRoles.aspx** file as shown:

CODE TO TYPE: ManageRoles.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="ManageRoles.aspx.cs" Inherits="FormAuthentication.Admin.ManageRoles" %>

<asp:Content ID="Content1" ContentPlaceHolderID="HeadContent" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <asp:Panel ID="RoleGridPanel" runat="server">
        <b>Current Roles</b>
        <asp:GridView ID="CurrentRolesGrid" runat="server" AutoGenerateColumns="false"
OnRowDeleting="CurrentRolesGrid_RowDeleting">
            <Columns>
                <asp:TemplateField HeaderText="Role">
                    <ItemTemplate>
                        <asp:Label runat="server" ID="RoleNameLabel" Text='<%# Containe
r.DataItem %>' />
                    </ItemTemplate>
                </asp:TemplateField>
                <asp:CommandField DeleteText="Delete" ShowDeleteButton="True" />
            </Columns>
        </asp:GridView>
    </asp:Panel>
    <br />
    <asp:Panel ID="CreateRolePanel" runat="server">
        <b>Create a New Role:</b>
        <asp:TextBox ID="RoleNameTextBox" runat="server"></asp:TextBox>
        <asp:Button ID="CreateRoleButton" runat="server" Text="Create Role" OnClick="Cr
eateRoleButton_Click" />
    </asp:Panel>
</asp:Content>
```



but don't run it. We still need to do some work on the code-behind file.

OBSERVE: ManageRoles.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="ManageRoles.aspx.cs" Inherits="FormAuthentication.Admin.ManageRoles" %>

<asp:Content ID="Content1" ContentPlaceHolderID="HeadContent" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <asp:Panel ID="RoleGridPanel" runat="server">
        <b>Current Roles</b>
        <asp:GridView ID="CurrentRolesGrid" runat="server" AutoGenerateColumns="false"
OnRowDeleting="CurrentRolesGrid_RowDeleting">
            <Columns>
                <asp:TemplateField HeaderText="Role">
                    <ItemTemplate>
                        <asp:Label runat="server" ID="lblRoleName" Text='<%# Container.Data
Item %>' />
                    </ItemTemplate>
                </asp:TemplateField>
                <asp:CommandField DeleteText="Delete" ShowDeleteButton="True" />
            </Columns>
        </asp:GridView>
    </asp:Panel>
    <br />
    <asp:Panel ID="CreateRolePanel" runat="server">
        <b>Create a New Role:</b>
        <asp:TextBox ID="RoleNameTextBox" runat="server"></asp:TextBox>
        <asp:Button ID="CreateRoleButton" runat="server" Text="Create Role" OnClick="Cr
eateRoleButton_Click" />
    </asp:Panel>
</asp:Content>
```

Here we put an ASP.NET GridView control onto our page. We don't want it to **AutoGenerateColumns** because we want the column we are going to show to have a meaningful name, rather than the word **Item**. The **TemplateField** we're creating contains two columns. The first column will be our Role; the second column will contain a delete link so that we can remove roles as needed. Notice that we used `<%# Container.DataItem %>` as our label's text. In our code-behind file, we'll bind the GridView to the roles table in the database. `<%# Container.DataItem %>` will populate this column's label with the role name automatically. In the GridView's declaration, we indicate that when a role is deleted, we want our code-behind files, **CurrentRolesGrid_RowDeleting** method to run. This is how we link events in ASP.NET. You can go to the design view and click on the GridView control and access the events in the properties window as well.

Note The GridView class is powerful and its full use is beyond the scope of this lesson. However, if you want to learn more about it on your own, see the MSDN article on the [Gridview Class](#).

Okay, now, let's look at the code-behind file for our **ManageRoles** page. Modify your **ManageRoles.aspx.cs** code-behind file as shown:

CODE TO TYPE: ManageRoles.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Security;

namespace FormAuthentication.Admin
{
    public partial class ManageRoles : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!Page.IsPostBack)
                DisplayRolesInGrid();
        }

        protected void CreateRoleButton_Click(object sender, EventArgs e)
        {
            string newRoleName = RoleNameTextBox.Text.Trim();
            if (!Roles.RoleExists(newRoleName))
            {
                Roles.CreateRole(newRoleName);
                DisplayRolesInGrid();
            }
            RoleNameTextBox.Text = string.Empty;
        }

        private void DisplayRolesInGrid()
        {
            CurrentRolesGrid.DataSource = Roles.GetAllRoles();
            CurrentRolesGrid.DataBind();
        }

        protected void CurrentRolesGrid_RowDeleting(object sender, GridViewDeleteEventArgs e)
        {
            Label lblRoleName = CurrentRolesGrid.Rows[e.RowIndex].FindControl("lblRoleName") as Label;
            Roles.DeleteRole(lblRoleName.Text, false);
            DisplayRolesInGrid();
        }
    }
}
```



Save the file and start debugging. If your application does not start at the

http://localhost:xxxxx/Admin/ManageRoles.aspx page, navigate to it. Create two roles, **Admin** and **Member**. Later, we'll create users and put them in these roles.

Let's take a look at this, method by method, starting with the **Page_Load()** method. Here, we are telling our program that, if this is the initial load of the page, display what is in the roles database. (If we are posting this page back from one of our methods, they will handle this for us):

OBSERVE: ManageRoles.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Security;

namespace FormAuthentication.Admin
{
    public partial class ManageRoles : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!Page.IsPostBack)
                DisplayRolesInGrid();
        }

        protected void CreateRoleButton_Click(object sender, EventArgs e)
        {
            string newRoleName = RoleNameTextBox.Text.Trim();
            if (!Roles.RoleExists(newRoleName))
            {
                Roles.CreateRole(newRoleName);
                DisplayRolesInGrid();
            }
            RoleNameTextBox.Text = string.Empty;
        }

        private void DisplayRolesInGrid()
        {
            CurrentRolesGrid.DataSource = Roles.GetAllRoles();
            CurrentRolesGrid.DataBind();
        }

        protected void CurrentRolesGrid_RowDeleting(object sender, GridViewDeleteEventArgs e)
        {
            Label lblRoleName = CurrentRolesGrid.Rows[e.RowIndex].FindControl("lblRoleName") as Label;
            Roles.DeleteRole(lblRoleName.Text, false);
            DisplayRolesInGrid();
        }
    }
}
```

The **CreateRoleButton_Click()** method is an event handler. When the **CreateRoleButton** on the **ManageRoles.aspx** page is clicked, this method will run. We will get the **newRoleName** from the **RoleNameTextBox** on the .aspx page and remove any spaces on the ends. If that role name does not exist already, we'll enter it into the roles table in our database, and post the page back with the new data. If the role already exists, we clear the RoleNameTextBox.

In the **DisplayRolesInGrid()** method, we use the ASP.NET **System.Web.Security.Roles** class to get all of the roles from the database. Then we bind that data to our **CurrentRolesGrid**. This causes the grid to be refreshed.

Finally, the **CurrentRolesGrid_RowDeleting()** method is run when the **CurrentRolesGrid** fires a deleting event, when we click the delete button beside a role. We are using the **FindControl()** method to find out which row in the grid is being deleted. **FindControl** can be tricky because it looks only in the control from which it is called, so nesting calls

might be necessary. Once we find the label, we can get its text and then delete the role. Then we call **DisplayRolesInGrid()** to rebind the data and show the new roles, less our deletion.

Note The **Roles** class can be used to get and modify information about the roles set in the database.

Now let's take a look at how we can create users with roles. Modify your **CreateUsers.aspx** file as shown:

CODE TO TYPE: CreateUsers.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="CreateUsers.aspx.cs" Inherits="FormAuthentication.Admin.CreateUsers" %>

<asp:Content ID="Content1" ContentPlaceHolderID="HeadContent" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <asp:Panel ID="CreateUserWizardPanel" runat="server">
        <asp:CreateUserWizard ID="CreateUserWithRoles" runat="server" ContinueDestinati
onPageUrl="~/Default.aspx"
            LoginCreatedUser="False" OnActiveStepChanged="CreateUserWithRoles_ActiveSte
pChanged">
            <WizardSteps>
                <asp:CreateUserWizardStep ID="CreateUserWithRolesWizardStep" runat="ser
ver">
                </asp:CreateUserWizardStep>
                <asp:WizardStep ID="RoleStep" runat="server" StepType="Step" Title="Rol
es" AllowReturn="False">
                    <asp:CheckBoxList ID="RoleCheckBoxList" runat="server">
                    </asp:CheckBoxList>
                </asp:WizardStep>
                <asp:CompleteWizardStep ID="CompleteWizardStep" runat="server">
                </asp:CompleteWizardStep>
            </WizardSteps>
        </asp:CreateUserWizard>
    </asp:Panel>
</asp:Content>
```



Save, but don't run it yet. We still need to modify the code-behind file:

OBSERVE: CreateUsers.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="CreateUsers.aspx.cs" Inherits="FormAuthentication.Admin.CreateUsers" %>

<asp:Content ID="Content1" ContentPlaceHolderID="HeadContent" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <asp:Panel ID="CreateUserWizardPanel" runat="server">
        <asp:CreateUserWizard ID="CreateUserWithRoles" runat="server" ContinueDestinati
onPageUrl="~/Default.aspx"
            LoginCreatedUser="False" OnActiveStepChanged="CreateUserWithRoles_ActiveSte
pChanged">
            <WizardSteps>
                <asp:CreateUserWizardStep ID="CreateUserWithRolesWizardStep" runat="ser
ver">
                    </asp:CreateUserWizardStep>
                    <asp:WizardStep ID="RoleStep" runat="server" StepType="Step" Title="Rol
es" AllowReturn="False">
                        <asp:CheckBoxList ID="RoleCheckBoxList" runat="server">
                        </asp:CheckBoxList>
                    </asp:WizardStep>
                    <asp:CompleteWizardStep ID="CompleteWizardStep" runat="server">
                    </asp:CompleteWizardStep>
                </WizardSteps>
            </asp:CreateUserWizard>
        </asp:Panel>
    </asp:Content>
```



Here, we create a wizard using the built-in wizard functions of Visual Studio and ASP.NET. We use the **CreateUserWizardStep** to create our user. Then we use **RoleStep** to select a role(s) for the user. We use **CompleteWizardStep** to finish the process. However, none of this will work correctly until we modify the code-behind file to link up the logic.

Modify the **CreateUsers.aspx.cs** code-behind file as shown:

CODE TO TYPE: CreateUsers.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Security;

namespace FormAuthentication.Admin
{
    public partial class CreateUsers : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!Page.IsPostBack)
            {
                WizardStep RolesStep = CreateUserWithRoles.FindControl("RoleStep") as WizardStep;
                CheckBoxList RoleList = RolesStep.FindControl("RoleCheckBoxList") as CheckBoxList;
                RoleList.DataSource = Roles.GetAllRoles();
                RoleList.DataBind();
            }
        }
        protected void CreateUserWithRoles_ActiveStepChanged(object sender, EventArgs e)
        {
            if (CreateUserWithRoles.ActiveStep.Title == "Complete")
            {
                WizardStep RoleStep = CreateUserWithRoles.FindControl("RoleStep") as WizardStep;
                CheckBoxList RoleCheckBoxList = RoleStep.FindControl("RoleCheckBoxList") as CheckBoxList;
                foreach (ListItem item in RoleCheckBoxList.Items)
                {
                    if (item.Selected)
                    {
                        Roles.AddUserToRole(CreateUserWithRoles.UserName, item.Text);
                    }
                }
            }
        }
    }
}
```

 and . If the browser does not start up on the CreateUsers page, navigate to the page **<http://localhost:xxxxx/Admin/CreateUsers.aspx>**. Create a new user named **Admin**. You can use any email address, but make sure to give the user the password of **Admin123**. This way, your mentor will have the ability to get into the site without having to modify the **Web.config** file, which we'll add at the end of this lesson.

Note

Admin users should be members of **All** roles, unless there is some reason you don't want them to view those resources.

OBSERVE: CreateUsers.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Security;

namespace FormAuthentication.Admin
{
    public partial class CreateUsers : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!Page.IsPostBack)
            {
                WizardStep RolesStep = CreateUserWithRoles.FindControl("RoleStep") as WizardStep;
                CheckBoxList RoleList = RolesStep.FindControl("RoleCheckBoxList") as CheckBoxList;
                RoleList.DataSource = Roles.GetAllRoles();
                RoleList.DataBind();
            }
        }
        protected void CreateUserWithRoles_ActiveStepChanged(object sender, EventArgs e)
        {
            if (CreateUserWithRoles.ActiveStep.Title == "Complete")
            {
                WizardStep RoleStep = CreateUserWithRoles.FindControl("RolesStep") as WizardStep;
                CheckBoxList RoleCheckBoxList = RoleStep.FindControl("RoleCheckBoxList") as CheckBoxList;
                foreach (ListItem item in RoleCheckBoxList.Items)
                {
                    if (item.Selected)
                    {
                        Roles.AddUserToRole(CreateUserWithRoles.UserName, item.Text);
                    }
                }
            }
        }
    }
}
```

The **Page_Load()** method is similar to our last listing. If we are not posting back from one of our methods, we just fill our RoleCheckBoxList with the roles that are already in the database.

The meat of this class is the **CreateUserWithRoles_ActiveStepChanged()** method. Here, we check to see if we have reached the "Complete" step, meaning that the user has been created and all of the roles that we want them to have, have been selected. The "Complete" step is a built-in feature of the Wizard. Once we get there, we can iterate through the **RoleCheckBoxList** and assign the checked roles to our user.

The next page we work with allows us to add and remove roles for specific users. Modify your **ManageUsersWithRoles.aspx** file as shown:

CODE TO TYPE: ManageUsersWithRoles.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="ManageUsersWithRoles.aspx.cs" Inherits="FormAuthentication.Admin.Manage
UsersWithRoles" %>

<asp:Content ID="Content1" ContentPlaceHolderID="HeadContent" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <asp:Panel ID="UserListDropDownPanel" runat="server">
        <b>Select User:</b>
        <asp:DropDownList ID="UserListDropDown" runat="server" AutoPostBack="True" Data
TextField="UserName"
            DataValueField="UserName" OnSelectedIndexChanged="UserDropDownList_Selected
IndexChanged">
        </asp:DropDownList>
    </asp:Panel>
    <br />
    <asp:Panel ID="UsersRoleListPanel" runat="server">
        <asp:Repeater ID="UsersRoleList" runat="server">
            <ItemTemplate>
                <asp:CheckBox runat="server" ID="RoleCheckBox" AutoPostBack="true" Text
='<%# Container.DataItem %>'
                    OnCheckedChanged="RoleCheckBox_CheckChanged" />
                <br />
            </ItemTemplate>
        </asp:Repeater>
    </asp:Panel>
    <br />
    <asp:Panel ID="RoleListDropDownPanel" runat="server">
    </asp:Panel>
    <asp:DropDownList ID="RoleDropDownList" runat="server" AutoPostBack="true" OnSelect
edIndexChanged="RoleDropDownList_SelectedIndexChanged">
    </asp:DropDownList>
    <br />
    <asp:Panel ID="RolesUserGridPanel" runat="server">
        <asp:GridView ID="RolesUserGrid" runat="server" AutoGenerateColumns="false" Emp
tyDataText="There are no users in this role."
            OnRowDeleting="RolesUserGrid_RowDeleting">
            <Columns>
                <asp:TemplateField HeaderText="Users">
                    <ItemTemplate>
                        <asp:Label runat="server" ID="UserNameLabel" Text='<%# Containe
r.DataItem %>'></asp:Label>
                    </ItemTemplate>
                </asp:TemplateField>
            </Columns>
        </asp:GridView>
    </asp:Panel>
    <br />
    <asp:Panel ID="AddUserToRolePanel" runat="server">
        <b>UserName:</b>
        <asp:TextBox ID="AddUserToRole" runat="server"></asp:TextBox>
        <asp:Button ID="AddUserToRoleButton" runat="server" Text="Add User to Role" OnClick
="AddUserToRoleButton_Click" />
    </asp:Panel>
</asp:Content>
```



Save, but don't run it. We still need to modify the code-behind file:

OBSERVE: ManageUsersWithRoles.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="ManageUsersWithRoles.aspx.cs" Inherits="FormAuthentication.Admin.Manage
    UsersWithRoles" %>

<asp:Content ID="Content1" ContentPlaceHolderID="HeadContent" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <asp:Panel ID="UserListDropDownPanel" runat="server">
        <b>Select User:</b>
        <asp:DropDownList ID="UserListDropDown" runat="server" AutoPostBack="True" Data
        TextField="UserName"
            DataValueField="UserName" OnSelectedIndexChanged="UserDropDownList_Selected
        IndexChanged">
        </asp:DropDownList>
    </asp:Panel>
    <br />
    <asp:Panel ID="UsersRoleListPanel" runat="server">
        <asp:Repeater ID="UsersRoleList" runat="server">
            <ItemTemplate>
                <asp:CheckBox runat="server" ID="RoleCheckBox" AutoPostBack="true" Text
                ='<%# Container.DataItem %>'
                    OnCheckedChanged="RoleCheckBox_CheckChanged" />
                <br />
            </ItemTemplate>
        </asp:Repeater>
    </asp:Panel>
    <br />
    <asp:Panel ID="RoleListDropDownPanel" runat="server">
        <asp:DropDownList ID="RoleDropDownList" runat="server" AutoPostBack="true" OnSe
        lectedIndexChanged="RoleDropDownList_SelectedIndexChanged">
        </asp:DropDownList>
    </asp:Panel>
    <br />
    <asp:Panel ID="RolesUserGridPanel" runat="server">
        <asp:GridView ID="RolesUserGrid" runat="server" AutoGenerateColumns="false" Emp
        tyDataText="There are no users in this role."
            OnRowDeleting="RolesUserGrid_RowDeleting">
            <Columns>
                <asp:TemplateField HeaderText="Users">
                    <ItemTemplate>
                        <asp:Label runat="server" ID="UserNameLabel" Text='<%# Containe
                        r.DataItem %>'></asp:Label>
                    </ItemTemplate>
                </asp:TemplateField>
            </Columns>
        </asp:GridView>
    </asp:Panel>
    <br />
    <asp:Panel ID="AddUserToRolePanel" runat="server">
        <b>UserName:</b>
        <asp:TextBox ID="AddUserToRole" runat="server"></asp:TextBox>
        <asp:Button ID="AddUserToRoleButton" runat="server" Text="Add User to Role" OnClick
        ="AddUserToRoleButton_Click" />
    </asp:Panel>
</asp:Content>
```

It may seem like we're doing a lot here, but most of it is wrapping panel objects. Our **UserDropDownList** control will be populated by the code-behind file. It will allow us to select a user from our user accounts. When the selection changes, the **UserDropDownList_SelectedIndexChanged** method in our code-behind file will be executed.

The **UsersRoleList** repeater contains our **RoleCheckBox** items. This list of checkboxes will be populated by our code-behind file. We are using the same method as before by getting this item from the **<%# Container.DataItem %>**. When a checkbox is checked or unchecked, the **RoleCheckBox_CheckChanged()** method is run from our code-behind file.

Similar to the **UserListDropDown** control above, the **RoleDropDownList** control will be populated by the code-

behind file; we will be able to select roles to see what users are in that role, using the **RolesUserGrid** control.

Finally, we have our **AddUserToRole** text box and our **AddUserToRoleButton** which allow us to add a user to the selected role.

Now, let's put together the code-behind file. Modify your **ManageUsersWithRoles.aspx.cs** file as shown:

CODE TO TYPE: ManageUsersWithRoles.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Security;

namespace FormAuthentication.Admin
{
    public partial class ManageUsersWithRoles : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!Page.IsPostBack)
            {
                BindUsersToUserList();
                BindRolesToList();
                CheckForUserRoleSelection();
                UserIsInRole();
            }
        }

        private void BindUsersToUserList()
        {
            MembershipUserCollection users = Membership.GetAllUsers();
            UserListDropDown.DataSource = users;
            UserListDropDown.DataBind();
        }

        private void BindRolesToList()
        {
            string[] roles = Roles.GetAllRoles();
            UsersRoleList.DataSource = roles;
            UsersRoleList.DataBind();
            RoleDropDownList.DataSource = roles;
            RoleDropDownList.DataBind();
        }

        private void CheckForUserRoleSelection()
        {
            string userNameSelection = UserListDropDown.SelectedValue;
            string[] userRolesSelection = Roles.GetRolesForUser(userNameSelection);
            foreach (RepeaterItem item in UsersRoleList.Items)
            {
                CheckBox RoleCheckBox = item.FindControl("RoleCheckBox") as CheckBox;
                if (userRolesSelection.Contains<string>(RoleCheckBox.Text))
                    RoleCheckBox.Checked = true;
                else
                    RoleCheckBox.Checked = false;
            }
        }

        protected void UserDropDownList_SelectedIndexChanged(object sender, EventArgs e)
        {
            CheckForUserRoleSelection();
        }

        protected void RoleCheckBox_CheckChanged(object sender, EventArgs e)
        {
            CheckBox RoleCheckBox = sender as CheckBox;
            string selectedUser = UserListDropDown.SelectedValue;
            string roleName = RoleCheckBox.Text;
            if (RoleCheckBox.Checked)
                Roles.AddUserToRole(selectedUser, roleName);
        }
    }
}
```

```

        else
            Roles.RemoveUserFromRole(selectedUser, roleName);
        UserIsInRole();
    }

    private void UserIsInRole()
    {
        string selectedRoleName = RoleDropDownList.SelectedValue;
        string[] userInRole = Roles.GetUsersInRole(selectedRoleName);
        RolesUserGrid.DataSource = userInRole;
        RolesUserGrid.DataBind();
    }

    protected void RoleDropDownList_SelectedIndexChanged(object sender, EventArgs e)
    {
        UserIsInRole();
    }

    protected void RolesUserGrid_RowDeleting(object sender, GridViewDeleteEventArgs e)
    {
        string selectedRoleName = RoleDropDownList.SelectedValue;
        Label UserNameLabel = RolesUserGrid.Rows[e.RowIndex].FindControl("UserNameLabel") as Label;
        Roles.RemoveUserFromRole(UserNameLabel.Text, selectedRoleName);
        UserIsInRole();
        CheckForUserRoleSelection();
    }

    protected void AddUserToRoleButton_Click(object sender, EventArgs e)
    {
        string selectedRoleName = RoleDropDownList.SelectedValue;
        string userNameToAddToRole = AddUserToRole.Text;
        if (userNameToAddToRole.Trim().Length == 0)
        {
            return;
        }
        MembershipUser userInfo = Membership.GetUser(userNameToAddToRole);
        if (userInfo == null)
        {
            return;
        }
        if (Roles.IsUserInRole(userNameToAddToRole, selectedRoleName))
        {
            return;
        }
        Roles.AddUserToRole(userNameToAddToRole, selectedRoleName);
        AddUserToRole.Text = string.Empty;
        UserIsInRole();
        CheckForUserRoleSelection();
    }
}

```

 and . Add a user named **Member** with password **Member123** and place them in the **Member** role.

Let's take it method by method. The Page_Load method is familiar. It displays the initial information for our page, if it is not a post back:

OBSERVE: ManageUsersWithRoles.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Security;

namespace FormAuthentication.Admin
{
    public partial class ManageUsersWithRoles : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!Page.IsPostBack)
            {
                BindUsersToUserList();
                BindRolesToList();
                CheckForUserRoleSelection();
                UserIsInRole();
            }
        }

        private void BindUsersToUserList()
        {
            MembershipUserCollection users = Membership.GetAllUsers();
            UserListDropDown.DataSource = users;
            UserListDropDown.DataBind();
        }

        private void BindRolesToList()
        {
            string[] roles = Roles.GetAllRoles();
            UsersRoleList.DataSource = roles;
            UsersRoleList.DataBind();
            RoleDropDownList.DataSource = roles;
            RoleDropDownList.DataBind();
        }

        private void CheckForUserRoleSelection()
        {
            string userNameSelection = UserListDropDown.SelectedValue;
            string[] userRolesSelection = Roles.GetRolesForUser(userNameSelection);
            foreach (RepeaterItem item in UsersRoleList.Items)
            {
                CheckBox RoleCheckBox = item.FindControl("RoleCheckBox") as CheckBox;
                if (userRolesSelection.Contains<string>(RoleCheckBox.Text))
                {
                    RoleCheckBox.Checked = true;
                }
                else
                {
                    RoleCheckBox.Checked = false;
                }
            }
        }

        protected void UserDropDownList_SelectedIndexChanged(object sender, EventArgs e)
        {
            CheckForUserRoleSelection();
        }

        protected void RoleCheckBox_CheckChanged(object sender, EventArgs e)
        {
            CheckBox RoleCheckBox = sender as CheckBox;
            string selectedUser = UserListDropDown.SelectedValue;
            string roleName = RoleCheckBox.Text;
            if (RoleCheckBox.Checked)
            {
                Roles.AddUserToRole(selectedUser, roleName);
            }
        }
    }
}
```

```

        else
            Roles.RemoveUserFromRole(selectedUser, roleName);
        UserIsInRole();
    }

    private void UserIsInRole()
    {
        string selectedRoleName = RoleDropDownList.SelectedValue;
        string[] userInRole = Roles.GetUsersInRole(selectedRoleName);
        RolesUserGrid.DataSource = userInRole;
        RolesUserGrid.DataBind();
    }

    protected void RoleDropDownList_SelectedIndexChanged(object sender, EventArgs e)
    {
        UserIsInRole();
    }

    protected void RolesUserGrid_RowDeleting(object sender, GridViewDeleteEventArgs e)
    {
        string selectedRoleName = RoleDropDownList.SelectedValue;
        Label UserNameLabel = RolesUserGrid.Rows[e.RowIndex].FindControl("UserNameLabel") as Label;
        Roles.RemoveUserFromRole(UserNameLabel.Text, selectedRoleName);
        UserIsInRole();
        CheckForUserRoleSelection();
    }

    protected void AddUserToRoleButton_Click(object sender, EventArgs e)
    {
        string selectedRoleName = RoleDropDownList.SelectedValue;
        string userNameToAddToRole = AddUserToRole.Text;
        if (userNameToAddToRole.Trim().Length == 0)
        {
            return;
        }
        MembershipUser userInfo = Membership.GetUser(userNameToAddToRole);
        if (userInfo == null)
        {
            return;
        }
        if (Roles.IsUserInRole(userNameToAddToRole, selectedRoleName))
        {
            return;
        }
        Roles.AddUserToRole(userNameToAddToRole, selectedRoleName);
        AddUserToRole.Text = string.Empty;
        UserIsInRole();
        CheckForUserRoleSelection();
    }
}

```

The **BindUsersToList()** method gets all users from the database and binds them to the **UserListDropDown** control.

The **BindRolesToList()** does the same for roles, however, it binds them to two controls: the **UsersRoleList** and the **RoleDropDownList**.

The **CheckForUserSelection** method looks at the **UserListDropDown** control to see which user is selected and then gets all of the roles of which the user is a member. It looks through all of the items in the **UserRoleList** and marks the roles the user is in as "selected," and makes sure that any role that the user is not in is marked as "not selected."

The **UserDropDownList_SelectedIndexChanged()** event handler calls the **CheckForUserRoleSelection()** method any time the user is changed on the **UserDropDownList** control.

The **RoleCheckBox_CheckedChanged** event handler adds or removes a user from a role based on the status of the **UserListDropDown** and the **RoleCheckBox**. Then it calls the **UserIsInRole()** method to update the **RolesUserGrid** control.

The **RoleDropDownList_SelectedIndexChanged** event handler calls **UserIsInRole()** to update the page showing which users are in this role when the drop-down list selection changes.

The **RolesUserGrid_RowDeleting()** event handler determines which role we are working with and then removes the user listed in that row of the grid from the role. Then it calls **UserIsInRole** and **CheckForUserRoleSelection** to update the page.

Finally, the **AddUserToRoleButton_Click()** event handler adds a user to the selected role, if the user exists in the database. This method and its associated control is useful if you know the name of the user and don't want to use the drop-down list.

Note The **Membership** class can be used to get and set information about the users in the database.

Authorization

At the beginning of this lesson, we edited a file named **Web.config**. There is one main **Web.config** file for the entire site, located in the project's root directory. However, each subdirectory can have its own **Web.config** as well. While the main **Web.config** file manages the entire site, the subdirectory **Web.config** files manage access to that particular directory and the files therein. There are two ways of controlling access to files in an ASP.NET site using the **Web.config** files. You can control everything from the main **Web.config** file or you can delegate access to the subdirectory **Web.config** files.

Let's look at how to control access from the main site's **Web.config** file first.

Note The following file is **NOT** from our project. It is just for example purposes.

OBSERVE: /Web.config

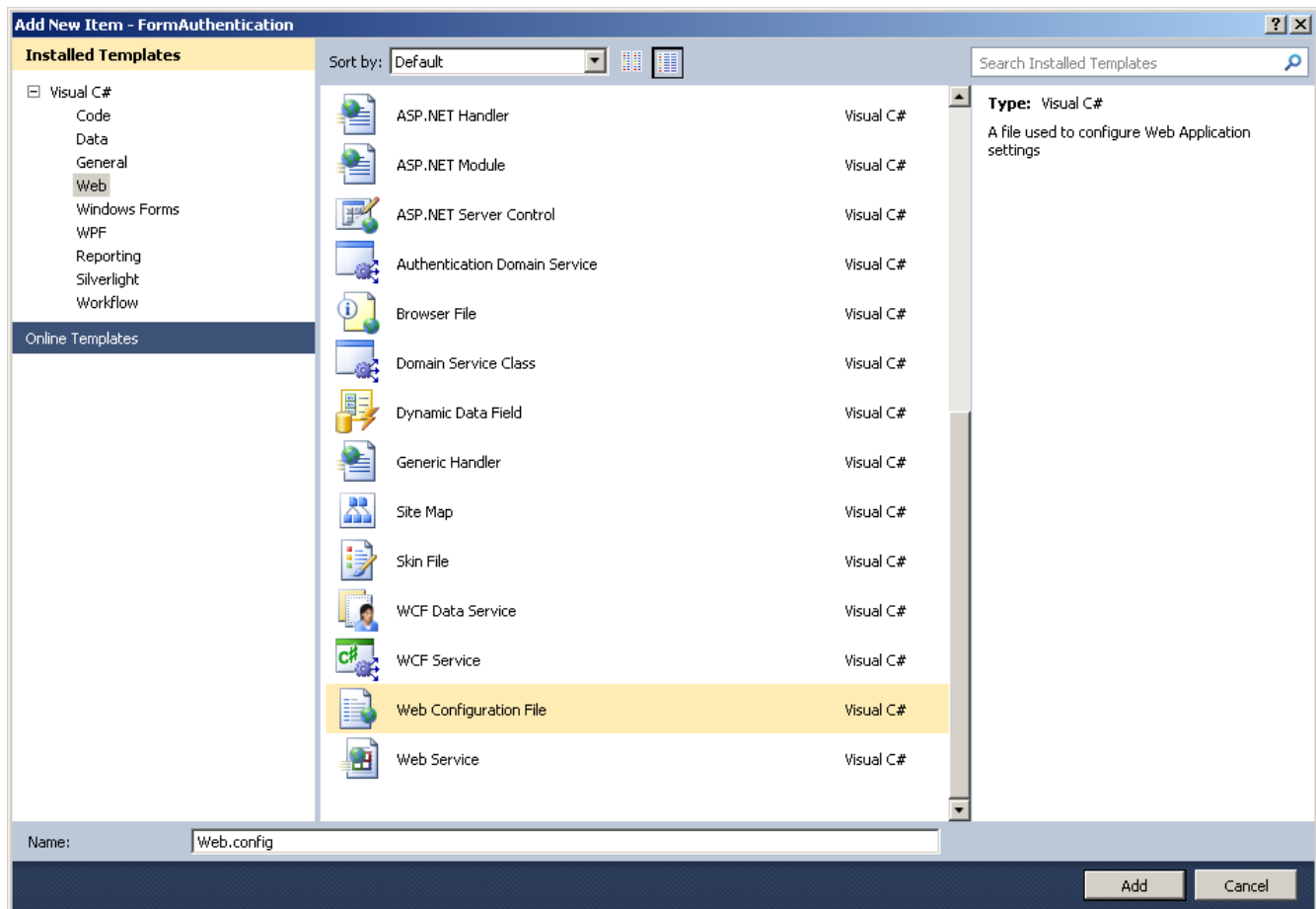
```
<configuration>
  <system.web>
    <authentication mode="Forms"/>
    <authorization> <deny users="?"/>
  </authorization>
</system.web>
  <location path="Account/register.aspx">
    <system.web>
      <authorization>
        <allow users="*" />
      </authorization>
    </system.web>
  </location>
</configuration>
```

In this **Web.config** file, the first **<system.web>** tag's authorization tag **denies access** to the entire site to all anonymous users. In the next section, we explicitly give access to all users to view the **Account/register.aspx** file. The **?** in the **<deny users="?" />** indicates all non-authenticated users. The ***** in the **<allow users="*" />** indicates *all users*.

Note We can control access to the site and to specific files and directories from the main **Web.config** file. Though on a large site, this file can become quite large. To modularize access rules, we can allow each directory to control access to itself and its own files.



Now, let's create a new **Web.config** file for our **Admin** directory. In the **Admin** directory, create a new file named **Web.config** as shown:

Note Make sure you have created the **Admin** user and put that user in the **Admin** role before doing this.



CODE TO TYPE: Admin\Web.config

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Admin"/>
      <deny users="*/>
    </authorization>
  </system.web>
</configuration>
```

 and . Log in as a non-admin user and attempt to go to the <http://localhost:xxxxx/Admin/CreateUsers.aspx> page. You are redirected to the **Login.aspx** page. Log in as **Admin** using the **Admin123** password. Try to go to the <http://localhost:xxxxx/Admin/CreateUsers.aspx> page again and note that you can now create a new user:

OBSERVE: Admin\Web.config

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Admin"/>
      <deny users="*/>
    </authorization>
  </system.web>
</configuration>
```

Note that the <system.web> section of the file is read top to bottom, so if you put the **deny users** tag before the **allow roles** tag, you won't ever get into that directory when running the site. You can allow multiple roles by adding them to the **allow roles** tag, separated by commas. You can **<allow users="UserName, UserName, UserName"/>**. The **"**"** character can be used to allow all users.

Note

For more information on the **allow** tag, see the MSDN article on [allow Element for authorization](#). Another good source on the **Web.config** file is this blog, [Setting authorization rules for a particular page or folder in web.config](#).

Tip

Keep [the ASP.NET](#) website open in a browser when working with ASP.NET projects. It's the definitive source for ASP.NET applications, and can provide much more information than this lesson can hope to provide.

As always, complete this lesson's homework before you move on to the next...

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Introduction to MVC and Razor

Lesson Objectives

In this lesson, you will:

- use the Model/View/Controller (MVC) design pattern and each of its components.
- use these components with one another to create a rich application.
- experiment to compare and contrast MVC with the Smart UI design pattern.
- use RouteConfig.cs file to add, manipulate and modify the route maps.
- use basic Razor syntax.

Defining MVC

Model

The Model is responsible for handling data from the application. It *implements the logic* for the applications data domain and is completely independent of the View and Controller.

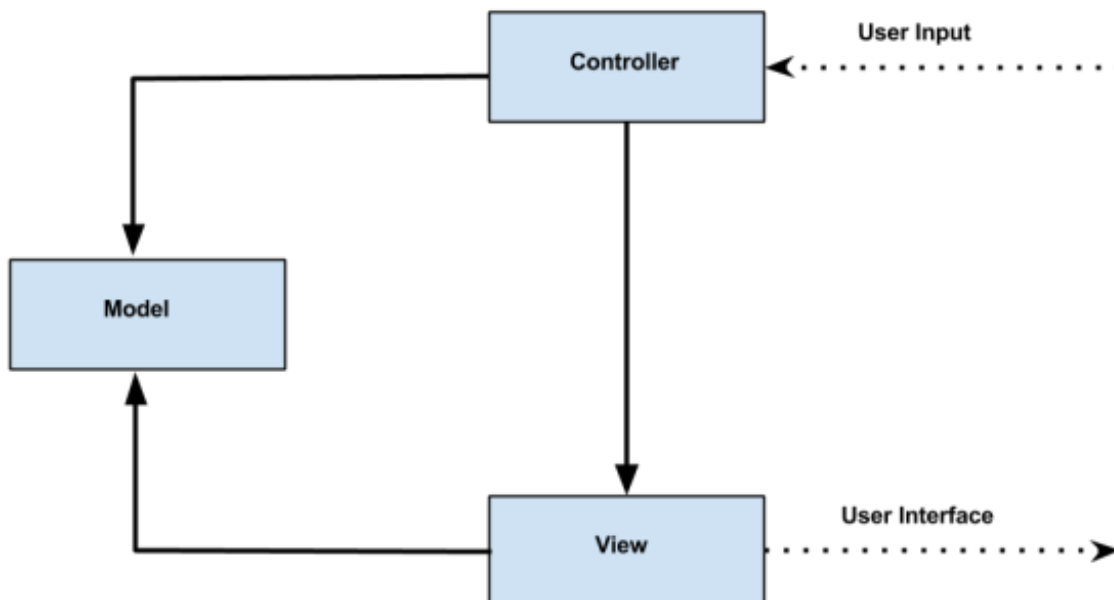
Controller

The *Controller translates the data* from the user (*server-side requests* and *HTTP requests*), whether it be mouse clicks or any other form of input. It informs the Model and/or View of the changes.

View

The View is responsible for *rendering the user interface* (browser window).

Model/View/Controller



Note

The View and Controller are dependent on the Model, but the Model is independent of both the View and the Controller.

MVC vs. Smart UI Design Pattern

The *Smart UI* design pattern is commonly referred to as the "anti-pattern," because the design pattern does not separate the logic from the interface. Essentially, it implements all of the business logic within the user interface and

performs updates by altering event handlers. It can be beneficial when you're creating a small application, because the implementation, user interface, and logic can be coded and designed quickly. However, if the application ever needs to expand, this method can become cumbersome and make it difficult to find bugs.

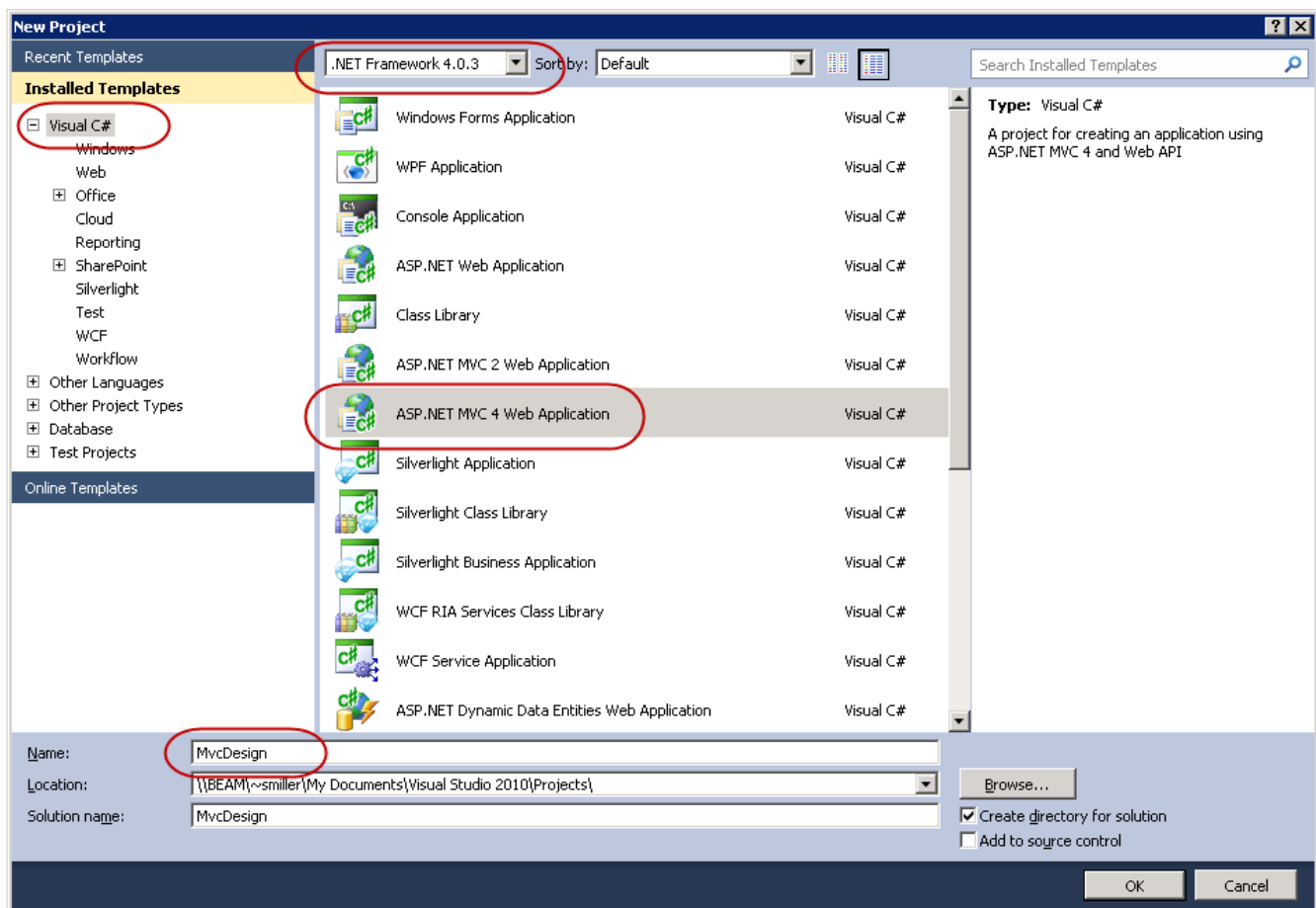
The MVC design pattern separates all instances and creates a logical pattern for communication between different objects. We have a View, which renders our user interface and notifies the Controller of any changes. The Controller performs the business logic that can be retrieved from the Model. The Controller then sends the updated information to the View to be rendered to the user. The Model is an "interface" where the variables for our database interactions and user input will be defined and retrieved.

So, the MVC design pattern is methodology that is implemented to allow for code reusability, and to allow for the creation of a readily expandable and modifiable code base. Debugging an MVC application is more straightforward since each component is kept separate; no component is encompassed within another component. With the Smart UI pattern, we lose code reusability and debugging takes longer because there is no separation of the individual components.

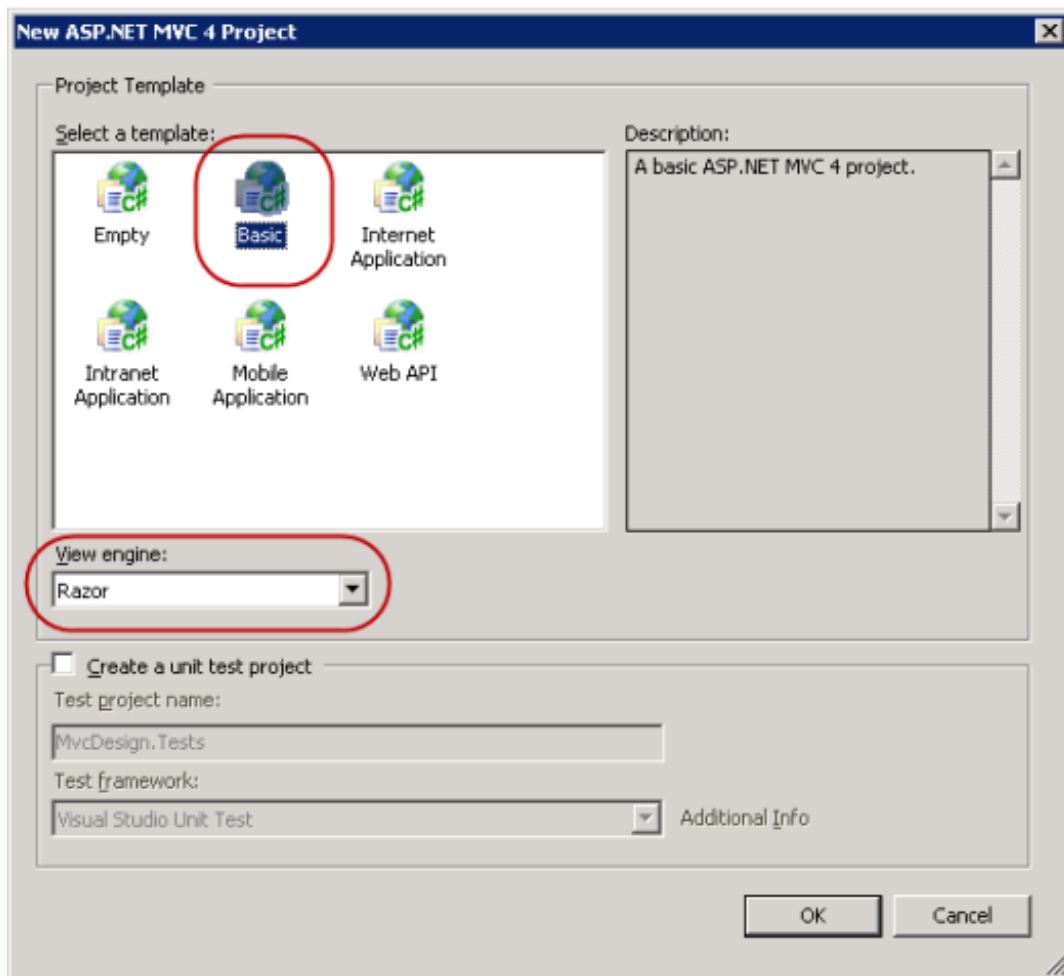
Creating a New Project

Create a new MVC C# 4 project. Click **File | New | Project**.

Select the **Visual C#** template and **ASP.NET MVC 4 Web Application**, and name your project **MvcDesign**:



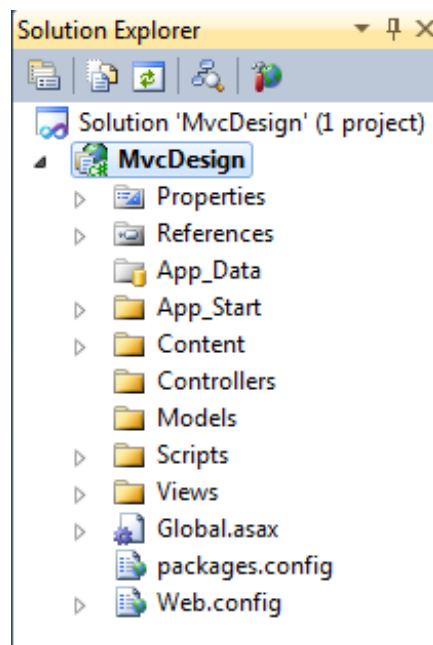
Click **OK**. Next, you're asked which type of application you want to create. Choose **Basic**. Make sure the **Razor** View engine is enabled:



Click **OK** to create the project.

Project Folders

Visual Studio creates some folders and files for us in Solution Explorer:

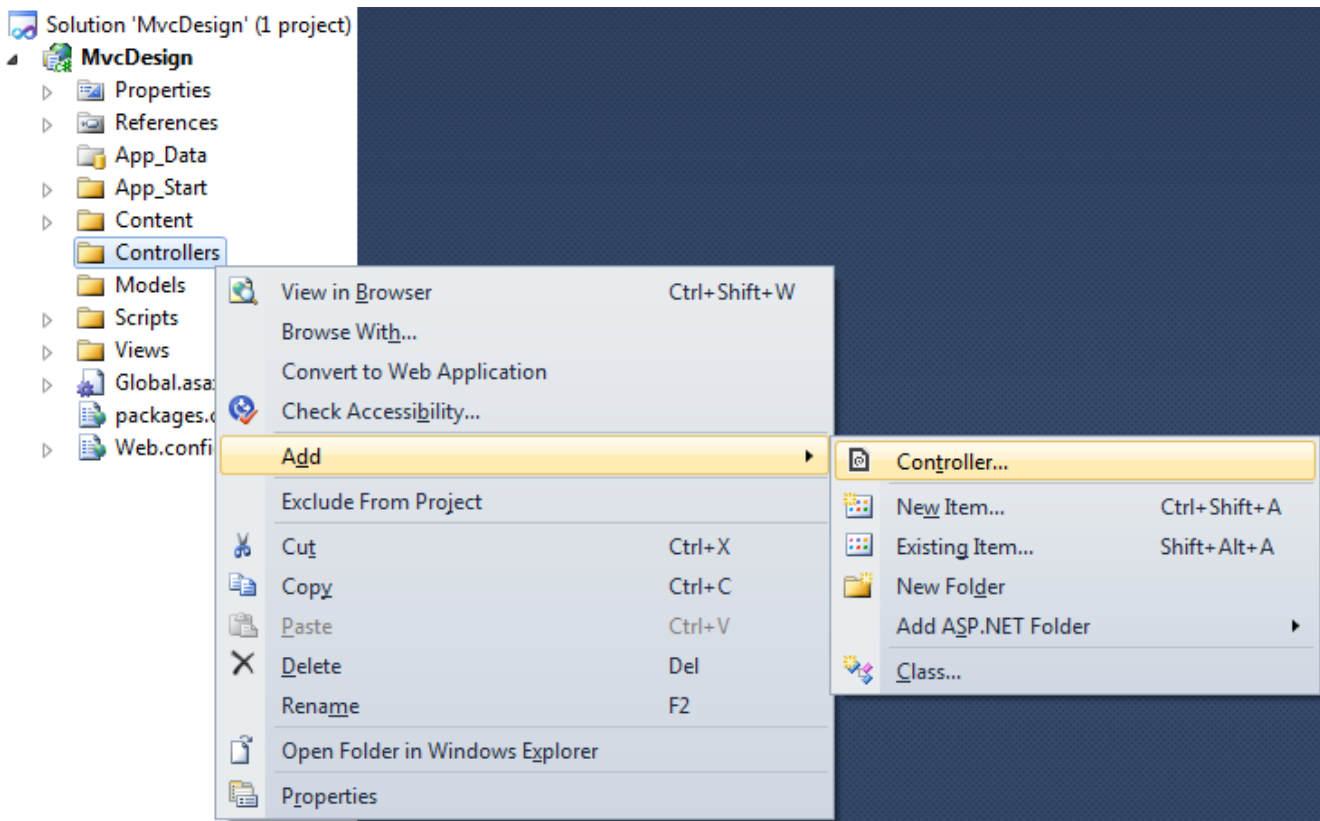


- **Properties Folder:** Contains the file **AssemblyInfo.cs** and has some general information regarding the assembly. In this file, you can change various default parameters pertaining to the build number, copyright information, company information, and so on.

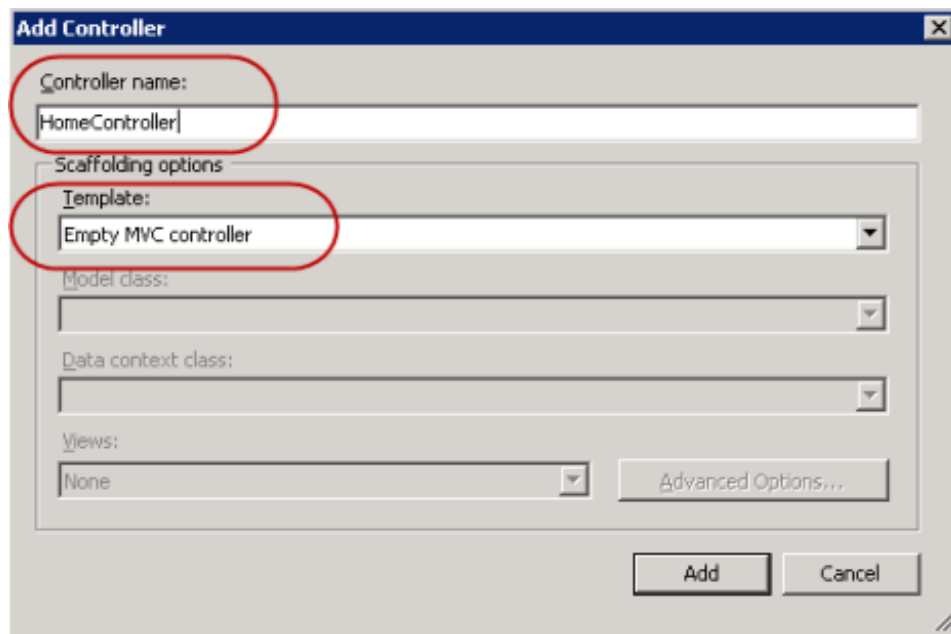
- **References Folder:** Contains all your C# API classes.
- **App_Start Folder:** Contains information about how the application loads various items at the start. It also defines the URL routes, and required .js files.
- **Content Folder:** Contains a default CSS file named Site.css, which the application references for its styling. It also contains an Images folder.
- **Controllers Folder:** Will contain the Controllers for your application.
- **Models Folder:** Will contain the Models for your application.
- **Views Folder:** Will contain the Views for your application.
- **Global.asax Folder:** Contains the **Global.asax.cs** file, which declares and handles application and session level events and objects.
- **Packages.config File:** An XML file that defines all the packages and dependencies needed to run your application.

The Controller

Right-click the **Controllers** folder and select **Add | Controller...**



In the Add Controller dialog, change the name from **Default1Controller** to **HomeController**, set the Template to **Empty MVC Controller**, and click **Add**:



Visual Studio creates a **HomeController.cs** file with some default code.

Note Visual Studio also adds a new folder and files to the Views folder.

Save and run. You see this message:

Server Error in '/' Application.

The view 'Index' or its master was not found or no view engine supports the searched locations. The following locations were searched:
~/Views/Home/Index.aspx
~/Views/Home/Index.ascx
~/Views/Shared/Index.aspx
~/Views/Shared/Index.ascx
~/Views/Home/Index.cshtml
~/Views/Home/Index.vbhtml
~/Views/Shared/Index.cshtml
~/Views/Shared/Index.vbhtml

This occurs because there is no View object for the HomeController to interact with. Modify **/Controllers/HomeController.cs** so it doesn't need a View object:


CODE TO TYPE:

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

namespace MvcDesign.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

        public ActionResult Index()
        public string Index()
        {
            return View();
            return "A Controller without a View!";
        }
    }
}
```



and . The controller returns the text.

When the application is loaded, it's set to load the /Home/ controller and Index method as defined in the **/App_Start/RouteConfig.cs** file:

OBSERVE: /App_Start/RouteConfig.cs

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

namespace MvcDesign
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id = UrlParamete
r.Optional }
            );
        }
    }
}
```

The **routes.MapRoute(...)** function defines how the application will handle URL paths. The **url:** line defines the default path for the application to interpret. The **defaults:** line defines the default controller action (in this case, **Home/Index/**), and an **Optional** parameter. The **Optional** parameter is generally an argument we want to pass to the controller for the View to display. This concept will be discussed in more depth a little later.

The View

Change **HomeController.cs** back to the way it was:

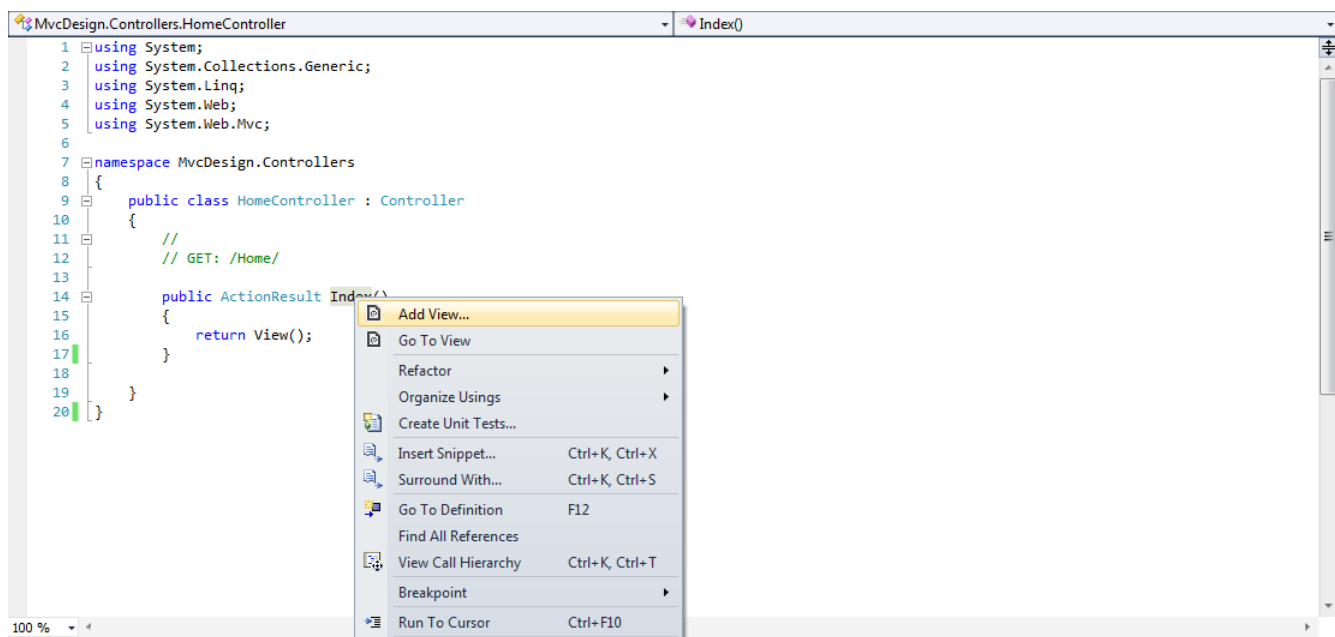
CODE TO TYPE:

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

namespace MvcDesign.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

        public string Index()
        public ActionResult Index()
        {
            return "A Controller without a View!";
            return View();
        }
    }
}
```

Now, create a View for the Index() method. Using Visual Studio, right-click on the Index() method and select **Add View...**:



Leave the defaults as they are for now:

Add View

View name:

View engine:
Razor (CSHTML) ▼

☐ Create a strongly-typed view

Model class:

Scaffold template:
Empty ▼ ☒ Reference script libraries

☐ Create as a partial view

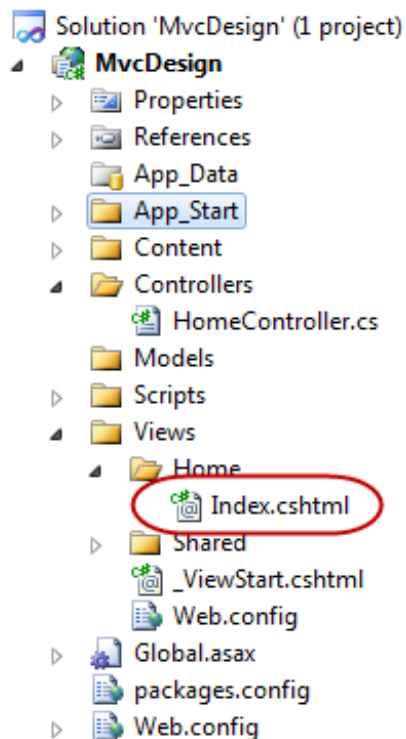
☒ Use a layout or master page:

...
(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

Add **Cancel**

Click **Add**. Visual Studio names the View after the Controller method, and opens the new **Index.cshtml** file that is stored in the **Views/Home** folder automatically. The folder is named after our Controller:



Now that we have a View to work with, let's modify the **/Views/Shared/_Layout.cshtml** file so we can begin to see the differences of our Controller methods by adding a navigation menu.

Note _Layout.cshtml persists through all Views. That's where we would make major changes to the design.

CODE TO TYPE: /Views/Shared/_Layout.cshtml

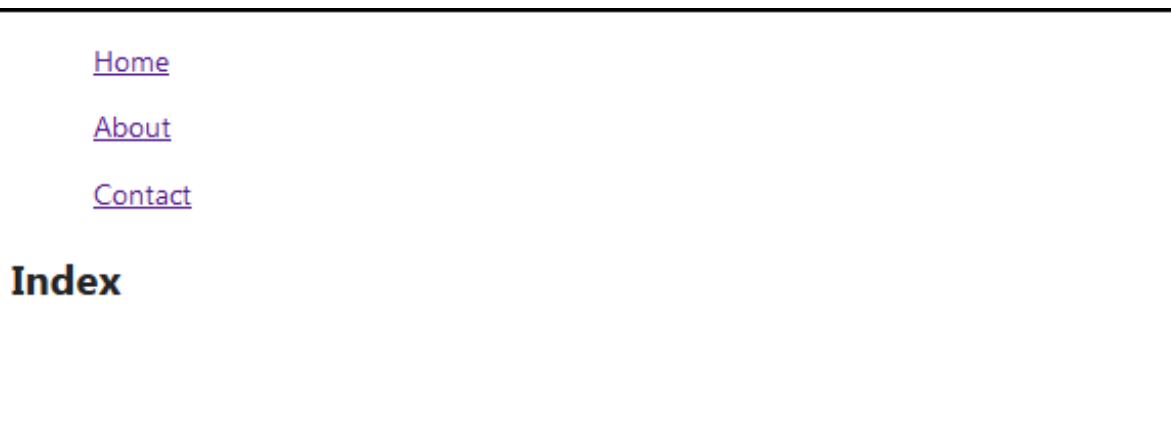
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>

<div>
    <li>
        <ul>
            <a href="/">Home</a>
        </ul>
        <ul>
            <a href="/Home/About/">About</a>
        </ul>
        <ul>
            <a href="/Home/Contact/">Contact</a>
        </ul>
    </li>
</div>

    @RenderBody()

    @Scripts.Render("~/bundles/jquery")
    @RenderSection("scripts", required: false)
</body>
</html>
```

 and .



We now need to add a View object for each of our links in **HomeController.cs**:

CODE TO TYPE: /Controllers/HomeController.cs

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

namespace MvcDesign.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

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

        public ActionResult About()
        {
            return View();
        }

        public string Contact()
        {
            return "This is the Contact page!";
        }
    }
}
```

OBSERVE:

```
public ActionResult About()
{
    return View();
}

public string Contact()
{
    return "This is the Contact page!";
}
```

One of the HomeController's new methods returns an **ActionResult** type; the other returns a **string** type. You'll see these differences more clearly when we run the application, but first we need to create Views for our new **About()** and **Contact()** methods. Follow the same steps as before for creating a View:

Add View

View name:

About

View engine:

Razor (CSHTML)

☐ Create a strongly-typed view

Model class:

Scaffold template:

Empty

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

...

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceholder ID:

MainContent

Add

Cancel

Add View

View name:

View engine:

Razor (CSHTML)

☐ Create a strongly-typed view

Model class:

Scaffold template:

Empty

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

...

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

Add Cancel

Modify **/Views/Home/About.cshtml** as shown:

CODE TO TYPE: /Views/Home/About.cshtml

```
@{
    ViewBag.Title = "About";
}

<h2>AboutThis is the About page!</h2>
```



The page displays the links we just added. Navigate using the links, paying attention to the address bar and how the URLs are displayed:

OBSERVE: URL listing

```
Home page: localhost:xxxxx  
About page: http://localhost:xxxxx/Home/About/  
Contact page: http://localhost:xxxxx/Home/Contact/
```

Note Notice how the URLs match the names of the Controller Actions; we'll discuss this in more detail shortly.

You can see the change we made to the About page:



Controller Actions

Controller Actions get exposed when a user either enters the URL in the navigation bar, or clicks on a link. An action is a Controller method that gets invoked when a certain URL is accessed. For example, we told the application that we want the **HomeController** to use the **Contact** method (which is a Controller Action) when we navigate to "**localhost:xxxxx/Home/Contact/**".

ActionResult is an Action that returns the View corresponding to the controller (in our example); this enables us to have different web pages rendered for different methods. There are many different types of Controller Action Methods, but they are all derived from the ActionResult class.

Let's take a look at a few of them to get a feel for how they work. Modify **/Controllers/HomeController.cs** as shown:

CODE TO TYPE: /Controllers/HomeController.cs

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

namespace MvcDesign.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/


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

        //
        // GET: /Home/About/
        public ActionResult About()
        {
            return View();
        }

        //
        // GET: /Home/Contact/
        public string Contact()
        {
            return "This is the Contact page!";
        }

        //
        // GET: /Home/Google
        // Redirect from our application to http://www.google.com
        public RedirectResult Google()
        {
            return Redirect("http://www.google.com");
        }

        public RedirectResult RedirectContact()
        {
            return Redirect("/Home/Contact/");
        }
    }
}
```

 Save, but don't run the application; we have a bit more editing to do.

The **RedirectResult** return type allows you to redirect to another action method by using its URL. As you can see in our **Google()** method, we redirect the user to Google. In our **RedirectContact()** method, we render our Contact view using the URL. Modify your **/Views/Shared/_Layout.cshtml** as shown.

CODE TO TYPE: /Views/Shared/_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
<div>
    <li>
        <ul>
            <a href="/">Home</a>
        </ul>
        <ul>
            <a href="/Home/About/">About</a>
        </ul>
        <ul>
            <a href="/Home/Contact/">Contact</a>
        </ul>
        <ul>
            <a href="/Home/Google/">Google</a>
        </ul>
        <ul>
            <a href="/Home/RedirectContact/">Redirect to Contact()</a>
        </ul>
    </li>
</div>
    @RenderBody()

    @Scripts.Render("~/bundles/jquery")
    @RenderSection("scripts", required: false)
</body>
</html>
```

 Save and  run, then test the **Google** and **Redirect to Contact()** links.

There are many other methods that are derived from the `ActionResult` class, but they're beyond the scope of this lesson. We encourage you to read about the various redirect methods at [Redirect Methods](#).

Introduction to Razor

We have been using the Razor syntax with ASP.NET and C# throughout the lesson. Let's talk a bit more about Razor itself, where we have used it, and why we care about it.

First let's look at our recently modified **About.cshtml** file:

OBSERVE: About.cshhtml

```
@{
    ViewBag.Title = "About";
}

<h2>This is the About page!</h2>

@{
    if (ViewBag.id != 0)
    {
        <p>We retrieved a value of: @ViewBag.id</p>
    }
    else
    {
        <p>There was nothing in the id value.</p>
    }
}
```

The **@** character allows us to use code in our HTML pages using the Razor syntax. **@ViewBag.id** is an inline code statement. **<p></p>** is HTML formatting that is stored in our Razor code, with Razor code placed in the HTML formatting. Any time we want to use a code statement that is in HTML formatting, we need to use the **@** symbol.

If we want to define a *string literal*, we need to append the **@** symbol before our string:

OBSERVE: String Literal

```
@{ var a_string = @"This is a string literal" }
```

Note Syntax and variable definitions are case-sensitive.

This is how we retrieve values from a form:

OBSERVE:

```
/* Multi-Statement Block */
@{
    var input_1 = Request["form_box_1"];
    var input_2 = Request["form_box_2"];
    int x = 5;
}

/* Inline-block statement */
<p>@Request.UrlPath</p>
// HTML form code
```

Request retrieves values from a form POST action. **@Request.UrlPath** returns the absolute path of the web page. We can declare a variable of any type using the **var** keyword or an explicit data type declaration such as **int**.

Let's take a look at some comments in the Razor syntax:

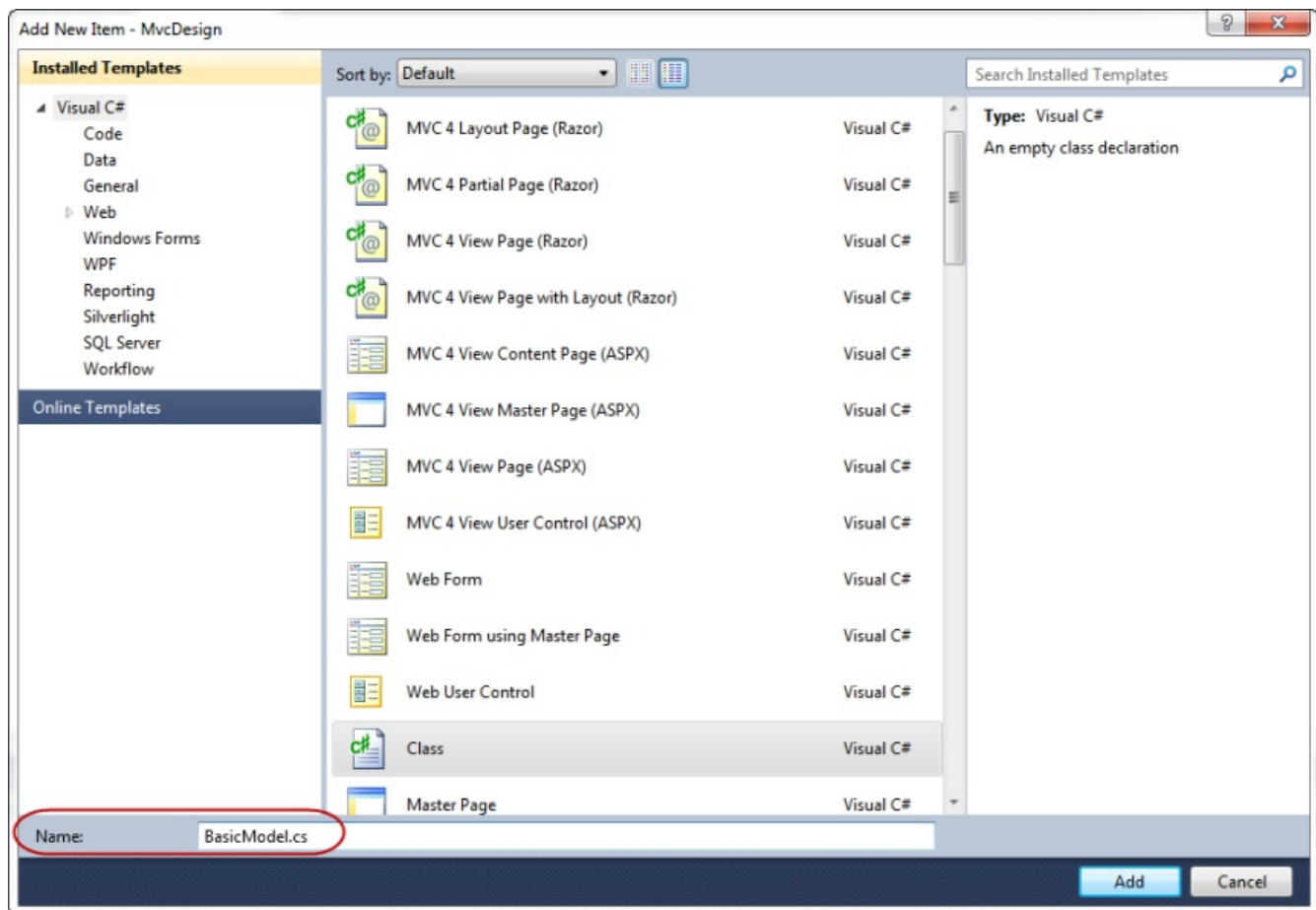
OBSERVE: Razor Comments

```
@* This is an inline Razor comment *@
@*
And this
is
a
Razor
Block Comment *@
```

@@** showed an inline comment, whereas **@**@** showed a block comment.

Let's practice the Razor syntax.

Right-click the **Models** folder and add a new class named **BasicModel.cs**:



Modify **/Models/BasicModel.cs** as shown:

CODE TO TYPE: /Models/BasicModel.cs

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

namespace MvcDesign.Models
{
    public class BasicModel
    {
        public int IntegerValue { get; set; }
        public string StringValue { get; set; }
    }
}
```

We now need to modify the **/Controllers/HomeController.cs** file:

CODE TO TYPE: /Controllers/HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using MvcDesign.Models;

namespace MvcDesign.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

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

        //
        // GET: /Home/About/
        public ActionResult About()
        {
            View();
        }

        //
        // GET: /Home/Contact/
        public string Contact()
        {
            return "This is the Contact page!";
        }

        //
        // GET: /Home/Google
        // Redirect from our application to http://www.google.com
        public RedirectResult Google()
        {
            return Redirect("http://www.google.com");
        }

        //
        // GET: /Home/Contact
        // Redirect from our link to the Contact view
        public RedirectResult RedirectContact()
        {
            return Redirect("/Home/Contact/");
        }

        public ActionResult IntMethod()
        {
            return View();
        }

        public ActionResult StrMethod()
        {
            return View();
        }
    }
}
```

Now that we have two action methods, we need views for each. Right-click **IntMethod()** and create a view for it. Make sure the **Create a strongly-typed view** option is checked, and choose **BasicModel (MvcDesign.Models)**. If you don't see the **BasicModel** option, try rebuilding the solution:

Add View [X]

View name:
IntMethod

View engine:
Razor (CSHTML) ▼

☒ Create a strongly-typed view

Model class:
BasicModel (MvcDesign.Models) ▼

Scaffold template:
Empty ▼

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

[] ...

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceholder ID:
MainContent

[Add] [Cancel]

Now, add this code to **IntMethod.cshtml**:

CODE TO TYPE: IntMethod.cshtml

```
@model MvcDesign.Models.BasicModel

@{
    ViewBag.Title = "IntMethod";
}

<h2>IntMethod</h2>

<p>
    @using (Html.BeginForm())
    {
        <fieldset>
            <legend>IntMethod</legend>
            Enter an Integer: @Html.EditorFor(model => model.IntegerValue)
            <input type="submit" value="Submit" />
        </fieldset>
    }

    @if (Model.IntegerValue == 0)
    {
        <div>Invalid integer has been entered.</div>
    }
    else
    {
        <div>Integer value entered: @Model.IntegerValue</div>
    }
</p>
```

Create a view for **StrMethod** just as we did with **IntMethod**, and insert this code into **StrMethod.cshtml**:

CODE TO TYPE: StrMethod.cshtml

```
@model MvcDesign.Models.BasicModel

@{
    ViewBag.Title = "StrMethod";
}

<h2>StrMethod</h2>

<p>
    @using (Html.BeginForm())
    {
        <fieldset>
            <legend>StrMethod</legend>
            Enter a string: @Html.EditorFor(model => model.StringValue)
            <input type="submit" value="Submit" />
        </fieldset>
    }

    @if (Model.StringValue == null)
    {
        <div>The value of the string was empty.</div>
    }
    else
    {
        <div>The string entered was: @Model.StringValue</div>
    }
</p>
```

Let's go over the new syntax:

OBSERVE:

```
@using (Html.BeginForm())
{
    <fieldset>
    <legend>StrMethod</legend>
    Enter a string: @Html.EditorFor(model => model.StringValue)
    <input type="submit" value="Submit" />
    </fieldset>
}
```

- **Html.BeginForm()** creates an HTML form using the **Html** helper object.
- **@Html.EditorFor** is a function that takes an object as an argument that identifies the value.
- **Model.StringValue** gives us access to all of our Model's members.

Now, edit **HomeController.cs** as shown:

CODE TO TYPE: HomeController.cs

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

namespace MvcDesign.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

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

        //
        // GET: /Home/About/
        public ActionResult About()
        {
            View();
        }

        //
        // GET: /Home/Contact/
        public string Contact()
        {
            return "This is the Contact page!";
        }

        //
        // GET: /Home/Google
        // Redirect from our application to http://www.google.com
        public RedirectResult Google()
        {
            return Redirect("http://www.google.com");
        }

        //
        // GET: /Home/Contact
        // Redirect from our link to the Contact view
        public RedirectResult RedirectContact()
        {
            return Redirect("/Home/Contact/");
        }

        public ActionResult IntMethod(BasicModel model)
        {
            model.IntegerValue = Convert.ToInt32(model.IntegerValue) + 10;
            return View(model);
        }

        public ActionResult StrMethod(BasicModel model)
        {
            return View(model);
        }
    }
}
```

We pass an object into the `IntMethod` and `StrMethod` functions so we can access our Model's members directly. Now we have the IntelliSense capabilities that are offered by Visual Studio.

We need to modify the `/Views/Home/Index.cshtml` file before we can run the application:



CODE TO TYPE: `/Views/Home/Index.cshtml`

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<form method="post" action="/Home/IntMethod">
    <table>
        <tr>
            <td>
                Input an Integer:
            </td>
            <td>
                <input id="IntegerValue" name="IntegerValue" type="text" value="" />
            </td>
        </tr>
    </table>
    <input type="submit" id="IntMethod" value="Return IntMethod" />
</form>

<form method="post" action="/Home/StrMethod">
    <table>
        <tr>
            <td>
                Input a string:
            </td>
            <td>
                <input id="StringValue" name="StringValue" type="text" value="" />
            </td>
        </tr>
    </table>
    <input type="submit" id="StrMethod" value="String Method" />
</form>
```

 Save and  run it.

We don't see any change to the index page, but when we enter a value for `IntMethod` or `StrMethod` we see a minor modification:

[Home](#)

[About](#)

[Contact](#)

[Google](#)

[Redirect to Contact\(\)](#)

IntMethod

IntMethod

Enter an Integer:

Integer value entered: 42

[Home](#)

[About](#)

[Contact](#)

[Google](#)

[Redirect to Contact\(\)](#)

StrMethod

StrMethod

Enter a string:

The string entered was: This is a test

In essence, we have been using the Razor syntax throughout this entire lesson. It's a powerful tool to use with ASP.NET with C# applications. It allows us to perform tasks just as we would using any other web programming language.

URLs and Routing

We introduced the **RouteConfig.cs** file earlier. Now, we'll discuss it in more detail so you can begin to create custom routes and control your URLs.

Basics of Routing

Open **/App_Start/RouteConfig.cs** and insert this code:

CODE TO TYPE: /App_Start/RouteConfig.cs

```
.  
.   
.   
    routes.MapRoute(  
        name: "About",  
        url: "Home/About/",  
        defaults: new { controller = "Home", action = "About" }  
    );  
  
    routes.MapRoute(  
        name: "Int Method",  
        url: "Home/IntMethod/{id}",  
        defaults: new { controller = "Home", action = "IntMethod", id = @"\d+" }  
    );  
  
    routes.MapRoute(  
        name: "Str Method",  
        url: "Home/StrMethod/{id}",  
        defaults: new { controller = "Home", action = "StrMethod", id = "[A-Z][a-z].+" }  
    );  
.   
.   
. 
```

id = @"\d+" specifies that this method will take only integers and **id = "[A-Z][a-z].+"** specifies strings.

Note

The regular expressions are not required, but they could be useful if you're using the same method with different data types.

The above routes for **IntMethod** and **StrMethod** help prevent the routes from becoming too "greedy" and causing a misuse of the id values.

Create a view in your **/App_Start/RouteConfig.cs** for each new method in **HomeController.cs**, and then add the code shown in the following sections.

Routing Constraints

Constraints on routing can help to minimize the possibility of errors in our URLs.

Currently, if we enter the URL to the About page (*localhost:xxxxx/Home/About/*) manually, and append a value at the end, we receive an error. This is where we want to use constraints.

In **/App_Start/RouteConfig.cs**, add the About route as shown:

CODE TO TYPE: /App_Start/RouteConfig.cs for About

```
.  
.   
.   
    routes.MapRoute(  
        name: "About",  
        url: "{controller}/{action}/{id}",  
        defaults: new { controller = "Home", action = "About", id = UrlParameter.Optional },  
        constraints: new { id = @"^[0-9]+$" }  
    );  
.   
.   
. 
```

The **constraints** setting allows us to define how we want to handle the possibility of an optional id value in the URL. Now we need to modify the **About** method in our **/Controllers/HomeController.cs** file:

CODE TO TYPE: /Controllers/HomeController.cs About method

```
.  
.br/>.br/>    public ActionResult About(int? id)  
    {  
        ViewBag.id = id  
        return View();  
    }  
.br/>.br/.
```

Here we accept a nullable integer item in our About action method, and assign it to the ViewBag object. Let's test this new idea and see how our routing constraint handles a word.

Let's add a little bit of logic to our **/Views/Home/About.cshtml** file:

CODE TO TYPE: /Views/Home/About.cshhtml

```
@{
    ViewBag.Title = "About";
}

<hgroup class="title">
    <h1>@ViewBag.Title.</h1>
    <h2>@ViewBag.Message</h2>
</hgroup>

<article>
    <p>
        Use this area to provide additional information.
    </p>
    <p>
        Use this area to provide additional information.
    </p>
    <p>
        Use this area to provide additional information.
    </p>
</article>

<aside>
    <h3>Aside Title</h3>
    <p>
        Use this area to provide additional information.
    </p>
    <ul>
        <li>@Html.ActionLink("Home", "Index", "Home")</li>
        <li>@Html.ActionLink("About", "About", "Home")</li>
        <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    </ul>
</aside>
<h2>This is the About page!</h2>

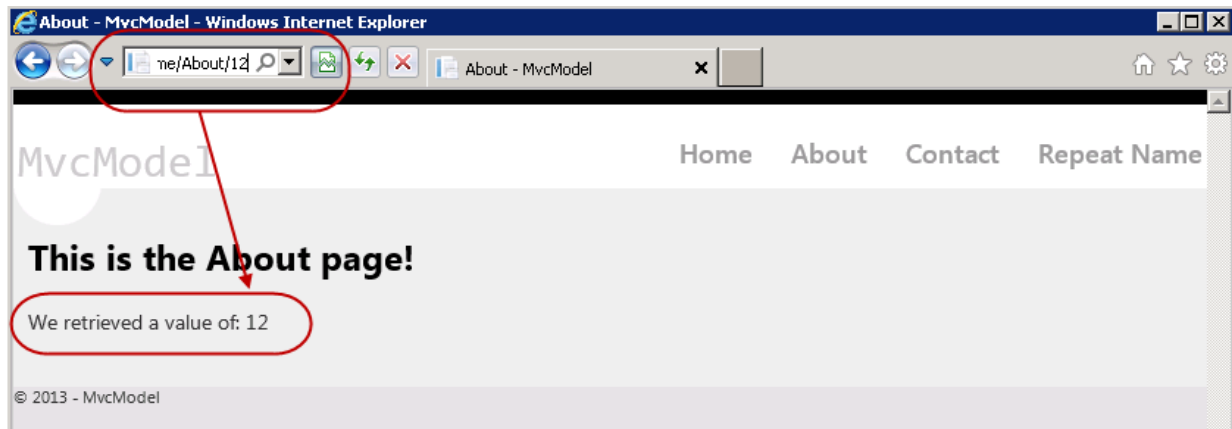
@{
    if (ViewBag.id != 0)
    {
        <p>We retrieved a value of: @ViewBag.id</p>
    }
    else
    {
        <p>There was nothing in the id value.</p>
    }
}
```




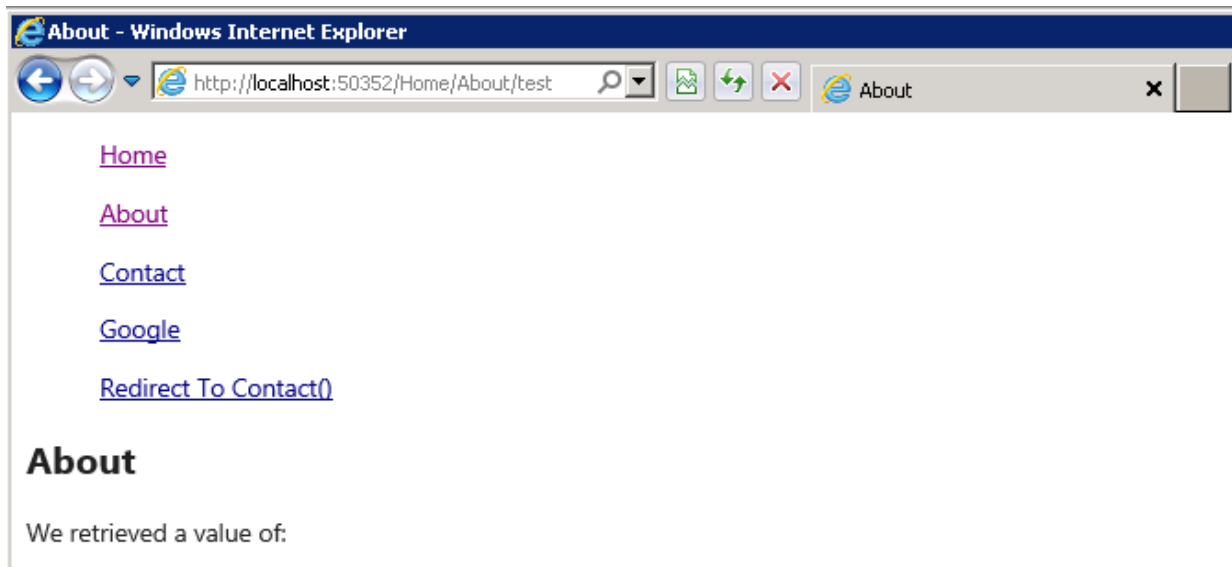
Save and



run it. Navigate to the About page using the previously established link. It displays "There was nothing in the id value." Append an integer value to the URL to see what happens. If you use a URL like **localhost:xxxxx/Home/About/12** and enter a string so the URL resembles **localhost:xxxxx/Home/About/test**, you'll see these results in the images:



save all the files and run the application using the  button, and navigate to localhost:xxxxx/Home/About



Let's change the About route in our **RouteConfig.cs** file to make the route more strict:

CODE TO TYPE: Removing UrlParameter.Optional from RouteConfig.cs


```
.  
. .  
. .  
. .  
  
routes.MapRoute(  
    name: "About",  
    url: "{Home}/{About}/{id}",  
    defaults: new { controller = "Home", action = "About", id = UrlParameter.Optional,  
    constraints: new { id = @"^[0-9]+$" }  
);  
. .  
. .  
. .
```

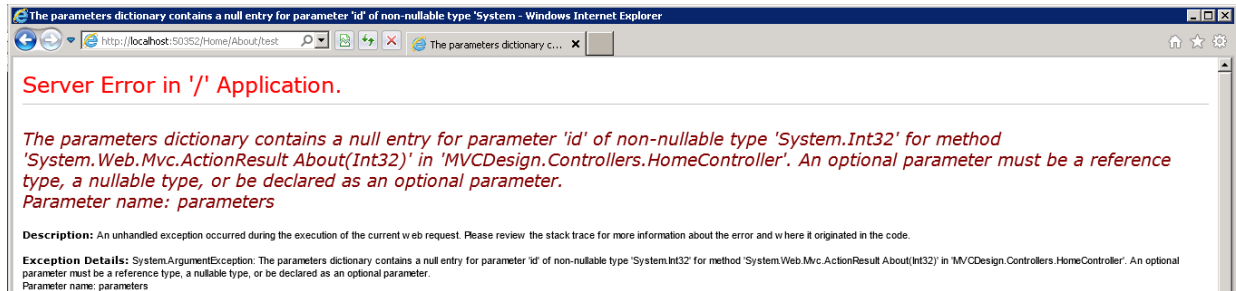
And now let's modify the About action method in the HomeController.cs file.

CODE TO TYPE: Removing nullable int parameter from About()

```
.  
. .  
public ActionResult About(int id)  
{  
    ViewBag.id = id;  
    return View();  
}  
. . .
```



and . Navigate to the About and enter a string so the URL resembles `localhost:xxxxx/Home/About/test`. You'll see this result:



We get an error stating what has gone wrong with the application. It's up to you to decide how errors and routing constraints should be handled by the application.

You're doing really well so far! Practice this stuff some more in the homework then move on to the next lesson.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

MVC

Lesson Objectives

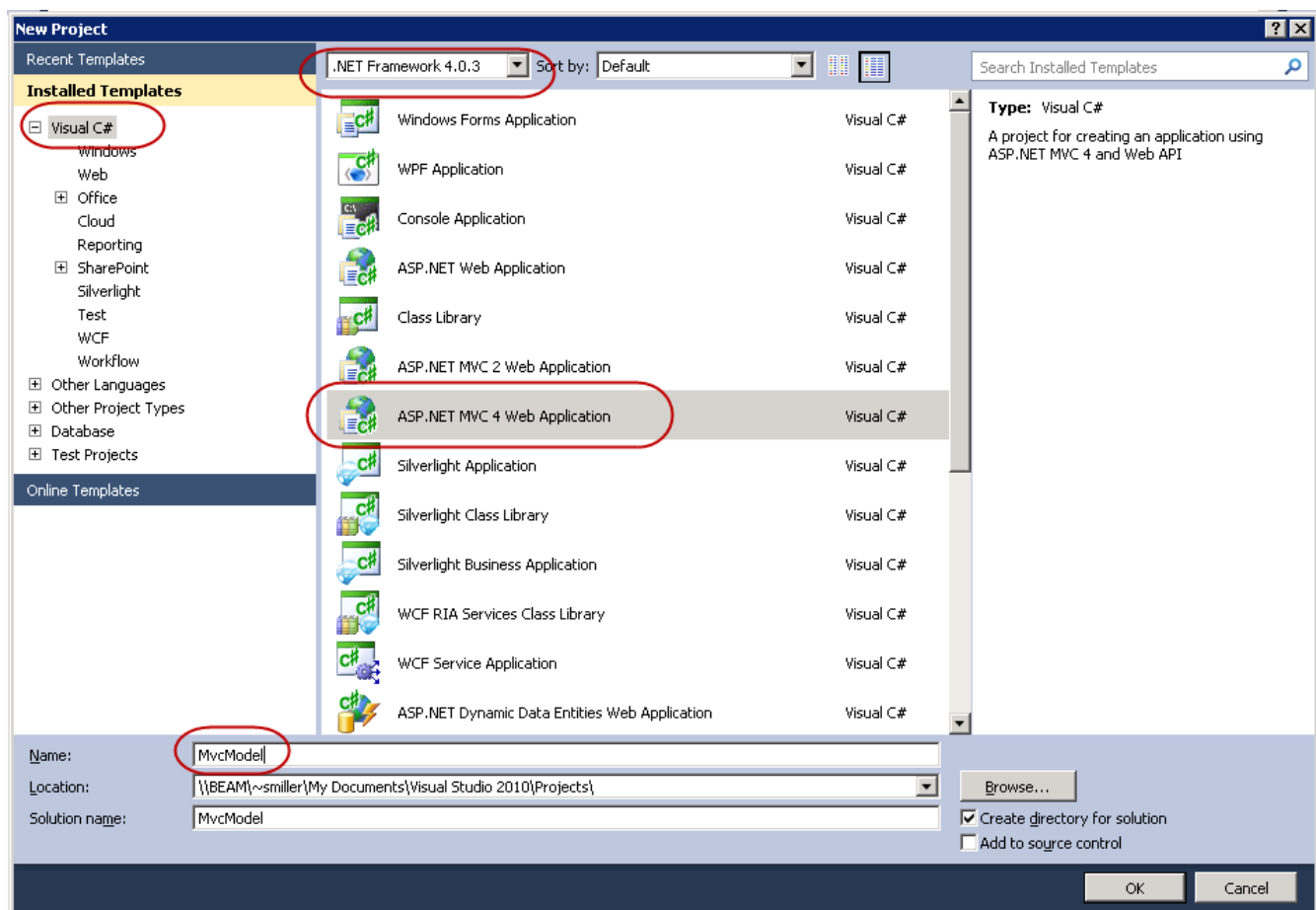
In this lesson, you will:

- use MVC with the Entity Framework.
- combine models and entities.
- add filters when working with entities.

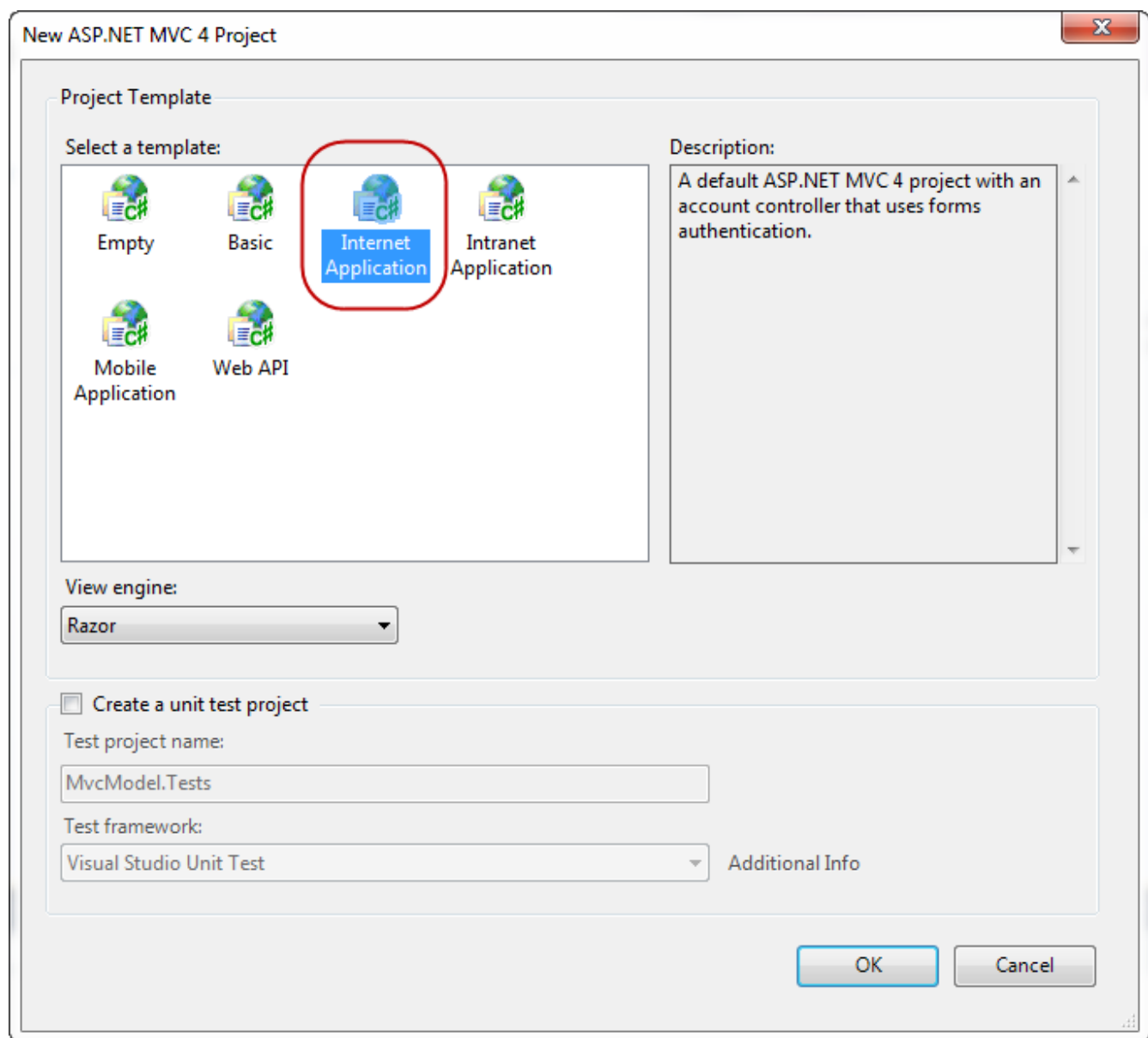
The Model

Since the Model *implements the data* of our application, we are going to create a new project to more effectively implement our Model with the default setup given by Visual Studio.

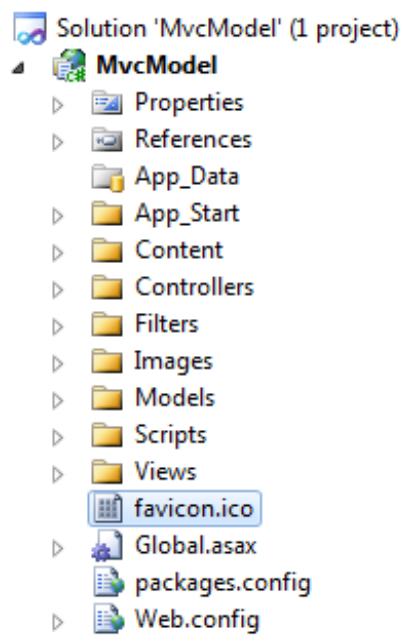
Select **File | New | Project**. Select the **ASP.NET MVC 4 Web Application**, and name your application **MvcModel**:



Click **OK**. In the next dialog, select **Internet Application**:



Click **OK**. Observe the folders that are created by default in our new application:



Our application creates a **HomeController** for us, as well as an entire web application template for us to use and modify, which reduces the amount of time we need to invest in a project startup. Modify **/Controllers/HomeController.cs** as shown:

/Controllers/HomeController.cs

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

namespace MvcModel.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Modify this template to jump start your ASP.NET MVC application. Welcome to the OST Forum!";


            return View();
        }

        public ActionResult About()
        {
            ViewBag.Message = "Your app description page.";

            return View();
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your contact page.";

            return View();
        }
    }
}
```

 and navigate around to get a feeling for the project.

Modify **/Views/Home/Index.cshtml** file as shown:

```

@{
    ViewBag.Title = "Home PageOST Forum";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2>@ViewBag.Message</h2>
            </hgroup>
            <p>
                To learn more about ASP.NET MVC visit
                <a href="http://asp.net/mvc" title="ASP.NET MVC Website">http://asp.net/mvc</a>.
                The page features <mark>videos, tutorials, and samples</mark> to help y
ou get the most from ASP.NET MVC.
                If you have any questions about ASP.NET MVC visit
                <a href="http://forums.asp.net/1146.aspx/1?MVC" title="ASP.NET MVC Foru
m">our forums</a>.
            </p>
        </div>
    </section>
}
<h3>We suggest the following:</h3>
<ol class="round">
    <li class="one">
        <h5>Getting Started</h5>
        ASP.NET MVC gives you a powerful, patterns based way to build dynamic websites
that
        enables a clean separation of concerns and that gives you full control over mar
kup
        for enjoyable, agile development. ASP.NET MVC includes many features that enabl
e
        fast, TDD friendly development for creating sophisticated applications that use
the latest web standards.
        <a href="http://go.microsoft.com/fwlink/?LinkId=245151">Learn more</a>
    </li>

    <li class="two">
        <h5>Add NuGet packages and jump start your coding</h5>
        NuGet makes it easy to install and update free libraries and tools.
        <a href="http://go.microsoft.com/fwlink/?LinkId=245153">Learn more</a>
    </li>

    <li class="three">
        <h5>Find Web Hosting</h5>
        You can easily find a web hosting company that offers the right mix of features
and price for your applications.
        <a href="http://go.microsoft.com/fwlink/?LinkId=245157">Learn more</a>
    </li>
</ol>

```

Your updated Index.cshtml looks like this:

OBSERVE: Updated Index.cshtml

```
@{
    ViewBag.Title = "OST Forum";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2>@ViewBag.Message</h2>
            </hgroup>
            <p>
            </p>
        </div>
    </section>
}
```

@{ ViewBag.Title = "Home Page" } assigns the name "Home Page" to the **Title** of a **ViewBag** object. **@ViewBag.Message** is defined in the **HomeController.cs** file, and displays a message to the user on the /Home/Index page.

The Entity Framework

Let's discuss the various approaches of the Entity Framework.

- **Database First:** This approach is used when we have an existing database and would like the Entity Framework to construct a data model for us. The database schema, model data, and mapping between them, are stored in an XML file with the extension *.edmx*.
- **Model First:** This approach is used when we don't have a database already and want Visual Studio's Entity Framework designer to generate the DDL (*Data Definition Language*) statements for us.
- **Code First:** This is the approach we'll use for this lesson. We use it when we don't have a database and want to create one. This approach uses information from the Model to create the schema of our database. We will use the **DbContext** class API. This API allows us to create, drop, or recreate the database if the model changes.

MVC and the Entity Framework

Let's build our application. Open **/Views/Shared/_Layout.cshtml** and add this code:

CODE TO TYPE: /Views/Shared/_Layout.cshtml

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title - My ASP.NET MVC ApplicationOST Forum</title>
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
    <meta name="viewport" content="width=device-width" />
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
  </head>
  <body>
    <header>
      <div class="content-wrapper">
        <div class="float-left">
          <p class="site-title">@Html.ActionLink("your logo hereOST Forum", "
Index", "Home")</p>
        </div>
        <div class="float-right">
          <section id="login">
            @Html.Partial("_LoginPartial")
          </section>
          <nav>
            <ul id="menu">
              <li>@Html.ActionLink("Home", "Index", "Home")</li>
              <li>@Html.ActionLink("About", "About", "Home")</li>
              <li>@Html.ActionLink("Forums", "Index", "Forum")</li>
              <li>@Html.ActionLink("Posts", "Index", "Posts")</li>
              <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
            </ul>
          </nav>
        </div>
      </div>
    </header>
    <div id="body">
      @RenderSection("featured", required: false)
      <section class="content-wrapper main-content clear-fix">
        @RenderBody()
      </section>
    </div>
    <footer>
      <div class="content-wrapper">
        <div class="float-left">
          <p>© @DateTime.Now.Year - My ASP.NET MVC ApplicationOST Forum</p>
        </div>
      </div>
    </footer>

    @Scripts.Render("~/bundles/jquery")
    @RenderSection("scripts", required: false)
  </body>
</html>
```

There are a couple of items to note here:

OBSERVE:

```
<title>@ViewBag.Title - OST Forum</title>
```

- **@ViewBag** is used to pass data from a controller to a view; ViewBag is a dynamic object that becomes null during redirection.
- **Title** assigns the name of the View object of each controller method.

@ViewBag.Title is wrapped in a **<title></title>** tag, so when we run our application, the title of the browser tab reflects the name of the "link" to which we have navigated. In the **Index.cshtml** file, the **@ViewBag.Title** is set to

"Home Page", and the titles are set similarly for each .cshtml file in the View folder.

Note @ViewBag is a dynamic object and can be extended to use any name we decide (for example, @ViewBag.Message).

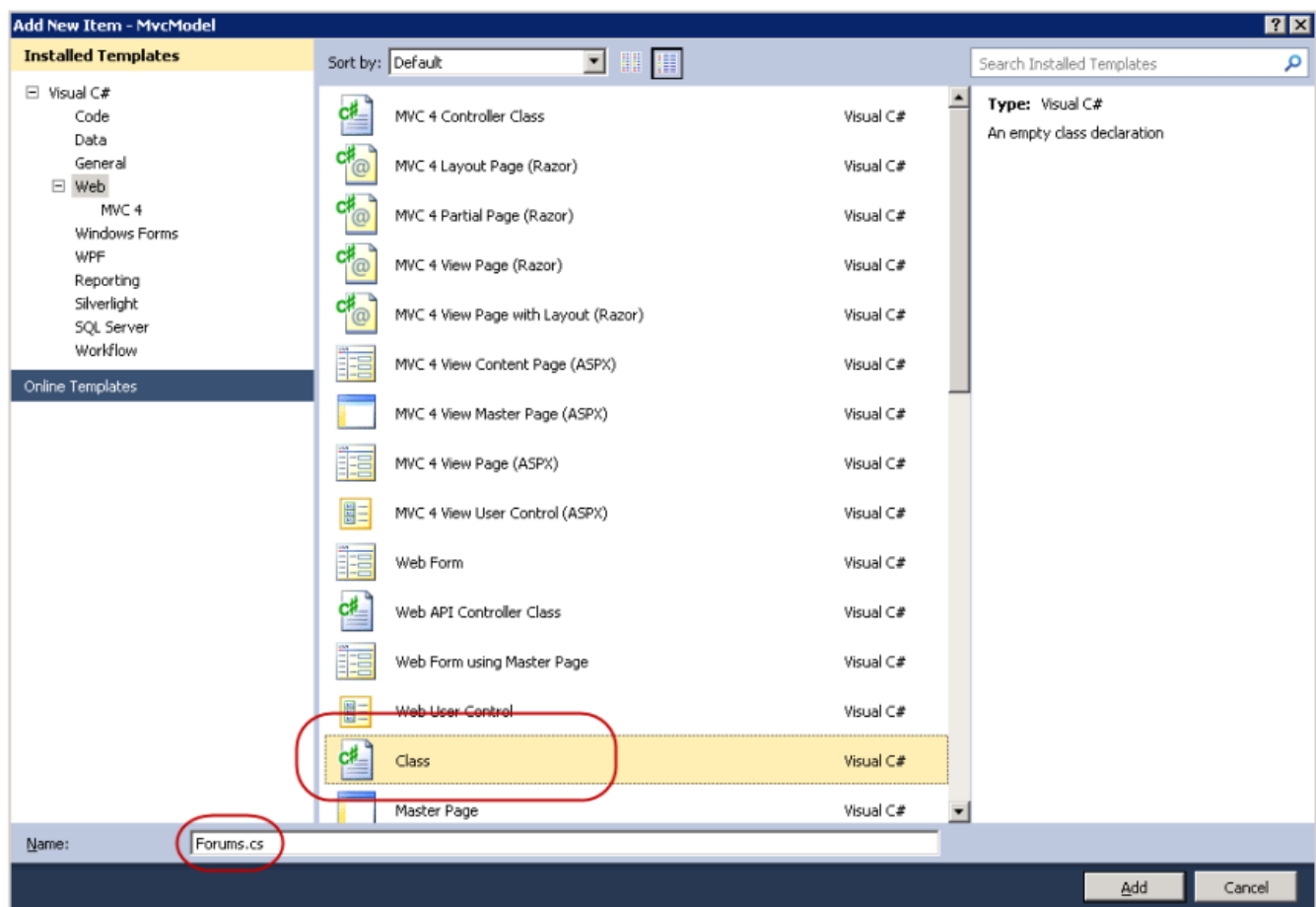
OBSERVE:

```
<ul id="menu">
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
```

- In **@Html.ActionLink**, **@Html** is a helper object that builds all the HTML controls needed for the input and output of data. **ActionLink** is a method of the object that allows us to define the path of the link we want.

Note If we want to have an anchor tag, we can use `Link Name`

Add a few Models by right-clicking the **Models** folder in the Solution Explorer and selecting **Add |** . We'll need **Posts**, **Users**, **Replies**, and **Forums** Models.



Edit **/Models/Posts.cs** as shown:

CODE TO TYPE: /Models/Posts.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Web;

namespace MvcModel.Models
{
    public class Posts
    {
        [Key]
        public int PostId { get; set; }
        public string postTitle { get; set; }
        public string postValue { get; set; }
        public string postForum { get; set; }
        public DateTime date { get; set; }
    }
}
```

Edit **/Models/Users.cs** as shown:

CODE TO TYPE: /Models/Users.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Web;

namespace MvcModel.Models
{
    public class Users
    {
        [Key]
        public int UserId { get; set; }
        public string userName { get; set; }
    }
}
```

Edit **/Models/Replies.cs** as shown:

CODE TO TYPE: /Models/Replies.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Web;

namespace MvcModel.Models
{
    public class Replies
    {
        [Key]
        public int ReplyId { get; set; }
        public string reply { get; set; }
        public DateTime replyDate { get; set; }
    }
}
```

Edit **/Models/Forums.cs** as shown:

CODE TO TYPE: /Models/Forums.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Web;

namespace MvcModel.Models
{
    public class Forums
    {
        public int id { get; set; }
    }
}
```

Create another new class named **/ForumContext.cs**. This class will contain the Database information for our different classes.

Modify **/ForumContext.cs** as shown:

CODE TO TYPE: ForumContext.cs

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using MvcModel.Models;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Linq;
using System.Web;

namespace MvcModel.EFData
{
    public class ForumContext : DbContext
    {
        public DbSet<Forums> forums { get; set; }
        public DbSet<Replies> reply { get; set; }
        public DbSet<Users> users { get; set; }
        public DbSet<Posts> posts { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

Let's take a closer look.

OBSERVE:

```
public class ForumContext : DbContext
{
    public DbSet<Forums> forums { get; set; }
    public DbSet<Replies> reply { get; set; }
    public DbSet<Users> users { get; set; }
    public DbSet<Posts> posts { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

ForumContext : DbContext creates our **ForumContext** class, but inherits from the **DbContext** class, so we can use the Entity Framework for our Data Models. **DbSet** creates a database for our *Entity Set*; rows in a database are generally referred to as an *Entity*. So in our code, **DbSet< Forums> forums** creates an **Entity Set** from the **Forums** class that we name **Forums**.

modelBuilder.Conventions.Remove<PluralizingTableNameConvention>(); prevents our tables from having plural names such as Forums, Users, and Posts. Instead they will be named Forum, User, and Post.

Now, we need to add a **Connection String** to our **Web.config** file:

CODE TO TYPE: Web.config

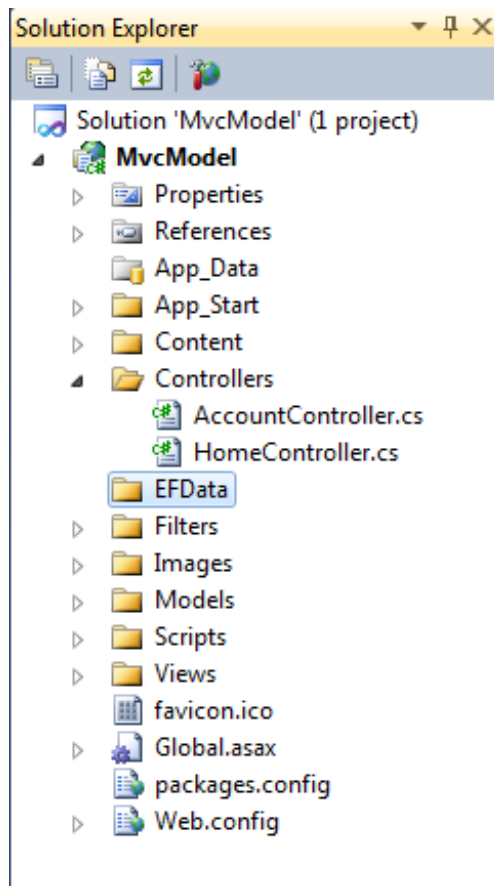
```
.  
.   
.   
<configuration>  
  <configSections>  
    <!-- For more information on Entity Framework configuration, visit http://go.micros  
oft.com/fwlink/?LinkID=237468 -->  
  </configSections>  
  <connectionStrings>  
    <del add name="DefaultConnection" connectionString="Data Source=.\SQLEXPRESS;Initial Ca  
talog=aspnet-MvcModel-20130516195933;Integrated Security=SSPI" providerName="System.Dat  
a.SqlClient"/>  
  
    <add name="ForumContext"  
      connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=Forums;AttachDBFile  
name=|DataDirectory|\Forums.mdf;Integrated Security=SSPI;User Instance=true"  
      providerName="System.Data.SqlClient"/>  
  
  </connectionStrings>  
.   
.   
. 
```

The Entity Framework automatically looks for a connection string named the same as our context class.



and then select **Build | Build Solution** to compile the project, but don't run the application.

Now that we have a fairly decent layout of a forum, let's incorporate the main class that incorporates the Entity Framework functionality for a Data Model. Create a new folder named **EFData**:



We now need to populate our database with default information. Create a new class in the **EFData** folder named **ForumInitializer.cs** and modify it as shown (it's okay to copy and paste this section, it's pretty repetitive):

CODE TO TYPE: /EFData/ForumInitializer.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using MvcModel.Models;

namespace MvcModel.EFData
{
    public class ForumInitializer : DropCreateDatabaseAlways<ForumContext>
    {
        protected override void Seed(ForumContext context)
        {
            var replies = new List<Replies>
            {
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia do
lor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2012-09-23")
            },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia do
lor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2001-05-12")
            },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia do
lor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2002-07-11")
            },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia do
lor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2011-07-04")
            },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia do
lor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2000-09-02")
            },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia do
lor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2010-05-14")
            }
        };
        replies.ForEach(s => context.reply.Add(s));
        context.SaveChanges();

        var users = new List<Users>
        {
            new Users { userName = "User_a" },
            new Users { userName = "User_b" },
            new Users { userName = "User_c" },
            new Users { userName = "User_d" },
            new Users { userName = "User_e" },
            new Users { userName = "User_f" },
        };
        users.ForEach(s => context.users.Add(s));
        context.SaveChanges();


        var posts = new List<Posts>
        {
            new Posts { postTitle = "A new Post", postValue = "this is a test post"
, postForum = "Forum_A", date = DateTime.Parse("2010-05-05") },
            new Posts { postTitle = "Another post", postValue = "This is another po
st", postForum = "Forum_B", date = DateTime.Parse("2010-12-03") },
            new Posts { postTitle = "A Third Post", postValue = "some kind of post
again", postForum = "Forum_C", date = DateTime.Parse("2005-01-04") },
            new Posts { postTitle = "Some post", postValue = "This is \"SomePost\""
, postForum = "Forum_A", date = DateTime.Parse("2003-04-04") },
            new Posts { postTitle = "the Fifth Post", postValue = "This is a fifth
post", postForum = "Forum_B", date = DateTime.Parse("2002-05-19") },
            new Posts { postTitle = "the sixth post", postValue = "This is the sixt
h post!", postForum = "Forum_C", date = DateTime.Parse("2006-08-19") },
            new Posts { postTitle = "Posting", postValue = "Yet Another Post", post
Forum = "Forum_A", date = DateTime.Parse("2012-12-12") },
            new Posts { postTitle = "Still posting", postValue = "We're coming to a

```

```

        close on posts", postForum = "Forum_A", date = DateTime.Parse("1999-12-23") },
            new Posts { postTitle = "Still Filling", postValue = "Isn't seeding a d
atabase fun?!", postForum = "Forum_C", date = DateTime.Parse("2006-09-09") },
            new Posts { postTitle = "Final Post Seed", postValue = "A final seed po
st value!", postForum = "Forum_B", date = DateTime.Parse("2004-08-19") }
        };
        posts.ForEach(s => context.posts.Add(s));
        context.SaveChanges();
    }
}
}

```

Again,  and build the solution to make sure we don't have any errors, but don't run the application yet—we still have a couple more files to modify. Modify **/Global.asax | Global.asax.cs** as shown:

CODE TO TYPE: /Global.asax.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using System.Data.Entity;
using MvcModel.Models;
using MvcModel.EFData;

namespace MvcModel
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

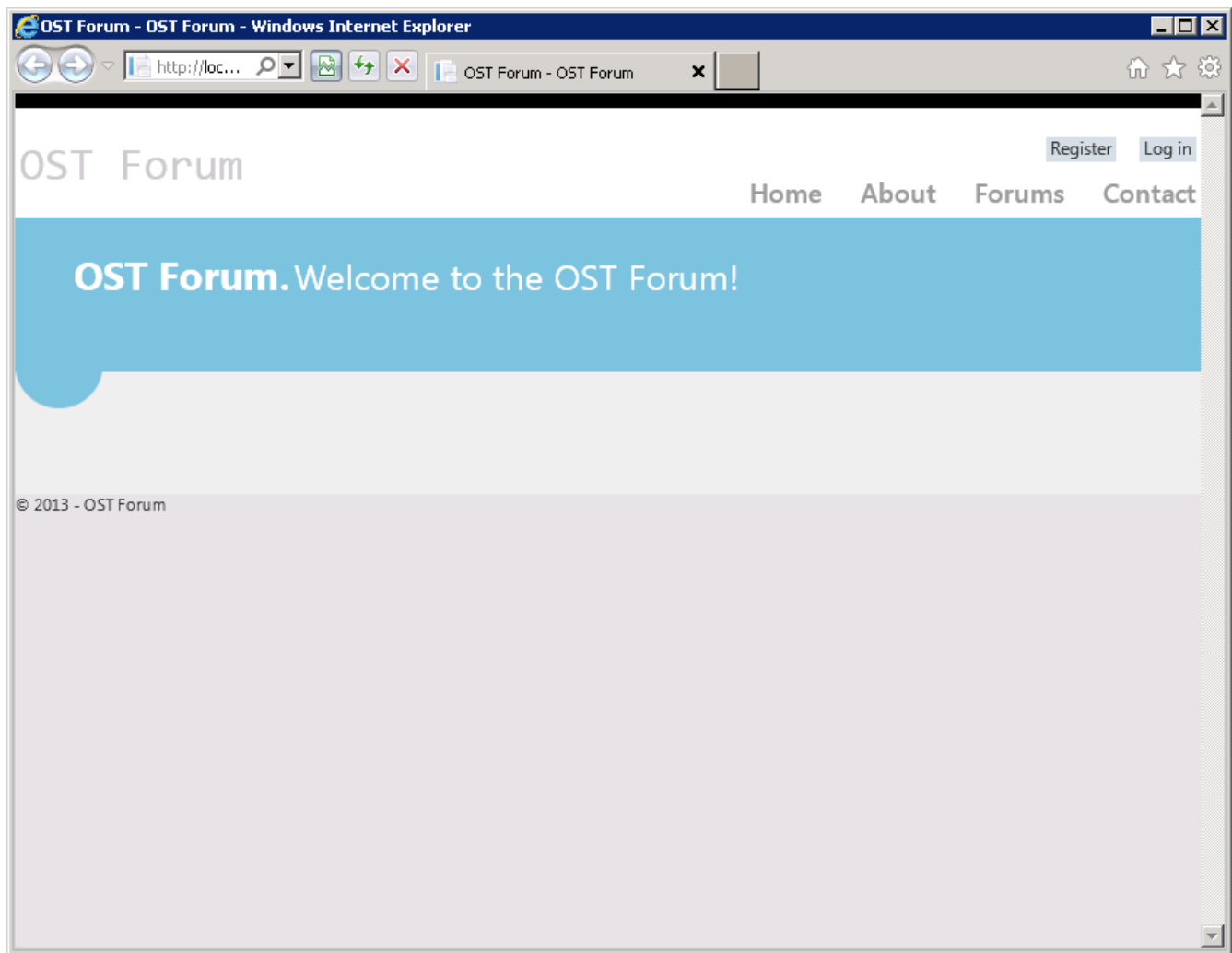
            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
AuthConfig.RegisterAuth();
            Database.SetInitializer<ForumContext>(new ForumInitializer());
        }
    }
}

```

We set up the application so that when it is run, the database is compared against our models; if the models have changed since the last run, the database will be dropped and recreated. The **using** statements we added allow the **Global.asax.cs** file to access the necessary models, whereas **Database.SetInitializer<ForumContext>(new ForumInitializer());** allows for our database to be initialized with our default information.

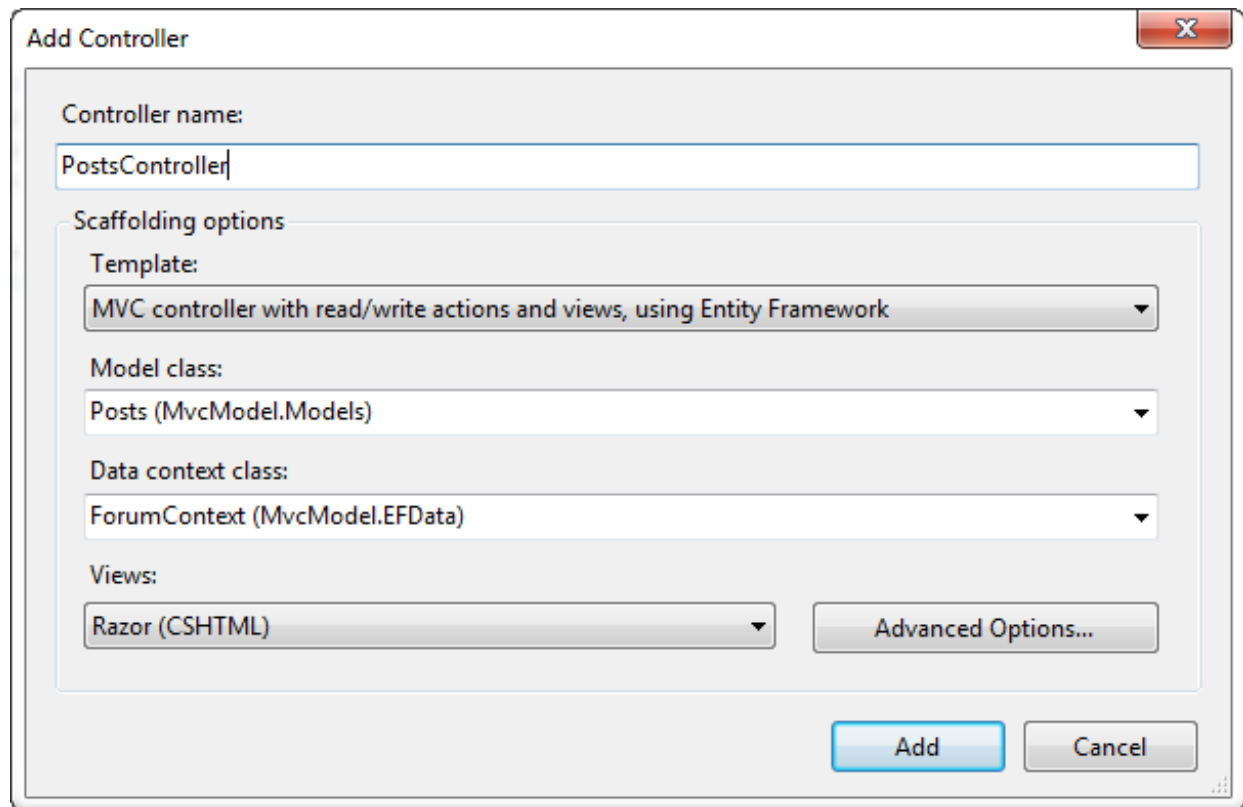
Tip If we set our application up for deployment, we would want to remove the code that seeds our database.

 and . The application looks like this:



No we'll begin using the Entity Framework with the MVC design pattern. The Entity Framework makes it convenient for us to do the stuff we want to do without having to focus on the preliminary coding.

Create a controller for the **Posts** model. Right-click the **Controllers** folder and select **Add | Controller....** Enter the Name **PostsController**; for Template, select **MVC Controller with read/write actions and views, using Entity Framework**; for Model class, select **Posts (MvcModel.Models)**; for Data context class, select **ForumContext (MvcModel.EFData)**; and for Views, select **Razor (CSHTML)**. When you're ready, click **Add**:



Add Controller

Controller name:
PostsController

Scaffolding options

Template:
MVC controller with read/write actions and views, using Entity Framework

Model class:
Posts (MvcModel.Models)

Data context class:
ForumContext (MvcModel.EFData)

Views:
Razor (CSHTML)

Advanced Options...

Add Cancel

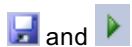
OBSERVE: PostsController.cs snippet

```
public class PostsController : Controller
{
    private ForumContext db = new ForumContext();

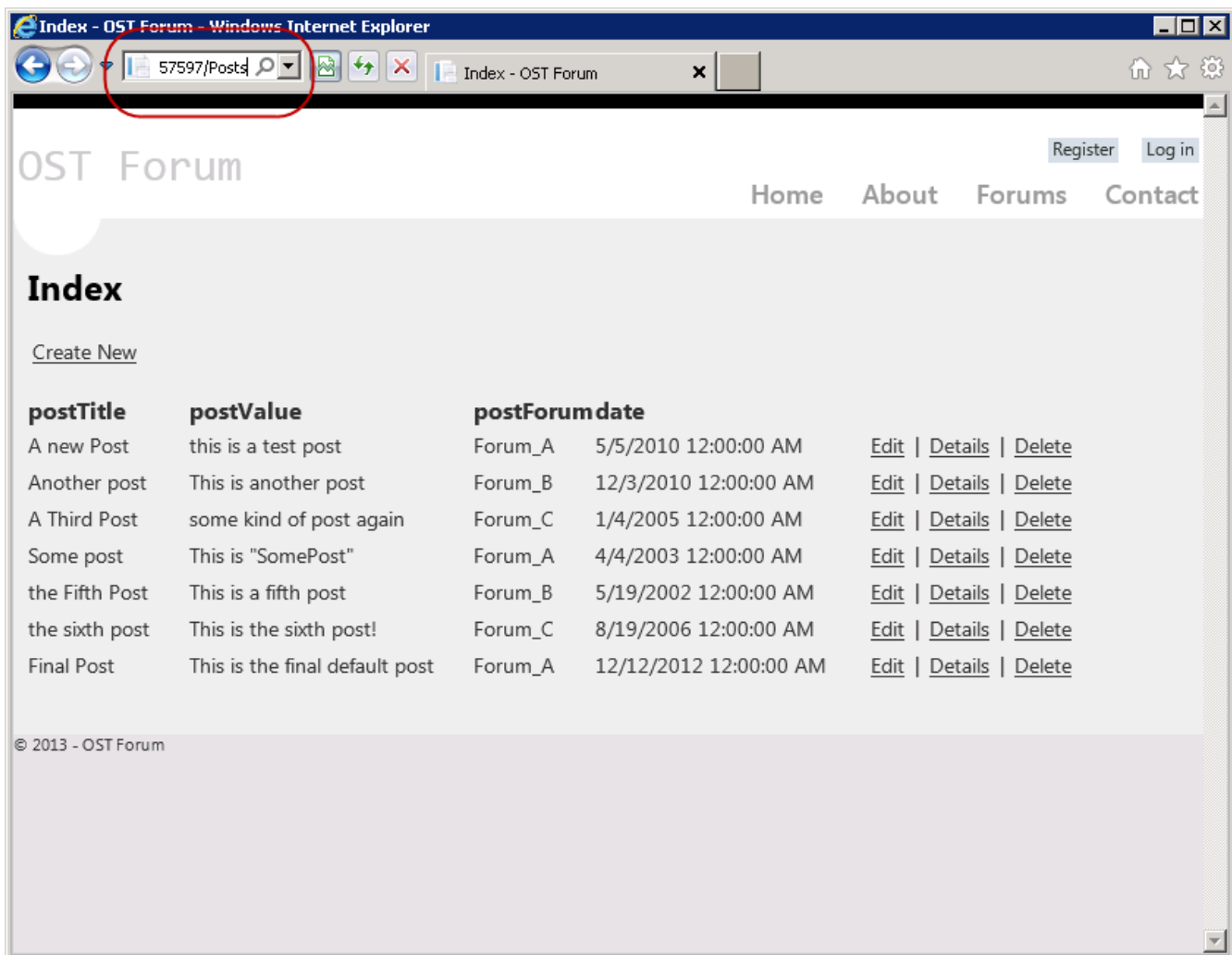
    //
    // GET: /Posts/

    public ActionResult Index()
    {
        return View(db.posts.ToList());
    }
}
```

Notice how Visual Studio creates a new variable **private ForumContext db = new ForumContext();**. This variable instantiates a new database object. **db.posts.ToList()** retrieves a list of database items from the Posts property of our database context instance, and since our database context contains *Entities* of our other models, Posts acts as a parent class.



and select **Posts** to see how the application looks now.



The folder in **/Views/Posts** that contains **Create.cshtml**, **Edit.cshtml**, **Delete.cshtml**, and **Details.cshtml** is *scaffolded* code that is generated for you by default. Use the **Create** links to add a post and explore the other options created by the scaffold.

Note In a production application, we would want to validate the data in our **Edit** and **Create** methods

Models and Entities

Now that we have an understanding of the Entity Framework, let's begin the transition from Models to Entities. We have an application that is ready to run, now let's go over the differences between a Model and an Entity:

- **Model:** A basic abstraction of information to separate the business logic of the controller. It is an object that retrieves and delivers information from a user that is then manipulated by a controller.
- **Entity:** An entity is basically a model, but it represents an object or element of an object that you want to place in a storage element (such as a database). It interacts with other entities to create a "Data Model" that has a *one-to-one*, *zero-or-one-to-many*, or *many-to-many* relationship.

The five states of an Entity

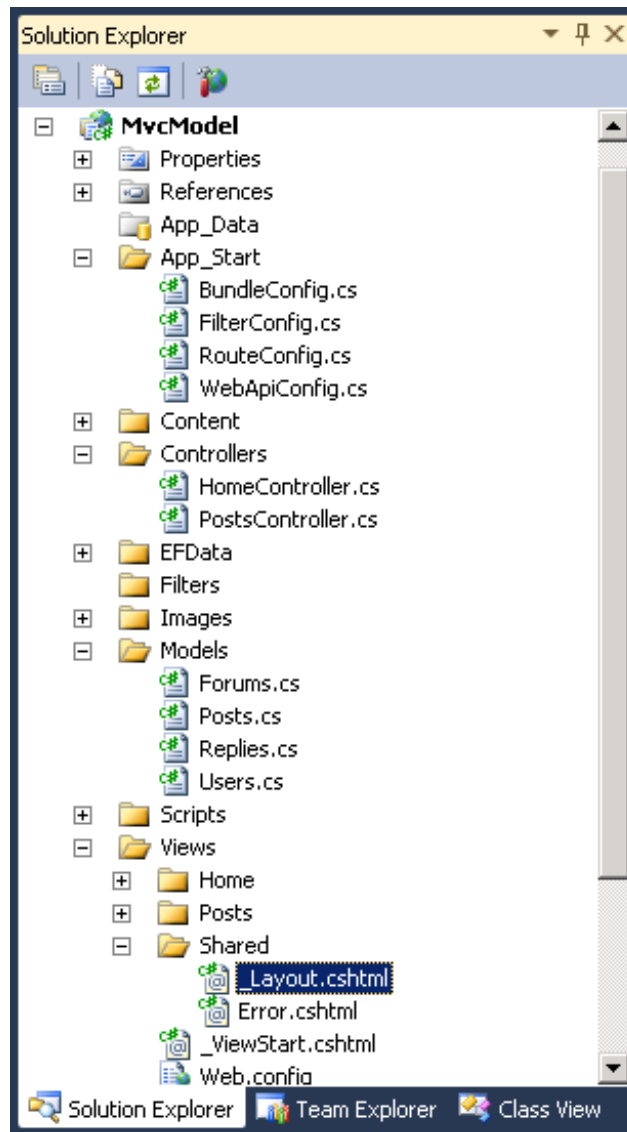
An Entity has five possible states, which is how a database context keeps entities in sync with the database:

- **Added:** An entity does not yet exist in a database. A method must issue an INSERT statement to the database. Ours uses *SaveChanges*.
- **Unchanged:** The default when an entity is read from the database and nothing needs to be done.
- **Modified:** Some or all of the entities' properties have changed and *SaveChanges* must issue an UPDATE statement to the database.
- **Deleted:** An entity has been marked for deletion and *SaveChanges* issues a DELETE command to the database.

- **Detached:** An entity is being tracked by the database context (ForumContext in this lesson).

Let's remove unnecessary clutter in our project. Delete these items:

- /App_Start/**AuthConfig.cs**
- /Controllers/**AccountController.cs**
- /Filters/**InitializeSimpleMembershipAttribute.cs**
- /Models/**AccountModels.cs**
- /Views/**Account** (folder)
- /Views/Shared/**_LoginPartial.cshtml**



Remove the reference to `_LoginPartial` from **Layout.cshtml** as shown:

CODE TO TYPE: /Views/Shared/_Layout.cshtml

```

.
.
.
        <div class="float-left">
            <p class="site-title">@Html.ActionLink("OST Forum", "Index", "Home"
) </p>
        </div>
        <div class="float-right">
            <section id="login">
            @Html.Partial("_LoginPartial")
            </section>
            <nav>
                <ul id="menu">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Forums", "Index", "Forum")</li>
                    <li>@Html.ActionLink("Posts", "Index", "Posts")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
            </nav>
        </div>
.
.
.

```

Modify **/Models/Posts.cs** as shown:

CODE TO TYPE: /Models/Posts.cs

```

.
.
.
[Key]
public int PostId { get; set; }

public int? ReplyId { get; set; }

[Required]
[Display(Name = "Title")]
[StringLength(100, ErrorMessage = "{0} must be a minimum of {2} and a maximum of {1} ch
aracters.", MinimumLength = 3)]
public string postTitle { get; set; }

[Required]
[Display(Name = "Post")]
[StringLength(1200, ErrorMessage = "Post must be less than {1} characters.")]
public string postValue { get; set; }

[Required]
[Display(Name = "Forum")]
public string postForum { get; set; }

[Display(Name = "Date")]
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
public DateTime date { get; set; }

public virtual Replies replies { get; set; }
public virtual Forums forum { get; set; }
public virtual ICollection<Forums> forums { get; set; }
.
.
.

```

The **[Required]** attribute makes an Entity property a required item.

StringLength(100, ErrorMessage = "{0} must be a minimum of {2} and a maximum of {1} characters.", MinimumLength = 3) sets the maximum string length that can be entered. **ErrorMessage** is the message to display if the length exceeds or does not meet the standards. **MinimumLength** is the shortest length a string can be.

Now, make the necessary changes to **/Models/Forums.cs**:

CODE TO TYPE: /Models/Forums.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcModel.Models
{
    public class Forum
    {
        [Key]
        public int idForumId { get; set; }

        public int? PostId { get; set; }

        [Display(Name = "Forum")]
        public string ForumName { get; set; }

        public virtual ICollection<Posts> posts { get; set; }
        public virtual ICollection<Users> users { get; set; }
    }
}
```

We need to add some attributes to the Replies entity in **/Models/Replies**:

/Models/Replies

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcModel.Models
{
    public class Replies
    {
        [Key]
        public int ReplyId { get; set; }

        [Required(ErrorMessage = "Can not have an empty reply.")]
        [MaxLength(500, ErrorMessage = "Reply must be less then {1} characters.")]
        public string reply { get; set; }

        [DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode=true)]
        public DateTime replyDate { get; set; }
    }
}
```

And we now need to add some data annotations to our Users entity:

CODE TO TYPE: Adding data annotations to /Models/Users.cs

```
[Key]
public int UserId { get; set; }

public int? PostId { get; set; }

[Display(Name = "User")]
public string userName { get; set; }

public virtual Posts posts { get; set; }
public virtual ICollection<Forums> forum { get; set; }
public virtual ICollection<Replies> reply { get; set; }
```

Let's add a bit of error checking to make our application just a bit more robust. Modify **/Controllers/PostsController.cs** as shown:

CODE TO TYPE: PostsController.cs

```
using System.Web.Routing;
.
.
.
[HttpPost]
public ActionResult Create(Posts posts)
{
    try
    {
        if (ModelState.IsValid)
        {
            db.posts.Add(posts);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (DataException err)
    {
        ModelState.AddModelError("Error in creating a new post. If the problem persists contact Customer Support.", "Error " + err);
    }

    return View(posts);
}
```

Let's try and create a new post; if it fails we want to log the error, let the user know about the error and what to do about it. We also attach information about the exception with the **DataException err** variable.

In **/Controllers/PostsController.cs**, edit the Delete **Get** method as shown:

CODE TO TYPE: /Controllers/PostsController.cs HttpGet Delete()

```
//
// GET: /Posts/Delete/5

public ActionResult Delete(bool? saveDeleteError, int? id = 0)
{
    Posts posts = db.posts.Find(id);
    if (posts == null!saveDeleteError.GetValueOrDefault())
    {
        ViewBag.Error = "An error has occurred. We are working hard to resolve it. Thank you.";
        return HttpNotFound();
        return View();
    }
    return View(posts db.posts.Find(id));
}
```

Finally, let's increase the efficiency of our **DeleteConfirmed** function:

CODE TO TYPE: /Controllers/PostsController.cs

```
//  
// POST: /Posts/Delete/5  
  
[HttpPost, ActionName("Delete")]  
public ActionResult DeleteConfirmed(int id)  
{  
    Posts post = db.post.Find(id);  
    db.posts.Remove(post);  
    db.SaveChanges();  
    try  
    {  
        var delete = new Posts() { PostId = id };  
        db.Entry(delete).State = EntityState.Deleted;  
        db.SaveChanges();  
    }  
    catch (DataException err)  
    {  
        return RedirectToAction("Delete", new RouteValueDictionary{ { "id", id }, { "saveChangesError", true } });  
    }  
  
    return RedirectToAction("Index");  
}
```

var delete = new Posts() { PostId = id }; instantiates a new Posts() object with only the row we are referencing, using the primary key PostId.

db.Entry(delete).State = EntityState.Deleted; uses the Entity Framework to set the state of the entity to deleted. This is all the Entity Framework needs in order to delete an entity.

RedirectToAction("Delete", new RouteValueDictionary{ { "id", id }, { "saveChangesError", true } }) catches an error with the delete and redirects to the Delete page, displaying the item that the user attempted to delete.

Modify **/Views/Posts/Delete.cshtml** as shown:

CODE TO TYPE: Delete.cshtml

```
@model MvcModel.Models.Posts  
  
@{  
    ViewBag.Title = "Delete";  
}  
  
<h2>Delete</h2>  
<p class="error">@ViewBag.Error</p>  
<h3>Are you sure you want to delete this?</h3>  
<fieldset>  
    <legend>Posts</legend>  
    .  
    .  
    .
```

As you may have noticed, our forum is not too forum-like. It shows no users and has no links. Let's fix that. Modify **/EFData/ForumInitializer.cs** as shown (copy and paste this code, if you like):

CODE TO TYPE: /EFData/ForumInitializer.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using MvcModel.Models;

namespace MvcModel.EFData
{
    public class ForumInitializer : DropCreateDatabaseIfModelChanges<ForumContext>
    {
        protected override void Seed(ForumContext context)
        {
            var replies = new List<Replies>
            {
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2012-09-23") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2001-05-12") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2002-07-11") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2011-07-04") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2000-09-02") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2010-05-14") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2002-07-11") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("2001-12-04") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("1990-09-02") },
                new Replies { reply = "Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...", replyDate = DateTime.Parse("1992-07-11") }
            };
            replies.ForEach(s => context.reply.Add(s));
            context.SaveChanges();

            var users = new List<Users>
            {
                new Users { UserId = 0, userName = "User_z", reply = new List<Replies>() },
                new Users { UserId = 1, userName = "User_a", reply = new List<Replies>() },
                new Users { UserId = 2, userName = "User_b", reply = new List<Replies>() },
                new Users { UserId = 3, userName = "User_c", reply = new List<Replies>() },
                new Users { UserId = 4, userName = "User_d", reply = new List<Replies>() },
                new Users { UserId = 5, userName = "User_e", reply = new List<Replies>() },
                new Users { UserId = 6, userName = "User_f", reply = new List<Replies>() }
            };
        }
    }
}

```

```

        users.ForEach(s => context.user.Add(s));
        context.SaveChanges();

        var posts = new List<Posts>
        {
            new Posts { postTitle = "A new Post", postValue = "this is a test post"
, postForum = "Forum_A", date = DateTime.Parse("2010-05-05") },
            new Posts { postTitle = "Another post", postValue = "This is another po
st", postForum = "Forum_B", date = DateTime.Parse("2010-12-03") },
            new Posts { postTitle = "A Third Post", postValue = "some kind of post
again", postForum = "Forum_C", date = DateTime.Parse("2005-01-04") },
            new Posts { postTitle = "Some post", postValue = "This is \"SomePost\""
, postForum = "Forum_A", date = DateTime.Parse("2003-04-04") },
            new Posts { postTitle = "the Fifth Post", postValue = "This is a fifth
post", postForum = "Forum_B", date = DateTime.Parse("2002-05-19") },
            new Posts { postTitle = "the sixth post", postValue = "This is the sixt
h post!", postForum = "Forum_C", date = DateTime.Parse("2006-08-19") },
            new Posts { postTitle = "Posting", postValue = "Yet Another Post", post
Forum = "Forum_A", date = DateTime.Parse("2012-12-12") },
            new Posts { postTitle = "Still posting", postValue = "We're coming to a
close on posts", postForum = "Forum_A", date = DateTime.Parse("1999-12-23") },
            new Posts { postTitle = "Still Filling", postValue = "Isn't seeding a d
atabase fun?!", postForum = "Forum_C", date = DateTime.Parse("2006-09-09") },
            new Posts { postTitle = "Final Post Seed", postValue = "A final seed po
st value!", postForum = "Forum_B", date = DateTime.Parse("2004-08-19") }
        };
        posts.ForEach(s => context.posts.Add(s));
        context.SaveChanges();

        var forum = new List<Forums>
        {
            new Forums { ForumName = "Forum_A" , posts = new List<Posts>() },
            new Forums { ForumName = "Forum_B" , posts = new List<Posts>() },
            new Forums { ForumName = "Forum_C" , posts = new List<Posts>() }
        };
        forum.ForEach(s => context.forums.Add(s));
        context.SaveChanges();

        forum[0].posts.Add(posts[0]);
        forum[0].posts.Add(posts[3]);
        forum[0].posts.Add(posts[6]);
        forum[0].posts.Add(posts[7]);
        forum[1].posts.Add(posts[1]);
        forum[1].posts.Add(posts[4]);
        forum[1].posts.Add(posts[9]);
        forum[2].posts.Add(posts[2]);
        forum[2].posts.Add(posts[5]);
        forum[2].posts.Add(posts[8]);
    }
}

```

Here, we just seeded some more values and assigned **Posts** to **Forums**. Now, make the necessary modifications to **/ForumContext.cs**:

CODE TO TYPE: /ForumContext.cs

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using MvcModel.Models;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace MvcModel.EFData
{
    public class ForumContext : DbContext
    {
        public DbSet<Forums> forum { get; set; }
        public DbSet<Users> user { get; set; }
        public DbSet<Posts> post { get; set; }
        public DbSet<Replies> reply { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

            /* (One -> zero) or (one -> many) relationship */
            modelBuilder.Entity<Users>().HasOptional(u => u.posts);

            /* Many to Many relationship */
            modelBuilder.Entity<Forums>().HasMany(m => m.posts).WithMany(p => p.forums)
                .Map(k => k.MapLeftKey("ForumId"))
                .MapRightKey("PostId")
                .ToTable("ForumTitlePost");

            modelBuilder.Entity<Posts>().HasOptional(r => r.replies);
        }
    }
}
```

Let's discuss these new Entity framework objects:

OBSERVE:

```
modelBuilder.Entity<Users>().HasOptional(u => u.posts);

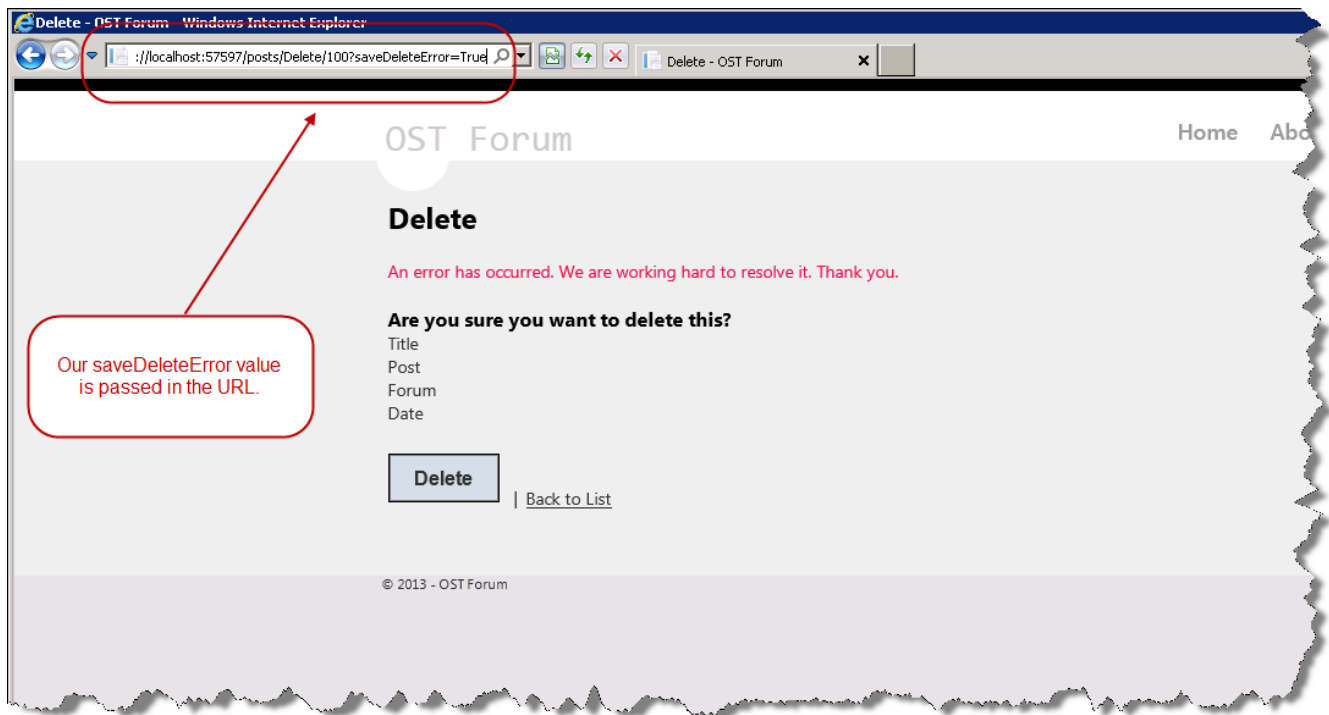
/* Many to Many relationship */
modelBuilder.Entity<Forums>().HasMany(m => m.posts).WithMany(p => p.forums
)
    .Map(k => k.MapLeftKey("ForumId"))
    .MapRightKey("PostId")
    .ToTable("ForumTitlePost");
```

modelBuilder.Entity creates a relationship between Entities. **.HasOptional** creates a one-to-zero or one-to-many relationship between two entities. For example, a user may have zero or many posts. **.HasMany | .WithMany** creates a many-to-many relationship. It can be read like this: A forum entity **HasMany** posts **WithMany** forums. **Map | MapLeftKey | MapRightKey** creates a mapping between two entities. The **MapLeftKey** is an ID of one entity and **MapRightKey** is an ID of the other entity. **ToTable** creates a table from the mapping that we just discussed.

Tip This is known as the Entity Framework *Fluent API*. For additional resources, see [Fluent API](#).



and . Add **/Posts/Delete/100** to the URL. When asked if you want to delete it, click **Delete**. Your web page displays an error message because item 100 doesn't exist in the database:



What if we want to display only Forum_A or we want to find a certain post? We can do this with Searching. Modify **/Controllers/PostsController.cs** as shown:

CODE TO TYPE: /Controllers/PostsController.cs

```
public ActionResult Index(string SortValue)
{
    ViewBag.ForumSort = String.IsNullOrEmpty(SortValue) ? "Forum" : "";
    ViewBag.DateSort = SortValue == "Date" ? "Date desc" : "Date";

    var post = from s in db.posts select s;
    switch (SortValue)
    {
        case "Forum":
            post = post.OrderByDescending(s => s.postForum);
            break;
        case "Date":
            post = post.OrderBy(s => s.date);
            break;
        case "Date desc":
            post = post.OrderBy(s => s.date);
            break;
        default:
            post = post.OrderByDescending(s => s.postForum);
            break;
    }
    return View(db.post.ToList());
}
```

The following code accepts a string value in the Index method and populates the appropriate ViewBag value, then creates a collection object that results from a single query:

OBSERVE:

```
public ActionResult Index(string SortValue)
{
    ViewBag.ForumSort = String.IsNullOrEmpty(SortValue) ? "Forum" : "";
    ViewBag.DateSort = SortValue == "Date" ? "Date desc" : "Date";

    var post = from s in db.post select s;
```

ViewBag.ForumSort = String.IsNullOrEmpty(SortValue) ? "Forum" : ""; is a ternary operator that sets the value of **ViewBag.ForumSort** with either *Forum* or an empty string. This ternary operator has the same functionality as this if-else construct:

OBSERVE: If-Else and Ternary

```
if (!String.IsNullOrEmpty(SortValue))
    ViewBag.ForumSort = "Forum";
else
    ViewBag.ForumSort = "";
```

Note

Ternary operators can clean up your code, but should not be used excessively; they can be difficult to read because lots of code can be crammed into a single line.

var post = from s in db.post select s; uses LINQ to Entities to specify a column by which to sort. It creates an IQueryable variable **post** at first. After the switch statement, the IQueryable object becomes a collection only after calling a method like *ToList()*. This method results in a single query of the database.

Now modify **/Views/Posts/Index.cshtml** as shown:

CODE TO TYPE: Index.cshtml (Posts)

```
<tr>
    <th>
        @Html.DisplayNameFor(model => model.postTitle)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.postValue)
    </th>
    <th>
        @Html.ActionLink("Forum", "Index", new { SortValue = ViewBag.ForumSort })
        @Html.DisplayNameFor(model => model.postForum)
    </th>
    <th>
        @Html.ActionLink("Date", "Index", new { SortValue = ViewBag.DateSort })
        @Html.DisplayNameFor(model => model.date)
    </th>
```



and . Click on either the **Forum** or **Date** heading to sort the posts by the value (the Forum sort is descending).

OST Forum

Login currently disabled.

[Home](#) [About](#) [Forum](#)

Index

[Create New](#)

Title	Post	Forum	Date	
Still posting	We're coming to a close on posts	Forum_A	12/23/1999	Edit Details Delete
the Fifth Post	This is a fifth post	Forum_B	5/19/2002	Edit Details Delete
Some post	This is "SomePost"	Forum_A	4/4/2003	Edit Details Delete
Final Post Seed	A final seed post value!	Forum_B	8/19/2004	Edit Details Delete
A Third Post	some kind of post again	Forum_C	1/4/2005	Edit Details Delete
the sixth post	This is the sixth post!	Forum_C	8/19/2006	Edit Details Delete
Still Filling	Isn't seeding a database fun?!	Forum_C	9/9/2006	Edit Details Delete
A new Post	this is a test post	Forum_A	5/5/2010	Edit Details Delete
Another post	This is another post	Forum_B	12/3/2010	Edit Details Delete
Posting	Yet Another Post	Forum_A	12/12/2012	Edit Details Delete

Note

You can use the `ActionLink` with methods like `ViewBag.ForumSort` or `ViewBag.DateSort` for sorting by any column of your data. It's a handy way to sort things, and saves you from having to write a lot of code.

Now, let's add a search bar so we can search our forum for specific elements. Edit `/Views/Posts/Index.cshtml` as shown:

CODE TO TYPE: /Views/Posts/Index.cshtml

```
<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    @using (Html.BeginForm())
    {
        <p>
            <b>Search Forum:</b> @Html.TextBox("SearchQuery", null, new { style = "width: 200px; height: 10px; font-size: 12px;" })
            <input type = "submit" value = "Search" />
        </p>
    }
}
```

We've created an HTML Form that contains a box for search queries. Let's discuss the `TextBox` properties:

OBSERVE:

```
<b>Search Forum:</b> @Html.TextBox("SearchQuery", null, new { style = "width: 200px; height: 10px; font-size: 12px;" })
```

TextBox creates a text box where our user can input queries. We store different Object attributes we may want to use in `null`. Ours has none, so it's null. (We'd give a different specific name if we wanted to use particular object attributes.) `new { style = "width: 200px; height: 14px; font-size: 12px;" }` defines a new inline CSS style element.

Now, let's add the necessary LINQ query for our search box. Modify the Index method in `/Controllers/PostsController.cs` as shown:

CODE TO TYPE: PostsController.cs

```
public ActionResult Index(string SortValue, string SearchQuery)
{
    ViewBag.ForumSort = String.IsNullOrEmpty(SortValue) ? "Forum" : "";
    ViewBag.DateSort = SortValue == "Date" ? "Date desc" : "Date";

    var post = from s in db.posts select s;

    if (!String.IsNullOrEmpty(SearchQuery))
    {
        post = post.Where(s => s.postTitle.ToUpper().Contains(SearchQuery.ToUpper()) ||
                               s.postValue.ToUpper().Contains(SearchQuery.ToUpper()) ||
                               s.postForum.ToUpper().Contains(SearchQuery.ToUpper()));
    }
    switch (SortValue)
    {
        case "Forum":
            post = post.OrderByDescending(s => s.postForum);
            break;
        case "Date":
            post = post.OrderBy(s => s.date);
            break;
        case "Date desc":
            post = post.OrderBy(s => s.date);
            break;
        default:
            post = post.OrderByDescending(s => s.postForum);
            break;
    }
    return View(post.ToList());
}
```

Let's discuss our new LINQ query.


OBSERVE:

```
post = post.Where(s => s.postTitle.ToUpper() .Contains(SearchQuery.ToUpper()) ||
```

Where selects only items from the database that return **true** from the query. **ToUpper()** causes a string element to be temporarily upper case. **Contains** returns **true** if the **postTitle**, **postValue**, or **postForum** contains an element of the search query.

Now we are able to search our forum:



and , then go to the Posts page and enter **Test**.

Index

[Create New](#)

Search Forum:

Title	Post	Forum	Date	
A new Post	this is a test post	Forum_A	5/5/2010	Edit Details Delete

© 2013 - MvcModel

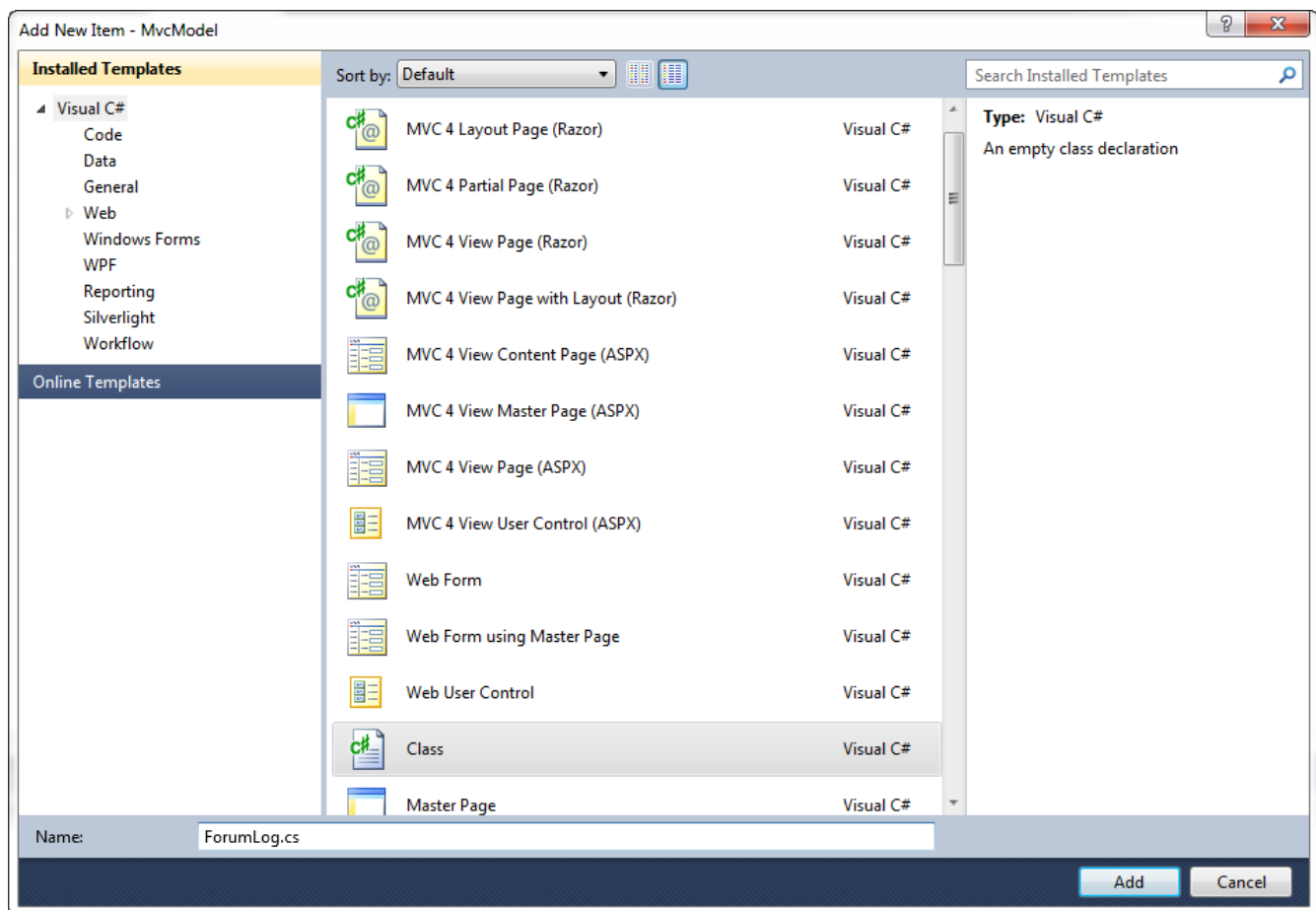
Filters

Filters are *custom* attributes that provide post/pre-action behavior on controller action methods. Filters can execute their logic either *before* or *after* an action method.

While we can create custom filters, ASP.NET MVC 4 has four default filters that we can use "out of the box":

- **Authorization:** Makes security decisions based on authentication or property validation.
- **Action:** Wraps the action method execution. It can provide extra data to method, inspect return value, or cancel execution of an action method.
- **Result:** Wraps execution of *ActionResult*. It can also perform additional processing of a result, or modify HTTP responses.
- **Exception:** Executes if an unhandled exception error occurs in the action method. It provides a hierarchy for handling exceptions. It begins at the top with *Authorization* and cascades down to execution of the *result*. These are useful for displaying error pages and logging.

Create a ForumLog model. Right-click the **Models** folder and add a new class named **ForumLog**:



Modify **/Models/ForumLog.cs** as shown:

CODE TO TYPE: /Models/ForumLog.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcModel.Models
{
    public class ForumLog
    {
        [Key]
        [Display(Name = "Log ID")]
        public int logID { get; set; }

        [Display(Name = "Controller")]
        public string Controller { get; set; }

        [Display(Name = "Action")]
        public string Action { get; set; }

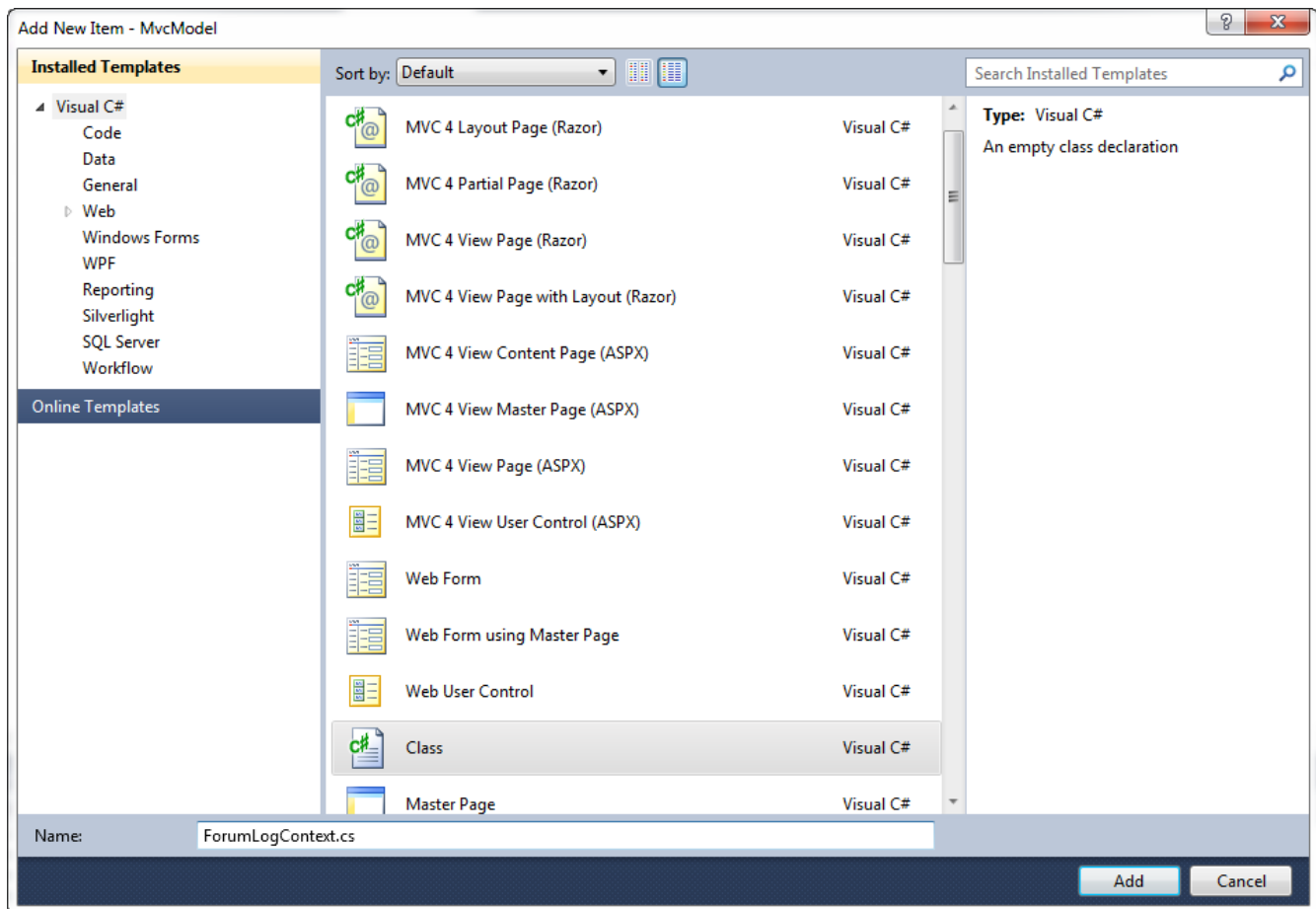
        [Display(Name = "Header")]
        public string BrowserHeader { get; set; }

        [Display(Name = "Browser Type")]
        public string BrowserType { get; set; }

        [Display(Name = "IP Address")]
        public string IP { get; set; }

        [Display(Name = "Access Date")]
        public DateTime Date { get; set; }
    }
}
```

Separate the forum database from the forum log database. Create a **ForumLogContext.cs** class file in the **EFData** folder:



Now add some code to it so we can create the database.

Note

We won't create a seed method because the table, or "forum log," receives data only after a forum user visits the site.

CODE TO TYPE: /EFData/ForumLogContext.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using MvcModel.Models;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace MvcModel.EFData
{
    public class ForumLogContext : DbContext
    {
        public DbSet<ForumLog> forumLog { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

We now need to add a new connection string to **Web.config** so we can save our results.

CODE TO TYPE: /Web.config

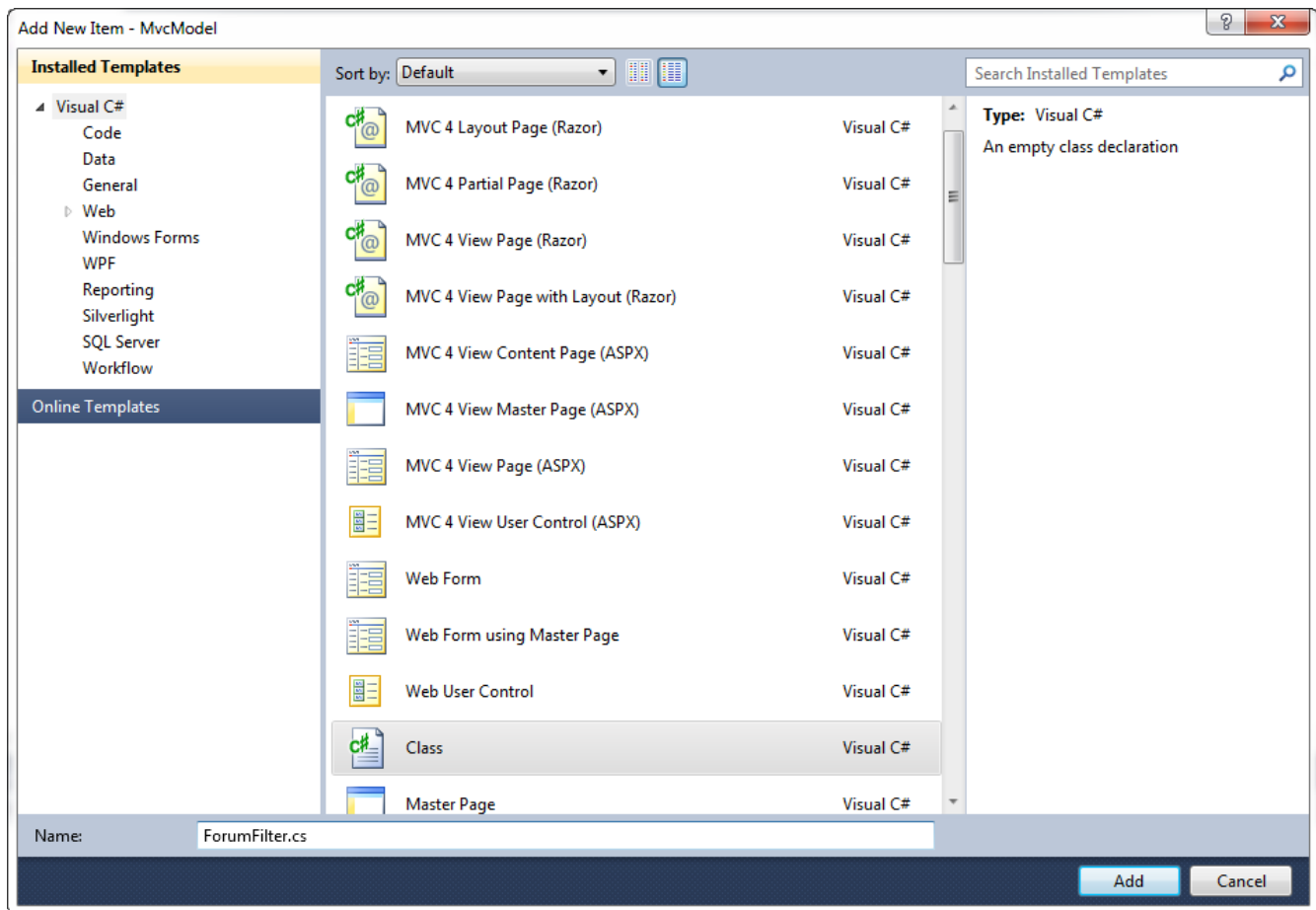
```
<connectionStrings>

    <add name ="ForumContext"
        connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=Forum;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\Forums.mdf;User Instance=true"
        providerName ="System.Data.SqlClient"/>

    <add name ="ForumLogContext"
        connectionString="data source=.\SQLEXPRESS;Initial Catalog=ForumLog;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\ForumLog.mdf;User Instance=true"
        providerName ="System.Data.SqlClient"/>

</connectionStrings>
```

Add a custom filter class. Right-click the **Filters** folder and add a new class named **ForumFilter.cs**:



Add thiscode to our custom filter:

CODE TO TYPE: /Filters/ForumFilter.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using System.Web.Mvc;
using MvcModel.Models;
using MvcModel.EFData;

namespace MvcModel.Filters
{
    public class ForumFilter : ActionFilterAttribute, IActionFilter
    {
        private ForumLogContext logDB = new ForumLogContext();

        void IActionFilter.OnActionExecuting(ActionExecutingContext actionContext)
        {
            ForumLog log = new ForumLog()
            {
                Controller = actionContext.ActionDescriptor.ControllerDescriptor.ControllerName,
                Action = actionContext.ActionDescriptor.ActionName + "(ForumFilter)",
                BrowserHeader = actionContext.HttpContext.Request.UserAgent,
                BrowserType = actionContext.HttpContext.Request.Browser.Browser,
                IP = actionContext.HttpContext.Request.UserHostAddress,
                Date = actionContext.HttpContext.Timestamp
            };

            logDB.forumLog.Add(log);
            logDB.SaveChanges();

            this.OnActionExecuting(actionContext);
        }
    }
}
```

Let's discuss some of this code.

/Filters/ForumFilter.cs

```
public class ForumFilter : ActionFilterAttribute, IActionFilter
{
    private ForumLogContext logDB = new ForumLogContext();

    void IActionFilter.OnActionExecuting(ActionExecutingContext actionContext)
    {
        ForumLog log = new ForumLog()
        {
            ...
        };

        logDB.forumLog.Add(log);
        logDB.SaveChanges();

        this.OnActionExecuting(actionContext);
    }
}
```

- **ForumFilter : ActionFilterAttribute, IActionFilter**: inherits from **ActionFilterAttribute**, and implement the **IActionFilter** interface.
- **ForumLogContext logDB = new ForumLogContext();**: gets access to our forum log database.
- **void IActionFilter.OnActionExecuting**: makes ForumFilter override the **OnActionExecuting** method.
- **ForumLog log = new ForumLog();**: instantiates a new ForumLog entity with the information gathered

from the action.

- **this.OnActionExecuting(actionContext);** OnActionExecuting uses the Entity Framework to create and populate an entity instance of *actionContext*.

Add a line to **/Global.asax.cs** in the *Application_Start* method so we can initialize the database:

CODE TO TYPE: /Global.asax.cs

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
    Database.SetInitializer<ForumContext>(new ForumInitializer());
    Database.SetInitializer<ForumLogContext>(new DropCreateDatabaseIfModelChanges<ForumLogContext>());
}
```

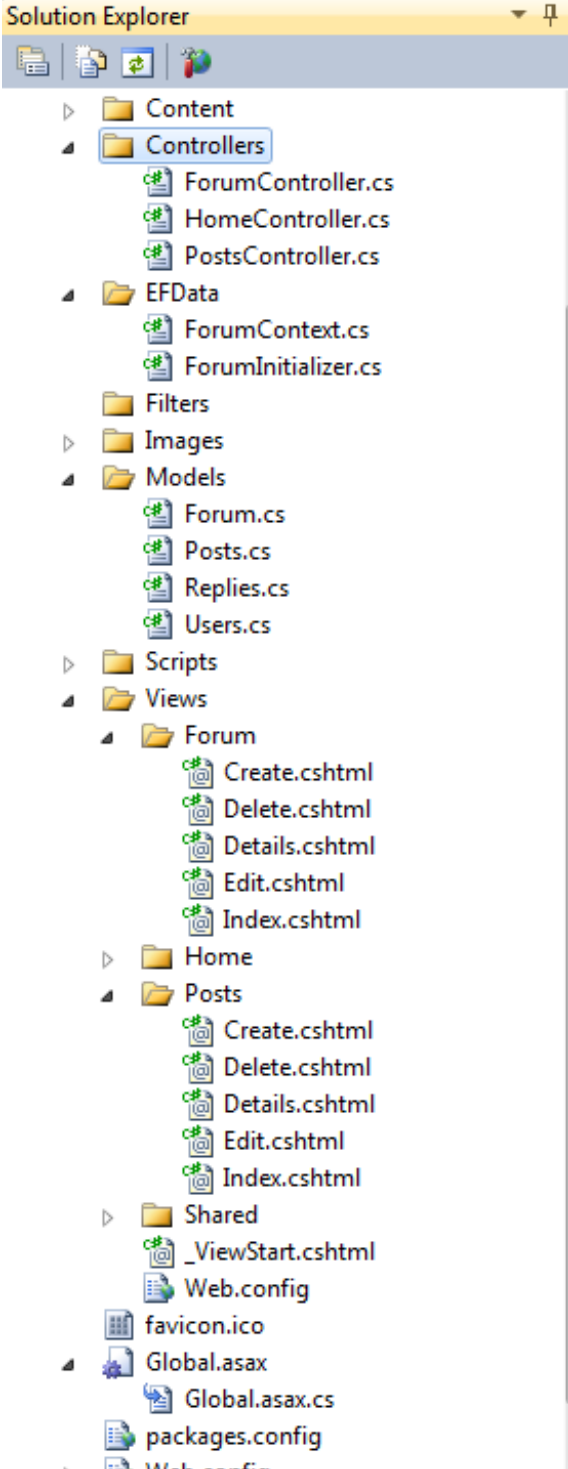


and **Build (F6)** the application, but don't run it yet.

Now instead of sending users to **Forums/Posts/**, we'll create an actual forum-like option using a Forum controller. Right-click the **Controllers** folder and select **Add | Controller**. Name it **ForumController.cs**, set the Model class to **Forums (MvcModel.Models)** and set the Data context class set to **ForumContext (MvcModel.EFData)**.

The screenshot shows the 'Add Controller' dialog box. The 'Controller name' field is filled with 'ForumController'. Under the 'Scaffolding options' section, the 'Template' dropdown is set to 'MVC controller with read/write actions and views, using Entity Framework'. The 'Model class' dropdown is set to 'Forum (MvcModel.Models)'. The 'Data context class' dropdown is set to 'ForumContext (MvcModel.EFData)'. The 'Views' dropdown is set to 'Razor (CSHTML)'. There are 'Add' and 'Cancel' buttons at the bottom right, and an 'Advanced Options...' button next to the Views dropdown.

Our Solution Explorer looks like this:



Add the custom filter to the methods we want to use it on. Modify **/Controllers/ForumController.cs** as shown:

CODE TO TYPE: /Controllers/ForumController.cs snippet

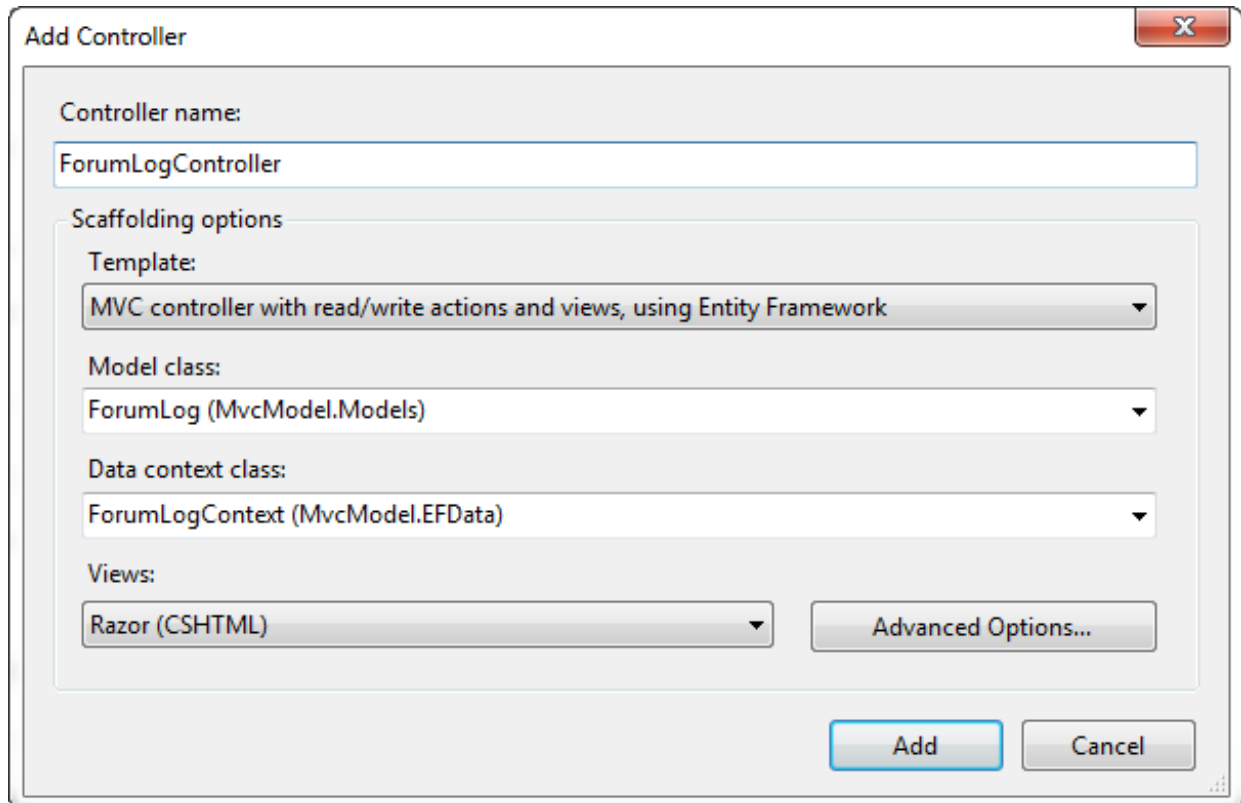
```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcModel.Models;
using MvcModel.EFData;
using MvcModel.Filters;

namespace MvcModel.Controllers
{
    [ForumFilter]
    public class ForumController : Controller
    {
        private ForumContext db = new ForumContext();

        //
        // GET: /Forum/
    }
}
```

We added our custom filter to *all* the actions in the ForumController. When we access any of the actions that use the forum controller, our custom filter will be invoked.

Now we need a way to see the information we store in the ForumLog database. For this, we need a controller and a view. Add a controller to the **Controllers** folder named ForumLogController.cs:



The screenshot shows the 'Add Controller' dialog box. The 'Controller name' field contains 'ForumLogController'. Under the 'Scaffolding options' section, the 'Template' is set to 'MVC controller with read/write actions and views, using Entity Framework'. The 'Model class' is set to 'ForumLog (MvcModel.Models)'. The 'Data context class' is set to 'ForumLogContext (MvcModel.EFData)'. The 'Views' dropdown is set to 'Razor (CSHTML)'. There are three buttons at the bottom: 'Add', 'Cancel', and 'Advanced Options...'.

Open **_Layout.cshtml** and create a link to our log:

CODE TO TYPE: _Layout.cshtml

```
<ul id="menu">
  <li>@Html.ActionLink("Home", "Index", "Home")</li>
  <li>@Html.ActionLink("About", "About", "Home")</li>
  <li>@Html.ActionLink("Forums", "Index", "Forum")</li>
  <li>@Html.ActionLink("Posts", "Index", "Posts")</li>
  <li>@Html.ActionLink("Forum Log", "Index", "ForumLog")</li>
  <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
```

Tip We can also inject the custom filter into individual methods of controller classes.

Modify **/Views/ForumLog/Index.cshtml** as shown:

CODE TO TYPE:

```
@model IEnumerable<MvcModel.Models.ForumLog>

@{
    ViewBag.Title = "IndexForum Log";
}

<h2>IndexForum Log</h2>
.
.
.
```



and . Go to the Forum Log page and create a forum log entry; the *Forum Log* looks like this.

OST Forum

Login currently disabled.

[Home](#) [About](#) [Forum](#) [Posts](#) [Forum Log](#)

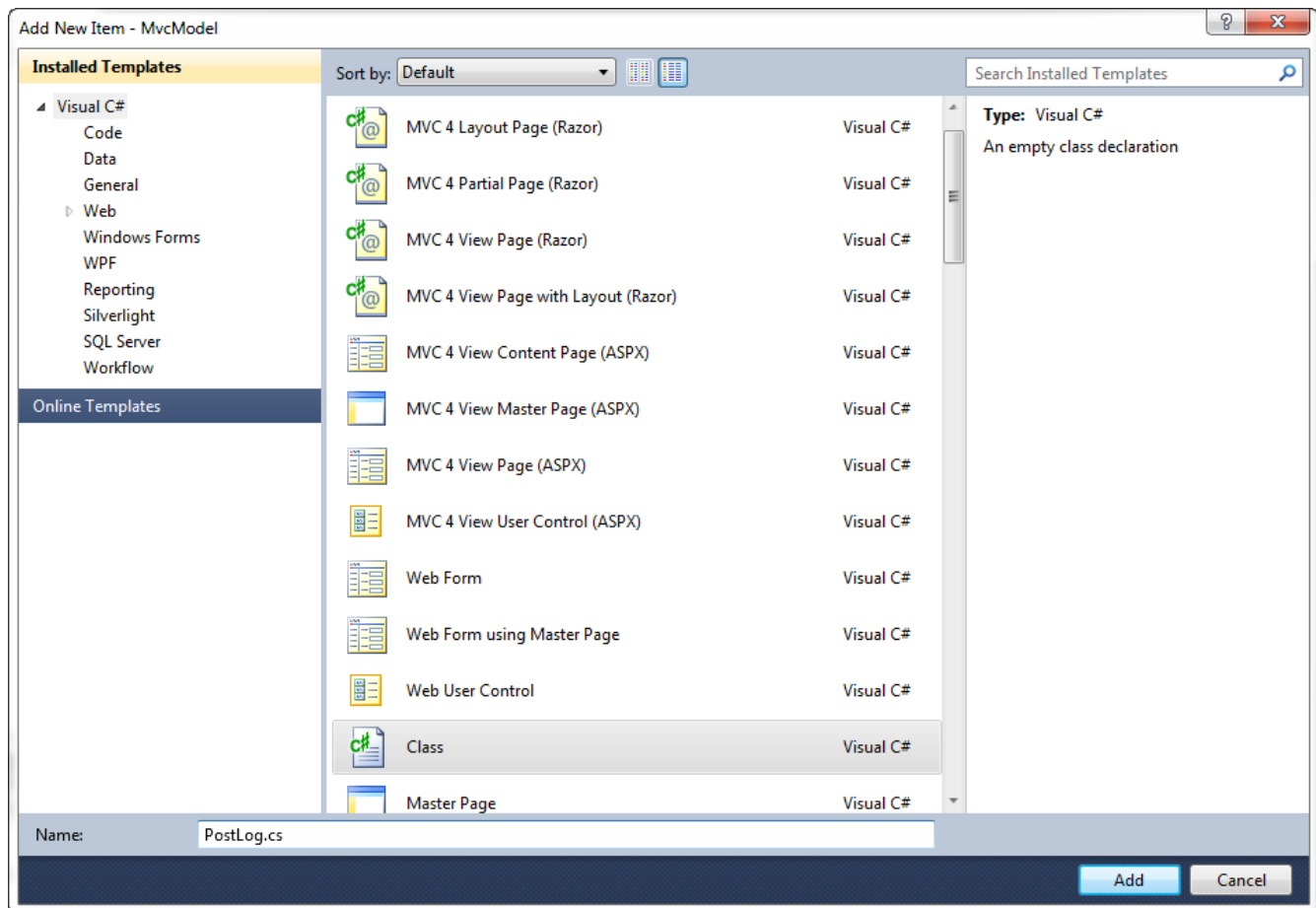
Index

[Create New](#)

ControllerAction		Browser Type	Header	IP Address	Access Date	
Forum	Index	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 5:39:18 PM	Edit Details Delete

Now we want to add multiple filters into a controller:

Add a **PostLog.cs** class in the *Models* folder:



Modify **/Models/PostLog.cs** as shown:

CODE TO TYPE: PostLog.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcModel.Models
{
    public class PostLog
    {
        [Key]
        public int postLogID { get; set; }

        [Display(Name = "Controller")]
        public string Controller { get; set; }

        [Display(Name = "Action")]
        public string Action { get; set; }

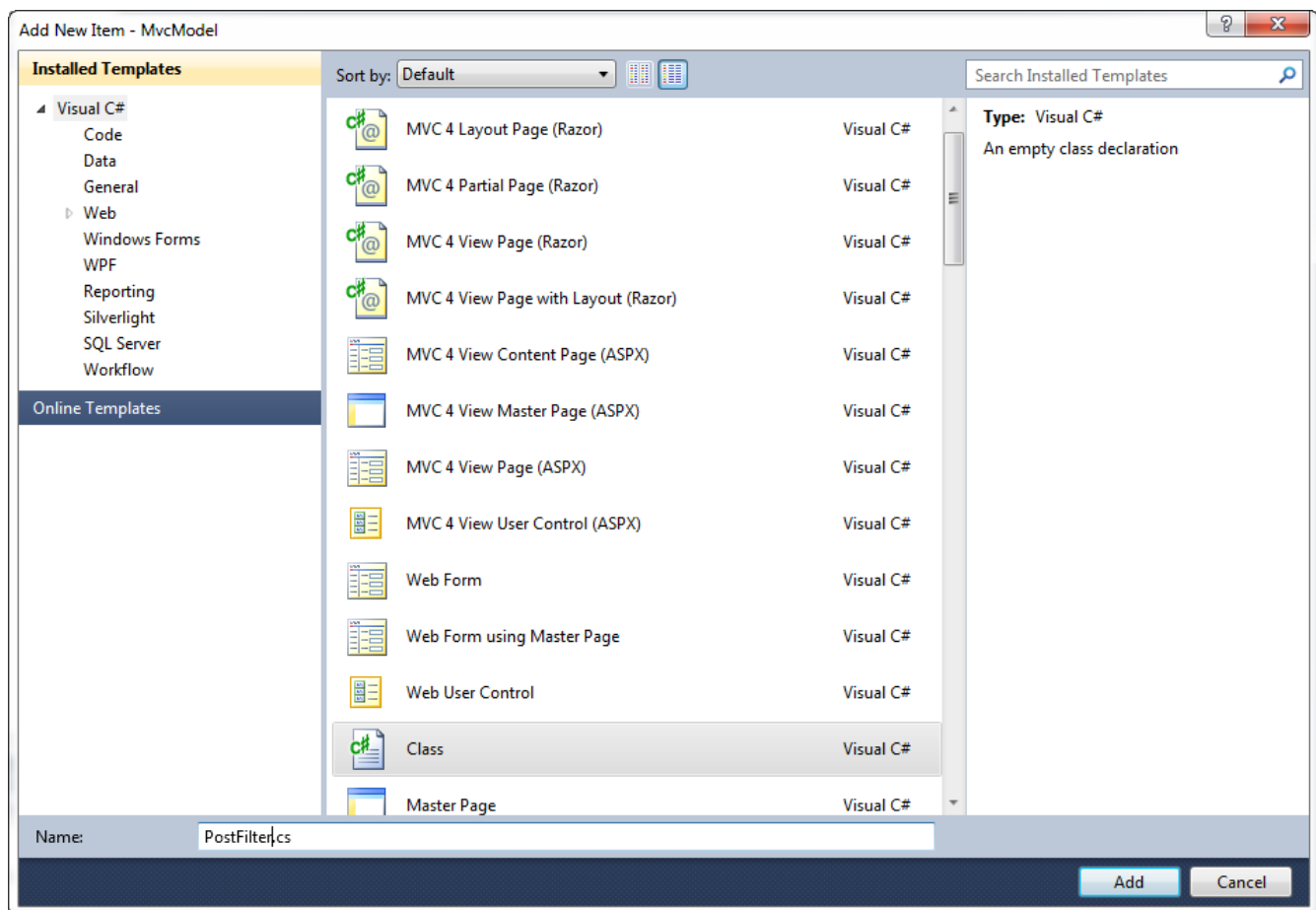
        [Display(Name = "Browser")]
        public string BrowserType { get; set; }

        [Display(Name = "Browser Header")]
        public string BrowserHeader { get; set; }

        [Display(Name = "IP Address")]
        public string IP { get; set; }

        [Display(Name = "Access Date")]
        public DateTime Date { get; set; }
    }
}
```

Create a new class in the **Filters** folder named **PostFilter.cs**:



This filter is identical to the ForumFilter. We're using it here to show how to use two different filters on the same class:

CODE TO TYPE: /Filters/PostFilter.cs

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

using System.Data.Entity;
using System.Web.Mvc;
using MvcModel.Models;
using MvcModel.EFData;

namespace MvcModel.Filters
{
    public class PostFilter : ActionFilterAttribute, IActionFilter
    {
        private ForumLogContext logDB_2 = new ForumLogContext();

        void IActionFilter.OnActionExecuting(ActionExecutingContext actionContext)
        {
            ForumLog log = new ForumLog()
            {
                Controller = actionContext.ActionDescriptor.ControllerDescriptor.ControllerName,
                Action = actionContext.ActionDescriptor.ActionName + "(PostFilter)",
                BrowserHeader = actionContext.HttpContext.Request.UserAgent,
                BrowserType = actionContext.HttpContext.Request.Browser.Browser,
                IP = actionContext.HttpContext.Request.UserHostAddress,
                Date = actionContext.HttpContext.Timestamp
            };
            logDB_2.forumLog.Add(log);
            logDB_2.SaveChanges();

            this.OnActionExecuting(actionContext);
        }
    }
}
```

Note We are using the same database, just adding a new table to it.

Now open **ForumLogContext.cs** in the *EFData* folder and add the PostLog filter to it:

CODE TO TYPE: /EFData/ForumLogContext.cs

```
public class ForumLogContext : DbContext
{
    public DbSet<ForumLog> forumLog { get; set; }
    public DbSet<PostLog> postLog { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

Add the filter to the class. Modify **/Controllers/ForumController.cs** as shown:

CODE TO TYPE: /Controllers/ForumController.cs

```
namespace MvcModel.Controllers
{
    [PostFilter(Order = 1)]
    [ForumFilter(Order = 2)]
    public class ForumController : Controller
    {
        private ForumContext db = new ForumContext();

        //
        // GET: /Forum/

        public ActionResult Index()
        {
            return View(db.forum.ToList());
        }
    }
}
```

We added **Order** to the filters to establish a hierarchy for how the filters will be applied.



and .

Forum Log

[Create New](#)

ControllerAction		Browser Type	Header	IP Address	Access Date	
Forum	Index(ForumFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 6:39:41 PM	Edit Details Delete
Forum	Index(ForumFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 6:41:52 PM	Edit Details Delete
Forum	Index(ForumFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 6:42:31 PM	Edit Details Delete
Forum	Index(PostFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 6:46:30 PM	Edit Details Delete
Forum	Index(ForumFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 6:46:30 PM	Edit Details Delete
Forum	Index(PostFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:37 PM	Edit Details Delete
Forum	Index(ForumFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:37 PM	Edit Details Delete
Forum	Details(PostFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:44 PM	Edit Details Delete
Forum	Details(ForumFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:44 PM	Edit Details Delete
Forum	Index(PostFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:45 PM	Edit Details Delete
Forum	Index(ForumFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:45 PM	Edit Details Delete
Forum	Edit(PostFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:47 PM	Edit Details Delete
Forum	Edit(ForumFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:47 PM	Edit Details Delete
Forum	Index(PostFilter)	Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0	::1	5/24/2013 7:12:48 PM	Edit Details Delete

Notice the order in which the filters are used. First **PostFilter** is accessed and then **ForumFilter** is accessed. We can use these methods to establish a hierarchy of filters that have varied levels of authentication or validation.

Tip You can find lots of good information about filtering at [MVC Filters](#).

Phew! That was a fairly lengthy lesson. Good job sticking with it. Practice what you've learned here in the homework to make sure it sticks with you!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Unit Testing

Lesson Objectives

In this lesson, you will:

- add test a test project to your ASP.NET MVC projects.
 - conduct testing as part of the software development process.
 - use unit testing, as well as integrated testing.
 - employ an automated testing process.
 - test to validate software.
-

Testing with Visual Studio

When you automate the software testing process, you can identify errors in your software code quickly, and enforce its original functional specifications and requirements. There are many different testing strategies:

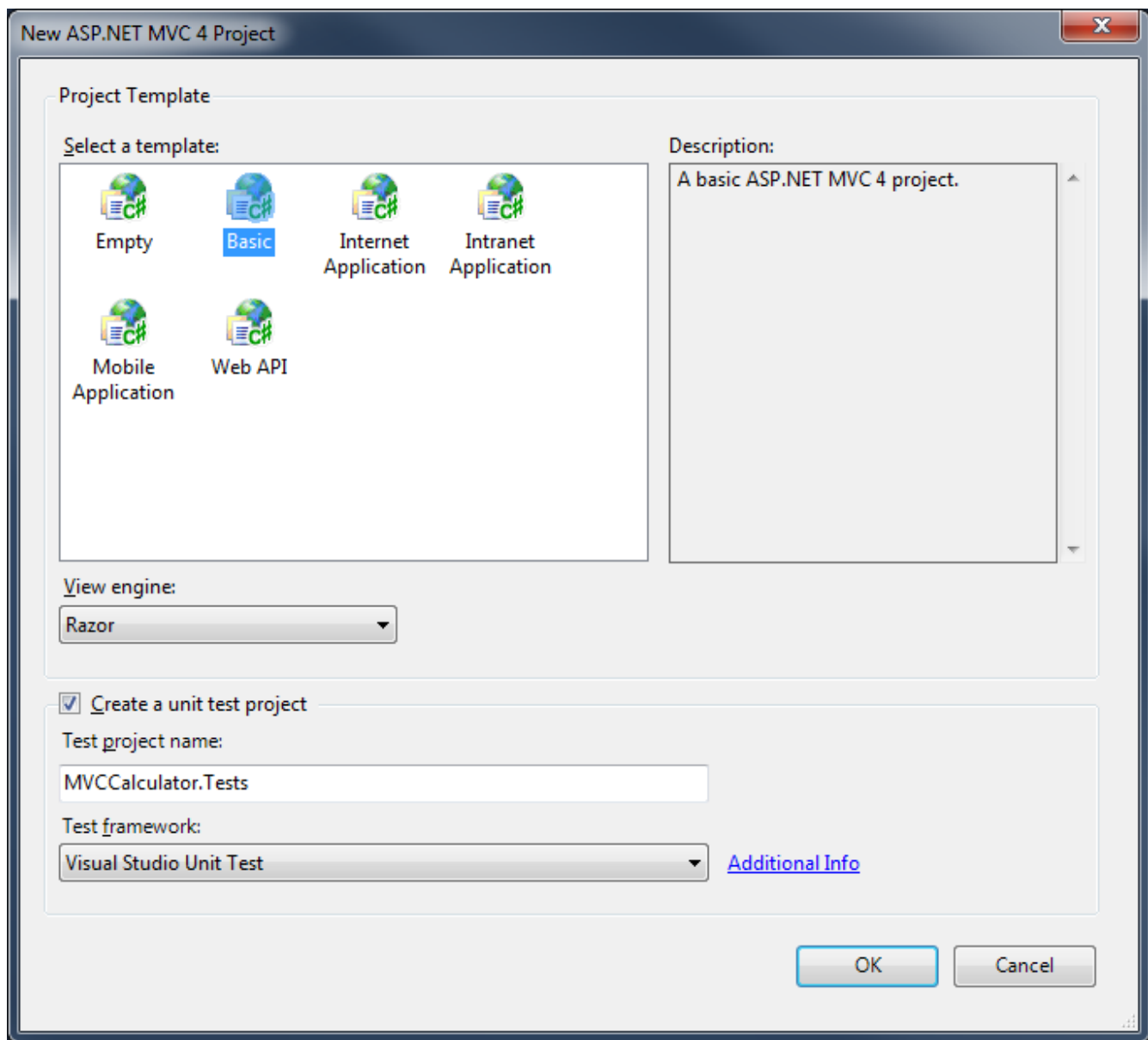
- Having a user run the software and test for errors.
- Utilize some type of testing framework or software.
- Integrate testing functions within the same project (with or without a testing framework).
- Create a separate testing project that "exercises" all of the software methods (with or without a testing framework).

For this lesson, we'll employ Visual Studio Test Professional integrated into Visual Studio. We'll create a companion test project that will co-exist as a separate project within our Visual Studio Solution. Let's get started!

Creating the Project

First, let's create the Studio Solution that will contain our application project and test project. For this lesson, we'll recreate the calculator project from an earlier lesson, but this time we will create the project using MVC.

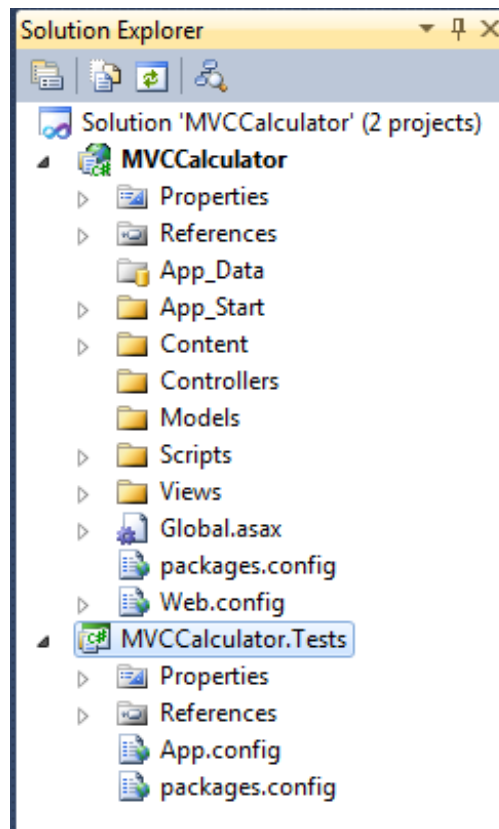
To create the project, select **File | New | Project**. In the New Project dialog box, make sure **Visual C#** is selected under Installed Templates, and select the **ASP.NET MVC 4 Web Application** template. Change the Name of the Project to **MVCCalculator**. Click **OK**. In the New ASP.NET MVC 4 Project dialog box, select the **Basic** template, make sure the **Create a unit test project** is checked, make sure the Test project name is **MVCCalculator.Tests**, and click **OK**:



Tip

It's a common naming convention to give the test project the same name as the project being tested, with **.Tests** appended to the original project name.

When Visual Studio finishes creating your projects, the Solution Explorer shows a Solution, as well as both the application project (MVCCalculator) and the test project (MVCCalculator.Tests). A Studio Solution enables you to group two or more projects or related items into a single unit.



Before we begin to work through testing concepts and add tests, let's recreate our calculator using MVC and Razor. We won't spend much time explaining how the calculator works, except where we've modified our code to work as an MVC application by using a model, view (using Razor), and a controller.

Add a model for the calculator. Right-click the **Models** folder and select **Add | Class**. In the Add New Item dialog box, make sure that the Visual C# template is selected, enter **Calculator** for the Name, and click **Add**. Modify **/Models/Calculator.cs** as shown:

CODE TO TYPE: /Models/Calculator.cs

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

namespace MVCCalculator.Models
{
    public class Calculator
    {
        public double CurrentValue { get; set; }
        public bool FirstOperation { get; set; }
        public double PendingValue { get; set; }
        public string PendingAction { get; set; }

        public Calculator()
        {
            CurrentValue = 0.0D;
            FirstOperation = true;
            PendingValue = 0.0D;
            PendingAction = "";
        }

        public void Process(string key)
        {
            if (string.IsNullOrEmpty(PendingAction)) PendingAction = "";

            double number;
            if (double.TryParse(key, out number))
                ProcessNumber(number);
            else
                ProcessAction(key);
        }

        private void ProcessAction(string currentAction)
        {
        }

        private void ProcessNumber(double number)
        {
        }

        private double DoCalculation(string action, double left, double right)
        {
        }
    }
}
```

We'll add code to the methods shortly, but first let's discuss this code and how it will work differently from the web forms code we used earlier.

/Models/Calculator.cs

```
.  
.   
.   
public double CurrentValue { get; set; }  
public bool FirstOperation { get; set; }  
public double PendingValue { get; set; }  
public string PendingAction { get; set; }  
  
public Calculator()  
{  
    CurrentValue = 0.0D;  
    FirstOperation = true;  
    PendingValue = 0.0D;  
    PendingAction = "";  
}  
  
public void Process(string key)  
{  
    if (string.IsNullOrEmpty(PendingAction)) PendingAction = "";  
  
    double number;  
    if (double.TryParse(key, out number))  
        ProcessNumber(number);  
    else  
        ProcessAction(key);  
}  
  
private void ProcessAction(string currentAction)  
{  
}  
  
private void ProcessNumber(double number)  
{  
}  
  
private double DoCalculation(string action, double left, double right)  
{  
}  
.  
.  
.
```

When you create a model, keep in mind that we will have access to the public member properties of the model within the view. We will be able to populate the model properties automatically within our Razor code, then have access to those properties in the controller via our model. Each public class property (**CurrentValue**, **FirstOperation**, **PendingValue**, **PendingAction**) helps enable the calculator functionality. We've omitted the string equivalent values of **CurrentValue** and **PendingValue** because they're not as useful in this revision. We've added a constructor **Calculator** to make sure that our class properties are initialized correctly, then we added methods that we'll call to distribute the actual calculator functionality. The only public method we're using is **Process** which is called to initiate each calculator response. The processing changes, depending on whether a number was selected (**ProcessNumber**), or we have an action (**ProcessAction**). We've broken down the processing a bit further, into a calculation method (**DoCalculation**).

Tip

Throughout these lessons, we've encouraged you to create small, encapsulated methods which helps to streamline code maintenance. This technique also simplifies the process of automated testing, which allows us to create isolated test scenarios more efficiently.

We use the **string.IsNullOrEmpty** method to address form submission behavior where our **PendingAction** string property becomes null. In other words, the method handles the case where a user tries to submit the form with an empty **PendingAction** string.

Let's add the actual code for the methods:

```

.
.
.
private void ProcessAction(string currentAction)
{
    bool pendingOperation = (PendingAction.Length > 0);
    if ("/*-= ".IndexOf(currentAction) >= 0)
    {
        if (pendingOperation)
            CurrentValue = DoCalculation(PendingAction, PendingValue, CurrentValue);

        FirstOperation = true;

        if (currentAction != "=")
        {
            PendingAction = currentAction;
            PendingValue = CurrentValue;
        }
        else
        {
            PendingAction = "";
            PendingValue = 0.0D;
        }
    }
    else
    {
        switch (currentAction)
        {
            case "C":
                // Reset calculator window and hidden fields
                CurrentValue = 0.0D;
                PendingAction = "";
                PendingValue = 0.0D;
                FirstOperation = true;
                break;
            case "CE":
                // Clear error
                CurrentValue = 0.0D;
                FirstOperation = true;
                break;
            case "<-":
                // Backspace - prevent leaving "bad" data in calculator window
                // Not implemented
                break;
            case ".":
                // Decimal point
                // Not implemented
                break;
            case "+/-":
                // Sign
                CurrentValue *= -1;
                break;
        }
    }
}

private void ProcessNumber(double number)
{
    if (FirstOperation)
        CurrentValue = number;
    else
        CurrentValue = CurrentValue * 10.0 + number;

    FirstOperation = false;
}

```

```
private double DoCalculation(string action, double left, double right)
{
    // Perform arithmetic calculations
    double result = 0.0;
    switch (action)
    {
        case "/":
            // Prevent divide by zero
            if (right != 0)
                result = left / right;
            else
            {
                // TODO: Handle divide by zero error
            }
            break;
        case "*":
            result = left * right;
            break;
        case "-":
            result = left - right;
            break;
        case "+":
            result = left + right;
            break;
    }
    return result;
}
.
.
.
```

We've left in the implementation conversion of the backspace and the decimal point actions for the lesson project.

Next, add a controller for our project. Right-click the **Controllers** folder and select **Add | Controller**. Change the ControllerName to **HomeController**, verify that **Empty MVC Controller** is selected as the Template under the Scaffolding options, and click **Add**:

The screenshot shows the 'Add Controller' dialog box. The 'Controller name' field is filled with 'HomeController'. Under the 'Scaffolding options' section, the 'Template' dropdown menu is open and shows 'Empty MVC controller' as the selected option. Below this, the 'Model class' and 'Data context class' fields are empty. The 'Views' dropdown menu is also open and shows 'None' as the selected option. There is an 'Advanced Options...' button to the right of the 'Views' dropdown. At the bottom of the dialog, there are 'Add' and 'Cancel' buttons.

Modify **/Controllers/HomeController.cs** as shown:

CODE TO TYPE:

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

namespace MVCCalculator.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

        public ActionResult Index()
        {
            return View(new Calculator());
        }

        [HttpPost]
        public ActionResult Index(Calculator calculator)
        {
            calculator.Process(Request.Form["key"]);
            ModelState.Clear();
            return View(calculator);
        }
    }
}
```

Most of this code will be familiar to you, but let's go over a couple of lines for clarification:

HomeController.cs

```
public ActionResult Index()
{
    return View(new Calculator());
}

[HttpPost]
public ActionResult Index(Calculator calculator)
{
    calculator.Process(Request.Form["key"]);
    ModelState.Clear();
    return View(calculator);
}
```

When the page first loads, a new **Calculator** is created. In response to POST actions from the form (**HttpPost**), a **calculator** instance is returned, but this time the instance was returned from the view, populated with previous values. We process the action based on the **"key"**, and then we **clear ModelState**. The ModelState contains the information from the POST action. We need the POST action object because it allows us to differentiate the previous values from the current values. We clear the current values so that our calculator model can populate the form fields again.

Finally, we'll add a view to enable the calculator to work. Right-click on the first Index method (not the version decorated with the HttpPost attribute), and select **Add View**. In the Add View dialog box, accept the default name **Index**, ensure the View engine is **Razor (CSHTML)**, and that **Create a strongly-typed view** is checked. Select the **Calculator** class as the model (if you don't see it in the list, rebuild the project), and click **Add**:

Add View [X]

View name:

View engine:

☒ Create a strongly-typed view

Model class:

 Scaffold template:
 ☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

Modify the Index.cshtml code as shown:

CODE TO TYPE: /Views/Home/Index.cshtml

```
@model MVCCalculator.Models.Calculator

@{
    ViewBag.Title = "IndexMVC Calculator";
    string[,] buttons = new string[,]
    {
        {"CE", "C", "<=", "/"},
        {"1", "2", "3", "*"},
        {"4", "5", "6", "-"},
        {"7", "8", "9", "+"},
        {"+/-", "0", ".", "="}
    };
}

<h2>Index@ViewBag.Title</h2>


@using (Html.BeginForm())
{
    @Html.HiddenFor(m => m.FirstOperation)
    @Html.HiddenFor(m => m.PendingValue)
    @Html.HiddenFor(m => m.PendingAction)

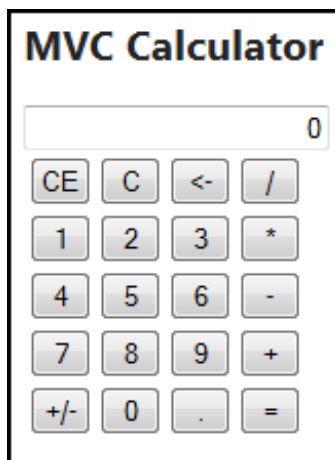
    <div class="editor-field">
        @Html.TextBoxFor(m => m.CurrentValue, new { style = "text-align: right;" })
    </div>

    <table>
        @for (int row = 0; row < buttons.GetLength(0); row++)
        {
            <tr>
                @for (int col = 0; col < buttons.GetLength(1); col++)
                {
                    <td><input type="submit" name="key" id="key" value="@buttons[row, col]"
style="width:40px" /></td>
                }
            </tr>
        }
    </table>
}
```

There are no surprises in the code, but it's simpler than our previous versions, and we can use an array for buttons.



and . You see the calculator website:



Now that we have a running MVC program, let's move on to testing.

Software Testing

Automating your testing is good! Let's review the process of software testing and its benefits.

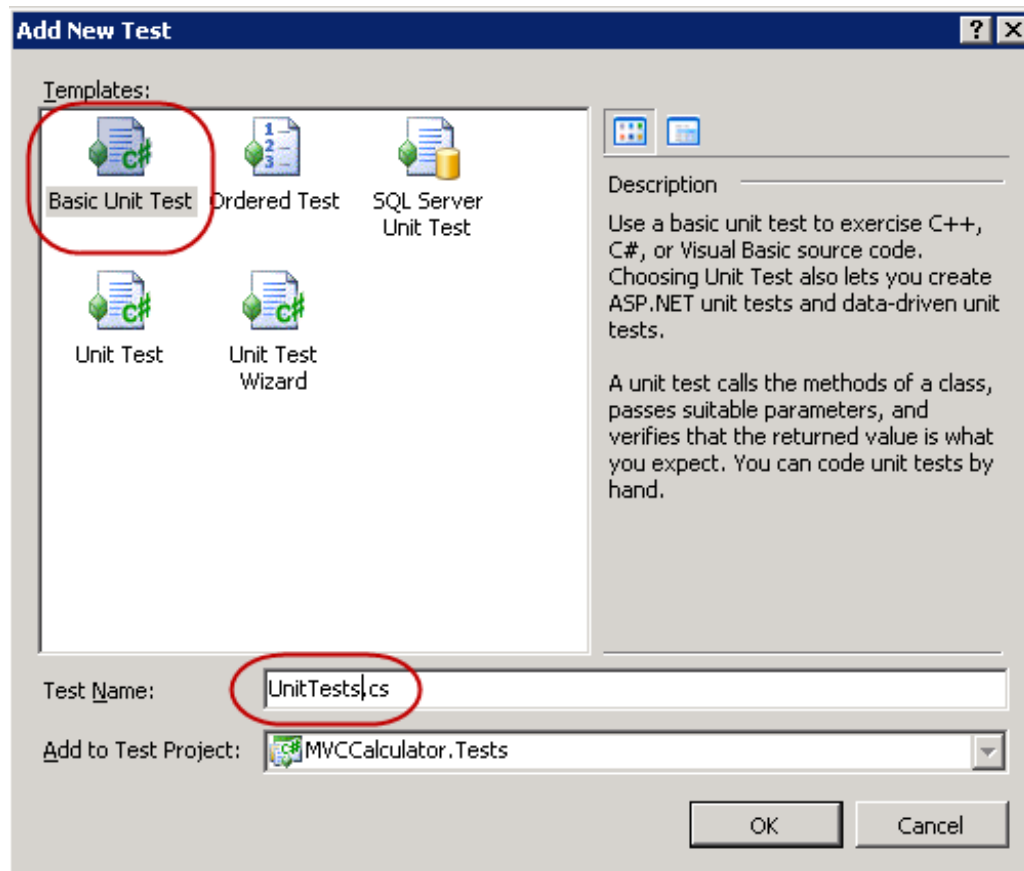
Software testing is the process of validating and verifying that a software product meets requirements, satisfies the needs of the stakeholders, and operates as expected. Thorough testing helps determine the quality of a software product.

In the process of software testing we develop a series of tests that can be run against the software to confirm that the software works as it should, and if not, to determine where the software fails. The extent to which a software application can be validated and verified using automated test tools depends on the quality of the automated tests.

We'll start by adding a new test to our **MVCCalculator.Tests** project to demonstrate the process of creating a test, and conducting a test run.

Unit Testing

Right-click the **MVCCalculator.Tests** project, and select **Add | New Test**. In the Add New Test dialog box, select **Basic Unit Test**, change the Test Name to **UnitTests.cs**, and click **OK**.



Modify **/MVCCalculator.Tests/UnitTests.cs** as shown:

CODE TO TYPE: /MVCCalculator.Tests/UnitTests.cs

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

namespace MVCCalculator.Tests
{
    [TestClass]
    public class UnitTests
    {
        // Description: Enter number "123"
        [TestMethod]
        public void TestMethod1EnterNumber()
        {
            // Arrange
            Calculator calculator = new Calculator();
            calculator.CurrentValue = 0.0D;
            calculator.FirstOperation = true;

            // Act
            // Update display
            calculator.Process("1");
            // Assert
            Assert.AreEqual(calculator.CurrentValue.ToString(), "1");

            // Act
            calculator.Process("2");

            // Assert
            Assert.AreEqual(calculator.CurrentValue.ToString(), "12");

            // Act
            calculator.Process("3");

            // Assert
            Assert.AreEqual(calculator.CurrentValue.ToString(), "123");
        }
    }
}
```

Let's discuss this code.

UnitTests.cs

```
[TestClass]
public class UnitTests
{
    // Description: Enter number "123"
    [TestMethod]
    public void EnterNumber()
    {
        // Arrange
        Calculator calculator = new Calculator();
        calculator.CurrentValue = 0.0D;
        calculator.FirstOperation = true;

        // Act
        calculator.Process("1");
        // Assert
        Assert.AreEqual(calculator.CurrentValue.ToString(), "1");

        .
        .
        .
    }
}
```

First, we renamed the test method **EnterNumber** to reflect the purpose of the test. For this first test, we want to verify that if the user clicks on number buttons, they'll see the number displayed. So, is that really what we're doing?

No, it isn't. We're actually creating an instance of our **Calculator** class, setting the initial state (which isn't really necessary since the Calculator constructor sets the initial state, but we're doing it anyway to illustrate the possibility), calling the **Process** method to simulate the user clicking on the "1" button, and then calling the **AreEqual** method of the **Assert** object to test the calculator **CurrentValue** for the correct value. The **Assert** class encapsulates the concept of an assertion, which verifies an assumption of truth about compared conditions. If the assertion fails, the Assert class will throw an **AssertFailedException** exception. The integrated testing component of Visual Studio will detect this assert failure.

We can employ various techniques to execute our tests, but for now, make sure you have the UnitTests.cs open, and then select **Test | Run | Tests in Current Context**. The Test Results tabbed pane will appear in Visual Studio, and the results of the test run will be displayed:

Test run completed Results: 1/1 passed; Item(s) checked: 0				
	Result	Test Name	Project	Error Message
<input checked="" type="checkbox"/>	Passed	EnterNumber	MVCCalculator.Tests	

Now let's change our code to force an assertion failure. Modify the UnitTests.cs code as shown in the code snippet below, then re-run the test:

UnitTests.cs

```
.
.
.
Assert.AreEqual(calculator.CurrentValue.ToString(), "129");
.
.
.
```

You see a failed test:

Test run failed Results: 0/1 passed; Item(s) checked: 1				
	Result	Test Name	Project	Error Message
<input checked="" type="checkbox"/>	Failed	EnterNumber	MVCCalculator.Tests	Assert.AreEqual failed. Expected:<12>. Actual:<19>.

The failed test result line indicates which assertion failed, why it failed ("12" was expected, but "19" was found), and the name of the test that failed. You can also double-click on the failed test to go directly to it in the code.

Reset your assert statement and re-run the test to be sure your code passes the test.

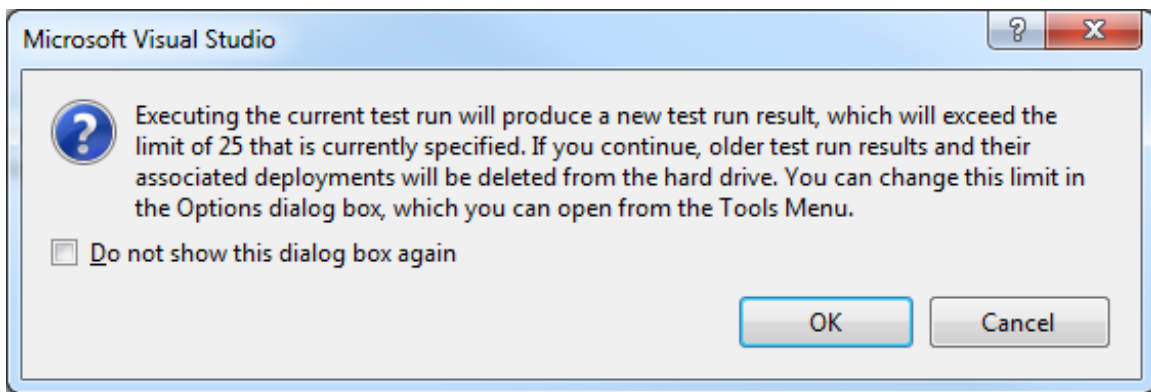
Adding a test is pretty straightforward, but there are lots of possible scenarios to test for and various patterns to use. We used the Arrange, Act, Assert pattern to test our code:

- Arrange: set up the conditions necessary for the test.
- Act: conduct the test.
- Assert: evaluate the results using an assertion statement.

That wasn't so bad, was it? We'll execute our next test on our MVC code, but before we do, let's discuss unit testing, and compare it to integrated testing.

Note

When running tests, you may see a "Test Run Result limit exceeded" dialog box similar to the one below. This dialog box indicates that Studio Test has reached your current threshold or limit (by default, 25 results) for saving and recording results. Conducting another test run will cause Studio Test to remove one (or more) of the currently saved results; that's okay. If you were testing a large project, you'd want to increase the threshold in **Tools | Options | Test Tools**. Whenever you see this dialog box during the course, just click **OK**.



Unit and Integrated Testing

The test we created above is known as a *unit test*. A *unit* is the smallest testable part of a software application. Where possible, we should create tests for these smallest components. However, our application is built upon these smaller units, and the larger components can also be considered units. As programs become more complex, you will need to test blocks of units. This is known as *integrated testing*. We won't specifically cover integrated testing in this course, so for now, you can just think of it as testing larger units.

MVC Unit Testing

In our previous unit testing, we were essentially testing our model. Now, let's test our controller, which will also include testing our view, and even more model testing. Let's add a test for our controller.

Right-click on the **MVCCalculator.Tests** project, and select **Add | New Test**. In the Add New Test dialog box, select **Basic Unit Test**, change the Test Name to **HomeControllerUnitTests.cs**, and click **OK**.

Modify **/MVCCalculator.Tests/HomeControllerUnitTests.cs** as shown:

CODE TO TYPE: /MVCCalculator.Tests/HomeControllerUnitTests.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Web.Mvc;
using MVCCalculator.Models;
using MVCCalculator.Controllers;

namespace MVCCalculator.Tests
{
    [TestClass]
    public class HomeControllerUnitTests
    {
        [TestMethod]
        public void TestMethodTestIndexView()
        {
            // Arrange
            var controller = new HomeController();

            // Act
            var result = controller.Index() as ViewResult;

            // Assert
            Assert.AreEqual("Index", result.ViewName);
        }
    }
}
```

Let's discuss this code.

/MVCCalculator.Tests/HomeControllerUnitTests.cs

```
.
.
.
// Arrange
var controller = new HomeController();

// Act
var result = controller.Index() as ViewResult;

// Assert
Assert.AreEqual("Index", result.ViewName);
.
.
.
```

This test code creates an instance of the **HomeController**, then uses the instance variable **controller** to call the **Index** method and return a **ViewResult** object. Then we test the assertion that **ViewName** is **Index**. Let's run the test and see what happens.

Make sure you have the HomeControllerUnitTests.cs open. Select **Test | Run | Tests in Current Context**. The TestResults tabbed pane appears in Studio, showing you the results of the test run.

Test run failed Results: 0/1 passed; Item(s) checked: 1			
	Result	Test Name	Error Message
<input checked="" type="checkbox"/>	Failed	TestIndexView	Assert.AreEqual failed. Expected:<Index>. Actual:<>.


The test failed! Why? In our HomeController.cs file, when returning a ViewResult, we use an overloaded version that doesn't specify a name for the view. MVC attempts to infer the name of the view, but is unsuccessful in the Index method. Let's add a name for our view, then re-run the test. Modify **/MVCCalculator.Tests/HomeController.cs** as

shown:

```
CODE TO TYPE: HomeController.cs

.
.
.
public ActionResult Index()
{
    return View("Index", new Calculator());
}
.
.
.
```

We've added a new first parameter to the View constructor, which specifies the view name. Rerun the test; it passes.

Test run completed Results: 1/1 passed; Item(s) checked: 0				
	Result	Test Name	Project	Error Message
<input type="checkbox"/>	 Passed	TestIndexView	MVCCalculator.Tests	

Does your Test Results tabbed pane show both of the tests we've added, rather than just one test? We can use keyboard shortcuts to run the tests; these allow you to select which tests to run, for example, you may want to run just the tests that failed the last run. You can right-click on the test source file and select **Run Tests**, or you can use one of these shortcuts:

Tip

- **Ctrl + R**, then press **A**: Runs all the tests in all projects.
- **Ctrl + R**, then press **D**: Runs all tests that were run in the last test run.
- **Ctrl + R**, then press **F**: Runs all tests in the last test run that did not pass.

So, what if we want to test the data returned from a method in the controller? Let's try that test next.

Add the following test method to **/MVCCalculator.Tests/HomeControllerUnitTests.cs**:

CODE TO TYPE: HomeControllerUnitTests.cs

```
.
.
.
[TestMethod]
public void TestIndexViewEnterNumber()
{
    // Arrange
    HomeController controller = new HomeController();
    Calculator setupCalculator = new Calculator
    {
        CurrentValue = 0.0D,
        FirstOperation = true,
        PendingValue = 0.0D,
        PendingAction = ""
    };

    // Act
    var result = controller.Index(setupCalculator) as ViewResult;
    var resultCalculator = (Calculator)result.ViewData.Model;

    // Assert
    Assert.AreEqual(1.0D, resultCalculator.CurrentValue);
}
.
.
.
```

Let's discuss this test code before we run it.

HomeControllerUnitTests.cs

```
[TestMethod]
public void TestIndexViewEnterNumber()
{
    // Arrange
    HomeController controller = new HomeController();
    Calculator setupCalculator = new Calculator
    {
        CurrentValue = 0.0D,
        FirstOperation = true,
        PendingValue = 0.0D,
        PendingAction = ""
    };

    // Act
    var result = controller.Index(setupCalculator) as ViewResult;
    var resultCalculator = (Calculator)result.ViewData.Model;

    // Assert
    Assert.AreEqual(1.0D, resultCalculator.CurrentValue);
}
```

We set up a controller, but this time we'll call the overloaded controller Index method that requires a **Calculator** object, so we'll need to create one. Once we've returned the ViewResult **result**, we'll use **result** to retrieve the Model from the ViewData, casting it as our **Calculator** to create **resultCalculator**. Using **resultCalculator**, we can test the assertion that the **CurrentValue** is 1.0D.

Run the test. Did it work? No? Let's run it again, but this time, let's actually debug into our test code to see if we can find out what's going on.

Add a Breakpoint to the first line of code in the **Act** section of our code, the line that starts with **var result = ...**, and then select **Test | Debug | Tests in Current Context**. When the execution of the test stops at the breakpoint, step into the code. Eventually, you'll see an error message on the line of code in the HomeController.cs file that calls the calculator Process method.

Note

You may need to review how to set breakpoints and how to step into code using Studio debugging techniques found on the **Debug** menu.

When testing controller methods that are decorated with **HttpPost** (or **HttpGet**), your controller method expects to have been called from a web page; underlying objects will have been created and populated with information from the web posting process. There are various techniques available to help you test for such scenarios, we'll begin by using one of the basic techniques.

We'll take advantage of features built into MVC to test the overloaded controller **Index** method. If we provide a class with properties that match the web form fields when a controller method is called in response to a view, the ASP.NET MVC engine will populate this class automatically, setting the values of the matching properties to the form field values. We will use this technique, which also makes our controller code easier to read. then add a new class named **CalculatorForm** to this folder. Add a single property called **key** as shown:

CODE TO TYPE: /FormModels/CalculatorForm.cs

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

namespace MVCCalculator
{
    public class CalculatorForm
    {
        public string key { get; set; }
    }
}
```

Modify **/Controllers/HomeController.cs** to use the new **CalculatorForm** class:

CODE TO TYPE: /Controllers/HomeController.cs

```
.
.
.
using MVCCalculator.FormModels;
.
.
.
[HttpPost]
public ActionResult Index(Calculator calculator, CalculatorForm calculatorForm)
{
    calculator.Process(Request.Form["key"]calculatorForm.key);
    ModelState.Clear();
    return View(calculator);
}
.
.
.
```

We've added a second parameter to our **Index** method, then used the **key** property to retrieve the id of the button clicked by the user. Run your **MVCCalculator** application to make sure it still works as you'd expect.

We changed the **Index()** method (in the previous LISTING) by requiring an additional **CalculatorForm** parameter, so we need to change the test program (which calls the **Index()** method) to pass an additional **CalculatorForm** parameter.

Update your test code to use this new method. Modify **/MVCCalculator.Tests/HomeControllerUnitTests.cs** as shown:

```

.
.
.
using MVCCalculator.FormModels;
.
.
.
[TestMethod]
public void TestIndexViewEnterNumber()
{
    // Arrange
    HomeController controller = new HomeController();
    Calculator setupCalculator = new Calculator
    {
        CurrentValue = 0.0D,
        FirstOperation = true,
        PendingValue = 0.0D,
        PendingAction = ""
    };

    // Act
    var result = controller.Index(setupCalculator, new CalculatorForm { key = "1" }) as
    ViewResult;
    var resultCalculator = (Calculator)result.ViewData.Model;

    // Assert
    Assert.AreEqual(1.0D, resultCalculator.CurrentValue);
}
.
.
.

```

We modified the code to create and populate an instance of the CalculatorForm. Now when you run the test, it will pass without generating an exception.

Now that you've worked through the test examples in this lesson, you can create your own test methods! With practice and some online research, you can create an extensive test suite for all of your future applications.

Additional Testing Templates

We'll only be using some of the existing testing templates during the course, but there are more available:

Ordered Test: allows you to execute existing tests in a specific order, which is useful when testing application scenarios or use cases.

Unit Test: provides for additional test methods for initialization and cleanup.

Unit Test Wizard: will analyze your project and create unit tests classes and stubs based on the contents of your project. Try this template and check out the resulting testing classes. You can always delete any of the classes (or all of them).

Final Thoughts

Adding software testing is a valuable resource to ensure that your code performs as you expect. There are lots of articles online that discuss testing strategies, breakdown what you should test, and how you should test. Most large development companies employ automated testing, but even as a sole developer, your code and development process will benefit from the development of test cases.

As always, dig in and practice this stuff in your homework. See you when you're done.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Client Solutions: JavaScript, jQuery, Ajax, XML, and JSON

Lesson Objectives

In this lesson you will:

- read about client-side and server-side development.
- use client-side solutions.
- use JavaScript, jQuery, and Ajax, while employing XML and JSON data formats.

Client-Side and Server-Side Development

As you've worked through the lessons in this course so far, we've discussed the concept of client-side and server-side; now I want to emphasize the difference.

We've primarily written web applications (with or without MVC) in C#, HTML, Aspx, Razor, and SQL. For our purposes, C# and SQL are server-side technologies, which means they are loaded and executed on a computer other than the computer or device the user is using. Aspx and Razor are also server-side technologies, but they're used specifically to create content that enables a web user interface. For the purposes of this lesson, we'll focus on technologies designed to execute on the client-side device, or as in the case of XML and JSON, used to allow the exchange and interpretation of meaningful data with and on the client-side device.

Before proceeding, be warned: this lesson will not teach JavaScript, jQuery, Ajax, XML or JSON. Each of those technologies could be a course unto itself. We'll be making limited use of them so you understand how they work with C#, ASP.NET, and MVC. (We offer courses in these tools as well. If you'd like to learn more about them, you know what to do. See our [course catalog](#).)

Note

This lesson will focus on technologies utilized on the client-side device, but the technologies are not limited to client computing. They are used in server environments as well. For example, JavaScript can also be coded and executed for server-side development. XML and JSON are data encoding schemes that can be used in almost any type of environment.

JavaScript

For our purposes, JavaScript is a client-side language that executes on the client device, such as a web browser running on a desktop computer, laptop, tablet, or mobile phone. JavaScript code can be added to web page content to respond to user interactions, manipulate web page content, and many other actions. Hundreds of thousands of web pages use JavaScript as a key feature of their functionality.

Note



Most web devices that can display web content and execute JavaScript have the ability to disable JavaScript. Current standards strongly discourage the creation of web content that requires JavaScript, but many developers ignore this recommendation. You can detect whether JavaScript support is enabled and if it isn't, you can redirect the user to an alternate, non-JavaScript-reliant version of your website, or notify the user that JavaScript is required.

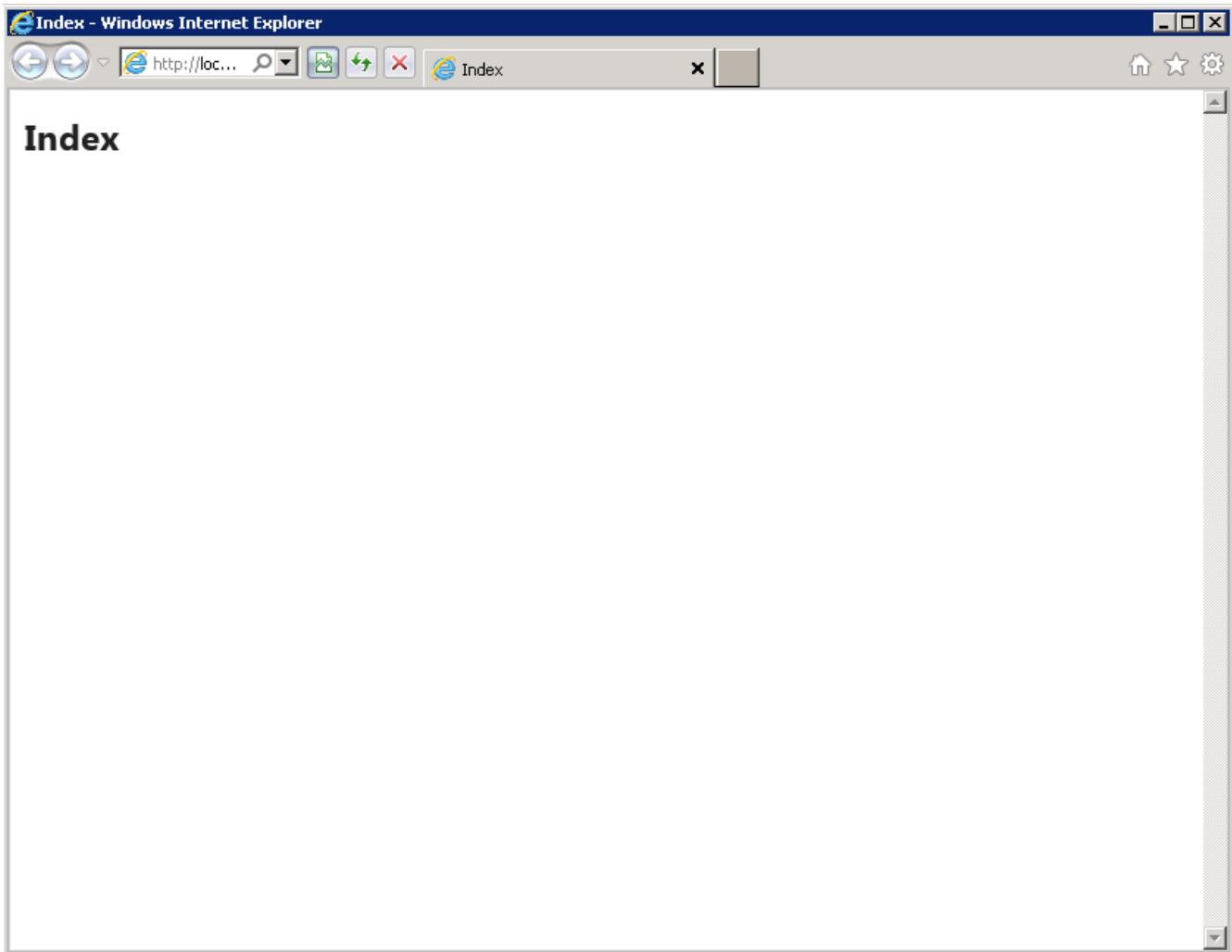
Okay, let's get started using JavaScript—we'll create a new MVC project specifically to add JavaScript to our view.

Select **File | New | Project**. Then select **Visual C# | ASP.NET MVC 4 Web Application**, change the project Name to **MVCJavaScript**, and click **OK**. On the New ASP.NET MVC Project dialog, select the **Basic** template and the **Razor** View engine, uncheck **Create a unit test project**, and click **OK**.

Once the MVCJavaScript Basic MVC solution and project is created, add an empty controller. Right-click the **Controllers** folder, select **Add | Controller**, change the controller name to **HomeController**, set the Scaffolding options Template to **Empty MVC controller**, and click **Add**.

Once the HomeController.cs file is displayed in Studio, right-click in the Index method, select **Add View**, leave the View Name **Index** and the View engine **Razor (CSHTML)**, uncheck **Create a strongly-typed view**, uncheck **Create as a partial view**, leave checked **Use a layout or master page**, and click **Add**.

 and  to ensure you can see the basic Index view page:





Next, let's add code to display the current date and time, using JavaScript code and Razor code. Modify the Index.cshtml file as shown:

CODE TO TYPE: Index.cshtml

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
Razor current date/time: @DateTime.Now<br />
JavaScript current date/time: <script type="text/javascript">window.onload = document.w
rite(new Date());</script><br />
</p>
```

 and . You see the index page with the date/time output from Razor and JavaScript:

Index

Razor current date/time: 6/19/2013 4:46:28 PM

JavaScript current date/time: Wed Jun 19 2013 16:46:29 GMT-0700 (Pacific Daylight Time)

JavaScript can be added into any Razor view page. Keep in mind that there's a difference between the Razor date/time call and the JavaScript date/time call: the Razor code was interpreted and resolved (the date/time determined) on the server, and the date/time result was sent from the server to the client computer as literal text and displayed by the browser; the JavaScript date/time was determined when the JavaScript code was loaded and executed within the web browser. This is an important distinction to understand. I'll describe an example to help illustrate. Suppose we want to create a current date/time display on our web page. Using Razor, we'd need to refresh the page constantly to execute the server-side Razor code to get the current date/time. Also, the date/time we would see is the date/time on the server. If we are in a different time zone than that of the server, the date/time would be incorrect. Using JavaScript, we could have a JavaScript timer constantly refresh a call to a function that gets the current date/time, and display the updated date/time without having to reload content from the server.

When using JavaScript (or any other client-side technologies or languages) in your Razor code, the Razor view engine will interpret the content either explicitly, such as when you use the `@` symbol or script tag, or implicitly by inferring the language based on context. Naturally, we can mix Razor and JavaScript, but sometimes you'll need to help the Razor view engine in determining the correct language. Let's work through an example where the code will create a Razor (C#) array that holds the days of the week, and then use JavaScript and Razor to loop through the Razor array, pushing each day of the week into a JavaScript array. Ultimately, the code will use JavaScript to output the days of the week from the JavaScript array.

Modify your **Index.cshtml** file as shown:

CODE TO TYPE: Index.cshtml

```
@{
    ViewBag.Title = "Index";
    string[] daysOfWeek = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };
}

<h2>Index</h2>

<p>
Razor current date/time: @DateTime.Now<br />
JavaScript current date/time: <script type="text/javascript">window.onload = document.write(new Date());</script><br />
</p>

<p>
<script type="text/javascript">
    var daysOfWeek = [];
    @foreach (var weekDay in daysOfWeek)
    {
        <text>
        daysOfWeek.push('@weekDay');
        </text>
    }
    document.write("Days of the week pushed into JavaScript array from Razor<br />\n");
    for (var i = 0; i < daysOfWeek.length; i++)
        document.write(daysOfWeek[i] + "<br/>\n");
</script>
</p>
```

 and . You see this:

Index

Razor current date/time: 6/19/2013 5:23:52 PM

JavaScript current date/time: Wed Jun 19 2013 17:23:52 GMT-0700 (Pacific Daylight Time)

Days of the week pushed into JavaScript array from Razor

Sunday

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

Now remove the `<text>` and `</text>` tags, and take a look at the error highlighted in red:

```
@foreach (var weekDay in daysOfWeek)
{
    daysOfWeek.push('@weekDay');
}
document.write("Days of the week pushed into JavaScript array from Razor: ");
for (var i = 0; i < daysOfWeek.length; i++)
```

Too many characters in character literal

The `<text>` and `</text>` tags tell Razor that the code inside those tags is *not* Razor C# code. Then, within the `<text>` block, we use `@weekDay` to indicate that `weekDay` is a Razor variable.

Tip

We recommend you view the HTML source of your web page that is generated by Studio. Take note of the way the Razor code was rendered in the HTML code. As a web developer, you'll examine rendered HTML frequently to make sure your source code is rendered into HTML as expected. .

Also, when you use Razor and JavaScript (or other languages) together, you may see a warning, underlined in green in the Studio Code Editor:

```
var daysOfWeek = [];
@foreach (var weekDay in daysOfWeek)
{
    Conditional compilation is turned off;
}
```

The error means exactly what it says: that you have conditional compilation turned off in Studio. According to Microsoft, "Conditional compilation enables JScript to use new language features without sacrificing compatibility with older versions that do not support the features. Some typical uses for conditional compilation include using new features in JScript, embedding debugging support into a script, and tracing code execution." We don't need this functionality for our code to work, so the view engine suggests that perhaps we meant something different. Since it's just a warning, we can ignore it and our code will still work. If it had been an error (with a red underline, rather than a green one), we still wouldn't need to turn on conditional compilation, but instead determine which Razor error created this conditional compilation error.

Tip

You can debug into your JavaScript code using Studio. Look into using a browser-based JavaScript debugging tool. Many browsers have such capability built in; many developers use a free add-in product called `TARGET=" blank">FireBug`.

If you find yourself struggling with Razor syntax, especially the inline syntax when you mix Razor and JavaScript, there's an excellent article entitled [Inline Razor Syntax Overview](#) by Mike Brind that you may find helpful.

jQuery

If you've spent much time studying JavaScript, you've likely run across references to jQuery. jQuery is a cross-browser

JavaScript library created to help streamline scripting. While most of the capabilities of jQuery could be coded in JavaScript, jQuery provides a library of code for common functions, saving you the trouble of writing code yourself. jQuery is used by so many developers that MVC included it. Unfortunately, the jQuery code in our example is in the wrong place; let's move it and then add a call to get the current date and time using jQuery.

Modify **/Views/Shared/_Layout.cshtml** as shown:

CODE TO TYPE: _Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
    @Scripts.Render("~/bundles/jquery")
</head>
<body>
    @RenderBody()

    @Scripts.Render("~/bundles/jquery")
    @RenderSection("scripts", required: false)
</body>
</html>
```

Note

jQuery is located at the bottom of the rendered view intentionally for performance purposes, but for this lesson we'll move it so we can access the jQuery object and methods more easily. Here's a great article that addresses website performance issues like script placement: [Best Practices for Speeding Up Your Web Site](#).

Next, modify the view code as shown:

CODE TO TYPE: /Views/Home/Index.cshtml

```
<p>
    Razor current date/time: @DateTime.Now<br />
    JavaScript current date/time: <script type="text/javascript">window.onload = do
cument.write(new Date());</script><br />
    jQuery current date/time: <script type="text/javascript">window.onload = docume
nt.write($.now());</script><br />
</p>
```



The jQuery date/time is displayed under the JavaScript date/time. Next up: JSON and XML data and Razor and MVC.

JSON

An effective user interface is essential to any well-developed website, but the interface would be useless without meaningful information or data. JavaScript and other programmers commonly use a standard data format named JSON. JSON is an acronym for JavaScript Object Notation, a text-based, human-readable data interchange format. JSON is used to represent data structures and associate arrays as objects. Even though it is related to JavaScript, JSON, as a data format, has become language-independent because a variety of languages implement the ability to encode and decode data into JSON format.

Tip

Another reason to use a tool that understands JavaScript (such as FireBug) is that most of these tools also understand JSON.

With MVC, the Controller object includes a method to create JSON from an object. Let's modify our view once again, so that it returns a JSON data structure.

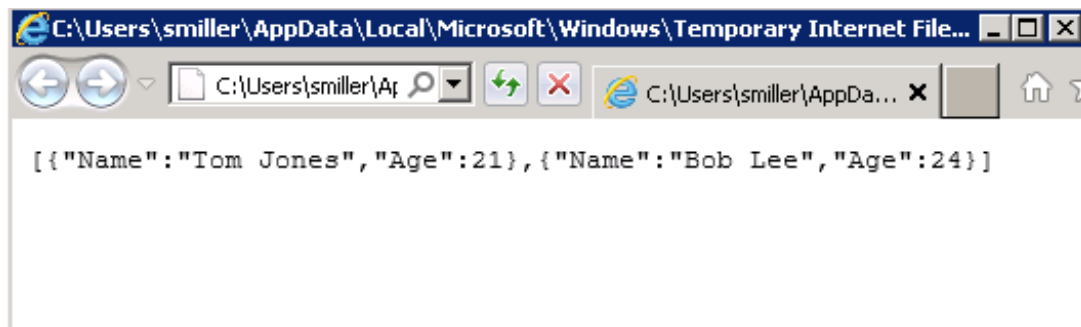
Modify **/Controllers/HomeController.cs** as shown:

CODE TO TYPE: **/Controllers/HomeController.cs**

```
.
.
.
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

[AcceptVerbs(HttpVerbs.Get)]
public ActionResult GetContent(string format = "JSON")
{
    Person[] person = new Person[] { new Person { Name = "Tom Jones", Age = 21 }, new P
erson { Name = "Bob Lee", Age = 24 } };
    if (format.ToLower().IndexOf("json") >= 0)
        return Json(person, JsonRequestBehavior.AllowGet);
    else
        return this.Content("Invalid format", "text/html");
}
.
.
.
```

Save and run the web application. Add **/Home/GetContent** to the URL. You can specify the format by adding **/Home/GetContent?format=JSON** to the URL, but the method defaults to JSON in the parameter list, so either URL works. You may be prompted to confirm that you want to open this URL (click **Open**), and to specify which application to use to open it (choose **Internet Explorer**). Once you confirm and specify, you see a window containing the JSON data:



Let's discuss this code:

/Controllers/HomeController.cs

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

[AcceptVerbs(HttpVerbs.Get)]
public ActionResult GetContent(string format = "JSON")
{
    Person[] persons = new Person[] { new Person { Name = "Tom Jones", Age = 21 }, new Pers
on { Name = "Bob Lee", Age = 24 } };
    if (format.ToLower().IndexOf("json") >= 0)
        return Json(persons, JsonRequestBehavior.AllowGet);
    else
        return this.Content("Invalid format", "text/html");
}
```

We create a **Person** class, and then an array of new **Person** instances. Our method has a string parameter **format** that defaults to "JSON." We'll use that to allow the user to specify a format (we'll use this feature when we discuss XML in a bit). Next, we call the **Json** method to convert our **persons** array into a JSON object. We add an option to the **Json** call **JsonRequestBehavior.AllowGet** to allow us to make a GET call to this method with JSON involved. (You can read more about this issue at [JSON Problem - JsonRequestBehavior to AllowGet](#).) When you save and run the code, you see this output in your browser:

```
[{"Name":"Tom Jones","Age":21},{"Name":"Bob Lee","Age":24}]
```

Now that you have some understanding of JSON data and MVC, let's move on to XML.

XML

XML, or eXtensible Markup Language, has evolved over time to work in many different environments, and offers an ever increasing menu of standards-based functionality and capability. XML looks a lot like HTML; it uses tags and tag attributes to wrap and describe data. Here's what the Person data we used in our JSON example looks like in XML:

OBSERVE:

```
<ArrayOfPerson>
  <Person>
    <Name>Tom Jones</Name>
    <Age>21</Age>
  </Person>
  <Person>
    <Name>Bob Lee</Name>
    <Age>24</Age>
  </Person>
</ArrayOfPerson>
```

This XML is "well-formed" in that it adheres to basic XML-formatting standards. Typically the document would also include a header line describing the file type and version with an optional encoding description, and an optional schema statement that references a file that describes how to interpret the XML data elements. Online resources have extensive coverage of XML, but for our purposes, the basic well-formed XML above is sufficient.

Let's modify our controller code to support returning our data in XML format. Modify HomeController.cs as shown (include the additional namespace):

CODE TO TYPE: HomeController.cs

```
.
.
.
using System.Xml;
using System.Xml.Serialization;
using System.IO;
using System.Text;
.
.
.
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult GetContent(string format = "JSON")
{
    Person[] persons = new Person[] { new Person { Name = "Tom Jones", Age = 21 }, new
Person { Name = "Bob Lee", Age = 24} };
    if (format.ToLower().IndexOf("json") >= 0)
        return Json(persons, JsonRequestBehavior.AllowGet);
    else if (format.ToLower().IndexOf("xml") >= 0)
    {
        string xml = ObjectToXML(persons);
        return this.Content(xml, "text/xml");
    }
    else
        return this.Content("Invalid format", "text/html");
}

private string ObjectToXML(object obj)
{
    StringBuilder strXML = new StringBuilder();
    try
    {
        Type objectType = obj.GetType();
        XmlSerializer xmlSerializer = new XmlSerializer(objectType);
        MemoryStream memoryStream = new MemoryStream();
        try
        {
            using (XmlTextWriter xmlTextWriter = new XmlTextWriter(memoryStream, Encodi
ng.UTF8) { Formatting = Formatting.Indented })
            {
                xmlSerializer.Serialize(xmlTextWriter, obj);
                memoryStream = (MemoryStream)xmlTextWriter.BaseStream;
                strXML.Append(new UTF8Encoding().GetString(memoryStream.ToArray()));
            }
        }
        finally
        {
            memoryStream.Dispose();
        }
    }
    catch (Exception ex)
    {
        strXML.Clear();
        strXML.Append("<Error>" + ex.Message + "</Error>\n");
    }
    return strXML.ToString();
}
.
.
.
```

Save and run the web application, this time using **http://localhost:xxxx/Home/GetContent?format=XML** for the URL.

Let's discuss the code:

HomeController.cs

```
.  
.   
.   
[AcceptVerbs(HttpVerbs.Get)]  
public ActionResult GetContent(string format = "JSON")  
{  
    Person[] persons = new Person[] { new Person { Name = "Tom Jones", Age = 21 }, new  
    Person { Name = "Bob Lee", Age = 24} };  
    if (format.ToLower().IndexOf("json") >= 0)  
        return Json(persons, JsonRequestBehavior.AllowGet);  
    else if (format.ToLower().IndexOf("xml") >= 0)  
    {  
        string xml = ObjectToXML(persons);  
        return this.Content(xml, "text/xml");  
    }  
    else  
        return this.Content("Invalid format", "text/html");  
}  
.   
.   
. 
```

ASP.NET does not have a simple call to convert an object to XML like it does with JSON, so we had to use the XML support within ASP.NET to serialize our object. If you search online, you'll find several ways to convert an object to XML, but the code we used here is fairly versatile. We return XML by converting our persons object array to XML by calling our new method **ObjectToXML**, then we set the result content to the string containing the XML, and then we specify the MIME type of the result content so the browser can interpret it. We could have used the same method with the JSON data, specifying the MIME type as "text/json," but most browsers would try to download the data. Most browsers can display the XML data, as shown below for Firefox.

```
- <ArrayOfPerson>  
  - <Person>  
    <Name>Tom Jones</Name>  
    <Age>21</Age>  
  </Person>  
  - <Person>  
    <Name>Bob Lee</Name>  
    <Age>24</Age>  
  </Person>  
</ArrayOfPerson>
```

Ajax

Now that we've discussed JavaScript, jQuery, JSON, and XML, let's turn our attention to Ajax. Ajax is a group of interrelated web development techniques used to create asynchronous communication from within a web application to the web server. Using Ajax, full page reloads are no longer required to retrieve information or populate or modify web page content, and individual elements on a web page can be refreshed from the server without impacting other elements. Ajax is implemented for deployment on the client side, and may be written in a variety of client-side languages, including JavaScript, but is commonly used in the jQuery form, so we'll use that form for this part of the lesson. Let's make a few more modifications to our project to add Ajax functionality. This time, we'll just update the Index.cshtml view. Add the code below to the bottom of **Views/Home/Index.cshtml**:

```

.
.
.
<p>
<script type="text/javascript">
    // Handle getJSON button click
    $.ready(function () {
        $("#getJSON").bind("click", function () {
            $("#divJSONResults").empty();
            $.ajax({
                type: "GET",
                contentType: "application/json; charset=utf-8",
                dataType: "json",
                url: '/Home/GetContent',
                data: { "format": "JSON" },
                success: function (data) {
                    if (data == null)
                        $("#divJSONResults").append("No data returned.");
                    else {
                        for (var i = 0; i < data.length; i++) {
                            $("#divJSONResults").append("Name/Age: " + data[i].Name + "
, " + data[i].Age + "<br />\n");
                        }
                    }
                }
            });
        });
    });
    // Handle getXML button click
    $.ready(function () {
        $("#getXML").bind("click", function () {
            $.ajax({
                type: "GET",
                contentType: "application/json; charset=utf-8",
                dataType: "text",
                url: '/Home/GetContent',
                data: { "format": "XML" },
                success: function (data) {
                    if (data == null)
                        document.getElementById('textAreaXMLResults').value = "No data
returned";
                    else {
                        document.getElementById('textAreaXMLResults').value = data;
                    }
                }
            });
        });
    });
</script>
</p>
<input type="submit" id="getXML" name="getXML" value="GET XML" /><br />
XML Results<br />
<textarea rows="15" cols="70" id="textAreaXMLResults" name="textAreaXMLResults"></textare
a><br />
<input type="submit" id="getJSON" name="getJSON" value="Get JSON" /><br />
JSON Results:<br />
<div id="divJSONResults" /><br />
.
.
.

```

When you save and run this code, and click on either button, you'll see results. We "tricked" the Ajax call for XML by specifying the data type is "text" so we could display the XML in a textarea HTML control. For JSON, we parsed the data and formatted the output.

In this section we've taken a pretty straightforward approach to using Ajax with MVC, but there is an MVC-standard way

of connecting data. MVC supports the concept of a partial view, a view that you would embed within another view. With Ajax, each partial view should be tied to a model. Although you don't need Ajax to enable and use partial views, when using Ajax for asynchronous communication you should use a partial view model and partial view for the display of any Ajax-generated results. Let's finish this lesson by developing a partial view.

Partial Views and Ajax

To demonstrate partial views and Ajax, let's update our project to display a list of students with paging capability. We could use JavaScript to receive JSON data and then create a student list, but let's take advantage of MVC. We'll need two new views (one will be a partial view) and associated controller methods, a couple of new models, and we'll use Ajax to switch the student listing depending on which page we're accessing.

Right-click the **/Models** folder, select **Add | Class**, name the class **Student**, and click **Add**. Modify **/Models/Student.cs** as shown.

CODE TO TYPE: /Models/Student.cs

```
.
.
.
namespace MVCJavaScript.Models
{
    public class Student
    {
        public string LastName { get; set; }
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public int Age { get; set; }
    }
}
```

Next, let's add a model for our list of students for the partial view that includes paging properties.

Right-click the **/Models** folder, select **Add | Class**, name the class **Students**, and click **Add**. Modify **/Models/Students.cs** as shown:

CODE TO TYPE: /Models/Students.cs

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

namespace MVCJavaScript.Models
{
    public class Students<Student>
    {
        public IEnumerable<Student> Data { get; set; }
        public int NumberOfPages { get; set; }
        public int CurrentPage { get; set; }
    }
}
```

The Students class model will serve as a container for a group of students, as well as to track paging information.

Next, we need to make a new view that will hold our partial view and Ajax code.

Right-click the **/Views/Home** folder, select **Add | View**, set the view name to **Student**, verify that only **Use a layout or master page** is checked, and click **Add**.

Modify **/Views/Home/Student.cshtml** as shown:

CODE TO TYPE: /Views/Home/Student.cshtml

```
@{
    ViewBag.Title = "Student List";
}

<script type="text/javascript">
function linkClick(pageNumber) {
    $.ajax({
        url: '@Url.Action("PartialStudent")',
        data: { "pageNumber": pageNumber },
        success: function (data) {
            $("#divPartialView").html(data);
        }
    });
}
</script>

<h2>@ViewBag.Title</h2>

<div id="divPartialView">
    @Html.Partial("PartialStudent")
</div>
```

We use a <div tag> to encapsulate our partial view. We also add a JavaScript function that we can call whenever a paging link is clicked, making a jQuery Ajax call requesting the correct page of data. Now we just need to create our partial view and update the controller.

Tip

When creating a strongly-typed view, you may need to clean your project (**Build | Clean Solution**), and then rebuild it so the Add View dialog is aware of any new models.

Right-click the **/Views/Home** folder, select **Add | View**, set the view name to **PartialStudent**, check and set the strongly-typed view dropdown to **Student**, check **Create as a Partial View**, and click **Add**. Modify **/Views/Home/PartialStudent.cshtml** as shown:

CODE TO TYPE: /Views/Home/PartialStudent.cshtml

```
@model MVCJavaScript.Models.Students<MVCJavaScript.Models.Student>

<table>
    <thead>
        <tr>
            <th>Last</th>
            <th>First</th>
            <th>Middle</th>
            <th>Age</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var person in Model.Data)
        {
            <tr>
                <td>@person.LastName</td>
                <td>@person.FirstName</td>
                <td>@person.MiddleName</td>
                <td>@person.Age</td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="4">
                @for (int pgNum = 1; pgNum <= Model.NumberOfPages; pgNum++)
                {
                    if (pgNum == Model.CurrentPage)
                    {
                        @pgNum
                    }
                    else
                    {
                        <a onclick="linkClick(@pgNum)" href="javascript:void(0)">@pgNum</a>
                    }
                }
            </td>
        </tr>
    </tfoot>
</table>
```

We modify the model used for the view to use the Students model rather than Student, and turn the model into a typed list for enumerations, which matches the Students model. Then we loop through each of the students, list the student information, and then display the page information. Now we'll update our controller.

Modify **/Controllers/HomeController.cs** as shown.

CODE TO TYPE: /Controllers/HomeController.cs

```
.
.
.

private Student[] studentData = new Student[]
{
    new Student { FirstName = "Tom", LastName = "Jones", Age = 21 },
    new Student { FirstName = "Bob", LastName = "Lee", Age = 24 },
    new Student { FirstName = "A", LastName = "Lee", Age = 25 },
    new Student { FirstName = "B", LastName = "Lee", Age = 26 },
    new Student { FirstName = "C", LastName = "Lee", Age = 27 },
    new Student { FirstName = "D", LastName = "Lee", Age = 28 },
    new Student { FirstName = "E", LastName = "Lee", Age = 29 },
    new Student { FirstName = "F", LastName = "Lee", Age = 30 },
    new Student { FirstName = "G", LastName = "Lee", Age = 31 },
    new Student { FirstName = "H", LastName = "Lee", Age = 32 },
    new Student { FirstName = "I", LastName = "Lee", Age = 33 },
    new Student { FirstName = "J", LastName = "Lee", Age = 34 },
    new Student { FirstName = "K", LastName = "Lee", Age = 35 },
    new Student { FirstName = "L", LastName = "Lee", Age = 36 }
};

private const int PageSize = 5;

public ActionResult Student()
{
    return View(GetStudents(1));
}

public ActionResult PartialStudent(int pageNumber = 1)
{
    return PartialView(GetStudents(pageNumber));
}

private Students<Student> GetStudents(int pageNumber)
{
    var students = new Students<Student>();
    students.Data = studentData.OrderBy(p => p.LastName).Skip(PageSize * (pageNumber -
1)).Take(PageSize).ToList();
    students.NumberOfPages = Convert.ToInt32(Math.Ceiling((double)studentData.Count() /
    PageSize));
    students.CurrentPage = pageNumber;
    return students;
}

.
.
.
```

In the controller, we add a data object seeded with test data for our demonstration, two methods that correspond to our views, and a method to extract the correct data from our data object. For a real application, we'd use a real database.



and



and add **/Home/Student** to the URL. You see output like this:

Student List

Last	First	Middle	Age
Jones	Tom		21
Lee	Bob		24
Lee	A		25
Lee	B		26
Lee	C		27
1 2 3			

Before you move on to the homework, here's a brief introduction to "view models" (they appear in the homework for this lesson). If this lesson's project was using a database, the Students model would not be represented in that database, because we don't store paging information there.

Frequently, when working with MVC, we'll add an additional model layer between the database model and the views, often referred to as view models. View models are usually kept in a separate folder, often called *ViewModels*, to differentiate a model intended for user interface from a model that represents database schema information.

Alright, off you go!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

ASP.NET and Databases

Lesson Objectives

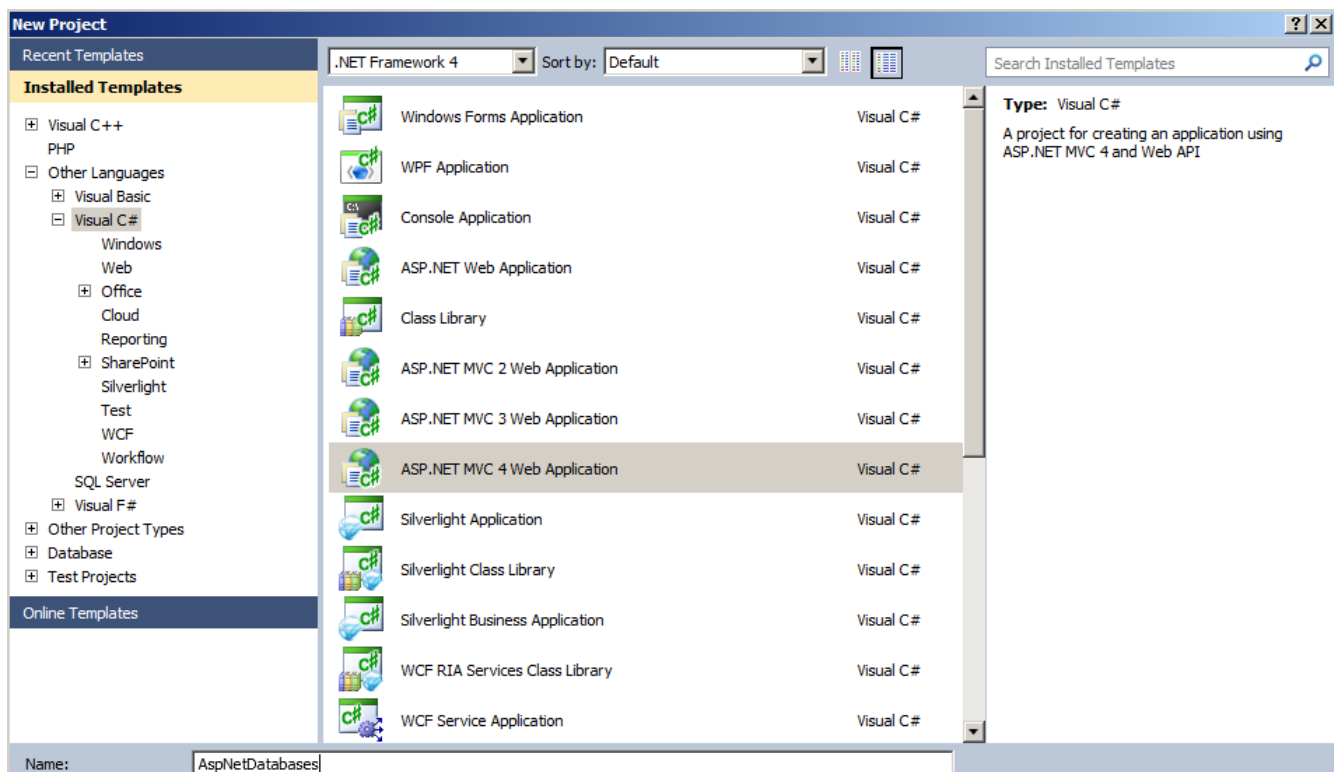
In the lesson, you will:

- add a database to an ASP.NET MVC Web Application.
- write code the system can use to automatically build a database.
- create an action you can use to add to a menu.
- use LINQ and data binding.

There are two main processes for working with databases in C#. The older method is to build the database and then code to that database. The newer method is to code first and have the system build the database from that code. We use the newer method in our lessons that cover databases.

Adding a Database

As we've seen before, adding a database to an ASP.NET project is a fairly straightforward process. Create a new **ASP.NET MVC 4 Web Application** project named **AspNetDatabases**.





Be sure to choose **Internet Application** and make sure the View Engine is set to **Razor**:


New ASP.NET MVC 4 Project


Project Template


Select a template:


 Empty

 Basic

 Internet Application

 Intranet Application

 Mobile Application

 Web API

Description:
A default ASP.NET MVC 4 project with an account controller that uses forms authentication.

View engine:
Razor

☐ Create a unit test project

Test project name:
AspNetDatabases.Tests

Test framework:
Visual Studio Unit Test

Additional Info

OK Cancel

Note

In a deployable application, we would want to implement security measures for various users and different degrees of access.

Creating a Menu Action

Open the **HomeController.cs** file and modify it as shown:

CODE TO TYPE: /Controllers/HomeController.cs

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

namespace AspNetDatabases.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Modify this template to jump start your ASP.NET MVC application.ASP.NET Database Lab";

            return View();
        }

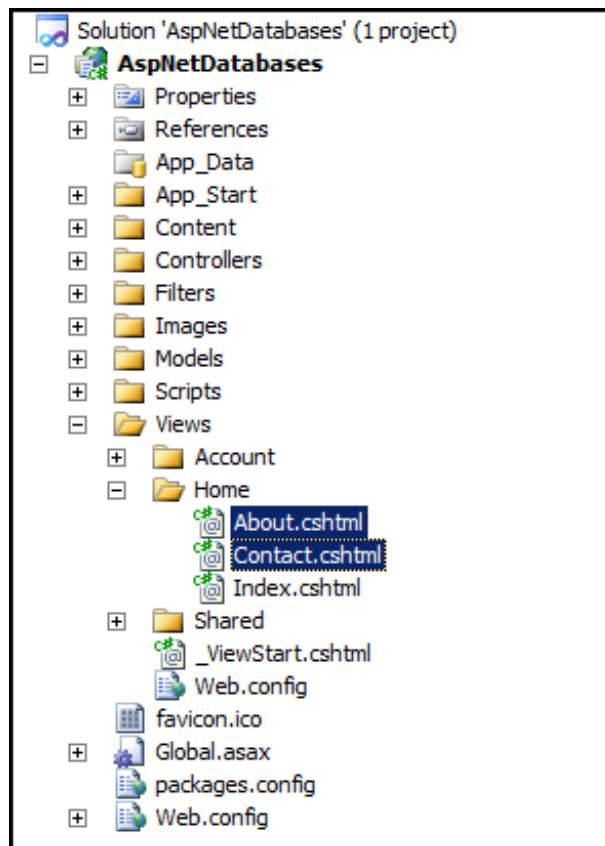
        public ActionResult About()
        {
            ViewBag.Message = "Your app description page.";

            return View();
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your contact page.";

            return View();
        }
    }
}
```

We've removed the About and Contact action methods, so we should remove them from our project. Delete them from the Views folder:



There's also some code we don't really need in the **Views/Shared/_Layout.cshtml** file. Open the file and modify it as shown:

CODE TO TYPE: /Views/Shared/_Layout.cshtml

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title - My ASP.NET MVC Application</title>
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
    <meta name="viewport" content="width=device-width" />
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
  </head>
  <body>
    <header>
      <div class="content-wrapper">
        <div class="float-left">
          <p class="site-title">@Html.ActionLink("your logo hereASP.NET Datab
ase Lab", "Index", "Home")</p>
        </div>
        <div class="float-right">
          <section id="login">
            @Html.Partial("_LoginPartial")
          </section>
          <nav>
            <ul id="menu">
              <li>@Html.ActionLink("Home", "Index", "Home")</li>
              <li>@Html.ActionLink("About", "About", "Home")</li>
              <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
            </ul>
          </nav>
        </div>
      </div>
    </header>
    <div id="body">
      @RenderSection("featured", required: false)
      <section class="content-wrapper main-content clear-fix">
        @RenderBody()
      </section>
    </div>
    <footer>
      <div class="content-wrapper">
        <div class="float-left">
          <p>© @DateTime.Now.Year - My ASP.NET MVC Application</p>
        </div>
      </div>
    </footer>

    @Scripts.Render("~/bundles/jquery")
    @RenderSection("scripts", required: false)
  </body>
</html>
```

Open the **/Views/Home/Index.cshtml** file and modify it as shown:

Note We'll leave the HTML structure in place, in case you need it later.

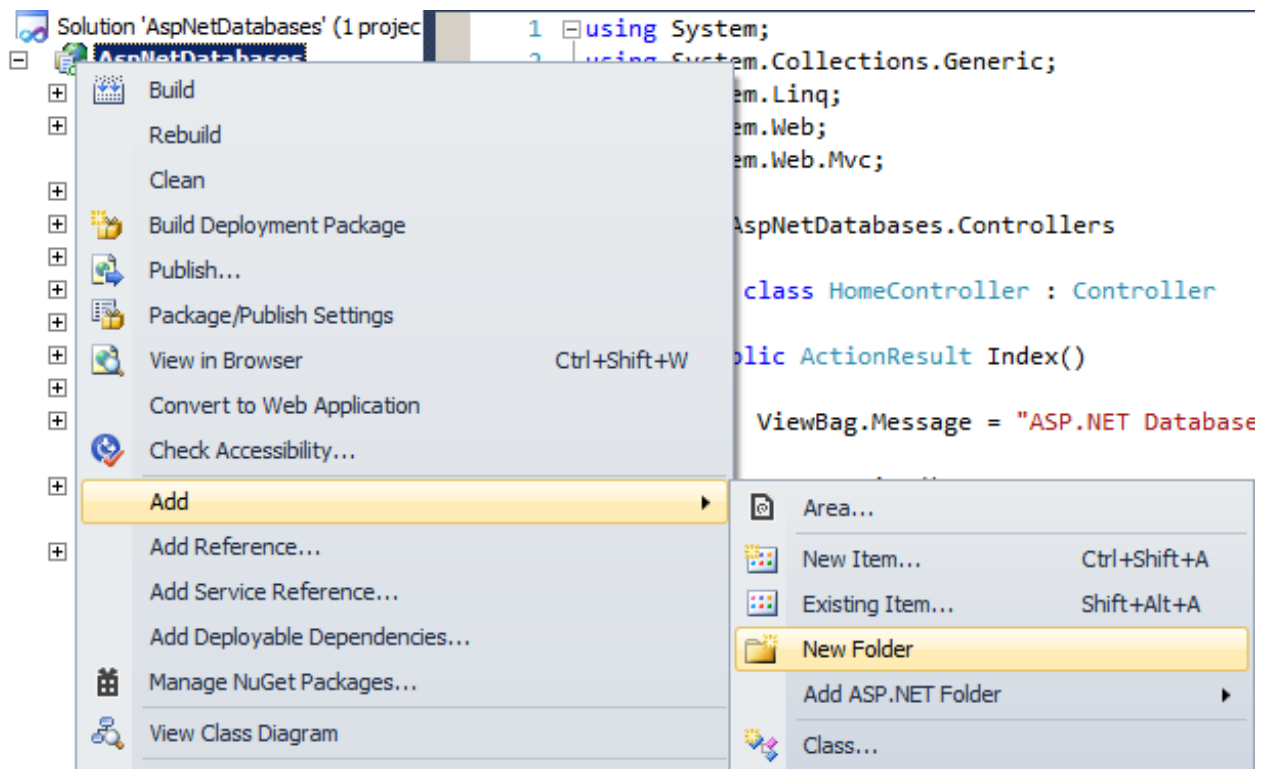
CODE TO TYPE: /Views/Home/Index.cshtml

```
@{
    ViewBag.Title = "Home Page";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2>@ViewBag.Message</h2>
            </hgroup>
            <p>
                To learn more about ASP.NET MVC visit
                <a href="http://asp.net/mvc" title="ASP.NET MVC Website">http://asp.net/mvc</a>.
                The page features <mark>videos, tutorials, and samples</mark> to help y
                ou get the most from ASP.NET MVC.
                If you have any questions about ASP.NET MVC visit
                <a href="http://forums.asp.net/1146.aspx/1?MVC" title="ASP.NET MVC Foru
                m">our forums</a>.
            </p>
        </div>
    </section>
}
<h3>We suggest the following:</h3>
<ol class="round">
    <li class="one">
        <h5>Getting Started</h5>
        ASP.NET MVC gives you a powerful, patterns based way to build dynamic websites
        that
        enables a clean separation of concerns and that gives you full control over mar
        kup
        for enjoyable, agile development. ASP.NET MVC includes many features that enabl
        e
        fast, TDD friendly development for creating sophisticated applications that use
        the latest web standards.
        <a href="http://go.microsoft.com/fwlink/?LinkId=245151">Learn more...</a>
    </li>

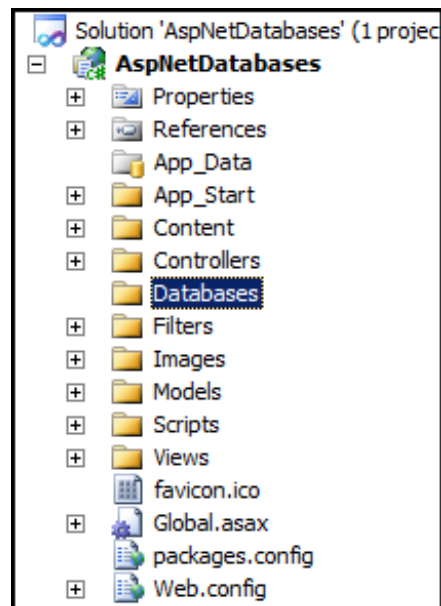
    <li class="two">
        <h5>Add NuGet packages and jump start your coding</h5>
        NuGet makes it easy to install and update free libraries and tools.
        <a href="http://go.microsoft.com/fwlink/?LinkId=245153">Learn more...</a>
    </li>

    <li class="three">
        <h5>Find Web Hosting</h5>
        You can easily find a web hosting company that offers the right mix of features
        and price for your applications.
        <a href="http://go.microsoft.com/fwlink/?LinkId=245157">Learn more...</a>
    </li>
</ol>
```

Create a new folder named **Databases** in your project:



Your new folder appears in the Solution Explorer:



Now, in the **/Databases** folder, create a new class named **CdDbInitializer**.

Modify the **/Databases/CdDbInitializer** class as shown:

CODE TO TYPE: /Databases/CdDbInitializer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using AspNetDatabases.Models;

namespace AspNetDatabases.Databases
{
    public class CdDbInitializer : DropCreateDatabaseIfModelChanges<CdDbContext>
    {
        protected override void Seed(CdDbContext context)
        {
            var CdDbItems = new List<CdDb>
            {
                new CdDb{ ID = 0, Title = "Desire", Artist = "Bob Dylan" },
                new CdDb{ ID = 1, Title = "Crossroads", Artist = "Eric Clapton" }
            };
            CdDbItems.ForEach(m => context.cddb.Add(m) );
            context.SaveChanges();
        }
    }
}
```



but don't run the application yet.

OBSERVE: /Databases/CdDbInitializer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using AspNetDatabases.Models;

namespace AspNetDatabases.Databases
{
    public class CdDbInitializer : DropCreateDatabaseIfModelChanges<CdDbContext>
    {
        protected override void Seed(CdDbContext context)
        {
            var CdDbItems = new List<CdDb>
            {
                new CdDb{ ID = 0, Title = "Desire", Artist = "Bob Dylan" },
                new CdDb{ ID = 1, Title = "Crossroads", Artist = "Eric Clapton" }
            };
            CdDbItems.ForEach(m => context.cddb.Add(m) );
            context.SaveChanges();
        }
    }
}
```

The **DropCreateDatabaseIfModelChanges** class will recreate our database the first time the context is used in our app domain. We are extending this class and overriding the **Seed** method so that when the database is created, it will be seeded with our default values.

Our **CdDbItems** list holds instances of our **CdDb** class, which will be used to seed the database's CdDb table.

When we **saveChanges()**, the database is seeded. It won't appear on our menu yet—we'll get to that later.

Now, we need to create a **CdDbContext** class in the **/Databases** folder that will serve as the context for our database CdDb table.

Modify the **/Databases/CdDbContext** class as shown:

CODE TO TYPE: /Databases/CdDbContext.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using AspNetDatabases.Models;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace AspNetDatabases.Databases
{
    public class CdDbContext : DbContext
    {
        public DbSet<CdDb> cddb { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```



but again, don't run it.

OBSERVE: /Databases/CdDbContext.cs

```
.
.
.
namespace AspNetDatabases.Databases
{
    public class CdDbContext : DbContext
    {
        public DbSet<CdDb> cddb { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

Here we set up the context by which to reference our **CdDb** table. We'll be able to access this context through our **cddb** property. When we **Remove()** the **PluralizingTableNameConvention** object from the conventions of our **modelBuilder**, we are indicating that we don't want our table names to be plural versions of the objects that create them. That is, our **DBSet<CdDb>** creates a **CdDb** table name rather than **CdDbs**.

In the **/Models** folder, create a new model named **CdDb**, and modify it as shown:

CODE TO TYPE: /Models/CdDb.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace AspNetDatabases.Models
{
    public class CdDb
    {
        [Key]
        public int ID { get; set; }
        public string Title { get; set; }
        public string Artist { get; set; }
    }
}
```



but don't run it.

OBSERVE: /Models/CdDb.cs

```
.
.
.
namespace AspNetDatabases.Models
{
    public class CdDb
    {

        [Key]
        public int ID { get; set; }
        public string Title { get; set; }
        public string Artist { get; set; }

    }
}
```

Here the **CdDb** class represents our CdDb table. The properties we are creating will map to the columns in the table. So we will have columns named **ID**, **Title**, and **Artist**, representing the ID of the CD, which is the primary key, the title of the album, and the artist. This is by no means all of the data we could put in this table.

We'll also need to modify the **/Global.asax** file to add the ability to use our database in our project. Let's do that now:

CODE TO TYPE: /Global.asax

```
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using System.Data.Entity;
using AspNetDatabases.Models;
using AspNetDatabases.Databases;

namespace AspNetDatabases
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            AuthConfig.RegisterAuth();

            Database.SetInitializer<CdDbContext>(new CdDbInitializer());
        }
    }
}
```



but don't run it.

OBSERVE: /Global.asax

```
.
.
.
    Database.SetInitializer<CdDbContext>(new CdDbInitializer());
.
.
.
```

Here we tell our application to use the **CdDbInitializer()** method to initialize **CdDbContext**, and set up our database. We'll add the connection string to our database file later in the **/Web.config** file.

In the **/Controllers** folder, create a new controller named **CdDbController** and modify it as shown:

Add Controller [X]

Controller name:
CdDbController

Scaffolding options

Template:
Empty MVC controller

Model class:

Data context class:

Views:
None

Advanced Options...


Add Cancel

CODE TO TYPE: /Controllers/CdDbController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AspNetDatabases.Databases;
using AspNetDatabases.Models;

namespace AspNetDatabases.Controllers
{
    public class CdDbController : Controller
    {
        private CdDbContext DB = new CdDbContext();
        //
        // GET: /CdDb/

        public ActionResult Index()
        {
            return View(DB.cddb.ToList());
        }
    }
}
```

 but don't run it.

OBSERVE: /Controllers/CdDbController.cs

```
.  
. .  
namespace AspNetDatabases.Controllers  
{  
    public class CdDbController : Controller  
    {  
        private CdDbContext DB = new CdDbContext();  
        //  
        // GET: /CdDb/  
  
        public ActionResult Index()  
        {  
            return View(DB.cddb.ToList());  
        }  
    }  
}
```

This controller returns a view that will display the CdDb table. We create a new variable named **DB** that is an instance of **CdDbContext**. We can get all of our rows from the **DB** as a list that we can then iterate through in the view.

Now we'll create a view from CdDbController:

Add View

View name:

View engine:

☒ Create a strongly-typed view

Model class:

Scaffold template:

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

Modify **/Views/CdDb/Index.cshtml** as shown:

CODE TO TYPE: /Views/CdDb/Index.cshtml

```
@model IEnumerable<AspNetDatabases.Models.CdDb>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Artist)
        </th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(model => item.Title)
            </td>
            <td>
                @Html.DisplayFor(model => item.Artist)
            </td>
        </tr>
    }
</table>
```



but don't run it. We still have a couple more things to do.

OBSERVE: /Views/CdDb/Index.cshtml

```
@model IEnumerable<AspNetDatabases.Models.CdDb>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Artist)
        </th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(model => item.Title)
            </td>
            <td>
                @Html.DisplayFor(model => item.Artist)
            </td>
        </tr>
    }
</table>
```

In this view, we create an HTML table that shows the contents of the database. We use the **CdDb** class as the model and we show the HTML table header with the name of the **Title** and **Artist** columns in our database. Then, we show the contents of those columns for each row in the database, using the **item** set up in our **foreach** loop. The **item** variable will represent a row in the database, and we can reference each data item by column name.

We're almost done, but we still need to configure our **Web.config** file so that we can use the database.

Add the connection string to **Web.config** as shown:

CODE TO TYPE: /Web.config

```
...

<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.micros
oft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.Entity
FrameworkSection, EntityFramework, Version=4.4.0.0, Culture=neutral, PublicKeyToken=b77
a5c561934e089" requirePermission="false" />
  </configSections>
  <connectionStrings>

    <add name="DefaultConnection"
      connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=aspnet-AspNetDataba
ses-20130608134956;Integrated Security=SSPI"
      providerName="System.Data.SqlClient"
    />

    <add name="CddbContext"
      connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=Cddb;AttachDBFile=|
DataDirectory|\Cddb.mdf;User Instance=true;Integrated Security=SSPI"
      providerName="System.Data.SqlClient"/>

  </connectionStrings>

...
```

Note

The connection string above ours is put into the file by Visual Studio. (Yours might differ slightly from the one shown here.) The **CddbContext** connection string gives us a name we can use to connect to the database.

Before we test it, we need to tell the system to add our action to the menu. Modify **/Views/Shared/_Layout.shtml** as shown:

CODE TO TYPE: /Views/Shared/_Layout.shtml

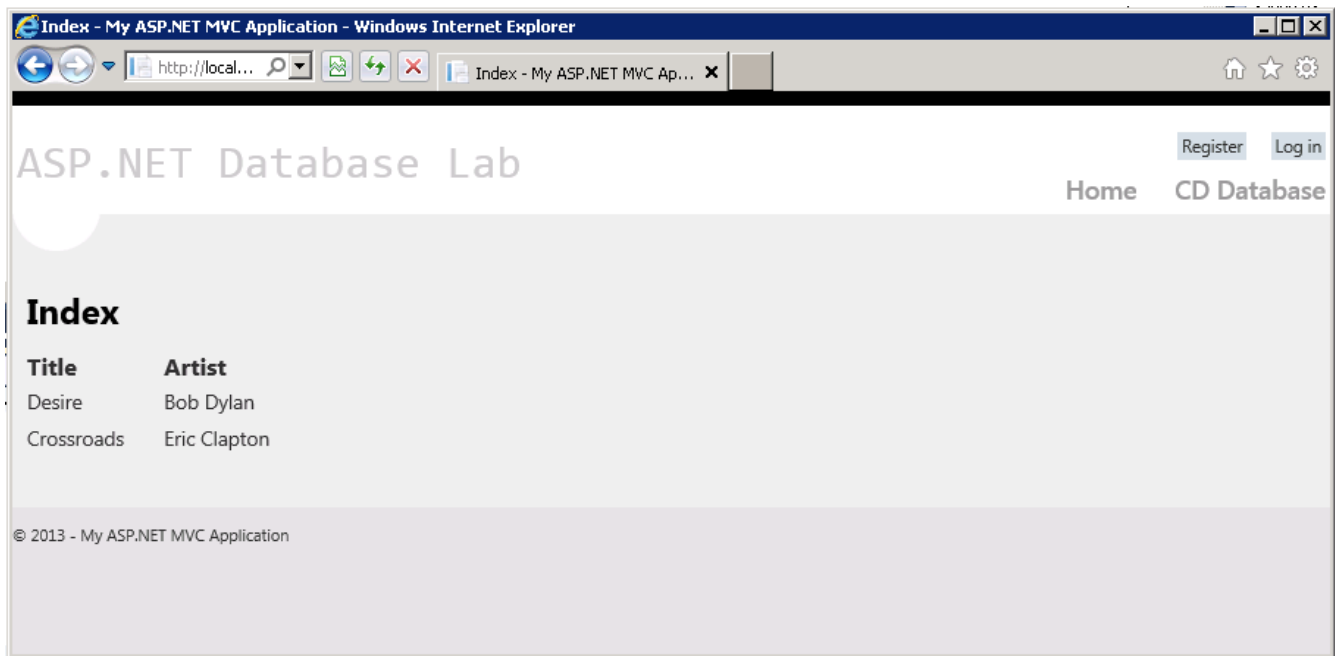
```
...

    <header>
      <div class="content-wrapper">
        <div class="float-left">
          <p class="site-title">@Html.ActionLink("ASP.NET Database Lab", "Index", "Home")</p>
        </div>
        <div class="float-right">
          <section id="login">
            @Html.Partial("_LoginPartial")
          </section>
          <nav>
            <ul id="menu">
              <li>@Html.ActionLink("Home", "Index", "Home")</li>
              <li>@Html.ActionLink("CD Database", "Index", "Cddb")</li>
            </ul>
          </nav>
        </div>
      </div>
    </header>

...
```



and  Click the **CD Database** menu item.



OBSERVE: /Views/Shared/_Layout.shtml

```
...  
  
    <header>  
        <div class="content-wrapper">  
            <div class="float-left">  
                <p class="site-title">@Html.ActionLink("ASP.NET Database Lab", "Index", "Home")</p>  
            </div>  
            <div class="float-right">  
                <section id="login">  
                    @Html.Partial("_LoginPartial")  
                </section>  
                <nav>  
                    <ul id="menu">  
                        <li>@Html.ActionLink("Home", "Index", "Home")</li>  
                        <li>@Html.ActionLink("CD Database", "Index", "Cddb")</li>  
                    </ul>  
                </nav>  
            </div>  
        </div>  
    </header>  
  
...
```

The `Html.ActionLink()` method uses the **CD Database** parameter as the clickable text for the link and will use the controller, **Cddb**, to invoke the action **Index**.

Note

You may get an error the first time you run the application. If you do, exit debugging and run the application again. Generally, this happens only the first time you run the app. However, if it continues, you may need to delete the `/App_Data/Cddb.mdf` file. In order to see if that's the case, click the **Show All Files** button in the Solution Explorer. If the **model** changes and the database and model are out of sync, delete the database file and let the app create a new one.



Creating, Editing, Deleting, Searching

Now we'll go over how to create, edit, delete, and search the data we get from the database. We'll update **/Views/CdDb/Index.shtml**, then we'll update the controller so that we can act on those requests. Finally, we'll create views as needed so that the user can complete the actions.

Modify **/Views/CdDb/Index.shtml** as shown:

CODE TO TYPE: /Views/CdDb/Index.shtml

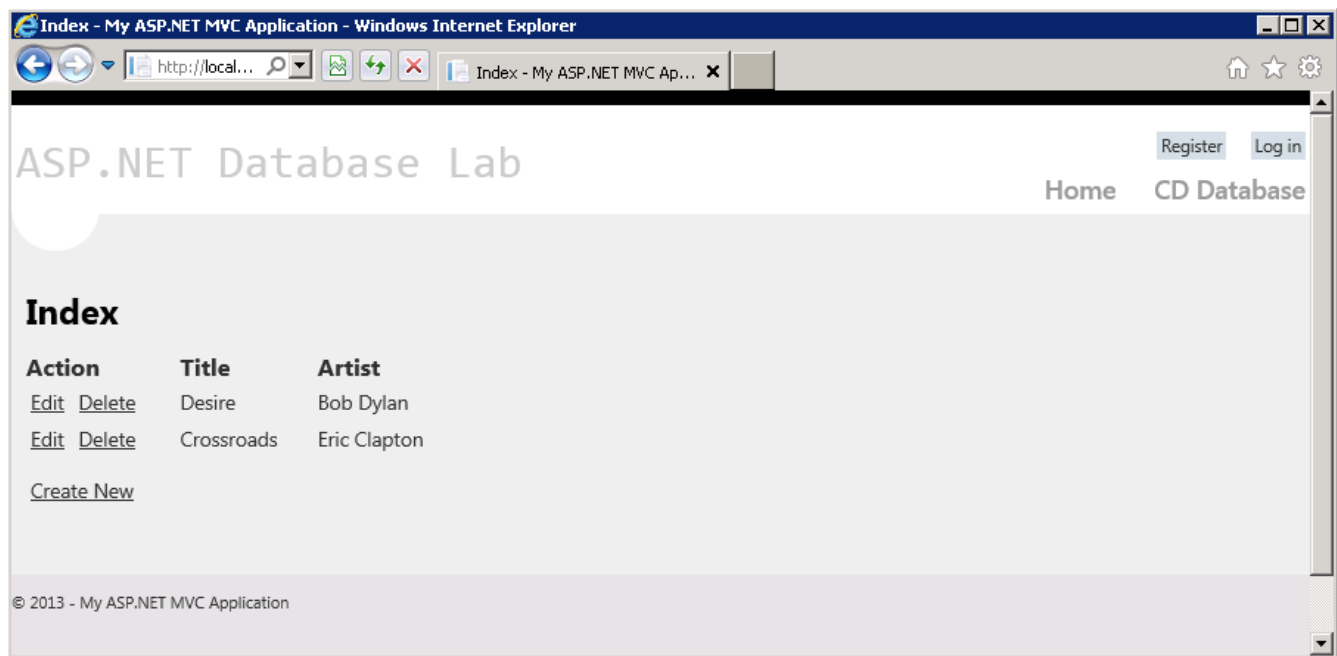
```
@model IEnumerable<AspNetDatabases.Models.CdDb>
@{
    ViewBag.Title = "Index";
}

<h2>
    Index</h2>
<table>
    <tr>
        <th>
            Action
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Artist)
        </th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.ActionLink("Edit", "Edit", new { ID = item.ID })
                @Html.ActionLink("Delete", "Delete", new { ID = item.ID })
            </td>
            <td>
                @Html.DisplayFor(model => item.Title)
            </td>
            <td>
                @Html.DisplayFor(model => item.Artist)
            </td>
        </tr>
    }
</table>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
```



Save your file and start debugging. Click the **CD Database** menu item. Now the view has some additional links, but they don't work. We'll fix that in a moment.



OBSERVE: /Views/CdDb/Index.shtml

```
.  
. .  
.  
@foreach (var item in Model)  
{  
    <tr>  
  
        <td>  
            @Html.ActionLink("Edit", "Edit", new { ID = item.ID })  
            @Html.ActionLink("Delete", "Delete", new { ID = item.ID })  
        </td>  
  
        <td>  
            @Html.DisplayFor(model => item.Title)  
        </td>  
        <td>  
            @Html.DisplayFor(model => item.Artist)  
        </td>  
    </tr>  
}  
</table>  
  
<p>  
    @Html.ActionLink("Create New", "Create")  
</p>
```

We created two action links that, when clicked, will call the **Edit** and **Delete** methods in our controller. We'll pass back the **ID** of the item to those methods.

Speaking of the controller, modify the **/Controllers/CdDbController.cs** file now:

CODE TO TYPE: /Controllers/CdDbController.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AspNetDatabases.Databases;
using AspNetDatabases.Models;

namespace AspNetDatabases.Databases
{
    public class CdDbController : Controller
    {
        private CdDbContext DB = new CdDbContext();
        //
        // GET: /CdDb/
        public ActionResult Index()
        {
            return View(DB.cddb.ToList());
        }

        //
        // GET: /CdDb/Create
        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /CdDb/Create
        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude = "ID")] CdDb cdToCreate)
        {
            if (!ModelState.IsValid)
            {
                return View();
            }
            DB.cddb.Add(cdToCreate);
            DB.SaveChanges();
            return RedirectToAction("Index");
        }
    }
}
```



. Let's discuss the new code.

OBSERVE: /Controllers/CdDbController.cs

```
.  
.   
.   
  
    //   
    // GET: /CdDb/Create   
    public ActionResult Create()   
    {   
        return View();   
    }   
  
    //   
    // POST: /CdDb/Create   
    [AcceptVerbs(HttpVerbs.Post)]   
    public ActionResult Create([Bind(Exclude = "ID")] CdDb cdToCreate)   
    {   
        if (!ModelState.IsValid)   
        {   
            return View();   
        }   
        DB.cddb.Add(cdToCreate);   
        DB.SaveChanges();   
        return RedirectToAction("Index");   
    }   
}
```

Here, we add two methods. The first method, **Create()**, returns the same view because we only want our data to be passed via the HTTP POST method. Since we are creating a new CD, this works well because it will give us an empty form. (Later we'll use the **Edit()** method to get the item we want.) The second method, **Create()**, submits an HTML form, which uses the HTML POST method. The line **[AcceptVerbs(HttpVerbs.Post)]** means that the **Create()** method is only used with the HTTP POST method.

We want our ID field to be updated automatically, so we exclude it from everything, knowing that SQL Server will enter it for us. We exclude our ID using the command **[Bind(Exclude = "ID")]** in the parameters of the method. The view that we'll create in a moment, will send the CdDb model object as the parameter for this method.

If our model's state is not valid, we don't want to update the database; **if it is not valid**, we return the view with no changes. If it is valid, we Add() the new object to the database and save the changes. Then we redirect to the Index action, which reloads our Index view.

Here comes the cool part. Normally, we'd have to build a form to call of these actions, but since our requirements are fairly common, Visual Studio can look at our Model and get all of the information it needs to create our View. All we will have to do is fill in some data in a dialog.

Right-click in the second **Create()** method and select **Add View** and complete the dialog as shown:

Add View

View name:
Create

View engine:
Razor (CSHTML)

☒ Create a strongly-typed view

Model class:
Cddb (AspNetDatabases.Models)

Scaffold template:
Create

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:
...
(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:
MainContent

Add Cancel

 and  Click the **CD Database** menu item and then click the **Create New** link. Fill in the form for a new CD and click the **Create** button. How cool is that?

We have the exact file structure we need in order to add an item to our database. Let's take a look at what Visual Studio did:

OBSERVE: /Views/CdDb/Create.cshtml

```
.
.
.
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>CdDb</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Artist)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Artist)
            @Html.ValidationMessageFor(model => model.Artist)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

First, the **@using (Html.BeginForm())** statement indicates that we're going to be using an HTML form. The "**@using (Html.BeginForm())**" statement will set up all the form content on the page and basic form-processing code for us.

@Html.ValidationSummary(true) allows the form to show a summary of validation errors when they appear. We don't have any validation code yet, so **@Html.ValidationSummary(true)** doesn't do anything right now.

The **@Html.EditorFor** methods give us an editor area that is tied to a field in our model object. Whatever we type into those fields will be added to the object that is passed to the **Create()** event in the controller.

The last item, **@section Scripts**, uses the **Scripts.Render()** method to bundle a bunch of jQuery scripts that are required to make this all work. (JavaScript will be discussed in greater detail in a later lesson.)

Note

The **Scripts.Render()** method bundles several scripts into a single JavaScript file before loading them. This is called **minifying** the scripts. It cuts down on the number of times the server has to be contacted. Without using this type of system, each file would have to be loaded separately. For more information on minifying scripts, read [this article](#)

If you don't put anything in the text boxes on the Create page, then run the program again, the object will be added to the database with empty items. Let's fix that now.

Modify **/Models/CdDb.cs** as shown:

CODE TO TYPE: /Models/CdDb.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace AspNetDatabases.Models
{
    public class CdDb
    {
        [Key]
        public int ID { get; set; }
        [Required(ErrorMessage = "Please enter a Title.")]
        public string Title { get; set; }
        [Required(ErrorMessage = "Please enter an Artist.")]
        public string Artist { get; set; }
    }
}
```



and . Click **CD Database** and then click **Create New**. Try to click **Create** with nothing in the text boxes. Cool, huh?

The **Required** attribute will force validation code to make sure that the data item is not empty or null. It will cause validation messages to be displayed using the error message in the attribute. It will also set the **ModelState.IsValid** to false:

OBSERVE: /Models/CdDb.cs

```
.
.
.
namespace AspNetDatabases.Models
{
    public class CdDb
    {
        [Key]
        public int ID { get; set; }
        [Required(ErrorMessage = "Please enter a Title.")]
        public string Title { get; set; }
        [Required(ErrorMessage = "Please enter an Artist.")]
        public string Artist { get; set; }
    }
}
```

By adding the **Required** option to each of the fields, we've made it so they cannot be empty strings.

Let's add code that will enable us to edit an entry that already exists in the database. Modify **/Controllers/CdDbController.cs** as shown:

CODE TO TYPE: /Controllers/CdDbController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AspNetDatabases.Databases;
using AspNetDatabases.Models;
using System.Data;
using System.Data.Entity;

namespace AspNetDatabases.Databases
{
    public class CdDbController : Controller
    {
        private CdDbContext DB = new CdDbContext();
        //
        // GET: /CdDb/
        public ActionResult Index()
        {
            return View(DB.cddb.ToList());
        }

        //
        // GET: /CdDb/Create
        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /CdDb/Create
        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude = "ID")] CdDb cdToCreate)
        {
            if (!ModelState.IsValid)
            {
                return View();
            }
            DB.cddb.Add(cdToCreate);
            DB.SaveChanges();
            return RedirectToAction("Index");
        }

        //
        // GET: /CdDb/Edit/
        public ActionResult Edit(int id = 0)
        {
            CdDb item = DB.cddb.Find(id);
            if (item == null)
            {
                return RedirectToAction("Index");
            }
            return View(item);
        }

        //
        // POST: /CdDb/Edit/
        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(CdDb cddbToEdit)
        {
            if (!ModelState.IsValid)
            {
                return View(cddbToEdit);
            }
            DB.Entry(cddbToEdit).State = EntityState.Modified;
            DB.SaveChanges();
            return RedirectToAction("Index");
        }
    }
}
```

```
}  
    }  
}
```



Save your file but don't run it.

OBSERVE: /Controller/CdDbController.cs

```
.  
. .  
. .  
. .  
  
    //  
    // GET: /CdDb/Edit/  
    public ActionResult Edit(int id = 0)  
    {  
        CdDb item = DB.cdDb.Find(id);  
        if (item == null)  
        {  
            return RedirectToAction("Index");  
        }  
        return View(item);  
    }  
    //  
    // POST: /CdDb/Edit/  
    [AcceptVerbs(HttpVerbs.Post)]  
    public ActionResult Edit(CdDb cddbToEdit)  
    {  
        if (!ModelState.IsValid)  
        {  
            return View(cddbToEdit);  
        }  
        DB.Entry(cddbToEdit).State = EntityState.Modified;  
        DB.SaveChanges();  
        return RedirectToAction("Index");  
    }  
}
```

Here's what happens when a user edits an entry:

1. The user clicks the **Edit** link beside an entry.
2. The browser sends a GET request to the **Edit** action with the ID of the requested entry.
3. The **Edit**(int id=0) method returns a View with that entry, which fills the text boxes with their data.
4. The user edits the entry and saves it.
5. The save submission is sent back to the server using a POST request.
6. The **Edit**(CdDb cddbToEdit) method is called.
7. The **Edit**(CdDb cddbToEdit) method sets the **State** of the entry to **Modified** and saves the changes.
8. The **Edit**(CdDb cddbToEdit) method redirects the user to the Index page where the listing is updated to show the changes in the database.

Of course, we can't do that yet since we don't have a view for this part of the controller.

Right-click in the second **Edit()** method and select **Add View**. Add a view as shown:

Add View [X]

View name:

View engine:

☒ Create a strongly-typed view

Model class:

Scaffold template:
 ☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:
 ...
(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

Modify **/Views/CdDb/Edit.cshtml** as shown:

CODE TO TYPE: /Views/CdDb/Edit.cshtml

```
@model AspNetDatabases.Models.CdDb

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>CdDb</legend>

        @Html.HiddenFor(model => model.ID)

        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Artist)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Artist)
            @Html.ValidationMessageFor(model => model.Artist)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```



and , click **CD Database**, select an entry to edit, modify the entry, and then save it. The changes are saved and displayed when you return to the Index page:

```
.
.
.
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>CdDb</legend>

        @Html.HiddenFor(model => model.ID)

        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Artist)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Artist)
            @Html.ValidationMessageFor(model => model.Artist)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

This probably looks familiar. It's similar to the **/Views/Create.cshtml** file. The only real difference is that there is a **hidden element for the ID** and the **submit button says Save** instead of Create.

Finally, we want to enable the user to delete an entry. Modify **/Controllers/CdDbController.cs** as shown:

CODE TO TYPE: /Controllers/CdDbController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AspNetDatabases.Databases;
using AspNetDatabases.Models;
using System.Data;
using System.Data.Entity;

namespace AspNetDatabases.Databases
{
    public class CdDbController : Controller
    {
        .
        .
        .

        //
        // POST: /CdDb/Edit/
        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(CdDb cddbToEdit)
        {
            if (!ModelState.IsValid)
            {
                return View(cddbToEdit);
            }
            DB.Entry(cddbToEdit).State = EntityState.Modified;
            DB.SaveChanges();
            return RedirectToAction("Index");
        }

        //
        // GET: /CdDb/Delete/
        public ActionResult Delete(int id = 0)
        {
            CdDb item = DB.cddb.Find(id);
            if (item == null)
            {
                return RedirectToAction("Index");
            }
            return View(item);
        }

        //
        // POST: /CdDb/Delete/
        [AcceptVerbs(HttpVerbs.Post), ActionName("Delete")]
        public ActionResult DeleteConfirmed(int id = 0)
        {
            CdDb item = DB.cddb.Find(id);
            if (item == null)
            {
                return RedirectToAction("Index");
            }
            DB.cddb.Remove(item);
            DB.SaveChanges();
            return RedirectToAction("Index");
        }
    }
}
```

OBSERVE: /Views/CdDbController.cs

```
.  
.    
.    
    //   
    // GET: /CdDb/Delete/   
    public ActionResult Delete(int id = 0)   
    {   
        CdDb item = DB.cddb.Find(id);   
        if (item == null)   
        {   
            return RedirectToAction("Index");   
        }   
        return View(item);   
    }   
    //   
    // POST: /CdDb/Delete/   
    [AcceptVerbs(HttpVerbs.Post), ActionName("Delete")]   
    public ActionResult DeleteConfirmed(int id = 0)   
    {   
        CdDb item = DB.cddb.Find(id);   
        if (item == null)   
        {   
            return RedirectToAction("Index");   
        }   
        DB.cddb.Remove(item);   
        DB.SaveChanges();   
        return RedirectToAction("Index");   
    }   
}   
}
```

Here, the first **Delete()** method finds the item and displays it by returning the view with the item as a parameter. The second method, **DeleteConfirmed()** is called when the user confirms the deletion. The **Remove()** method marks the item to be removed from the database. The **SaveChanges()** method actually causes the deletion.

Make a view for the page (using the first Delete() method) as we did before; make sure to use the **Delete** Scaffold Template.



and . Delete an entry in the database, using the new view you created.

The last topic we'll cover here is "searching."

Modify **/Controllers/CdDbController.cs** as shown.


CODE TO TYPE: /Controllers/CdDbController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using AspNetDatabases.Databases;
using AspNetDatabases.Models;
using System.Data;
using System.Data.Entity;

namespace AspNetDatabases.Databases
{
    public class CdDbController : Controller
    {
        .
        .
        .

        //
        // POST: /CdDb/Delete/
        [AcceptVerbs(HttpVerbs.Post), ActionName("Delete")]
        public ActionResult DeleteConfirmed(int id = 0)
        {
            CdDb item = DB.cddb.Find(id);
            if (item == null)
            {
                return RedirectToAction("Index");
            }
            DB.cddb.Remove(item);
            DB.SaveChanges();
            return RedirectToAction("Index");
        }

        //
        // GET: /CdDb/SearchIndex
        public ActionResult SearchIndex(string searchString)
        {
            var discs = from cd in DB.cddb
                        select cd;
            if (!String.IsNullOrEmpty(searchString))
            {
                discs = discs.Where(disc => disc.Title.Contains(searchString));
            }
            return View(discs);
        }
    }
}
```

 but don't run it.

OBSERVE: /Controllers/CdDbController.cs

```
.  
.   
.   
   
    //   
    // GET: /CdDb/SearchIndex   
    public ActionResult SearchIndex(string searchString)   
    {   
        var discs = from cd in DB.cdadb   
                     select cd;   
        if (!String.IsNullOrEmpty(searchString))   
        {   
            discs = discs.Where(disc => disc.Title.Contains(searchString));   
        }   
        return View(discs);   
    }   
}
```

We used the variable names `cd` and `disc`, but there is no indication of what these are. This could be one of the more frustrating aspects of C# and LINQ / lambda expressions. How and where they are used determines what they are. The quickest way to determine what they are to hover the mouse cursor over them in the editor. The code insight will tell you what they represent. In this case, they represent a model object, `CdDb`. The `cd` variable is a range variable, because it is a range in a LINQ query. `disc` is a parameter to the `Where()` method. The names of the variables are arbitrary; you can name them whatever you want.

We select all of the CDs from the `CdDb` table in our database. Then, we use the `Where()` method to match those objects with our `searchString`. When the View is rendered, the query is run on the list we gave it and only the objects that match will be displayed.

We need to create a view from this method, so do that now:

Add View [X]

View name:

View engine:

☒ Create a strongly-typed view

Model class:

Scaffold template:
 ☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

Modify the generated **/Views/CdDb/SearchIndex.cshtml** as shown:

CODE TO TYPE: /Views/SearchIndex.cshtml

```
@model IEnumerable<AspNetDatabases.Models.Cddb>

@{
    ViewBag.Title = "SearchIndex";
}

<h2>SearchIndex</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using (Html.BeginForm()) {
    <p> Title: @Html.TextBox("SearchString")<br />
    <input type="submit" value="Search" /></p>
}

<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Artist)
        </th>
        <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Artist)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
                @Html.ActionLink("Details", "Details", new { id=item.ID }) +
                @Html.ActionLink("Delete", "Delete", new { id=item.ID })
            </td>
        </tr>
    }
</table>
```

Let's discuss the new code.

OBSERVE: /Views/SearchIndex.cshtml

```
@model IEnumerable<AspNetDatabases.Models.Cddb>

@{
    ViewBag.Title = "SearchIndex";
}

<h2>SearchIndex</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using (Html.BeginForm()) {
    <p> Title: @Html.TextBox("SearchString")<br />
    <input type="submit" value="Search" /></p>
}

<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Artist)
        </th>
        <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Artist)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.ID })
            </td>
        </tr>
    }
</table>
```

Here, we **enable the user to type a search term**. If we didn't add that capability, the user would have to tack the search term onto the URL, which isn't very user-friendly. We also removed the "Details" link, since we haven't implemented that functionality.

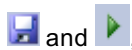
Note You rarely need to modify the generated code like this, but we wanted to show you how just in case.

Now let's give the user a way to get to the search page. Modify **/Views/Cddb/Index.cshtml** as shown:

CODE TO TYPE: /Views/CdDb/Index.cshtml

```
@model IEnumerable<AspNetDatabases.Models.CdDb>
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
@Html.ActionLink("Search", "SearchIndex")
<table>
.
.
.
</table>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
```



and , and enter a title to test out the search functionality.

Great work in this lesson. Practice what you've learned here in your homework. We'll meet again in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Object Relational Mapping: Entity Framework and Data Abstraction

Lesson Objectives

In this lesson, you will:

- implement and understand how the Entity Framework creates and implements mappings between entities and types.
 - implement Complex type to simplify an abstract entity design.
 - implement data abstraction in singular and generalized repositories.
 - use the Unit of Work design pattern and interact with a generic repository.
 - lock down the "behind-the-scenes" code to limit other developers from changing how the application interacts with the database.
 - keep entities in sync and limit database context creations to a single connection, rather than creating a different context for each entity.
-

During this lesson, you will create and implement a specialized repository, a generic repository, and a Unit of Work design pattern to abstract data from the application and begin to build an API for an application that can be used in a collaborative design team environment.

Entity Framework

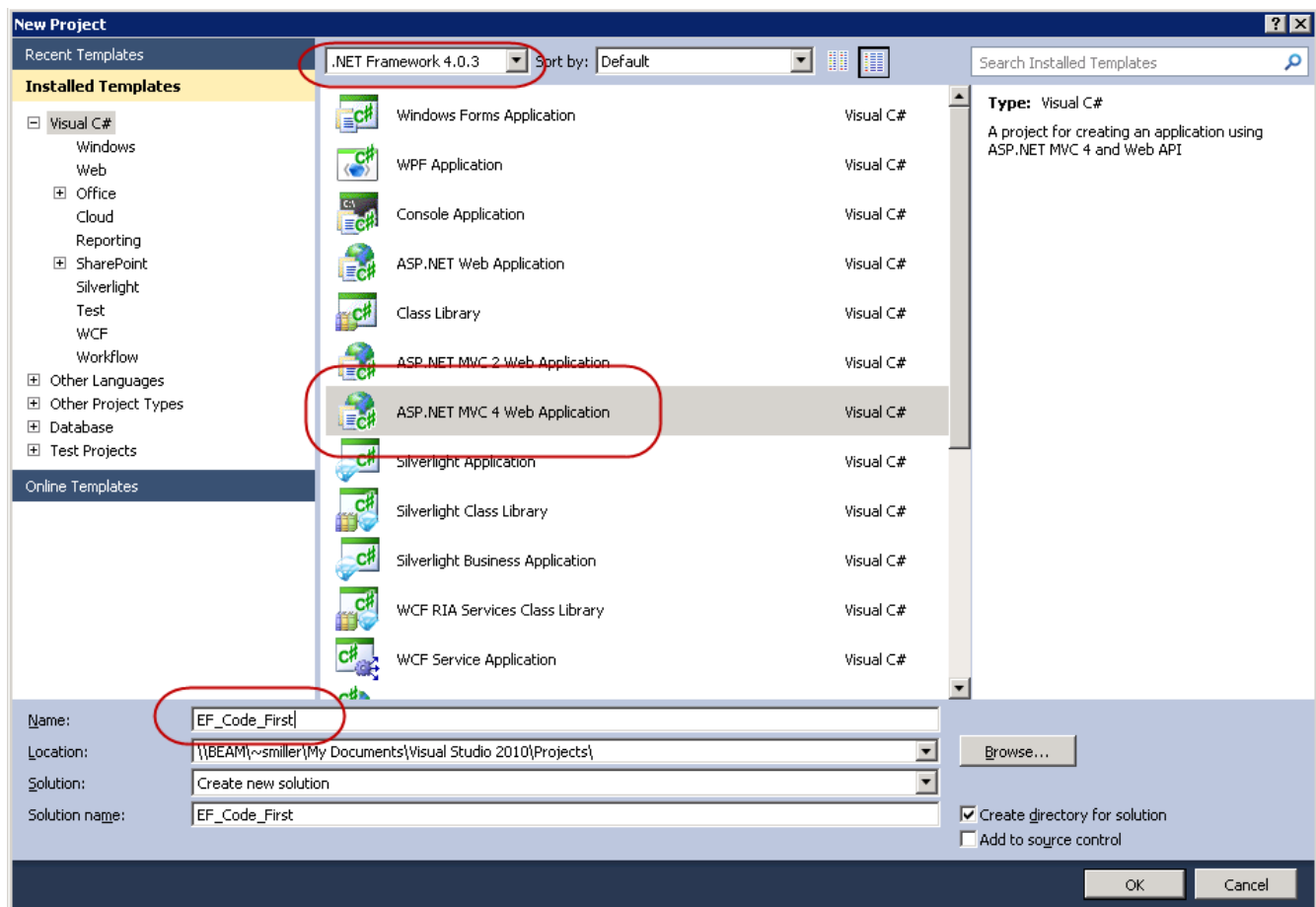
The Entity Framework allows us to work at a higher level of abstraction without worrying about the way data is stored in a database. This framework also allows us to write less code to perform the same operations in traditional applications. Entity Framework (EF) allows us to work with data in the form of domain-specific objects and properties.

This basic design approach to an application divides the application into three parts:

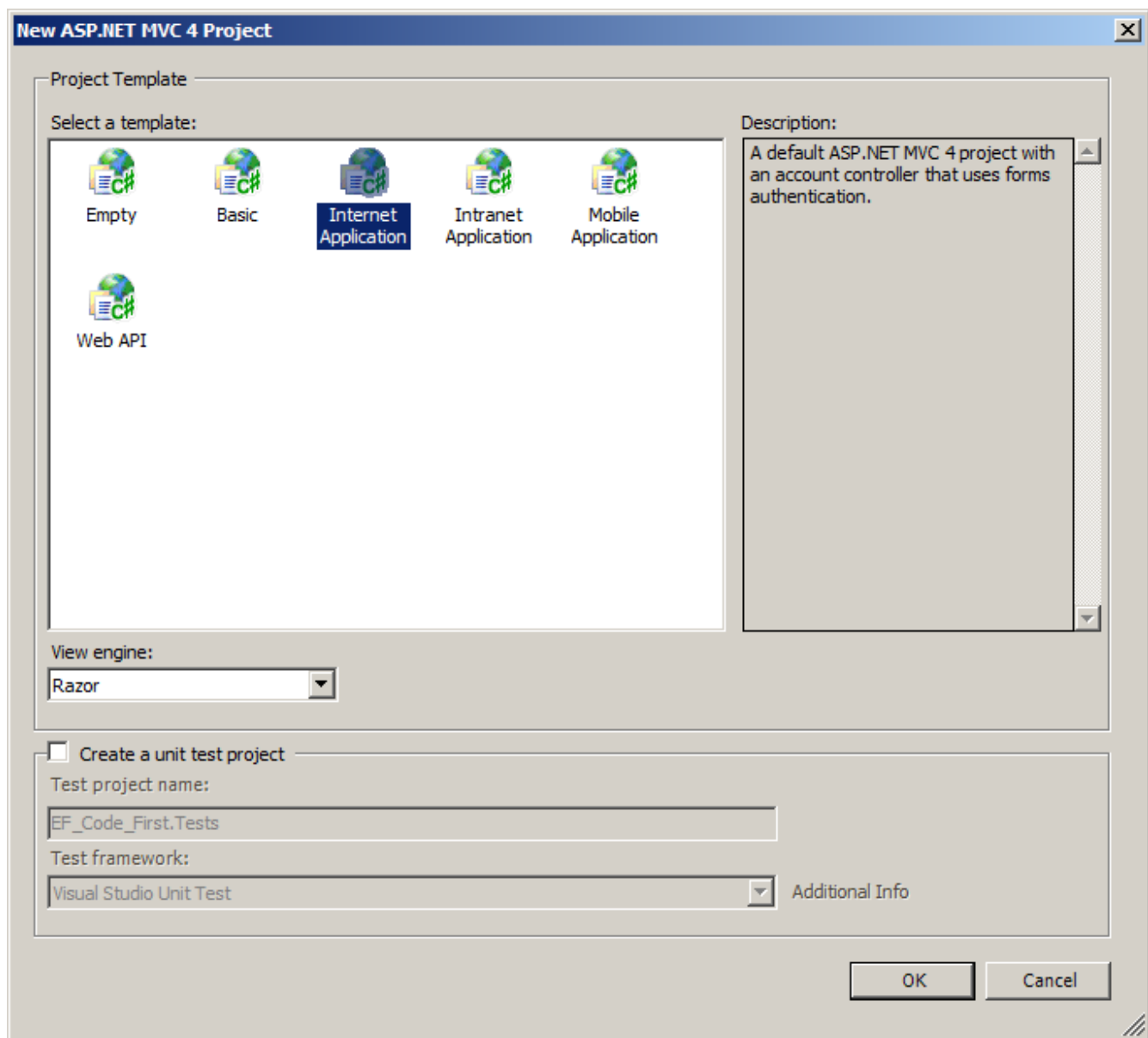
- **Physical Model:** addresses and defines capabilities of data engine's specifying storage details.
- **Logical Model:** SQL queries and stores procedure calls, normalizes entities and relationships into tables, using foreign key constraints.
- **Domain Model:** defines entities and relationships in the modeled system (Mappings).

The Entity Framework using the Code First approach allows us to use our own domain classes to represent the model that EF will use to perform queries, updates, and track changes. This approach uses a programming pattern known as *Convention over Configuration*. This means that EF will assume we are using the EF conventions.

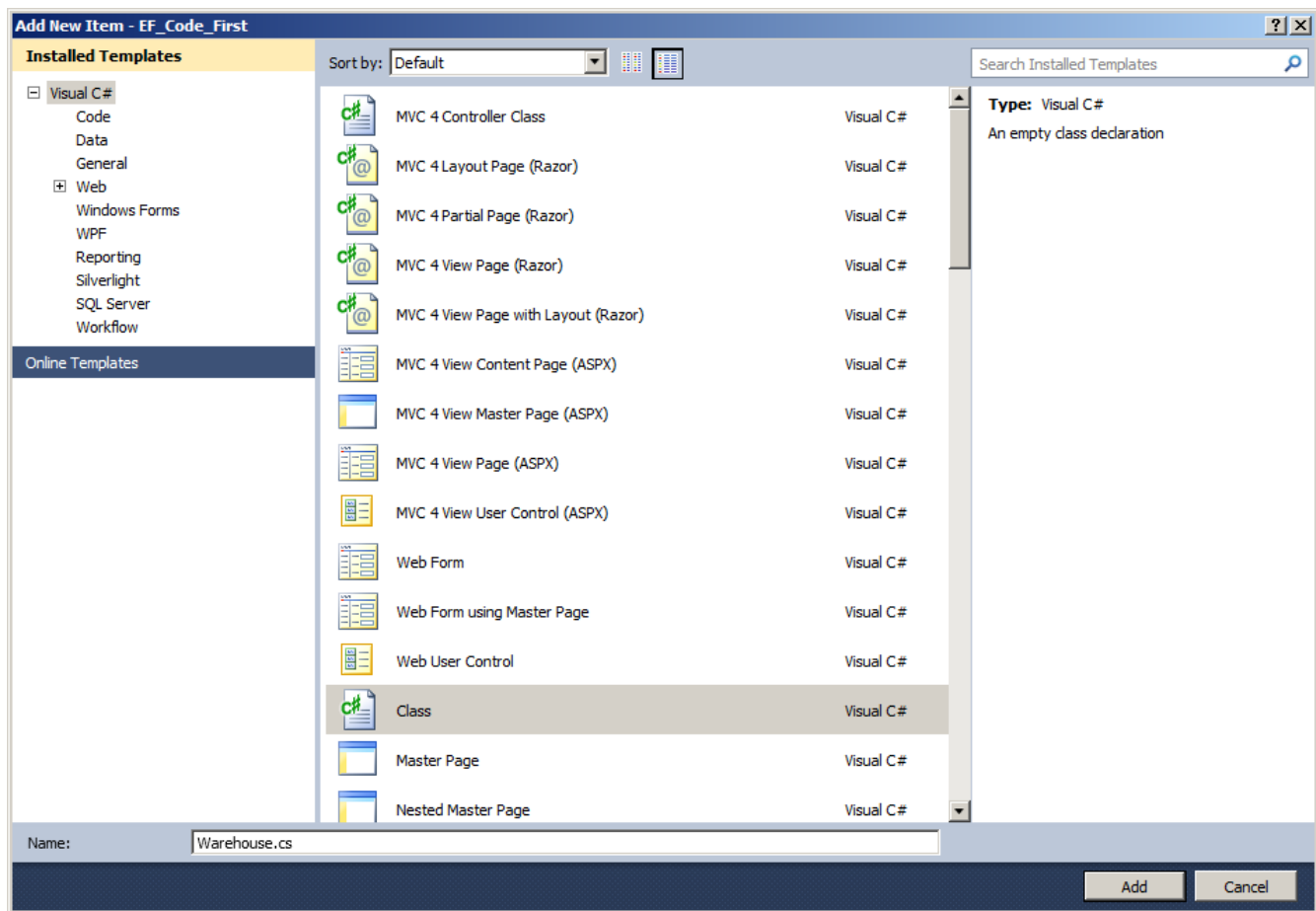
Let's begin! Create a new project named **EF_Code_First**. Set the project type to **ASP.NET MVC 4 Web Application**:



Use the **Internet Application** template and set **Razor** as the View Engine:



Now that we have a clean slate to work on, let's create our first Entity. Right-click the **Models** folder and create a new model named **Warehouse**:



You see the basic code that is created for every model:

OBSERVE: Default generated code for Warehouse.cs

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

namespace EF_Code_First.Models
{
    public class Warehouse
    {
    }
}
```

A warehouse sends and receives shipments, but the shipments are separate, smaller entities of the warehouse that must communicate with the warehouse to be handled properly. Add a new entity named **Shipments**, following the same instructions we used for Warehouse.

You see the same generated code as for the Warehouse entity:

OBSERVE: Default generated code for /Models/Shipments.cs

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

namespace EF_Code_First.Models
{
    public class Shipments
    {
    }
}
```

A shipment has a sender address, receiver address, tracking number (defined to be a package's Primary Key), a sent and received date/time, and a property for the number of items per shipment. Let's add these properties to our **/Models/Shipments** entity. Modify your code as shown:

CODE TO TYPE: /Models/Shipments.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace EF_Code_First.Models
{
    public class Shipments
    {
        [Key]
        public int TrackingNumber { get; set; }
        public int NumberOfItems { get; set; }
        public string SendToAddress { get; set; }
        public string SentFromAddress { get; set; }
        public DateTime ShippedDateTime { get; set; }
        public DateTime ReceivedDateTime { get; set; }
    }
}
```

EF's primary key naming convention is **Id** or **{EntityName}Id** (for Warehouse, the default primary key would be **Id** or **WarehouseId**). We added the DataAnnotation **[Key]** to let the Entity Framework know that we want to use the TrackingNumber as a primary key, because we didn't follow the default naming convention. Data annotations are located in the System.ComponentModel.DataAnnotations namespace.

Now that we have a shipment entity, we need to give the Warehouse entity some properties. Modify your code as shown:

CODE TO TYPE: /Models/Warehouse.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace EF_Code_First.Models
{
    public class Warehouse
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
    }
}
```

We didn't need the **Key** data annotation above `Id`, because EF automatically recognizes this as the primary key.

Currently, the models are not entities because they have no relationship to one another. A warehouse may have many shipments, but a shipment may only go to one warehouse or be shipped from one warehouse. Let's add some relationships and create proper Entities.

Modify **/Models/Shipments.cs** as shown:

CODE TO TYPE: /Models/Shipments.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace EF_Code_First.Models
{
    public class Shipments
    {
        [Key]
        public int TrackingNumber { get; set; }
        public int NumberOfItems { get; set; }
        public string SendToAddress { get; set; }
        public string SentFromAddress { get; set; }
        public DateTime ShippedDateTime { get; set; }
        public DateTime ReceivedDateTime { get; set; }
        public int WarehouseId { get; set; }
    }
}
```

Now modify **/Models/Warehouse.cs** as shown:

CODE TO TYPE: /Models/Warehouse.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace EF_Code_First.Models
{
    public class Warehouse
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
        public ICollection<Shipments> shipments { get; set; }
    }
}
```

In the Shipments entity, we added **WarehouseId**. By adding this property, we tell the Entity Framework that there is a relationship between Warehouse and Shipments. **ICollection<Shipments>** defines a collection of Shipments entities.

Now that we have a couple of entities with a one-to-many relationship, let's see how EF represents them in a database.

Open **/Web.config** and add the code below (your **/Web.config** file may differ):

CODE TO TYPE: /Web.config

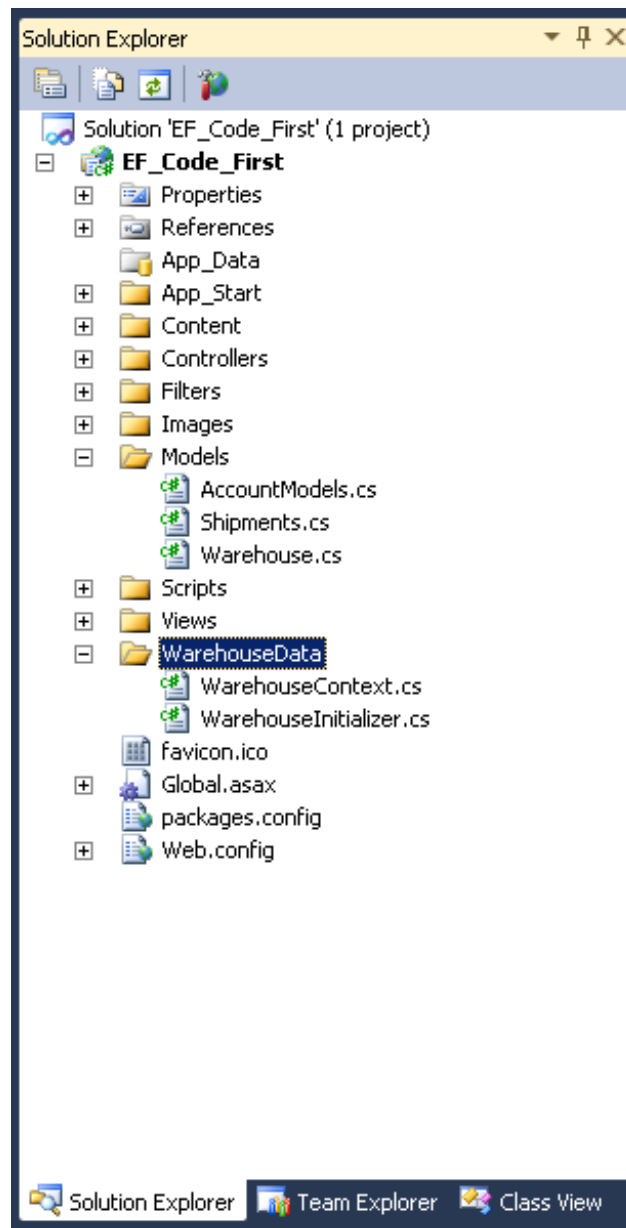
```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=169433
-->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkId=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=4.4.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <connectionStrings>

    <add name="DefaultConnection" connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=aspnet-EF_Code_First-20130614163900;Integrated Security=SSPI" providerName="System.Data.SqlClient" />

    <add name="WarehouseContext"
        connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=Warehouse;AttachDBF=|DataDirectory|\Warehouse.mdf;Integrated Security=SSPI;User Instance=true"
        providerName="System.Data.SqlClient"
    />
  </connectionStrings>
  .
  .
  .
```

Create a new folder named **/WarehouseData** and add two new classes to it: **WarehouseContext** and **WarehouseInitializer**.

Your Solution Explorer looks like this:



Modify **/WarehouseData/WarehouseContext.cs** as shown:

CODE TO TYPE: /WarehouseData/WarehouseContext.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using EF_Code_First.Models;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace EF_Code_First.WarehouseData
{
    public class WarehouseContext : DbContext
    {
        public DbSet<Warehouse> Warehouse { get; set; }
        public DbSet<Shipments> Shipments { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

Let's discuss what we just did.

OBSERVE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using EF_Code_First.Models;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace EF_Code_First.WarehouseData
{
    public class WarehouseContext : DbContext
    {
        public DbSet<Warehouse> Warehouse { get; set; }
        public DbSet<Shipments> Shipments { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

- **System.Data.Entity.ModelConfiguration.Conventions** is the namespace that allows us to modify EF conventions.
- We inherit the **DbContext** class so we can create and use database objects.
- **public DbSet<Warehouse>** creates a table in our database with the name of our entity.
- **OnModelCreating** defines what to do when our models are created. We can perform many operations in this function to properly define our entities if we so choose.
- **modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();** removes the convention to pluralize the entity names when they are created as tables.

Now we need to set up the **/WarehouseData/WarehouseInitializer** so we can create a database and insert our tables. Modify your code as shown:

CODE TO TYPE: /WarehouseData/WarehouseInitializer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using EF_Code_First.Models;

namespace EF_Code_First.WarehouseData
{
    public class WarehouseInitializer : DropCreateDatabaseAlways<WarehouseContext>
    {
        protected override void Seed(WarehouseContext context)
        {
        }
    }
}
```

Let's discuss what we did in the WarehouseInitializer.

CODE TO TYPE: /WarehouseData/WarehouseInitializer.cs

```
namespace EF_Code_First.WarehouseData
{
    public class WarehouseInitializer : DropCreateDatabaseAlways<WarehouseContext>
    {
        protected override void Seed(WarehouseContext context)
        {
        }
    }
}
```

- **DropCreateDatabaseAlways** will always drop and recreate the database, and optionally seed the database the first time a context is used in the application domain.
- **protected override void Seed** is a member function of the namespace System.Data.Entity that seeds the database with default values if we choose to do so.

We need to modify one more file before we can run our application. Modify **/Global.asax.cs** as shown:

CODE TO TYPE: /Global.asax.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using System.Data.Entity;
using EF_Code_First.WarehouseData;

namespace EF_Code_First
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            AuthConfig.RegisterAuth();
            var context = new WarehouseContext();
            context.Database.Initialize(true);
        }
    }
}
```

/Global.asax.cs

```
var context = new WarehouseContext();
context.Database.Initialize(true);
```

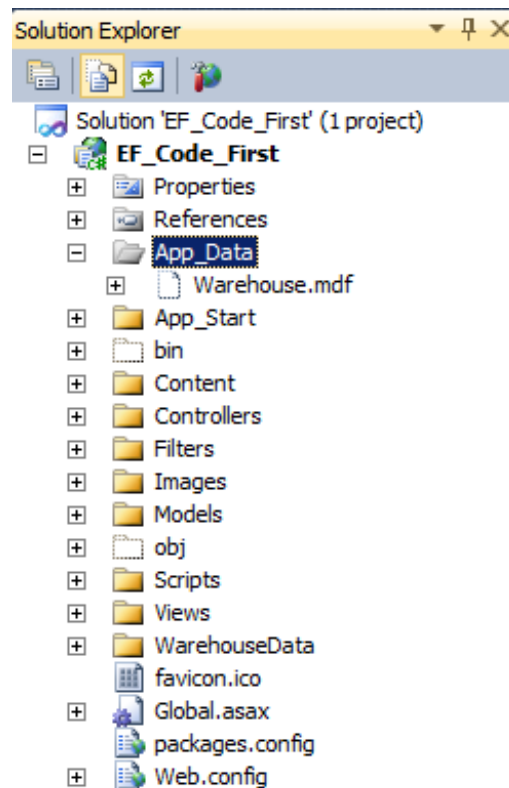
- **var context = new WarehouseContext();** instantiates a new WarehouseContext object.
- **context.Database.Initialize(true);** forces our database to be initialized and inserts our tables.



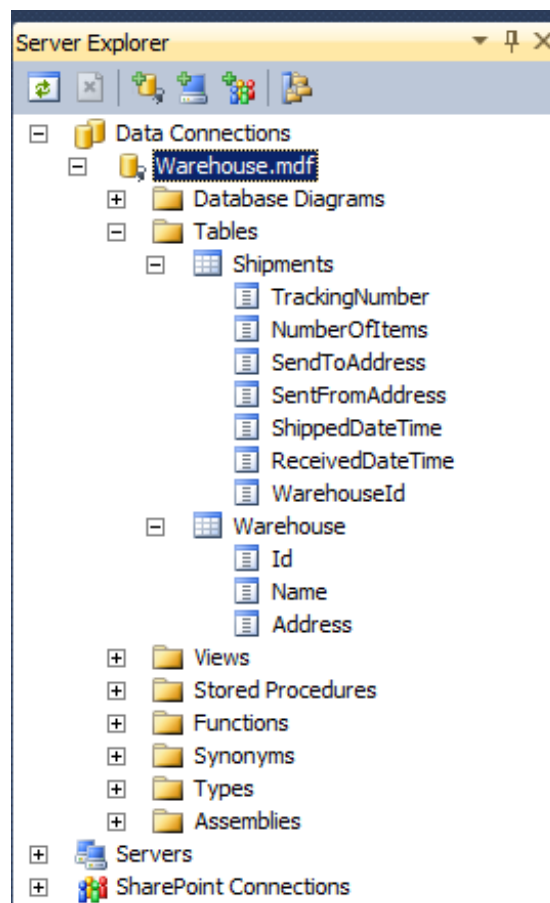
. All of our changes were on the back end of the application, so there is no visible change to the default application.

We've created a database named Warehouse, and inserted tables named Shipments and Warehouse. Click the Show

All Files  button in Solution Explorer and expand the **App_Data** folder, and you'll see our Warehouse.mdf file.



Double-click the **Warehouse.mdf** file; then, in the Server Explorer on the left, select and expand **Warehouse.mdf**. You see all the tables under **Data Connections | Tables**.



A common practice in domain drive design is to create models that are considered Complex. These Complex types are layered to create a complete entity. Our Shipments model contains properties that could be abstracted to a Complex type. Let's remove the **SendToAddress** and **SentFromAddress** properties and place these in a complex type called **Address**.

Create a new class in the **/Models** folder and name it **Address**, then modify it as shown:

CODE TO TYPE: /Models/Address.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations.Schema;

namespace EF_Code_First.Models
{
    [ComplexType]
    public class Address
    {
        public string Country { get; set; }
        public string State { get; set; }
        public string City { get; set; }
        public string Street { get; set; }
        public int Zip { get; set; }
        public string HousingNumber { get; set; }
    }
}
```

We created a new Model named Address that is a complex type, meaning we don't want to add it to any table, and we don't want to create any entities with it. Now we need to add this to our Shipments and Warehouse entities. Modify **/Models/Shipments.cs** as shown:

CODE TO TYPE: /Models/Shipments.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace EF_Code_First.Models
{
    public class Shipments
    {
        public Shipments()
        {
            Sender = new Address();
            Recipient = new Address();
        }

        [Key]
        public int TrackingNumber { get; set; }
        public int NumberOfItems { get; set; }
public string SendToAddress { get; set; }
public string SentFromAddress { get; set; }
        public DateTime ShippedDateTime { get; set; }
        public DateTime ReceivedDateTime { get; set; }
        public Address Sender { get; set; }
        public Address Recipient { get; set; }
        public int WarehouseId { get; set; }
        public virtual Warehouse WarehouseNavigation { get; set; }
    }
}
```

public virtual Warehouse WarehouseNavigation { get; set; } defines a navigation property to our Warehouse entity. We'll explain this concept in more depth in the LINQ to Entities section.

Modify **/Models/Warehouse.cs** again, as shown:

CODE TO TYPE: /Models/Warehouse.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace EF_Code_First.Models
{
    public class Warehouse
    {
        public Warehouse() { WarehouseAddress = new Address(); }
        public int Id { get; set; }
        public string Name { get; set; }
public string Address { get; set; }
        public Address WarehouseAddress { get; set; }
        public ICollection<Shipments> shipments { get; set; }
    }
}
```

Before we can compile and run our application, we need to change our Seed method to include this new information. Modify **/WarehouseData/WarehouseInitializer.cs** as shown.

CODE TO TYPE: WarehouseData/WarehouseInitializer.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using EF_Code_First.Models;

namespace EF_Code_First.WarehouseData
{
    public class WarehouseInitializer : DropCreateDatabaseIfModelChanges<WarehouseContext>
    {
        protected override void Seed(WarehouseContext context)
        {
            var _warehouse = new List<Warehouse>()
            {
                new Warehouse { Name = "Warehouse 1", WarehouseAddress = { Country = "Japan", State = "Province B", City = "Xiangzhou", Zip = 65147, Street = "Warehouse Way", HousingNumber = "1000"}, shipments = new List<Shipments>() },
                new Warehouse { Name = "Warehouse 2", WarehouseAddress = { Country = "United States", State = "LA", City = "Tabasco", Zip = 91542, Street = "Warehouse Way", HousingNumber = "2000"}, shipments = new List<Shipments>() },
                new Warehouse { Name = "Warehouse 3", WarehouseAddress = { Country = "Brazil", State = "Small", City = "Beach Town", Zip = 89245, Street = "Warehouse Way", HousingNumber = "3000"}, shipments = new List<Shipments>() }
            };
            _warehouse.ForEach(x => context.Warehouse.Add(x));
            context.SaveChanges();

            var _shipments = new List<Shipments>()
            {
                new Shipments { NumberOfItems = 5, Recipient = {Country = "United States", State = "NV", City = "Tahoe", Zip = 89712, Street = "Main St", HousingNumber = "588 Ste. A"}, Sender = {Country = "United States", State = "MN", City = "Lakeview", Zip = 56487, Street = "Kilimanjaro Ln", HousingNumber = "6589"}, ShippedDateTime = new System.DateTime(2001, 12, 28), ReceivedDateTime = new System.DateTime(2001, 12, 22), WarehouseId = _warehouse[0].Id },
                new Shipments { NumberOfItems = 1, Recipient = {Country = "United States", State = "CA", City = "San Diego", Zip = 92120, Street = "Testing Circle", HousingNumber = "415 Apt. J"}, Sender = {Country = "United States", State = "SD", City = "Fargo", Zip = 65874, Street = "Cyber Way", HousingNumber = "241 Ste J"}, ShippedDateTime = new System.DateTime(2004, 1, 13), ReceivedDateTime = new System.DateTime(2004, 1, 15), WarehouseId = _warehouse[0].Id },
                new Shipments { NumberOfItems = 10, Recipient = {Country = "United States", State = "NY", City = "New York", Zip = 87871, Street = "Empire Ln", HousingNumber = "2145"}, Sender = {Country = "United States", State = "ND", City = "Northern", Zip = 62513, Street = "Nougat Ct", HousingNumber = "6589 A"}, ShippedDateTime = new DateTime(2011, 2, 13), ReceivedDateTime = new DateTime(2011, 2, 23), WarehouseId = _warehouse[0].Id },
                new Shipments { NumberOfItems = 15, Recipient = {Country = "United States", State = "MO", City = "Bozeman", Zip = 89201, Street = "Chicken Way", HousingNumber = "128"}, Sender = {Country = "United States", State = "SD", City = "Slates", Zip = 31524, Street = "Leik Circle", HousingNumber = "124"}, ShippedDateTime = new DateTime(2011, 2, 13), ReceivedDateTime = new DateTime(2011, 2, 23), WarehouseId = _warehouse[1].Id },
                new Shipments { NumberOfItems = 3, Recipient = {Country = "United States", State = "OR", City = "Salem", Zip = 20152, Street = "Milky Way Ln", HousingNumber = "6542"}, Sender = {Country = "United States", State = "CA", City = "Sharp", Zip = 98457, Street = "Nimdooy Ct", HousingNumber = "6578"}, ShippedDateTime = new DateTime(2011, 2, 13), ReceivedDateTime = new DateTime(2011, 2, 23), WarehouseId = _warehouse[1].Id },
                new Shipments { NumberOfItems = 8, Recipient = {Country = "United States", State = "KY", City = "Campbellsville", Zip = 42718, Street = "Williams St", HousingNumber = "4233"}, Sender = {Country = "United States", State = "TX", City = "Mackay", Zip = 31542, Street = "Walkens Way", HousingNumber = "5487"}, ShippedDateTime = new DateTime(2012, 5, 1), ReceivedDateTime = new DateTime(2012, 5, 21), WarehouseId = _warehouse[1].Id },
            };
            context.Shipments.AddRange(_shipments);
            context.SaveChanges();
        }
    }
}

```

```

        new Shipments { NumberOfItems = 45, Recipient = {Country = "United States", State = "TX", City = "Honeybun", Zip = 43770, Street = "Milk and Cookies Way", HousingNumber = "5487"}, Sender = {Country = "United States", State = "ID", City = "Manchester", Zip = 59014, Street = "Quahog St", HousingNumber = "6958"}, ShippedDateTime = new DateTime(2001, 8, 5), ReceivedDateTime = new DateTime(2012, 3, 12), WarehouseId = _warehouse[2].Id },

        new Shipments { NumberOfItems = 24, Recipient = {Country = "United States", State = "MA", City = "Middleton", Zip = 65879, Street = "One Way E. Ln", HousingNumber = "215"}, Sender = {Country = "United States", State = "UT", City = "Milk", Zip = 20154, Street = "Juke Way", HousingNumber = "3654 Bldg B"}, ShippedDateTime = new DateTime(2009, 8, 5), ReceivedDateTime = new DateTime(2009, 9, 1), WarehouseId = _warehouse[2].Id },

        new Shipments { NumberOfItems = 12, Recipient = {Country = "United States", State = "MN", City = "Minnetonka", Zip = 31254, Street = "Informational Way", HousingNumber = "2121"}, Sender = {Country = "United States", State = "AZ", City = "Snickers", Zip = 35126, Street = "Kreet Ln", HousingNumber = "5487 Ste A"}, ShippedDateTime = new DateTime(1995, 4, 12), ReceivedDateTime = new DateTime(1995, 4, 14), WarehouseId = _warehouse[2].Id }
    };
    _shipments.ForEach(x => context.Shipments.Add(x));
    context.SaveChanges();

    _warehouse[0].shipments.Add(_shipments[0]);
    _warehouse[0].shipments.Add(_shipments[1]);
    _warehouse[0].shipments.Add(_shipments[2]);
    _warehouse[1].shipments.Add(_shipments[3]);
    _warehouse[1].shipments.Add(_shipments[4]);
    _warehouse[1].shipments.Add(_shipments[5]);
    _warehouse[2].shipments.Add(_shipments[6]);
    _warehouse[2].shipments.Add(_shipments[7]);
    _warehouse[2].shipments.Add(_shipments[8]);
    context.SaveChanges();
}
}
}

```

Now that our Entities are properly designed, we need to change our **/Global.asax** file to seed our database using our WarehouseInitializer. Modify the code as shown:

CODE TO TYPE: /Global.asax

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using System.Data.Entity;
using EF_Code_First.WarehouseData;

namespace EF_Code_First
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            AuthConfig.RegisterAuth();
var context = new WarehouseContext();
context.Database.Initialize(true);
            Database.SetInitializer<WarehouseContext>(new WarehouseInitializer());
        }
    }
}
```

Now, let's create a Controller for our Shipments Model. Right-click the **Controllers** folder and select **Add | Controller....** Name it **ShipmentsController**. Set the template to **Empty MVC controller**:

The screenshot shows the 'Add Controller' dialog box with the following configuration:

- Controller name: ShipmentsController
- Scaffolding options:
 - Template: Empty MVC controller
 - Model class: (empty)
 - Data context class: Address (EF_Code_First.Models)
 - Views: None
- Buttons: Add, Cancel, and Advanced Options...

Modify the controller as shown:

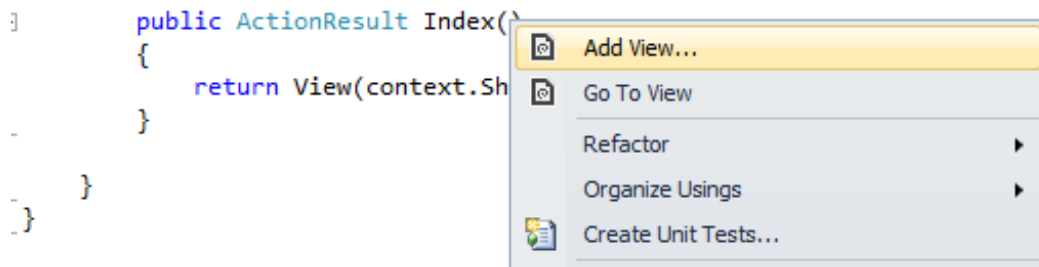
CODE TO TYPE: /Controllers/ShipmentsController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using EF_Code_First.Models;
using EF_Code_First.WarehouseData;

namespace EF_Code_First.Controllers
{
    public class ShipmentsController : Controller
    {
        WarehouseContext context = new WarehouseContext();
        //
        // GET: /Shipments/

        public ActionResult Index()
        {
            return View(context.Shipments.ToList());
        }
    }
}
```

We need to add a View so we can see what's being listed. Right-click the Index method and choose **Add View**:



Set the Scaffold Template to **List** and set the Model class to **Shipments (EF_Code_First.Models)** as shown:

Add View [X]

View name:

View engine:

☒ Create a strongly-typed view

Model class:

Scaffold template:
 ☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:
 ...
(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

Entity Framework creates a basic layout for us in **/Views/Shipments/Index.cshtml**:

/Views/Shipments/Index.cshtml View file

```
@model IEnumerable<EF_Code_First.Models.Shipments>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.NumberOfItems)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ShippedDateTime)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ReceivedDateTime)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.WarehouseId)
        </th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.NumberOfItems)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ShippedDateTime)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReceivedDateTime)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.WarehouseId)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.TrackingNumber }) |
                @Html.ActionLink("Details", "Details", new { id=item.TrackingNumber }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.TrackingNumber })
            </td>
        </tr>
    }
</table>
```



and



and navigate to the Shipments Index by adding **/Shipments** to the URL in the address bar:

Index

[Create New](#)

NumberOfItemsShipped	DateTime	ReceivedDateTime	WarehouseId	
5	12/28/2001 12:00:00 AM	12/22/2001 12:00:00 AM	1	Edit Details Delete
1	1/13/2004 12:00:00 AM	1/15/2004 12:00:00 AM	1	Edit Details Delete
10	2/13/2011 12:00:00 AM	2/23/2011 12:00:00 AM	1	Edit Details Delete
15	2/13/2011 12:00:00 AM	2/23/2011 12:00:00 AM	2	Edit Details Delete
3	2/13/2011 12:00:00 AM	2/23/2011 12:00:00 AM	2	Edit Details Delete
8	5/1/2012 12:00:00 AM	5/21/2012 12:00:00 AM	2	Edit Details Delete
45	8/5/2001 12:00:00 AM	3/12/2012 12:00:00 AM	3	Edit Details Delete
24	8/5/2009 12:00:00 AM	9/1/2009 12:00:00 AM	3	Edit Details Delete
12	4/12/1995 12:00:00 AM	4/14/1995 12:00:00 AM	3	Edit Details Delete

Note

If you encounter an error message like "Cannot create file because it already exists," click the **Show All Files** icon at the top of the Solution Explorer, expand the **App_Data** folder, expand the **Warehouse.mdf** file, delete the files with the **.ldf** extension, and try again. These are log files used for tracking what happens in the SQL database. In a production environment, you would perform a complete backup of your data and log files before removing these files, but in this case, because we're creating and seeding a new database that won't be used in a production context, it's okay to delete them.

Notice there are no addresses listed. Let's fix that. Modify **/Views/Shipments/Index.cshtml** as shown:

CODE TO TYPE: Adding addresses to the index view of Shipments

```
@model IEnumerable<EF_Code_First.Models.Shipments>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>


<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.NumberOfItems)
        </th>
        <th>
            Sender Address
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ShippedDateTime)
        </th>
        <th>
            Recipient Address
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ReceivedDateTime)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.WarehouseId)
        </th>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.NumberOfItems)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Sender.Country)<br />
                @Html.DisplayFor(modelItem => item.Sender.HousingNumber) @Html.DisplayFor(modelItem => item.Sender.Street), @Html.DisplayFor(modelItem => item.Sender.State)<br />
                @Html.DisplayFor(modelItem => item.Sender.Zip)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ShippedDateTime)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Recipient.Country)<br />
                @Html.DisplayFor(modelItem => item.Recipient.HousingNumber) @Html.DisplayFor(modelItem => item.Recipient.Street), @Html.DisplayFor(modelItem => item.Recipient.State)<br />
                @Html.DisplayFor(modelItem => item.Recipient.Zip)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReceivedDateTime)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.WarehouseId)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.TrackingNumber }) |
                @Html.ActionLink("Details", "Details", new { id=item.TrackingNumber }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.TrackingNumber })
            </td>
        </tr>
    }
```

```

    </tr>
  }
</table>

```



and  and navigate to the Index of Shipments again. It now displays all the information we want.

Index - My ASP.NET MVC Application - Windows Internet Explorer

http://localhost:62418/Shipments

your logo here

Register Log in

Home About Contact

Index

[Create New](#)

NumberOfItems	Sender Address	ShippedDateTime	Recipient Address	ReceivedDateTime	WarehouseId	
5	United States 6589 Kilimanjaro Ln, MN 56487	12/28/2001 12:00:00 AM	United States 588 Ste. A Main St, NV 89712	12/22/2001 12:00:00 AM	1	Edit Details Delete
1	United States 241 Ste J Cyber Way, SD 65874	1/13/2004 12:00:00 AM	United States 415 Apt. J Testing Circle, CA 92120	1/15/2004 12:00:00 AM	1	Edit Details Delete
10	United States 6589 A Nouget Ct, ND 62513	2/13/2011 12:00:00 AM	United States 2145 Empire Ln, NY 87871	2/23/2011 12:00:00 AM	1	Edit Details Delete
15	United States 124 Leik Circle, SD 31524	2/13/2011 12:00:00 AM	United States 128 Chicken Way, MO 89201	2/23/2011 12:00:00 AM	2	Edit Details Delete
3	United States 6578 Nimdoy Ct, CA 98457	2/13/2011 12:00:00 AM	United States 6542 Milky Way Ln, OR 20152	2/23/2011 12:00:00 AM	2	Edit Details Delete
8	United States 5487 Walkens Way, TX 31542	5/1/2012 12:00:00 AM	United States 4233 Williams St, KY 42718	5/21/2012 12:00:00 AM	2	Edit Details Delete
45	United States 6958 Quahog St, ID 59014	8/5/2001 12:00:00 AM	United States 5487 Milk and Cookies Way, TX 43770	3/12/2012 12:00:00 AM	3	Edit Details Delete
	United States		United States			

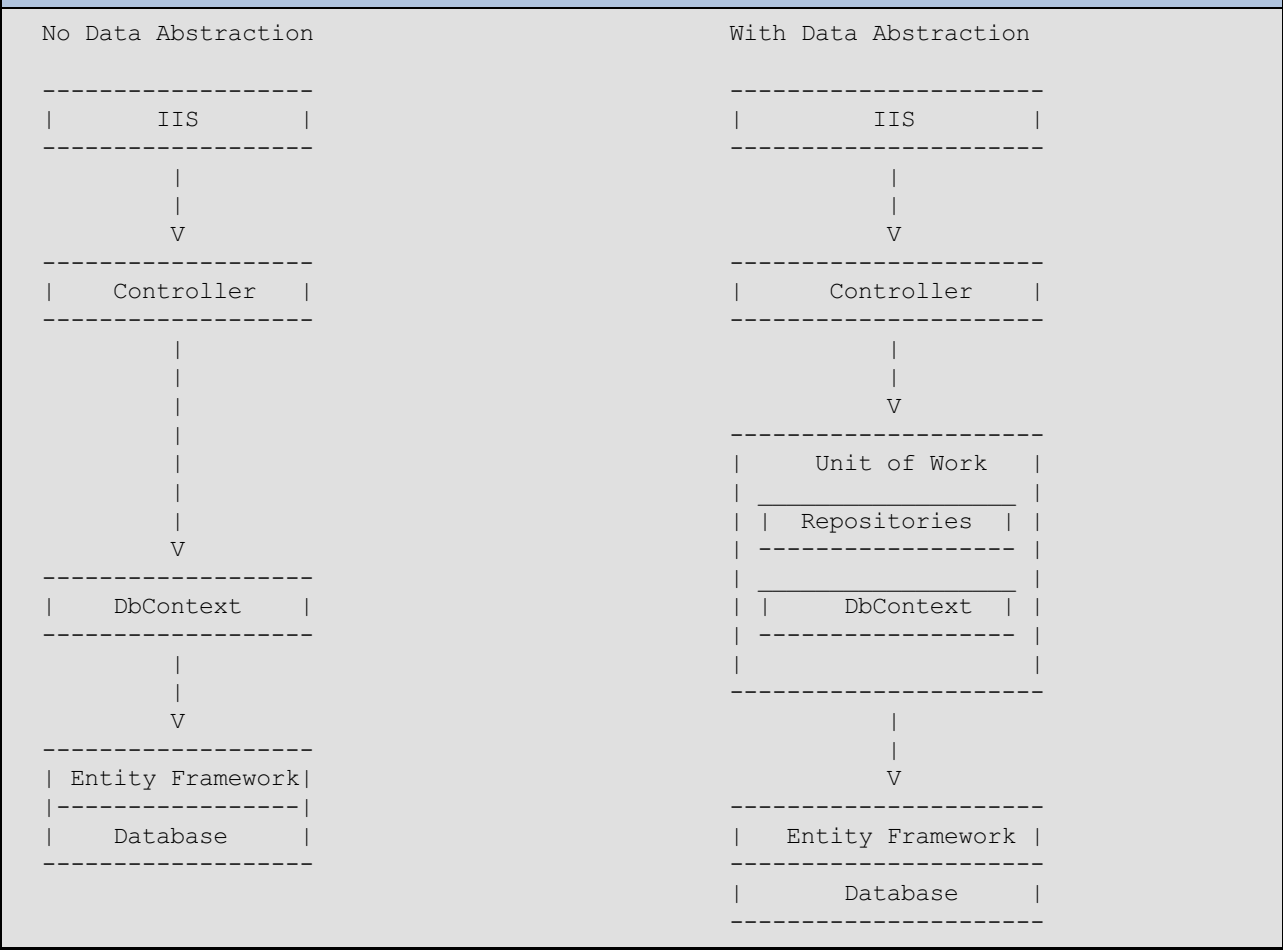
We'll cover the Database First approach in the Entity Data Model (EDM) section.

Data Abstraction

In this section, we'll cover Data Abstraction in terms of the Repository and Unit of Work patterns. These patterns create an abstraction layer between the data access and business logic layer of our applications.

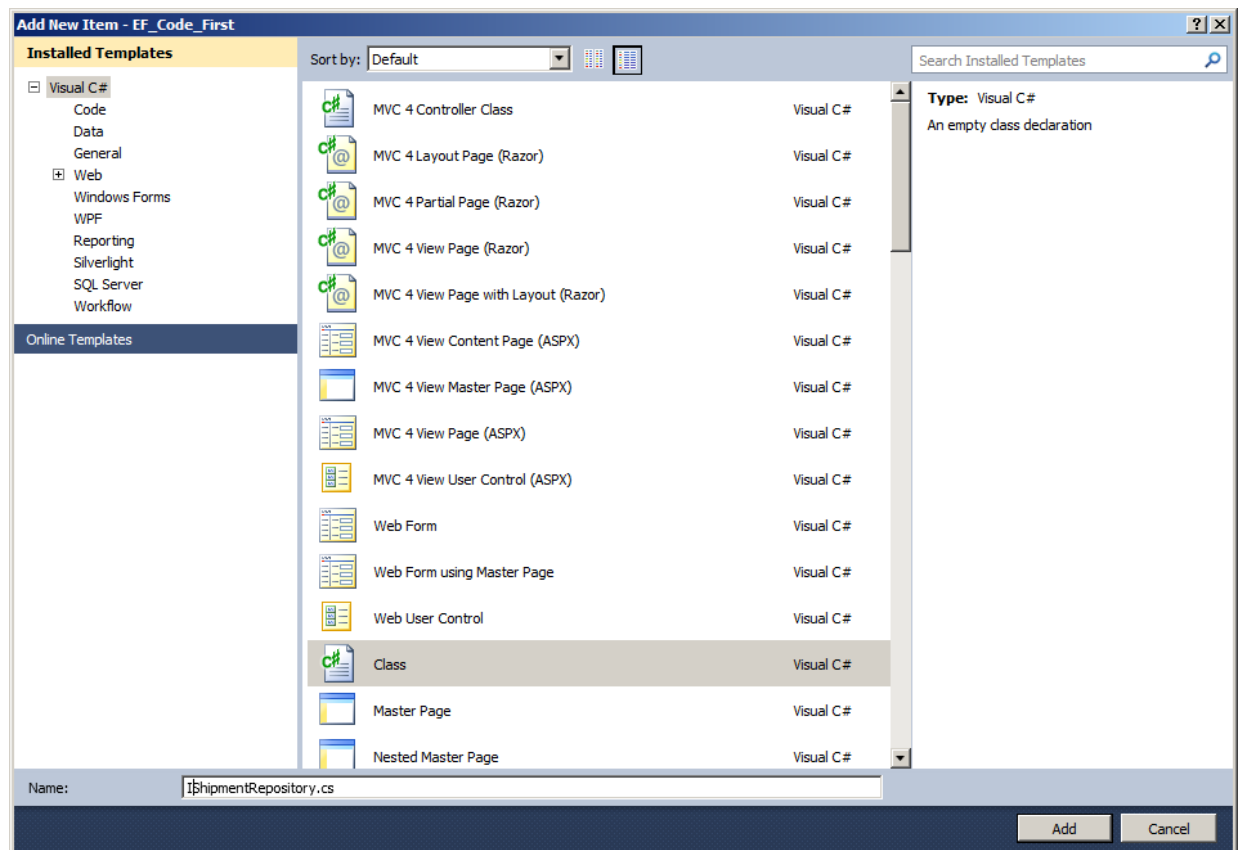
The method we've been using to create relationships has been a straightforward method that goes from IIS to the controller, and directly from the controller to the DbContext, and then to the Entity Framework and database. The Unit of Work and Repository patterns add another level of abstraction between the controller and Entity Framework and database phase.

OBSERVE: Difference between no abstraction and Unit of Work and Repository



Repository

Let's put Data Abstraction to work. Create a folder named **AbstractionLayer**. In this folder, add a class named **IShipmentRepository**:



Modify this new file as shown:

CODE TO TYPE: IShipmentRepository.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using EF_Code_First.Models;

namespace EF_Code_First.AbstractionLayer
{
    public class interface IShipmentRepository : IDisposable
    {
        IEnumerable<Shipments> GetAllShipments();
        Shipments GetByTrackingNumber(int trackingNumber);
        void CreateNewShipment(Shipments shipment);
        void RemoveShipment(int trackingNumber);
        void ModifyShipment(Shipments shipment);
        void saveShipment();
    }
}
```

Note **interface** will be discussed in a future lesson. In case you want a preview of coming attractions, here is an [interface reference](#).

Let's discuss this class declaration:

OBSERVE:

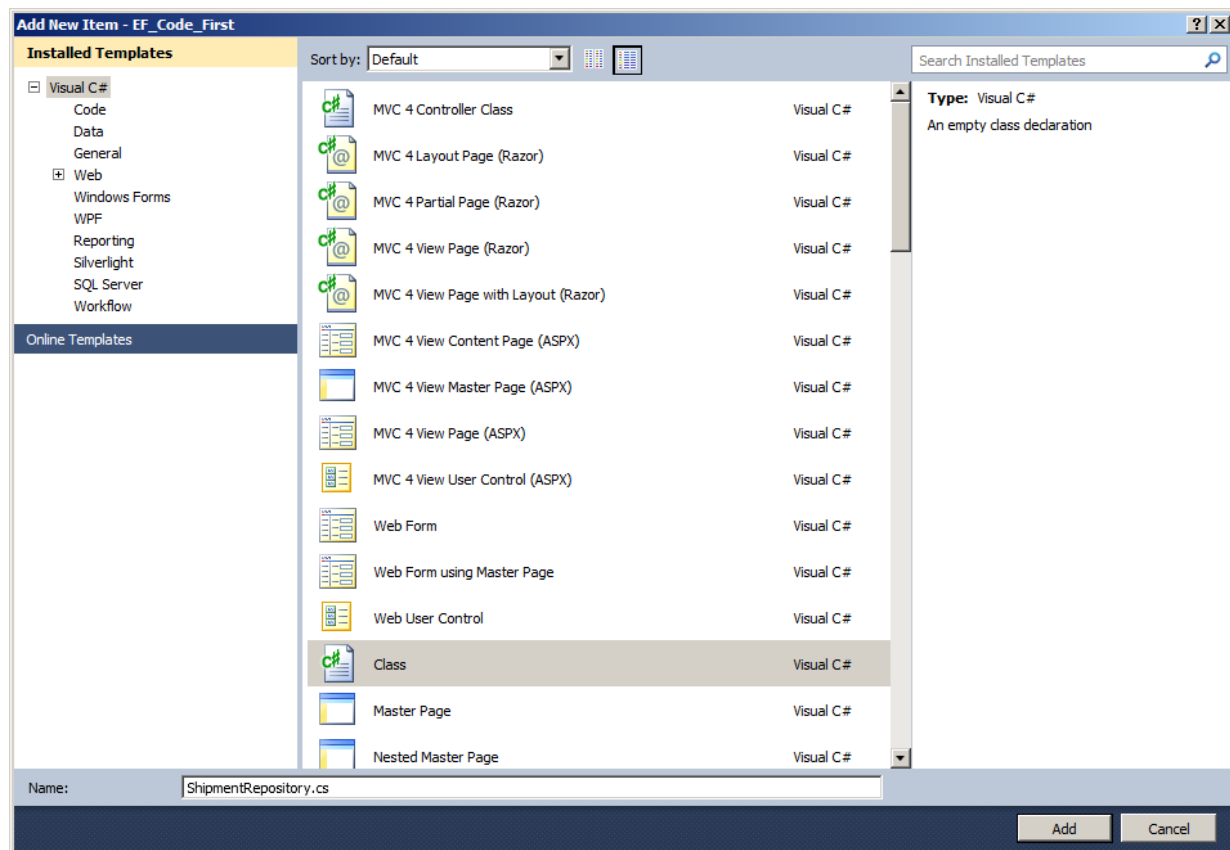
```
public interface IShipmentRepository : IDisposable
```

IShipmentRepository inherits the **IDisposable** interface as a means to release allocated services. In our case, IShipmentRepository will be the DbContext. We've set up our interface for the next lesson to use the

Entity Frameworks CRUD operations.

Tip CRUD is an acronym for Create, Read, Update, Delete.

To implement our newly created repository, add a new class to the **AbstractionLayer** folder named **ShipmentRepository**.



Modify the **ShipmentRepository.cs** class so we can begin to use the interface we just created:

CODE TO TYPE: ShipmentRepository.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data;
using EF_Code_First.Models;
using EF_Code_First.WarehouseData;

namespace EF_Code_First.AbstractionLayer
{
    public class ShipmentRepository : IShipmentRepository, IDisposable
    {
        private WarehouseContext context;

        public ShipmentRepository(WarehouseContext context)
        {
            this.context = context;
        }

        public IEnumerable<Shipments> GetAllShipments()
        {
            return context.Shipments.ToList();
        }

        public Shipments GetByTrackingNumber(int trackingNumber)
        {
            return context.Shipments.Find(trackingNumber);
        }

        public void CreateNewShipment(Shipments shipment)
        {
            context.Shipments.Add(shipment);
        }

        public void RemoveShipment(int trackingNumber)
        {
            Shipments shipment = context.Shipments.Find(trackingNumber);
            context.Shipments.Remove(shipment);
        }

        public void ModifyShipment(Shipments shipment)
        {
            context.Entry(shipment).State = EntityState.Modified;
        }

        public void saveShipment()
        {
            context.SaveChanges();
        }

        private bool ContextDisposed = false;

        protected virtual void Dispose(bool disposal)
        {
            if (!this.ContextDisposed)
            {
                if (disposal)
                {
                    context.Dispose();
                }
            }
            this.ContextDisposed = true;
        }

        public void Dispose()
        {
        }
    }
}
```



```

        Dispose(true);
        GC.SuppressFinalize(this);
    }
}

```

We've added quite a bit of code, and some of it may be a little confusing. Let's discuss what we've done with the repository.

OBSERVE:

```

.
.
.
    private WarehouseContext context;

    public ShipmentRepository(WarehouseContext context)
.
.
.
    private bool ContextDisposed = false;

    protected virtual void Dispose(bool disposal)
    {
        if (!this.ContextDisposed)
        {
            if (disposal)
            {
                context.Dispose();
            }
        }
        this.ContextDisposed = true;
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
.
.
.

```

- **private WarehouseContext context** defines a local variable to hold our database context.
- **public ShipmentRepository(WarehouseContext context)** defines a Constructor for our repository that expects a WarehouseContext object to be passed in when the repository is instantiated.
- **private bool ContextDisposed = false** declares a local boolean variable to use as a flag that indicates whether our context has been disposed.
- The **public void Dispose()** function accepts no arguments, but makes a self-referential call to an overloaded function **protected virtual void Dispose(bool disposal)**. (This usage will become clearer when we create our Controller.)
- **GC.SuppressFinalize(this);** tells our Garbage Collector(GC) that our object was cleaned up properly and that it does not need to go into the finalizer queue.

Tip

SuppressFinalize should only be called on classes that have finalizers. Finalizers mimic C++ destructors, but perform completely different operations.



and Build the project to make sure we have no errors.

Unit of Work

Now, we'll create our Unit of Work class. Before we can add the items that are necessary for this section, we

need to create a new Entity and complex model.

Add a new model to the **/Models** folder named **Person**, and modify it as shown:

CODE TO TYPE: /Models/Person.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations.Schema;

namespace EF_Code_First.Models
{
    [ComplexType]
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string MiddleName { get; set; }
        public string SocialSecurity { get; set; }
        public string FullName
        {
            get
            {
                return FirstName + " " + MiddleName[0] + ". " + LastName;
            }
        }
    }
}
```

Right-click the **/Models** folder and add a new **Employee** class. Add some properties for our Employee entity as shown:

CODE TO TYPE: /Models/Employee.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace EF_Code_First.Models
{
    public class Employee
    {
        public Employee() { person = new Person(); }

        [Key]
        [Required]
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int EmployeeId { get; set; }
        public string Position { get; set; }
        public Person person { get; set; }
        public int WarehouseId { get; set; }
        public virtual Warehouse EmployeeLocation { get; set; }
    }
}
```

OBSERVE:

```
[Key]
[Required]
[DatabaseGenerated(DatabaseGeneratedOption.None)]
```

In the code above, **[DatabaseGenerated(DatabaseGeneratedOption.None)]** tells EF we want to define a key attribute manually for this entity and that the database should not do it for us.

Add a list of Employees to the Warehouse Entity. Modify **/Models/Warehouse.cs** as shown:

CODE TO TYPE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace EF_Code_First.Models
{
    public class Warehouse
    {
        public Warehouse() { WarehouseAddress = new Address(); }

        public int Id { get; set; }
        public string Name { get; set; }
        public Address WarehouseAddress { get; set; }
        public ICollection<Shipments> shipments { get; set; }
        public ICollection<Employee> employees { get; set; }
    }
}
```

Add the Employee Entity to our **/WarehouseData/WarehouseContext.cs** file so we can create a table in our database:

CODE TO TYPE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using EF_Code_First.Models;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace EF_Code_First.WarehouseData
{
    public class WarehouseContext : DbContext
    {
        public DbSet<Warehouse> Warehouse { get; set; }
        public DbSet<Shipments> Shipments { get; set; }
        public DbSet<Employee> Employees { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

Add some initializer information for our newly created Employee entity in **WarehouseInitializer.cs**:

CODE TO TYPE: Seeding Employee Entity in WarehouseInitializer.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using EF_Code_First.Models;

namespace EF_Code_First.WarehouseData
{
    public class WarehouseInitializer : DropCreateDatabaseAlways<WarehouseContext>
    {
        protected override void Seed(WarehouseContext context)
        {
            var _warehouse = new List<Warehouse>()
            {
                new Warehouse { Name = "Warehouse 1", WarehouseAddress = { Country = "Japan", State = "Province B", City = "Xiangzhou", Zip = 65147, Street = "Warehouse Way", HousingNumber = "1000"}, shipments = new List<Shipments>(), employees = new List<Employee>() },
                new Warehouse { Name = "Warehouse 2", WarehouseAddress = { Country = "United States", State = "LA", City = "Tabasco", Zip = 91542, Street = "Warehouse Way", HousingNumber = "2000"}, shipments = new List<Shipments>(), employees = new List<Employee>() },
                new Warehouse { Name = "Warehouse 3", WarehouseAddress = { Country = "Brazil", State = "Small", City = "Beach Town", Zip = 89245, Street = "Warehouse Way", HousingNumber = "3000"}, shipments = new List<Shipments>(), employees = new List<Employee>() }
            };
            _warehouse.ForEach(x => context.Warehouse.Add(x));
            context.SaveChanges();

            .
            .
            .

            var _employees = new List<Employee>()
            {
                new Employee { EmployeeId = 15789, Position = "Administrator", person = { FirstName = "Robert", LastName = "Mikolnag", MiddleName = "Indo", SocialSecurity = "012-034-0678" }, WarehouseId = _warehouse[0].Id },
                new Employee { EmployeeId = 18754, Position = "Shipping Tagger", person = { FirstName = "Melissa", LastName = "Lockie", MiddleName = "Emerald", SocialSecurity = "012-067-0578" }, WarehouseId = _warehouse[0].Id },
                new Employee { EmployeeId = 12458, Position = "Receiving Coordinator", person = { FirstName = "Shawndra", LastName = "Namind", MiddleName = "Ko", SocialSecurity = "012-015-0789" }, WarehouseId = _warehouse[0].Id },
                new Employee { EmployeeId = 22541, Position = "Administrator", person = { FirstName = "Michael", LastName = "Hilter", MiddleName = "May", SocialSecurity = "012-023-0875" }, WarehouseId = _warehouse[1].Id },
                new Employee { EmployeeId = 29875, Position = "Floor Manager", person = { FirstName = "Miranda", LastName = "Miltred", MiddleName = "Liner", SocialSecurity = "012-098-0632" }, WarehouseId = _warehouse[1].Id },
                new Employee { EmployeeId = 27898, Position = "Forklift Operator", person = { FirstName = "Joe", LastName = "Banderfield", MiddleName = "Barry", SocialSecurity = "012-032-0808" }, WarehouseId = _warehouse[1].Id },
                new Employee { EmployeeId = 32565, Position = "Janitor", person = { FirstName = "Bob", LastName = "Kaler", MiddleName = "Kalifa", SocialSecurity = "012-032-0505" }, WarehouseId = _warehouse[2].Id },
                new Employee { EmployeeId = 30054, Position = "Logistics Administrator", person = { FirstName = "Misty", LastName = "Contreal", MiddleName = "Nancy", SocialSecurity = "012-075-3030" }, WarehouseId = _warehouse[2].Id },
                new Employee { EmployeeId = 30821, Position = "Web Developer", person = { FirstName = "Carrie", LastName = "Aderly", MiddleName = "Kimto", SocialSecurity = "012-054-7077" }, WarehouseId = _warehouse[2].Id }
            };
            _employees.ForEach(x => context.Employees.Add(x));
        }
    }
}

```

```

        context.SaveChanges();

        .
        .
        .

        _warehouse[0].employees.Add(_employees[0]);
        _warehouse[0].employees.Add(_employees[1]);
        _warehouse[0].employees.Add(_employees[2]);
        _warehouse[1].employees.Add(_employees[3]);
        _warehouse[1].employees.Add(_employees[4]);
        _warehouse[1].employees.Add(_employees[5]);
        _warehouse[2].employees.Add(_employees[6]);
        _warehouse[2].employees.Add(_employees[7]);
        _warehouse[2].employees.Add(_employees[8]);
        context.SaveChanges();
    }
}

```



and Build the application, but don't run it yet.

Now that we've established our Entity Framework, let's begin building our Unit of Work model. We'll start by building a Generic Repository. It will be much like the Shipment Repository, but far less dependent on a type.

In the **/AbstractionLayer** folder, create a new class named **GenericRepo.cs**. Add some functionality to our newly created repository as shown:

CODE TO TYPE: /AbstractionLayer/GenericRepo.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data;
using System.Data.Entity;
using System.Linq.Expressions;
using EF_Code_First.Models;
using EF_Code_First.WarehouseData;

namespace EF_Code_First.AbstractionLayer
{
    public class GenericRepo<EntityType> where EntityType : class
    {
        internal WarehouseContext context;
        internal DbSet<EntityType> DatabaseSet;

        public GenericRepo(WarehouseContext context)
        {
            this.context = context;
            this.DatabaseSet = context.Set<EntityType>();
        }

        public virtual IEnumerable<EntityType> Retrieve(
            Expression<Func<EntityType, bool>> filter = null,
            Func<IQueryable<EntityType>, IOrderedQueryable<EntityType>> orderBy
= null,
            string includeProperties = "")
        {
            IQueryable<EntityType> DbQuery = DatabaseSet;

            if (filter != null)
            {
                DbQuery = DbQuery.Where(filter);
            }

            foreach (var incProp in includeProperties.Split(new char[] { ',' },
StringSplitOptions.RemoveEmptyEntries))
            {
                DbQuery = DbQuery.Include(incProp);
            }

            if (orderBy != null)
            {
                return orderBy(DbQuery).ToList();
            }
            else
            {
                return DbQuery.ToList();
            }
        }

        public virtual EntityType GetEntitiesById(object id)
        {
            return DatabaseSet.Find(id);
        }

        public virtual void AddEntity(EntityType entityName)
        {
            DatabaseSet.Add(entityName);
        }

        public virtual void RemoveEntityEntry(object id)
        {
            EntityType EntityToRemove = DatabaseSet.Find(id);
            RemoveEntityEntry(EntityToRemove);
        }
    }
}
```

```

        public virtual void RemoveEntityEntry(EntityType entity)
        {
            if (context.Entry(entity).State == EntityState.Detached)
            {
                DatabaseSet.Attach(entity);
            }
            DatabaseSet.Remove(entity);
        }

        public virtual void UpdateEntity(EntityType entity)
        {
            DatabaseSet.Attach(entity);
            context.Entry(entity).State = EntityState.Modified;
        }
    }
}

```

Let's discuss what we've done in our new repository.

OBSERVE:

```

    internal DbSet<EntityType> DatabaseSet;

    public GenericRepo(WarehouseContext context)
    {
        this.context = context;
        this.DatabaseSet = context.Set<EntityType>();
    }

    public virtual IEnumerable<EntityType> Retrieve(
        Expression<Func<EntityType, bool>> filter = null,
        Func<IQueryable<EntityType>, IOrderedQueryable<EntityType>> orderBy
= null,
        string includeProperties = "")
    {
        .
        .
        .

        public virtual EntityType GetEntitiesById(object id)
    }

```

- **internal DbSet<EntityType> DatabaseSet** creates a class variable for the entity set for which the repository is instantiated.
- The **Constructor** accepts our warehouse context object and initializes our DatabaseSet variable.
- The **Retrieve** method uses Lambda Expressions to allow for an optional method to filter the results or order them by an entity property. It receives a comma-delimited set of strings that represent navigation properties. This method provides an eager-loading expression.
- The **GetEntitiesById(object id)** method returns a single item that is retrieved using a property of an entity.
- The rest of the functions perform basic operations such as Insert, Delete, and Update.

So, why would we want a generic repository when we could simply create a repository for every entity we create? By creating a generic repository that is extensible and independent of types, we reduce the amount of code redundancy, and create a centralized area where we can make modifications to the way we interact with our information.

Now we can create a Unit of Work class to ensure that our Entities share the same context. In this way, the Entities will not get out of sync and will update as we expect them to, without our having to update each entity manually.

In the **/AbstractionLayer** folder, create a new class named **WarehouseWorkUnit.cs**, and modify it as shown:

CODE TO TYPE: /AbstractionLayer/WarehouseWorkUnit.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using EF_Code_First.Models;
using EF_Code_First.WarehouseData;

namespace EF_Code_First.AbstractionLayer
{
    public class WarehouseWorkUnit : IDisposable
    {
        private WarehouseContext context = new WarehouseContext();
        private GenericRepo<Warehouse> _WarehouseRepository_;
        private GenericRepo<Employee> _EmployeeRepository_;

        public GenericRepo<Warehouse> WarehouseRepository
        {
            get
            {
                if (this._WarehouseRepository_ == null)
                {
                    this._WarehouseRepository_ = new GenericRepo<Warehouse>(context);
                }
                return _WarehouseRepository_;
            }
        }

        public GenericRepo<Employee> EmployeeRepository
        {
            get
            {
                if (this._EmployeeRepository_ == null)
                {
                    this._EmployeeRepository_ = new GenericRepo<Employee>(context);
                }
                return _EmployeeRepository_;
            }
        }

        public void Save()
        {
            context.SaveChanges();
        }

        private bool disposed = false;
        protected virtual void Dispose(bool disposal)
        {
            if (!this.disposed)
            {
                if (disposal)
                {
                    context.Dispose();
                }
                this.disposed = true;
            }
        }

        public void Dispose()
        {
            context.Dispose();
            GC.SuppressFinalize(this);
        }
    }
}

```


Let's discuss some of this code.

```
/AbstractionLayer/WarehouseWorkUnit.cs

.
.
.

namespace EF_Code_First.AbstractionLayer
{
    public class WarehouseWorkUnit : IDisposable
    {
        private WarehouseContext context = new WarehouseContext();
        private GenericRepo<Warehouse> _WarehouseRepository;
        private GenericRepo<Employee> _EmployeeRepository;

        public GenericRepo<Warehouse> WarehouseRepository
        {
            get
            {
                if (this._WarehouseRepository == null)
                {
                    this._WarehouseRepository = new GenericRepo<Warehouse>(context);
                }
                return _WarehouseRepository;
            }
        }

        public GenericRepo<Employee> EmployeeRepository
        {
            get
            {
                if (this._EmployeeRepository == null)
                {
                    this._EmployeeRepository = new GenericRepo<Employee>(context);
                }
                return _EmployeeRepository;
            }
        }
    }
}
.
.
.
```

- **private WarehouseContext context = new WarehouseContext();** instantiates a new WarehouseContext object.
- **private GenericRepo<>** creates our repository objects by using the appropriate entity in the templated GenericRepo.
- In the **WarehouseRepository** and **EmployeeRepository** methods, we check to see if there is an existing context object; if there isn't, we instantiate a new object by passing in our context object.



and Build the solution. Now that we have a Generic Repository and a Unit of Work class, we will stop this lesson here. In the next lesson, we'll finish this project. See you there!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Object Relational Mapping: CRUD and Automatic Code Generation, LINQ, and Entity Data Model

Lesson Objectives

In this lesson, you will:

- implement application views with Entity Frameworks CRUD design and modify them to use different design patterns to increase efficiency and productivity.
- implement data abstraction to create a more robust application for real-world development.
- review different design patterns that make collaborative design more effective and efficient.
- perform LINQ queries using the different syntaxes to retrieve information from tables using navigation properties.
- create an application from an existing database using ADO.NET to incorporate application data from a pre-existing application.

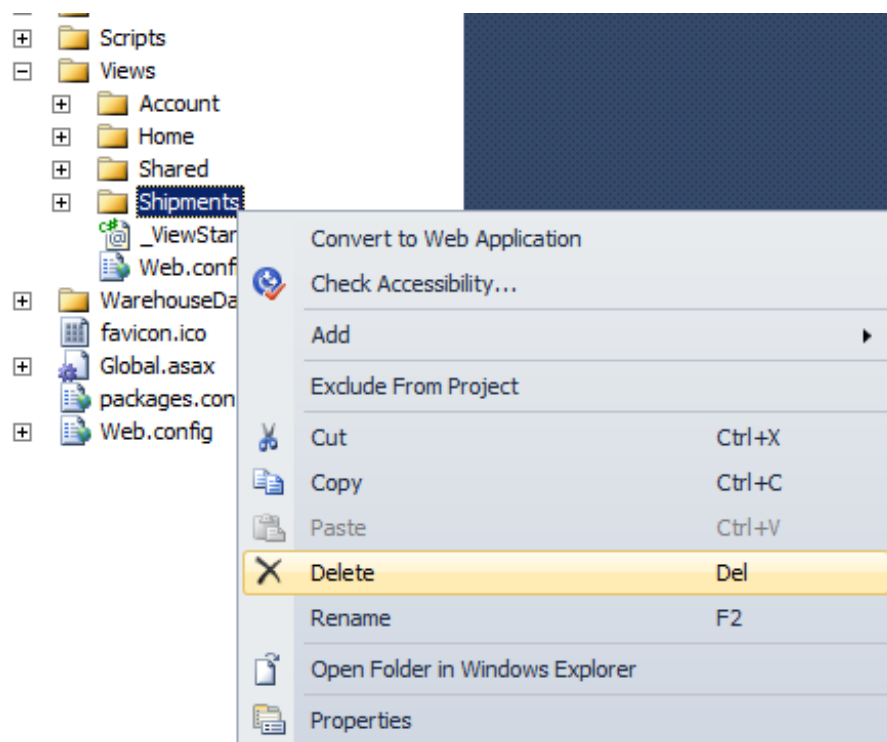
This lesson explores the benefits of abstracting data across an application, as well as using LINQ to create data technology independent solutions. You will learn to develop an application using Entity Frameworks "Create, Read, Update, Delete (CRUD)" generic design pattern, and to modify and extend the existing ASP.NET design pattern. You will also learn to implement an existing database into a new or existing application.

In this continuation of Object Relational Mapping, we will incorporate CRUD routines and interact with these routines using the Unit of Work design pattern. We'll also cover the Database First approach using the ADO.NET framework.

CRUD and Automatic Code Generation

Repository with CRUD

As promised, we'll continue working with the **EF_Code_First** solution, so go ahead and open that if it's not open already. Remove the Controller and View for the Shipments entity we created in the Entity Framework section. Right-click the Shipments View folder and select **Delete**:



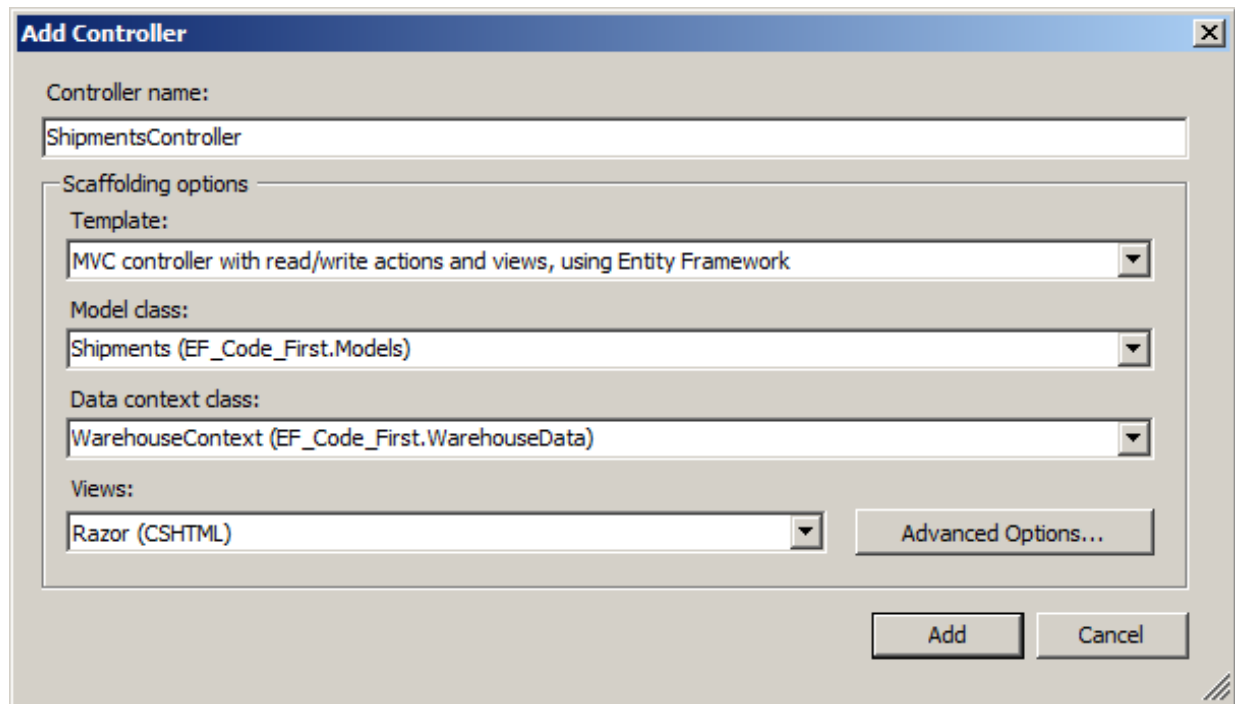
Do the same with the **/Controllers/ShipmentsController.cs** file.

Now create a Shipments controller using EF's CRUD option:

Right-click the **Controllers** folder and choose **Add | Controller...**

Set the controller name to **ShipmentsController** with these options:

- **Template:** MVC controller with Read/Write actions and Views, using Entity Framework
- **Model class:** Shipments (EF_Code_First.Models)
- **Data context class:** WarehouseContext (EF_Code_First.WarehouseData)
- **Views:** Razor (CSHTML)



The screenshot shows the 'Add Controller' dialog box. The 'Controller name' field contains 'ShipmentsController'. The 'Scaffolding options' section contains four dropdown menus: 'Template' (MVC controller with read/write actions and views, using Entity Framework), 'Model class' (Shipments (EF_Code_First.Models)), 'Data context class' (WarehouseContext (EF_Code_First.WarehouseData)), and 'Views' (Razor (CSHTML)). There is an 'Advanced Options...' button to the right of the 'Views' dropdown. At the bottom right are 'Add' and 'Cancel' buttons.

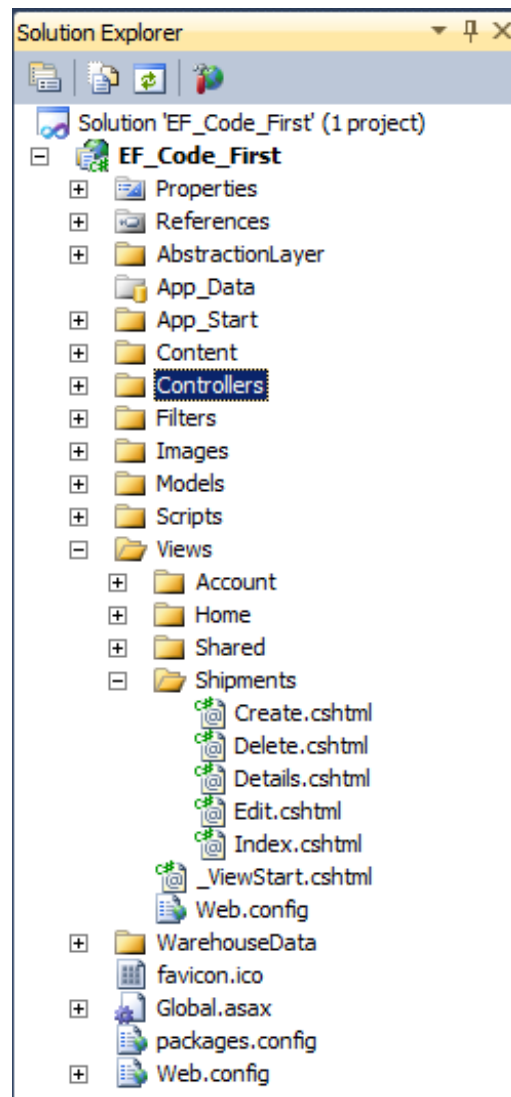
Note If these options don't appear in the drop-down lists, press **F6** to rebuild the application and retry.

Entity Framework creates default CRUD code for the Shipments controller:

OBSERVE: /Controllers/ShipmentsController.cs default CRUD code

```
.  
.   
.   
namespace EF_Code_First.Controllers  
{  
    public class ShipmentsController : Controller  
    {  
        private WarehouseContext db = new WarehouseContext();  
  
        //   
        // GET: /Shipments/  
  
        public ActionResult Index()  
        {  
            return View(db.Shipments.ToList());  
        }  
  
        //   
        // GET: /Shipments/Details/5  
  
        public ActionResult Details(int id = 0)  
        {  
            Shipments shipments = db.Shipments.Find(id);  
            if (shipments == null)  
            {  
                return HttpNotFound();  
            }  
            return View(shipments);  
        }  
  
        //   
        // GET: /Shipments/Create  
  
        public ActionResult Create()  
        {  
            return View();  
        }  
  
        .  
        .  
        .  
    }  
}
```



It also creates Views for each function in the controller:



Let's create a link to our Shipments Index page so we can see what EF has done. Open **/Views/Shared/_Layout.cshtml** and add a link to our Shipments index page as shown:

CODE TO TYPE: Modify /Views/Shared/_Layout.cshtml to add link to Shipments index

```
.
.
.
.
<nav>
  <ul id="menu">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    <li>@Html.ActionLink("Shipments Index", "Index", "Shipments")</li>
  </ul>
</nav>
.
.
.
```

 and  and navigate to the Shipments Index link; you see only a few data elements, because EF doesn't know how to pull the extra information we have stored in our Shipments table.

Index

[Create New](#)

NumberOfItemsShipped	DateTime	ReceivedDateTime	WarehouseId	
5	12/22/2001 12:00:00 AM	12/28/2001 12:00:00 AM	1	Edit Details Delete
1	1/15/2004 12:00:00 AM	1/13/2004 12:00:00 AM	1	Edit Details Delete
10	2/23/2011 12:00:00 AM	2/13/2011 12:00:00 AM	1	Edit Details Delete
15	2/23/2011 12:00:00 AM	2/13/2011 12:00:00 AM	2	Edit Details Delete
3	2/23/2011 12:00:00 AM	2/13/2011 12:00:00 AM	2	Edit Details Delete
8	5/21/2012 12:00:00 AM	5/1/2012 12:00:00 AM	2	Edit Details Delete
45	3/12/2012 12:00:00 AM	8/5/2001 12:00:00 AM	3	Edit Details Delete
24	9/1/2009 12:00:00 AM	8/5/2009 12:00:00 AM	3	Edit Details Delete
12	4/14/1995 12:00:00 AM	4/12/1995 12:00:00 AM	3	Edit Details Delete

We need to go back to **WarehouseData/WarehouseInitializer.cs** and change one line:

CODE TO TYPE: Changing WarehouseInitializer.cs to allow for persistent information

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using EF_Code_First.Models;

namespace EF_Code_First.WarehouseData
{
    public class WarehouseInitializer : DropCreateDatabaseAlwaysDropCreateDatabaseIfModelChanges<WarehouseContext>
    {
        protected override void Seed(WarehouseContext context)
        {
            .
            .
            .
        }
    }
}
```

Note **DropCreateDatabaseIfModelChanges** will recreate and reseed the database *only* if we change our model.

Now we'll add some Views for our Warehouse model. Right-click the **Controllers** folder and select **Add | Controller....** Name the controller **WarehouseController** and use the options as shown:

Add Controller

Controller name:
WarehouseController

Scaffolding options

Template:
MVC controller with read/write actions and views, using Entity Framework

Model class:
Warehouse (EF_Code_First.Models)

Data context class:
WarehouseContext (EF_Code_First.WarehouseData)

Views:
Razor (CSHTML)

Advanced Options...

Add Cancel



Now add a link to the Warehouse index to ensure that the controller works as we expect. Again, modify **/Views/Shared/_Layout.cshtml** and add another ActionLink:

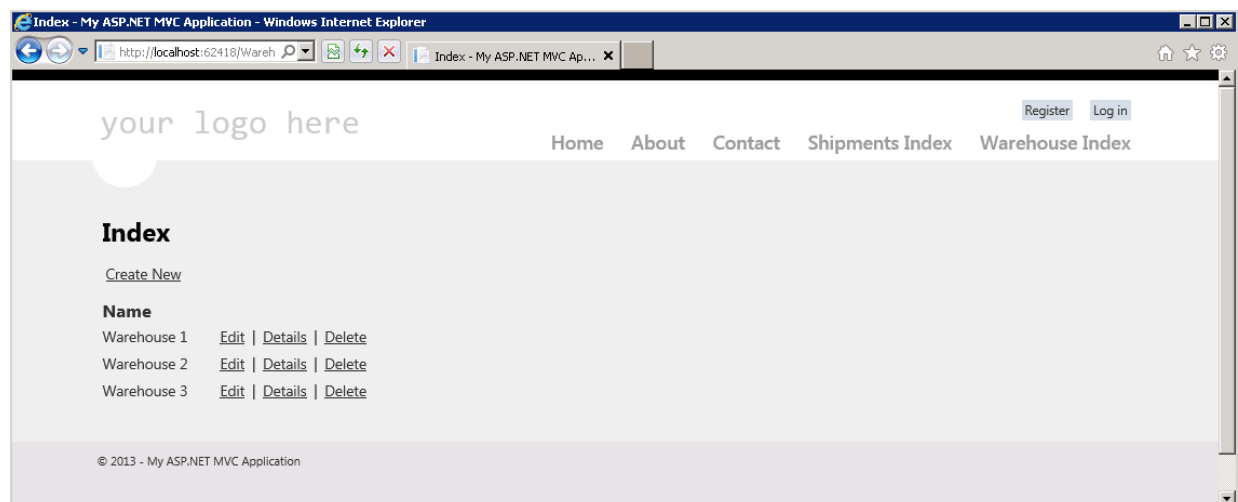
CODE TO TYPE:

```

.
.
.
<nav>
  <ul id="menu">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    <li>@Html.ActionLink("Shipments Index", "Index", "Shipments")</li>
    <li>@Html.ActionLink("Warehouse Index", "Index", "Warehouse")</li>
  </ul>
</nav>
.
.
.

```

 and  and click the **Warehouse Index** link, and you see a listing of our warehouses.



Now we can begin using our Repository. Modify **/Controllers/ShipmentsController.cs** as shown:

CODE TO TYPE:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using EF_Code_First.Models;
using EF_Code_First.WarehouseData;
using EF_Code_First.AbstractionLayer;

namespace EF_Code_First.Controllers
{
    public class ShipmentsController : Controller
    {
        private WarehouseContext db = new WarehouseContext();
        private IShipmentRepository ShipmentRepository;

        public ShipmentsController()
        {
            this.ShipmentRepository = new ShipmentRepository(new WarehouseContext());
        }

        public ShipmentsController(IShipmentRepository ShipmentRepository)
        {
            this.ShipmentRepository = ShipmentRepository;
        }

        //
        // GET: /Shipments/

        public ActionResult Index()
        {
            var shipmentsList = from s in ShipmentRepository.GetAllShipments() s
            select s;
            return View(db.Shipments.ToList());
            return View(shipmentsList);
        }

        //
        // GET: /Shipments/Details/5

        public ActionResult Details(int id = 0)
        {
            Shipments shipments = db.Shipments.Find(id); ShipmentRepository.GetByTrackingNumber(id);
            if (shipments == null)
            {
                return HttpNotFound();
            }
            return View(shipments);
        }

        //
        // GET: /Shipments/Create

        public ActionResult Create()
        {
            ViewBag.WarehouseId = new SelectList(db.Warehouse, "Id", "Name");
            return View();
        }

        //
        // POST: /Shipments/Create
    }
}
```

```

[HttpPost]
public ActionResult Create(Shipments shipments)
{
    try
    {
        if (ModelState.IsValid)
        {
db.Shipments.Add(shipments);
db.SaveChanges();
            ShipmentRepository.CreateNewShipment(shipments);
            ShipmentRepository.saveShipment();

            return RedirectToAction("Index");
        }
    }
    catch (DataException)
    {
        ModelState.AddModelError("", "An error occurred. We are working
hard to resolve this.");
    }

ViewBag.WarehouseId = new SelectList(db.Warehouse, "Id", "Name", shi
pments.WarehouseId);
    return View(shipments);
}

//
// GET: /Shipments/Edit/5

public ActionResult Edit(int id = 0)
{
    Shipments shipments = db.Shipments.Find(id); ShipmentRepository.GetBy
TrackingNumber(id);
    if (shipments == null)
    {
        return HttpNotFound();
    }
    return View(shipments);
}

//
// POST: /Shipments/Edit/5

[HttpPost]
public ActionResult Edit(Shipments shipments)
{
    try
    {
        if (ModelState.IsValid)
        {
db.Entry(shipments).State = EntityState.Modified;
db.SaveChanges();
            ShipmentRepository.ModifyShipment(shipments);
            ShipmentRepository.saveShipment();
            return RedirectToAction("Index");
        }
    }
    catch (DataException)
    {
        ModelState.AddModelError("", "Unable to edit shipment. The error
has been sent to the System Administrator.");
    }

ViewBag.WarehouseId = new SelectList(db.Warehouse, "Id", "Name", shi
pments.WarehouseId);
    return View(shipments);
}

```

```

//
// GET: /Shipments/Delete/5

public ActionResult Delete(int id = 0)
{
    Shipments shipments = db.Shipments.Find(id); ShipmentRepository.GetByTrackingNumber(id);
    if (shipments == null)
    {
        return HttpNotFound();
    }
    return View(shipments);
}

//
// POST: /Shipments/Delete/5

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    Shipments shipments = db.Shipments.Find(id);
    Shipments shipments = ShipmentRepository.GetByTrackingNumber(id);
    db.Shipments.Remove(shipments);
    db.SaveChanges();
    ShipmentRepository.RemoveShipment(id);
    ShipmentRepository.saveShipment();
    return RedirectToAction("Index");
}

protected override void Dispose(bool disposing)
{
    db.Dispose();
    base.Dispose(disposing);
    ShipmentRepository.Dispose();
    base.Dispose(disposing);
}
}

```

We replaced code that used the DbContext class with our repository.

OBSERVE:

```

private IShipmentRepository ShipmentRepository;

public ShipmentsController()
{
    this.ShipmentRepository = new ShipmentRepository(new WarehouseContext());
}

public ShipmentsController(IShipmentRepository ShipmentRepository)
{
    this.ShipmentRepository = ShipmentRepository;
}

```

Here's what's happening in our repository code:

- **private IShipmentRepository ShipmentRepository;** creates a local ShipmentRepository variable of the repository interface.
- **ShipmentsController()** creates a new ShipmentRepository context instance rather than using the DbContext class.
- **ShipmentsController(IShipmentRepository ShipmentRepository)** creates an optional constructor that allows the caller to pass in an instance of a context.

In the other methods, we replace our automatically generated code with the Shipments repository using our

interface.

We need to make one more change to the generated **/Views/Shipments/Create.html**. By default, the Entity Framework scaffold (CRUD) tries to set up the warehouse ID field as a dropdown list. To fully implement this, we need to add a new method into the **IShipmentRepository** and also define it in **ShipmentRepository**. This new method require us to pull the warehouseID from the shipments database. For now, we won't implement it as a dropdown; let's change the code accordingly:

CODE TO TYPE: Modification to /Views/Shipments/Create.cshtml

```
@model EF_Code_First.Models.Shipments

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)


    <fieldset>
    .
    .
    .

        <div class="editor-label">
            @Html.LabelFor(model => model.WarehouseId, "WarehouseNavigation")
            @Html.LabelFor(model => model.WarehouseId)
        </div>
        <div class="editor-field">
            @Html.DropDownList("WarehouseId", String.Empty)
            @Html.ValidationMessageFor(model => model.WarehouseId)
            @Html.EditorFor(model => model.WarehouseId)
            @Html.ValidationMessageFor(model => model.WarehouseId)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
```

Our repository code looks like the previous code, but does it work? Let's check it and see.



and , navigate to the Shipments Index and try to create a new shipment. Notice there's no place to enter the addresses of the Sender or Recipient, and the Index does not show any information pertaining to Senders or Recipients:

Create

NumberOfItems

ShippedDateTime

ReceivedDateTime

WarehouseId

[Back to List](#)

© 2013 - My ASP.NET MVC Application

We need to add this information manually in **/Views/Shipments/Create.cshtml** and **/Views/Shipments/Index.cshtml** files:

CODE TO TYPE: Modification to /Views/Shipments/Create.cshtml to include addresses

```
@model EF_Code_First.Models.Shipments

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Shipments</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.NumberOfItems)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.NumberOfItems)
            @Html.ValidationMessageFor(model => model.NumberOfItems)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Sender.Country)
        </div>
        <div class="editor-field">
            <b>Sender</b>: @Html.EditorFor(model => model.Sender.Country)    <b>
Recipient</b>: @Html.EditorFor(model => model.Recipient.Country)
            @Html.ValidationMessageFor(model => model.Sender.Country)
            @Html.ValidationMessageFor(model => model.Recipient.Country)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Sender.State)
        </div>
        <div class="editor-field">
            <b>Sender: </b>@Html.EditorFor(model => model.Sender.State)    <b>Re
cipient</b>: @Html.EditorFor(model => model.Recipient.State)
            @Html.ValidationMessageFor(model => model.Sender.State)
            @Html.ValidationMessageFor(model => model.Recipient.State)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Sender.City)
        </div>
        <div class="editor-field">
            <b>Sender: </b>@Html.EditorFor(model => model.Sender.City)    <b>Re
cipient</b>: @Html.EditorFor(model => model.Recipient.City)
            @Html.ValidationMessageFor(model => model.Sender.City)
            @Html.ValidationMessageFor(model => model.Recipient.City)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Sender.Street)
        </div>
        <div class="editor-field">
            <b>Sender: </b>@Html.EditorFor(model => model.Sender.Street)    <b>
Recipient</b>: @Html.EditorFor(model => model.Recipient.Street)
            @Html.ValidationMessageFor(model => model.Sender.Street)
            @Html.ValidationMessageFor(model => model.Recipient.Street)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Sender.HousingNumber)
        </div>
        <div class="editor-field">
```

```

        <b>Sender: </b>@Html.EditorFor(model => model.Sender.HousingNumber)
    <b>Recipient</b>: @Html.EditorFor(model => model.Recipient.HousingNumber)
        @Html.ValidationMessageFor(model => model.Sender.HousingNumber)
        @Html.ValidationMessageFor(model => model.Recipient.HousingNumber)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Sender.Zip)
    </div>
    <div class="editor-field">
        <b>Sender: </b>@Html.EditorFor(model => model.Sender.Zip)        <b>Rec
ipient</b>: @Html.EditorFor(model => model.Recipient.Zip)
        @Html.ValidationMessageFor(model => model.Sender.Zip)
        @Html.ValidationMessageFor(model => model.Recipient.Zip)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.ShippedDateTime)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.ShippedDateTime)
        @Html.ValidationMessageFor(model => model.ShippedDateTime)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.ReceivedDateTime)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.ReceivedDateTime)
        @Html.ValidationMessageFor(model => model.ReceivedDateTime)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.WarehouseId)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.WarehouseId)
        @Html.ValidationMessageFor(model => model.WarehouseId)
    </div>
    <p>
        <input type="submit" value="Create" />
    </p>
</fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```

Update **/Views/Shipments/Index.cshtml** to display the appropriate information:

CODE TO TYPE:

```
@model IEnumerable<EF_Code_First.Models.Shipments>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.NumberOfItems)
        </th>
        <th>
            Sender
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ShippedDateTime)
        </th>
        <th>
            Recipient
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ReceivedDateTime)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.WarehouseId)
        </th>
        <th></th>
    </tr>



    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.NumberOfItems)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Sender.Country)
                @Html.DisplayFor(modelItem => item.Sender.HousingNumber) @Html.DisplayFor(modelItem => item.Sender.Street), @Html.DisplayFor(modelItem => item.Sender.State),
                @Html.DisplayFor(modelItem => item.Sender.Zip)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ShippedDateTime)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Recipient.Country)
                @Html.DisplayFor(modelItem => item.Recipient.HousingNumber) @Html.DisplayFor(modelItem => item.Recipient.Street), @Html.DisplayFor(modelItem => item.Recipient.State),
                @Html.DisplayFor(modelItem => item.Recipient.Zip)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReceivedDateTime)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.WarehouseId)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.TrackingNumber }) |
                @Html.ActionLink("Details", "Details", new { id=item.TrackingNumber })
            </td>
        </tr>
    }
}
```



```

        @Html.ActionLink("Delete", "Delete", new { id=item.TrackingNumber })
    </td>
</tr>
}
</table>

```

Now  and , navigate to Shipments index and create a new shipment as shown:

Create

NumberOfItems

Country

Sender:

Recipient:

State

Sender:

Recipient:

City

Sender:

Recipient:

Street

Sender:

Recipient:

HousingNumber

Sender:

Recipient:

Zip

Sender:

Recipient:

ShippedDateTime

ReceivedDateTime

WarehouseId

Create

Click **Create**. You see the newly added shipment in the list:

[Create New](#)

NumberOfItems	Sender	ShippedDateTime	Recipient	ReceivedDateTime	WarehouseId	
5	United States 6589 Kilamanjaro Ln, MN, 56487	12/28/2001 12:00:00 AM	United States 588 Ste. A Main St, NV, 89712	12/22/2001 12:00:00 AM	1	Edit Details Delete
1	United States 241 Ste J Cyber Way, SD, 65874	1/13/2004 12:00:00 AM	United States 415 Apt. J Testing Circle, CA, 92120	1/15/2004 12:00:00 AM	1	Edit Details Delete
10	United States 6589 A Nouget Ct, ND, 62513	2/13/2011 12:00:00 AM	United States 2145 Empire Ln, NY, 87871	2/23/2011 12:00:00 AM	1	Edit Details Delete
15	United States 124 Leik Circle, SD, 31524	2/13/2011 12:00:00 AM	United States 128 Chicken Way, MO, 89201	2/23/2011 12:00:00 AM	2	Edit Details Delete
3	United States 6578 Nimdoy Ct, CA, 98457	2/13/2011 12:00:00 AM	United States 6542 Milky Way Ln, OR, 20152	2/23/2011 12:00:00 AM	2	Edit Details Delete
8	United States 5487 Walkens Way, TX, 31542	5/1/2012 12:00:00 AM	United States 4233 Williams St, KY, 42718	5/21/2012 12:00:00 AM	2	Edit Details Delete
45	United States 6958 Quahog St, IO, 59014	8/5/2001 12:00:00 AM	United States 5487 Milk and Cookies Way, TX, 43770	3/12/2012 12:00:00 AM	3	Edit Details Delete
24	United States 3654 Bldg B Juke Way, UT, 20154	8/5/2009 12:00:00 AM	United States 215 One Way E. Ln, MA, 65879	9/1/2009 12:00:00 AM	3	Edit Details Delete
12	United States 5487 Ste A Kreet Ln, AZ, 35126	4/12/1995 12:00:00 AM	United States 2121 Informational Way, MN, 31254	4/14/1995 12:00:00 AM	3	Edit Details Delete
12	Russia 4587 Some St, Province A, 98123	4/12/2005 12:00:00 AM	Brazil 1213 Ste F Another Way, Province C, 41578	4/29/2005 12:00:00 AM	2	Edit Details Delete

Now we can put our Shipments repository to use, and implement a repository.

Let's learn how to use our Unit of Work pattern so we can begin creating highly extensible and maintainable applications.

Unit of Work with CRUD

Modify the Warehouse controller to use the **WarehouseWorkUnit** class we created. Modify **/Controllers/WarehouseController.cs** as shown:

CODE TO TYPE: Implementing WarehouseWorkUnit in /Controllers/WarehouseController.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Data.Entity;
using System.Web;
using System.Web.Mvc;
using EF_Code_First.Models;
using EF_Code_First.WarehouseData;
using EF_Code_First.AbstractionLayer;

namespace EF_Code_First.Controllers
{
    public class WarehouseController : Controller
    {
        private WarehouseContext db = new WarehouseContext();
        private WarehouseWorkUnit WorkUnit = new WarehouseWorkUnit();
        //
        // GET: /Warehouse/

        public ActionResult Index()
        {
            var warehouses = WorkUnit.WarehouseRepository.Retrieve(includeProperties: "shipments, employees");
            return View(db.Warehouse.ToList()warehouses.ToList());
        }

        //
        // GET: /Warehouse/Details/5

        public ActionResult Details(int id = 0)
        {
            Warehouse warehouse = db.Warehouse.Find(id)WorkUnit.WarehouseRepository.GetEntitiesById(id);
            if (warehouse == null)
            +
            return HttpNotFound();
            +
            return View(warehouse);
        }

        //
        // GET: /Warehouse/Create

        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /Warehouse/Create

        [HttpPost]
        public ActionResult Create(Warehouse warehouse)
        {
            if (ModelState.IsValid)
            {
                WorkUnit.WarehouseRepository.AddEntity(warehouse);
                WorkUnit.Save();
                db.Warehouse.Add(warehouse);
                db.SaveChanges();
                return RedirectToAction("Index");
            }

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

```

//
// GET: /Warehouse/Edit/5

public ActionResult Edit(int id = 0)
{
    Warehouse warehouse = db.Warehouse.Find(id);WorkUnit.WarehouseRepository.GetEntitiesById(id);
    if (warehouse == null)
    {
        return HttpNotFound();
    }
    return View(warehouse);
}

//
// POST: /Warehouse/Edit/5

[HttpPost]
public ActionResult Edit(Warehouse warehouse)
{
    if (ModelState.IsValid)
    {
        WorkUnit.WarehouseRepository.UpdateEntity(warehouse);
        WorkUnit.Save();
        db.Entry(warehouse).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(warehouse);
}

//
// GET: /Warehouse/Delete/5

public ActionResult Delete(int id = 0)
{
    Warehouse warehouse = db.Warehouse.Find(id);WorkUnit.WarehouseRepository.GetEntitiesById(id);
    if (warehouse == null)
    {
        return HttpNotFound();
    }
    return View(warehouse);
}

//
// POST: /Warehouse/Delete/5

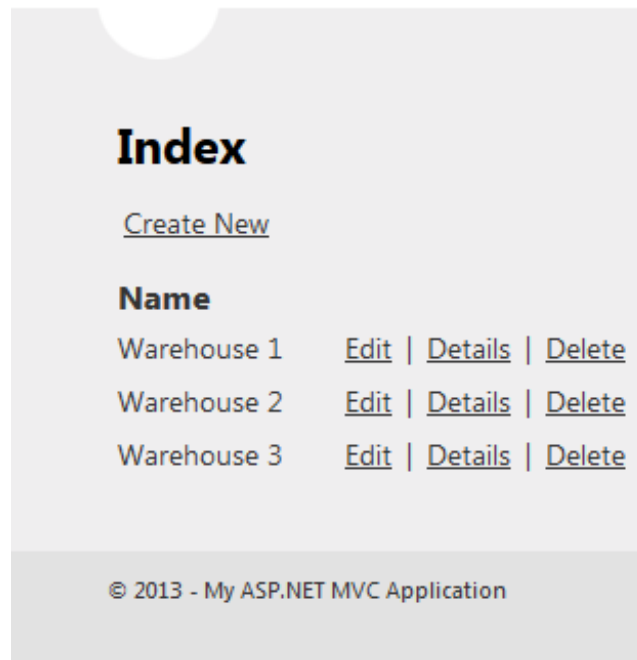
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    Warehouse warehouse = db.Warehouse.Find(id);WorkUnit.WarehouseRepository.GetEntitiesById(id);
    db.Warehouse.Remove(warehouse);WorkUnit.WarehouseRepository.RemoveEntityEntry(warehouse);
    db.SaveChanges();WorkUnit.Save();
    return RedirectToAction("Index");
}

protected override void Dispose(bool disposing)
{
    db.Dispose();WorkUnit.Dispose();
    base.Dispose(disposing);
}
}

```



and

and navigate to the **Warehouse Index** link. Everything works the same way it did before.

We'll reinforce just how useful the Unit of Work pattern can be, by adding an employee to a warehouse.

Open **/Views/Warehouse/Index.cshtml** and add two links to it:

CODE TO TYPE: Adding links to create a warehouse or employee to Index.cshtml

```
@model IEnumerable<EF_Code_First.Models.Warehouse>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("New Employee", "Create", new { NewWarehouse = false } )
    @Html.ActionLink("New Warehouse", "Create", new { NewWarehouse = true } )
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
        <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
                @Html.ActionLink("Details", "Details", new { id=item.Id }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.Id })
            </td>
        </tr>
    }
</table>
```

Modify **/Controllers/WarehouseController** as shown:

CODE TO TYPE: /Controllers/WarehouseController.cs Create changes

```
.
.
.

    //
    // GET: /Warehouse/Details/5

    public ActionResult Details(int id = 0)
    {
        Warehouse warehouse = WorkUnit.WarehouseRepository.GetEntitiesById(i
d);
        return View(warehouse);
    }

    //
    // GET: /Warehouse/Create

    public ActionResult Create(bool NewWarehouse = true)
    {
        if (NewWarehouse)
            return View();
        else
        {
            CreateDropDownList();
            return RedirectToAction("Create", "Employee");
        }
    }

.
.
.

    //
    // POST: /Warehouse/Delete/5

    [HttpPost, ActionName("Delete")]
    public ActionResult DeleteConfirmed(int id)
    {
        Warehouse warehouse = WorkUnit.WarehouseRepository.GetEntitiesById(i
d);
        WorkUnit.WarehouseRepository.RemoveEntityEntry(warehouse);
        WorkUnit.Save();
        return RedirectToAction("Index");
    }

    private void CreateDropDownList(object chosenWarehouse = null)
    {
        var existingWarehouses = WorkUnit.WarehouseRepository.Retrieve(order
By: x => x.OrderBy(y => y.Name));
        ViewBag.WarehouseDropDown = new SelectList(existingWarehouses, "Id",
"Name", chosenWarehouse);
    }

    protected override void Dispose(bool disposing)
    {
        WorkUnit.Dispose();
        base.Dispose(disposing);
    }
}
```

Create a Controller for the Employee model. Right-click the Controllers folder and add a new Controller.

Add Controller

Controller name:
EmployeeController

Scaffolding options

Template:
MVC controller with read/write actions and views, using Entity Framework

Model class:
Employee (EF_Code_First.Models)

Data context class:
WarehouseContext (EF_Code_First.WarehouseData)

Views:
Razor (CSHTML)

Advanced Options...

Add Cancel

Modify **/Controllers/EmployeeController.cs** to use our Unit of Work as shown:

CODE TO TYPE: Incorporating Unit of Work into /Controllers/EmployeeController.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using EF_Code_First.Models;
using EF_Code_First.WarehouseData;
using EF_Code_First.AbstractionLayer;

namespace EF_Code_First.Controllers
{
    public class EmployeeController : Controller
    {
        private WarehouseContext db = new WarehouseContext();
        private WarehouseWorkUnit WorkUnit = new WarehouseWorkUnit();
        //
        // GET: /Employee/

        public ActionResult Index()
        {
            var EmployeeList = WorkUnit.EmployeeRepository.Retrieve(includeProperties: "EmployeeLocation");
            return View(EmployeeList.ToList());

            var employees = db.Employees.Include(e => e.EmployeeLocation);
            return View(employees.ToList());
        }

        //
        // GET: /Employee/Details/5

        public ActionResult Details(int id = 0)
        {
            Employee employee = db.Employees.Find(id);
            if (employee == null)
            {
                return HttpNotFound();
            }
            return View(employee);
        }

        //
        // GET: /Employee/Create

        public ActionResult Create()
        {
            ViewBag.WarehouseId = new SelectList(db.Warehouse, "Id", "Name");
            CreateDropDownLists();
            return View();
        }

        //
        // POST: /Employee/Create

        [HttpPost]
        public ActionResult Create(Employee employee)
        {
            if (ModelState.IsValid)
            {
                WorkUnit.EmployeeRepository.AddEntity(employee);
                WorkUnit.Save();
                db.Employees.Add(employee);
                db.SaveChanges();
            }
        }
    }
}
```



```

        return RedirectToAction("Index");
    }

    ViewBag.WarehouseId = new SelectList(db.Warehouse, "Id", "Name", emp
loyee.WarehouseId);
    CreateDropDownLists(employee.Position, employee.EmployeeLocation);
    return View(employee);
}

//
// GET: /Employee/Edit/5

public ActionResult Edit(int id = 0)
{
    Employee employee = db.Employees.Find(id);
    if (employee == null)
    {
        return HttpNotFound();
    }

    ViewBag.WarehouseId = new SelectList(db.Warehouse, "Id", "Name", emp
loyee.WarehouseId);
    return View(employee);
}

//
// POST: /Employee/Edit/5

[HttpPost]
public ActionResult Edit(Employee employee)
{
    if (ModelState.IsValid)
    {
        db.Entry(employee).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    ViewBag.WarehouseId = new SelectList(db.Warehouse, "Id", "Name", emp
loyee.WarehouseId);
    return View(employee);
}

//
// GET: /Employee/Delete/5

public ActionResult Delete(int id = 0)
{
    Employee employee = WorkUnit.EmployeeRepository.GetEntitiesById(id);
    Employee employee = db.Employees.Find(id);
    if (employee == null)
    {
        return HttpNotFound();
    }
    return View(employee);
}

//
// POST: /Employee/Delete/5

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    Employee employee = WorkUnit.EmployeeRepository.GetEntitiesById(id);
    WorkUnit.EmployeeRepository.RemoveEntityEntry(employee);
    WorkUnit.Save();
    Employee employee = db.Employees.Find(id);
    db.Employees.Remove(employee);

```

```

        db.SaveChanges();
        return RedirectToAction("Index");
    }

    private void CreateDropDownLists(object position = null, object chosenWarehouse = null)
    {
        var existingPositions = WorkUnit.EmployeeRepository.Retrieve(orderBy: x => x.OrderBy(y => y.Position));
        ViewBag.PositionsDropDown = new SelectList(existingPositions, "Position", "Position", position);

        var existingWarehouses = WorkUnit.WarehouseRepository.Retrieve(orderBy: x => x.OrderBy(y => y.Name));
        ViewBag.WarehouseDropDown = new SelectList(existingWarehouses, "Id", "Name", chosenWarehouse);
    }

    protected override void Dispose(bool disposing)
    {
        WorkUnit.Dispose();
        db.Dispose();
        base.Dispose(disposing);
    }
}

```

We've already seen most of this code in the Warehouse controller. Let's discuss the new stuff.

OBSERVE:

```

    private void CreateDropDownLists(object position = null, object chosenWarehouse = null)
    {
        var existingPositions = WorkUnit.EmployeeRepository.Retrieve(orderBy: x => x.OrderBy(y => y.Position));
        ViewBag.PositionsDropDown = new SelectList(existingPositions, "Position", "Position", position);

        var existingWarehouses = WorkUnit.WarehouseRepository.Retrieve(orderBy: x => x.OrderBy(y => y.Name));
        ViewBag.WarehouseDropDown = new SelectList(existingWarehouses, "Id", "Name", chosenWarehouse);
    }

```

- **CreateDropDownLists(object position = null, object chosenWarehouse = null)** is a private function that takes two optional parameters. One of them is a position, and the other is a warehouse; both of them are objects.
- **var existingPositions = WorkUnit.EmployeeRepository.Retrieve(orderBy: x => x.OrderBy(y => y.Position));** creates a local variable using our EmployeeRepository from WorkUnit that we use as our IEnumerable type in the SelectList constructor.
- We treat **existingWarehouses** like existingPositions, except that **existingWarehouses** holds a list of our warehouses.

Now, modify the **/Views/Employee/Index.cshtml** page to display the information we want:

CODE TO TYPE: Employee Index.cshtml file

```
@model IEnumerable<EF_Code_First.Models.Employee>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Position)
        </th>
        <th>
            Employee Name
        </th>
        <th>
            Employee Location
        </th>
        <th>
            @Html.DisplayNameFor(model => model.EmployeeLocation.Name)
        </th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Position)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.person.FullName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EmployeeLocation.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EmployeeLocation.Name)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.EmployeeId }) |
                @Html.ActionLink("Details", "Details", new { id=item.EmployeeId }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.EmployeeId })
            </td>
        </tr>
    }
</table>
```

Now add some code to the Create.cshtml file for our Employee views:

CODE TO TYPE: Create.cshtml for Employee

```
@model EF_Code_First.Models.Employee

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Employee</legend>
        <div class="editor-label">
            @Html.LabelFor(model => model.EmployeeId)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.EmployeeId)
            @Html.ValidationMessageFor(model => model.EmployeeId)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.person.FirstName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.person.FirstName)
            @Html.ValidationMessageFor(model => model.person.FirstName)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.person.LastName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.person.LastName)
            @Html.ValidationMessageFor(model => model.person.LastName)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.person.MiddleName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.person.MiddleName)
            @Html.ValidationMessageFor(model => model.person.MiddleName)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.person.SocialSecurity)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.person.SocialSecurity)
            @Html.ValidationMessageFor(model => model.person.SocialSecurity)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Position)
        </div>
        <div class="editor-field">
@Html.EditorFor(model => model.Position)
@Html.ValidationMessageFor(model => model.Position)
            @Html.DropDownListFor(model => model.Position, (SelectList)ViewBag.P
ositionsDropDown)
        </div>
        <div class="editor-label">
@Html.LabelFor(model => model.WarehouseId, "EmployeeLocation")
            Warehouse Location
        </div>
        <div class="editor-field">
@Html.DropDownList("WarehouseId", String.Empty)
@Html.ValidationMessageFor(model => model.WarehouseId)
            @Html.DropDownListFor(model => model.WarehouseId, (SelectList)ViewBa
g.WarehouseDropDown)
        </div>
    </fieldset>
}

</div>
```

```

        </div>
        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```



and



, and go to the Warehouse Index. Now you see a list of our warehouses as well as two new links:

Index

[New Employee](#)
[New Warehouse](#)
[Create New](#)

Name	
Warehouse 1	Edit Details Delete
Warehouse 2	Edit Details Delete
Warehouse 3	Edit Details Delete

© 2013 - My ASP.NET MVC Application

Click **New Employee** to create a new employee:

your logo here

Create

EmployeeId

32768

FirstName

Richard

LastName

Rios

MiddleName

M

SocialSecurity

012-34-5678

Position

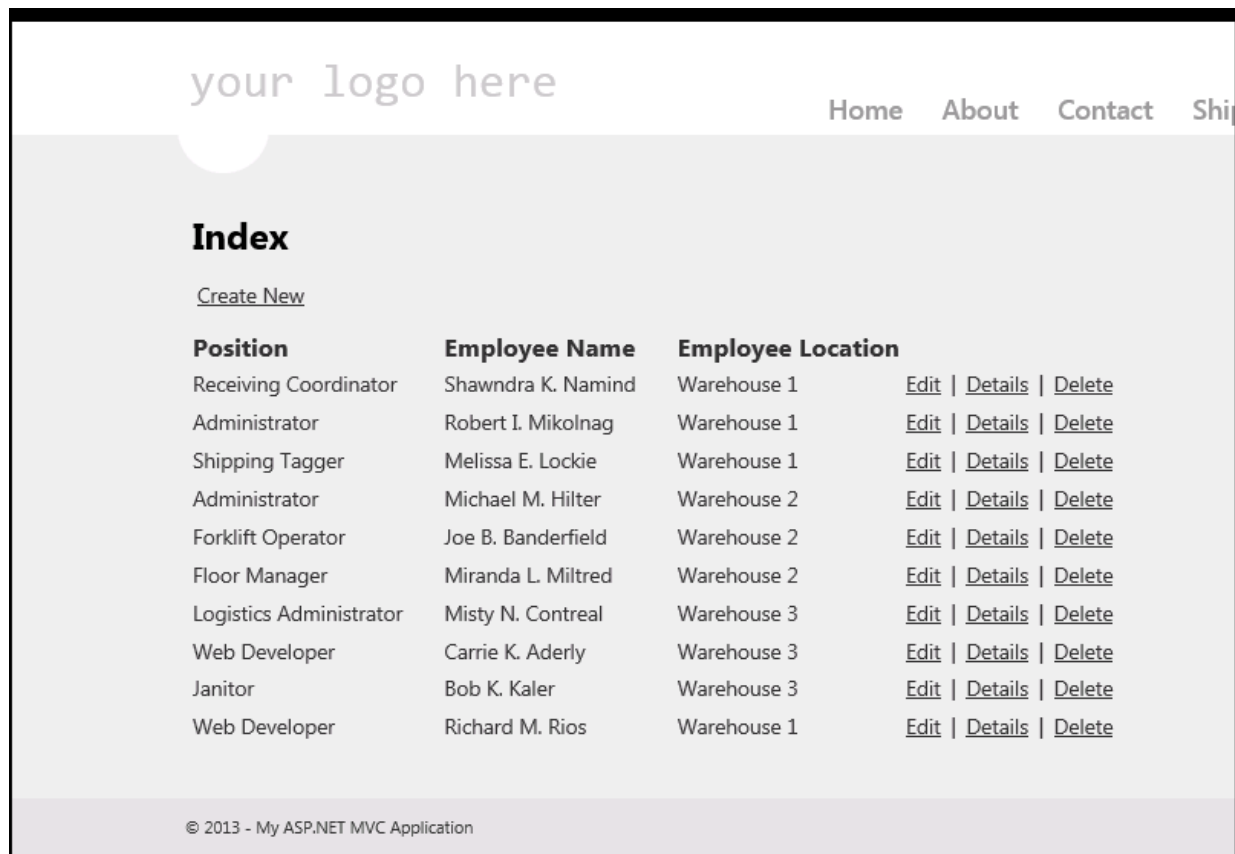
Web Developer

Warehouse Location

Warehouse 1

Create

Click **Create**. You're redirected to the Employee index page, where you see all of our employees:



Note You can find additional references for the SelectList class [here](#).

If we wanted to create a list of positions that weren't already defined, we could create a new entity and store our positions in a table in our database and reference that database to populate a position element.

LINQ to Entities

The Entity Framework provides two methods of performing queries: Method Syntax and Query Syntax.

- **Query Syntax:** must be translated into method calls during compile time since .NET Common Language Runtime (CLR) does not support this type on its own. It can become cumbersome to write, but many find it easier to read and understand.
- **Method Syntax:** native to .NET and does not need to be translated during compile time. Most LINQ references are written using the Method syntax.

Let's see these methods in action. First, we'll create a new controller named **LINQController** and choose the **Empty MVC controller** template.

Add one action for Method and one action for Query to the LINQController and some code to see the difference between the Query and Method syntax:

CODE TO TYPE: LINQController.cs with Method and Query syntax

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using EF_Code_First.Models;
using EF_Code_First.WarehouseData;

namespace EF_Code_First.Controllers
{
    public class LINQController : Controller
    {
        private WarehouseContext db = new WarehouseContext();

        public ActionResult Query()
        {
            var method = (from x in db.Shipments where (x.NumberOfItems < 30 && x.NumberOfItems > 10) select x).ToList();
            return View(method);
        }

        public ActionResult Method()
        {
            var query = db.Shipments.Where(x => (x.NumberOfItems < 30 && x.NumberOfItems > 10)).ToList();
            return View(query);
        }

        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }

        //
        // GET: /LINQ/
        public ActionResult Index()
        {
        return View();
        }
    }
}
```

Let's discuss the new code.

OBSERVE:

```
public ActionResult Query()
{
    var method = (from x in db.Shipments where (x.NumberOfItems < 30 && x.NumberOfItems > 10) select x).ToList();
    return View(method);
}

public ActionResult Method()
{
    var query = db.Shipments.Where(x => (x.NumberOfItems < 30 && x.NumberOfItems > 10)).ToList();
    return View(query);
}
```


- Our **Method()** action uses the Method syntax to return items in the Shipments table that have between 10 and 30 items.
- The **Query()** action uses the Query syntax and Lambda expressions to perform the same operation as the Method() action, but with less room for error.

These two statements are equivalent:

- **from x in db.Shipments** is equivalent to **db.Shipments**.
- **where (x.NumberOfItems < 30 && x.NumberOfItems > 10) select x** is equivalent to **Where(x => (x.NumberOfItems < 30 && x.NumberOfItems > 10))**.

Create a view for the Method() action. Check the **Strongly-Typed view** option and select the **Shipments** model as our model class. Set the template to **List**:

```

LINQ/Method List EF view

@model IEnumerable<EF_Code_First.Models.Shipments>

@{
    ViewBag.Title = "Method";
}

<h2>Method</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
<tr>
    <th>
        @Html.DisplayNameFor(model => model.NumberOfItems)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.ShippedDateTime)
    </th>
    .
    .
    .

```

Do the same for the Query() method; you'll see the same code and the same output.

Query and Method syntax are semantically equivalent. However, Query is not supported by the .NET CLR on its own and must be interpreted at compile time, while the Method syntax is supported natively and not translated during compile.

Using the Entity relationships we defined earlier in the lesson, we can retrieve multiple or single elements of a database. Add a new action to the LINQController named **MethodMany** as shown:

CODE TO TYPE: MethodMany action in LINQController.cs

```
.  
.br/>.br/>public ActionResult MethodMany(int id = 0)  
{  
    if (id == 0)  
    {  
        var methodMany = db.Warehouse  
            .Include(x => x.shipments)  
            .Include(x => x.employees)  
            .ToList();  
        return View(methodMany);  
    }  
    else  
    {  
        var methodMany = db.Warehouse  
            .Where(x => x.Id == id)  
            .Include(x => x.shipments)  
            .Include(x => x.employees)  
            .ToList();  
        return View(methodMany);  
    }  
}  
.br/>.br/.
```

Now, create a view for **MethodMany**; select the **Warehouse (EF_Code_First.Models)** model and an **Empty** template. The only property is the warehouse name, because EF doesn't know how to access our other entities and properties. Let's modify the view so we can retrieve all the information we need, as well as provide the option to view only one warehouse worth of information. This will allow us to see how our Method syntax can filter queries by values. Modify the code as shown:

CODE TO TYPE: Modifying MethodMany.cshtml to access other entities

```
@model IEnumerable<EF_Code_First.Models.Warehouse>

@{
    ViewBag.Title = "MethodMany";
}

<h2>MethodMany</h2>

<p>
    @Html.ActionLink("Show All", "MethodMany", new { id = 0 })
</p>
<table>
    @foreach (var item in Model)
    {
        <tr>
            <th>@Html.ActionLink(@item.Name.ToString(), "MethodMany", new { id = item.I
d }) </th>
            </tr>
            @foreach (var emp in item.employees)
            {
                <tr>
                    <td>@emp.Position</td>
                    <td>@emp.person.FullName</td>
                </tr>
            }
            <tr style="border-bottom: 1px solid black; border-right: 1px solid black;">
                @foreach (var ship in item.shipments)
                {
                    <td><b>Recipient</b>: @ship.Recipient.City, @ship.Recipient.State<br />
                    <b>Sender</b>: @ship.Sender.City, @ship.Sender.State<br /></td>
                }
            </tr>
        }
    }
</table>
```

Let's discuss some of the code.

OBSERVE:

```
@foreach (var item in Model)
{
    <tr>
        <th>@Html.ActionLink(@item.Name.ToString(), "MethodMany", new { id = item.I
d }) </th>
        </tr>
        @foreach (var emp in item.employees)
        {
            <tr>
                <td>@emp.Position</td>
                <td>@emp.person.FullName</td>
            </tr>
        }
        <tr style="border-bottom: 1px solid black; border-right: 1px solid black;">
            @foreach (var ship in item.shipments)
```

- The first **foreach** loop gives us access to the properties in our Model by declaring a variable named item that gains access to our Warehouse entity.
- In the second **foreach**, we declare a new variable that is valid only for the scope of this loop. By using our previous **item** variable, we can gain access to our employees entity. The same occurs with the **next loop**.

If we run our application and navigate to our MethodMany view, we should see the information listed. When we click on a warehouse name, we receive only that warehouse's information.

MethodMany

[Show All](#)

Warehouse 1

Receiving Coordinator Shawndra K. Namind

Administrator Robert I. Mikolnag

Shipping Tagger Melissa E. Lockie

Recipient: Tahoe, NV

Recipient: San Diego, CA

Recipient: New York, NY

Sender: Lakeview, MN

Sender: Fargo, SD

Sender: Northern, ND

Warehouse 2

Administrator Michael M. Hilter

Forklift Operator Joe B. Banderfield

Floor Manager Miranda L. Miltred

Recipient: Bozeman, MO

Recipient: Salem, OR

Recipient: Campbellsville, KY

Sender: Slates, SD

Sender: Sharp, CA

Sender: Mackay, TX

Warehouse 3

Logistics Administrator Misty N. Contreal

Web Developer Carrie K. Aderly

Janitor Bob K. Kaler

Recipient: Honeybun, TX

Recipient: Middleton, MA

Recipient: Minnetonka, MN

Sender: Manchester, IO

Sender: Milk, UT

Sender: Snickers, AZ

MethodMany

[Show All](#)

Warehouse 2

Administrator Michael M. Hilter

Forklift Operator Joe B. Banderfield

Floor Manager Miranda L. Miltred

Recipient: Bozeman, MO

Sender: Slates, SD

Recipient: Salem, OR

Sender: Sharp, CA

Recipient: Campbellsville, KY

Sender: Mackay, TX

© 2013 - My ASP.NET MVC Application

We won't go any further into the Query syntax now; instead let's focus on tools that are native to the .NET framework. Here's a sample code snippet that shows some of the elements required to get the same Entity information as the MethodMany() action:

OBSERVE: Query syntax for MethodMany action

```
.
.
.
from _warehouse in db.Warehouse
    join _shipments
        in db.Shipments
        on _warehouse.Id equals _shipments.WarehouseId
    into shipGroup
    from ship in shipGroup
    join _employees in db.Employees
    on _warehouse.Id equals _employees.WarehouseId
    into empGroup
    from emp in empGroup
where _warehouse.Id == id
.
.
.
```

Note You can find references for Query and Method Syntax [here](#).

Entity Data Model (EDM)

Now that we've worked with the Code First approach to the Entity Framework, let's discuss the Database First approach with the Entity Data Model.

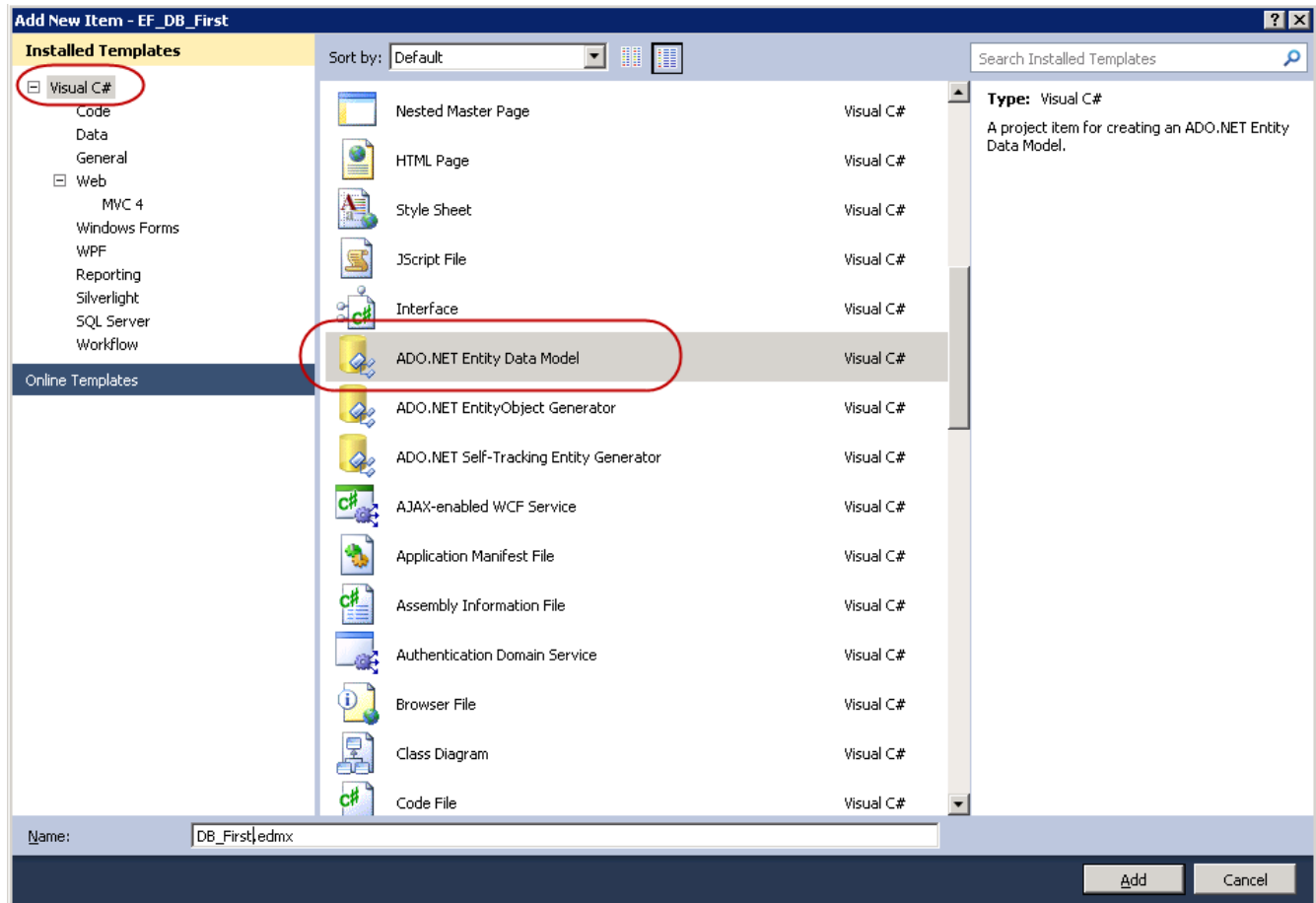
The Conceptual Model is a specific representation of the way data is represented. It uses Entities and relationships to model a relationship from our data, which allows the application to grow without scalability issues. Conceptual models are represented in a domain-specific language; specifically, in entities and relationships the language is referred to as Conceptual Schema Definition Language (CSDL).

Create a new project named **EF_DB_First**.

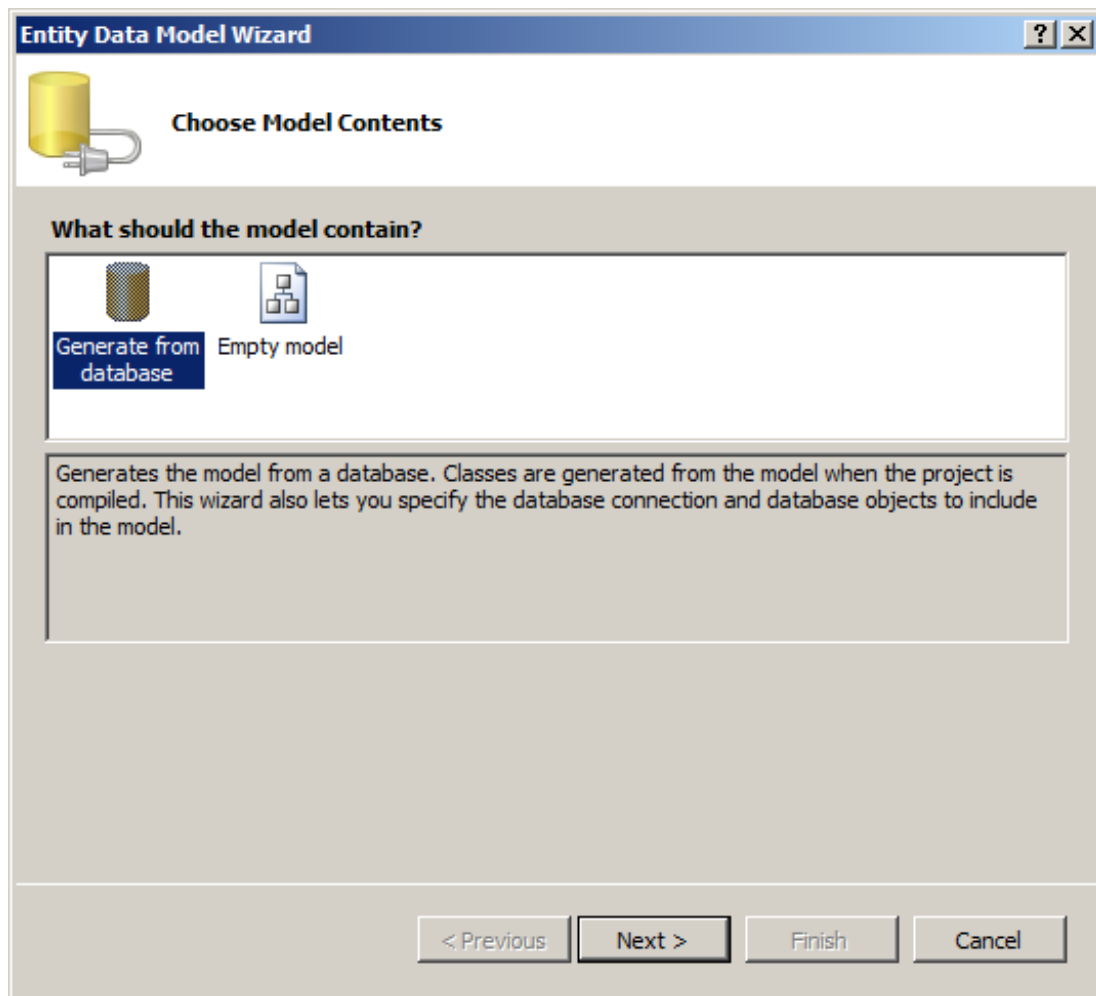
Once the project has been created, we create our EDM. Right-click on the root project folder and select **Add | New**

Item....

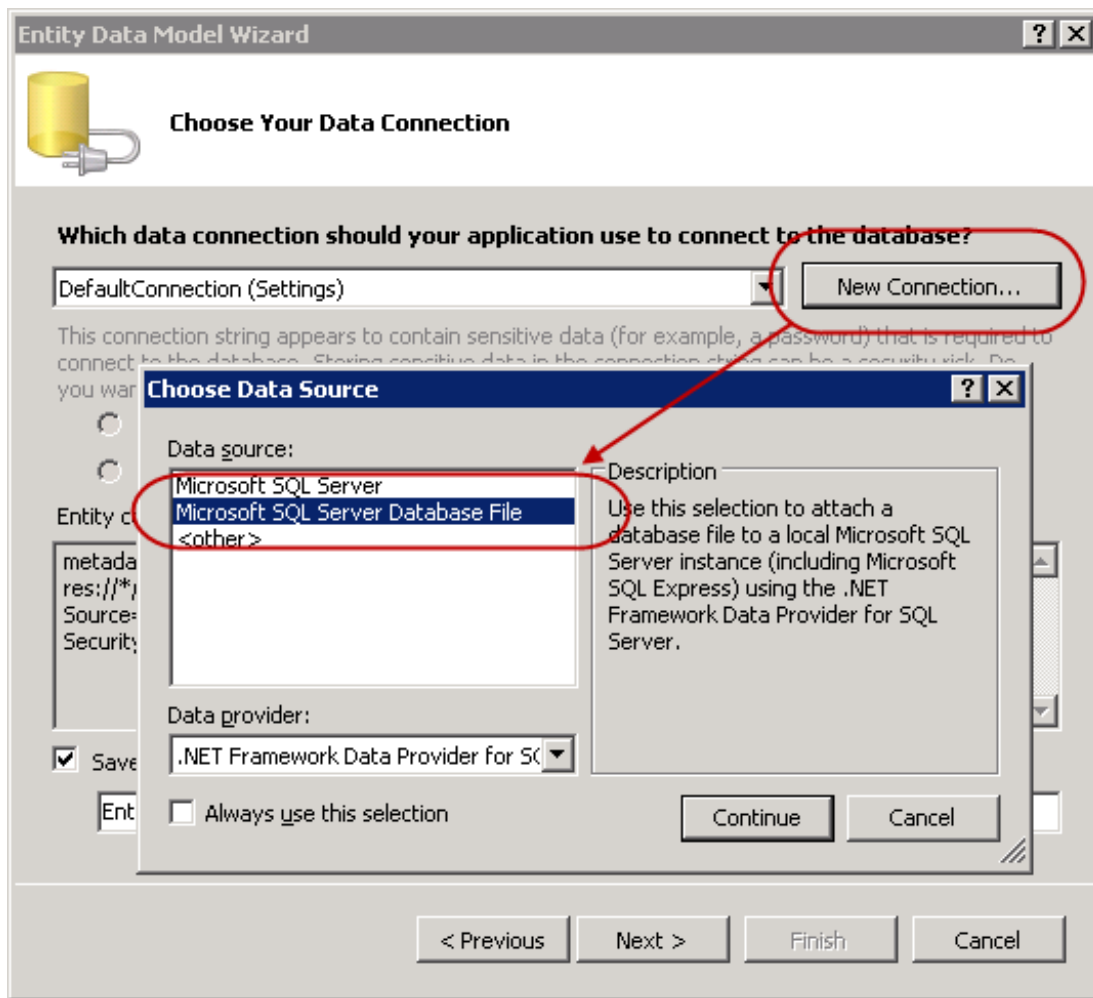
In the menu, under Installed Templates, select **Visual C#**, find **ADO.NET Entity Data Model** and enter the name **DB_First**:



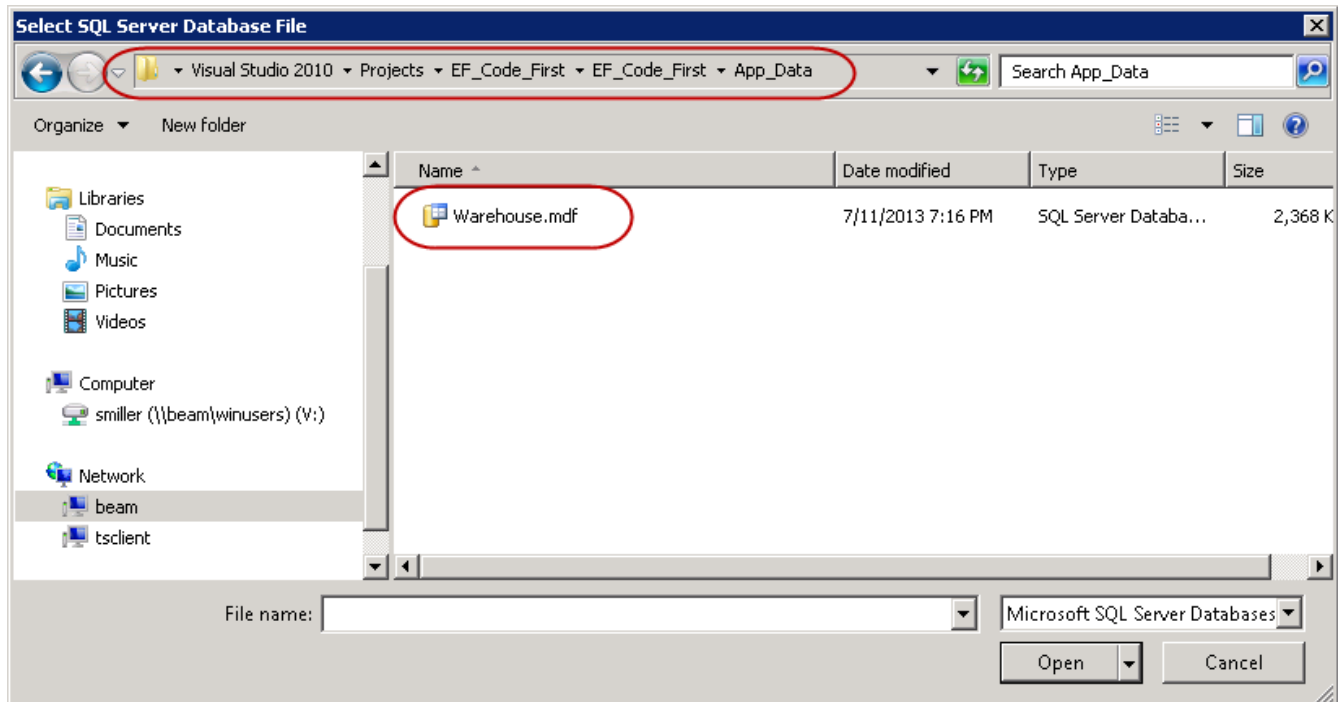
Click **Add**. In the Entity Data Model Wizard, select the **Generate From Database** option since we will be using the database from our Code-First sections:



Click **Next**. Click **New Connection** to change from the DefaultConnection and set it to **Microsoft SQL Server Database File**:




Click **Continue** and then **Browse** to navigate to and choose the existing **Warehouse.mdf** database from our **EF_Code_First** project:



Click **OK** and then click **Next**. When the wizard asks if you want to copy the database and modify the connection, click **Yes**. In the next window, enable the **Tables** only, and *uncheck* the **Pluralize or Singularize generated object names** box:

Entity Data Model Wizard

 Choose Your Database Objects

Which database objects do you want to include in your model?

- ☒ Tables
- ☐ Views
- ☐ Stored Procedures

☐ Pluralize or singularize generated object names

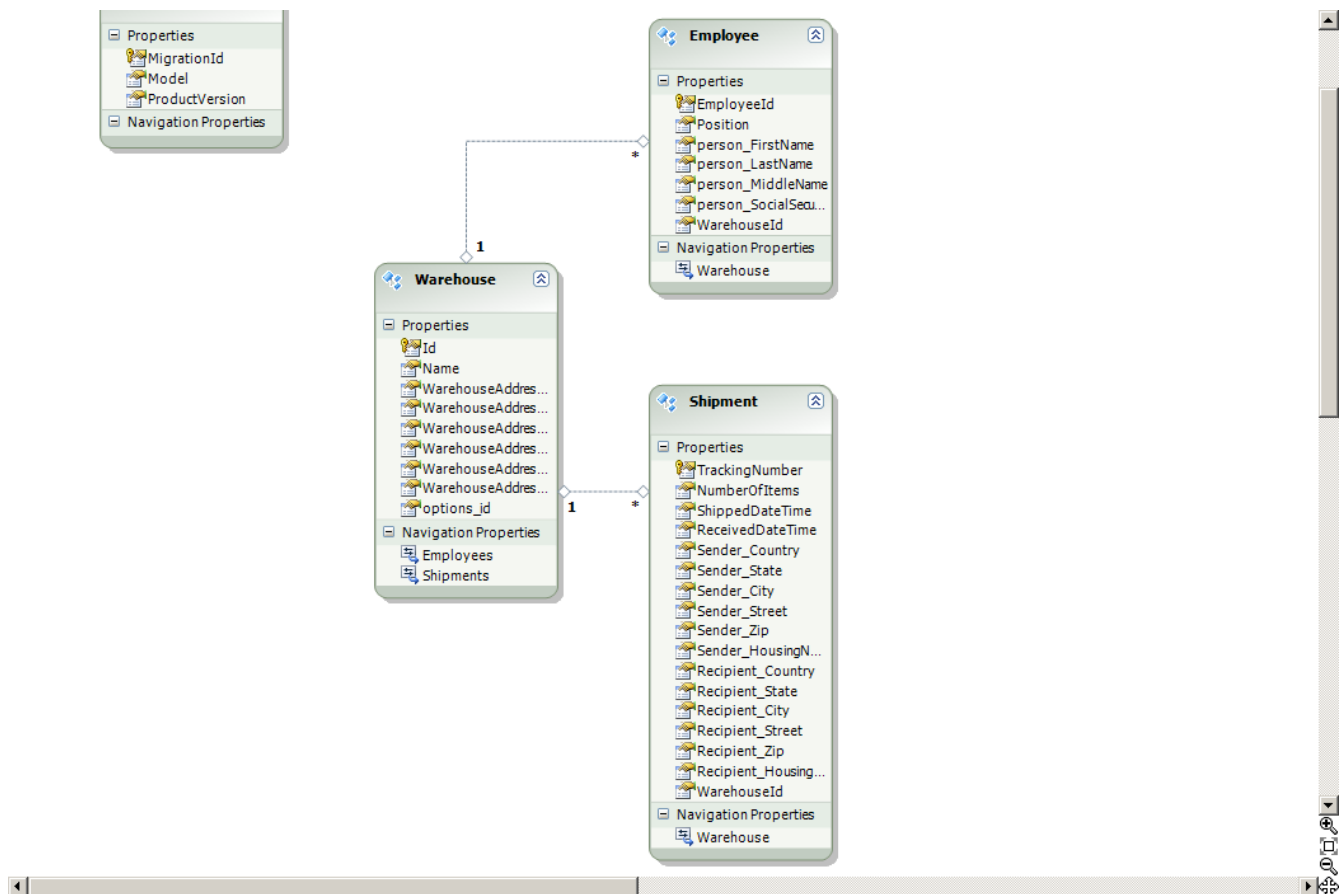
☒ Include foreign key columns in the model

Model Namespace:

WarehouseModel

< Previous Next > Finish Cancel

Click **Finish**. Visual Studio creates a file named DB_First.edmx and within that is the DB_First.Designer.edmx file, which displays our entities' properties:

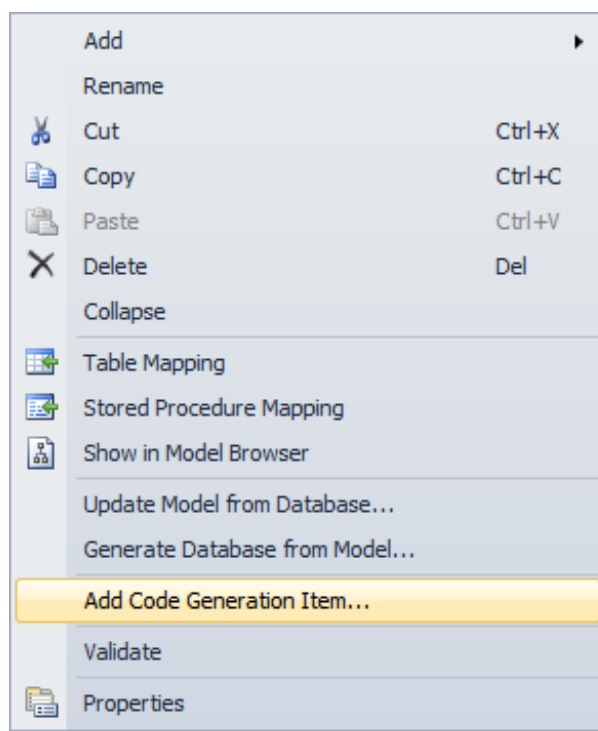


Note

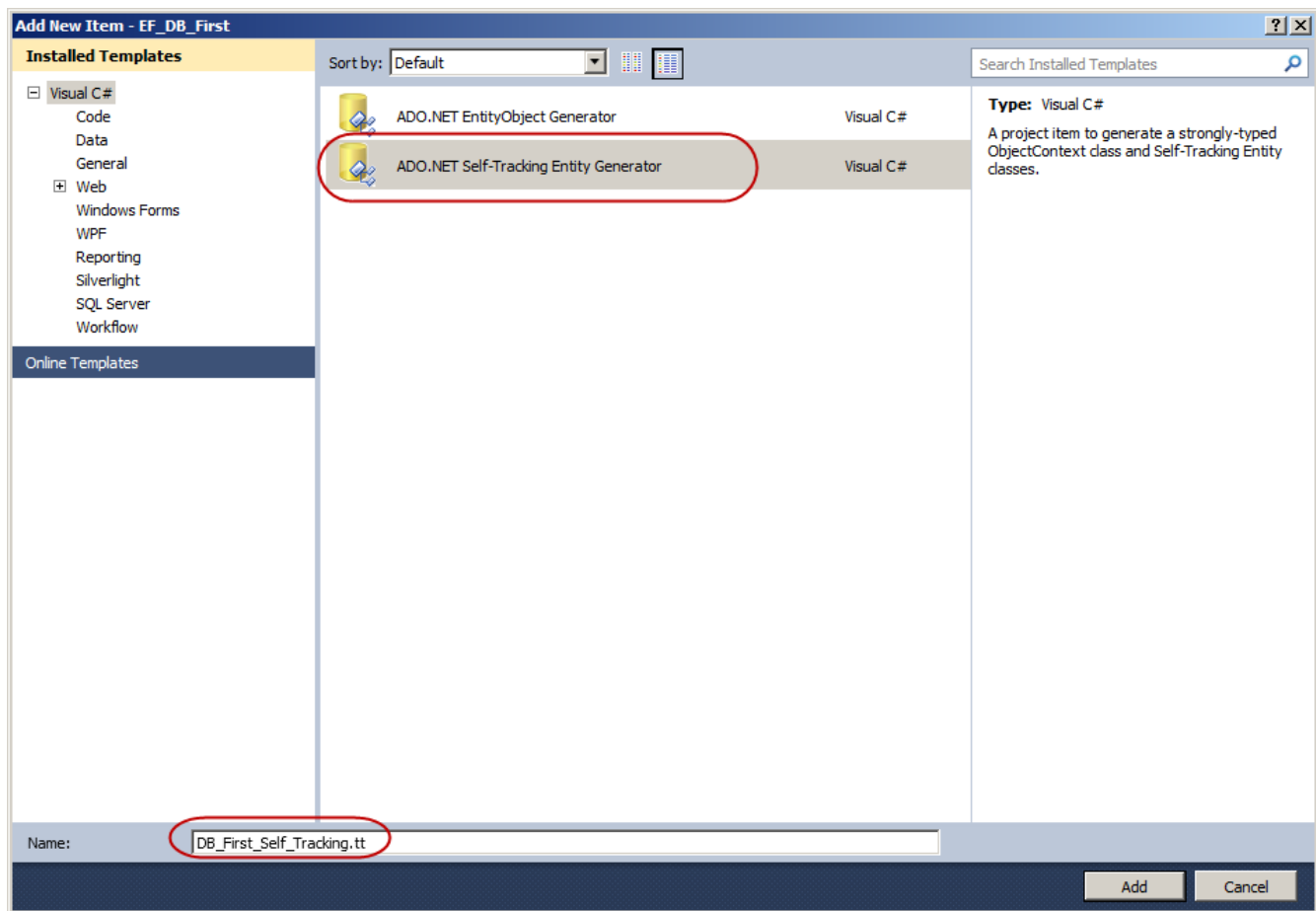
The relationships are represented as 1...* (one-to-many), *...*, (many-to-many), 1...1 (one-to-one), and 0...* (zero-or-one-to-many).

The ADO.NET EF has designed our Entities. We have entity relationships and a database.

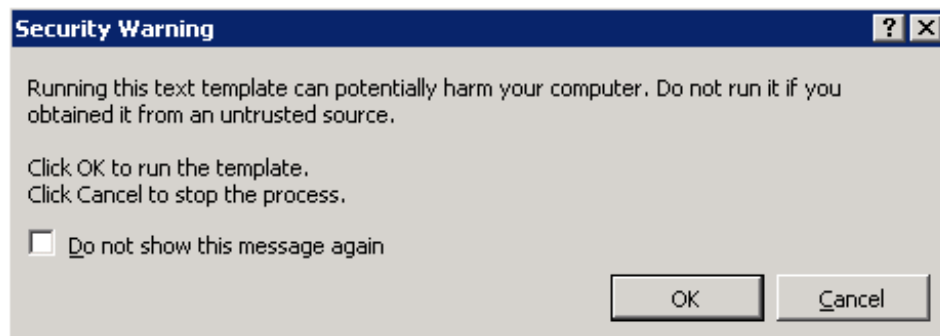
Right-click in the area where the edmx file is displayed and select **Add Code Generation Item...**:



Choose the **ADO.NET Self-Tracking Entity Generator** and name it **DB_First_Self_Tracking**:

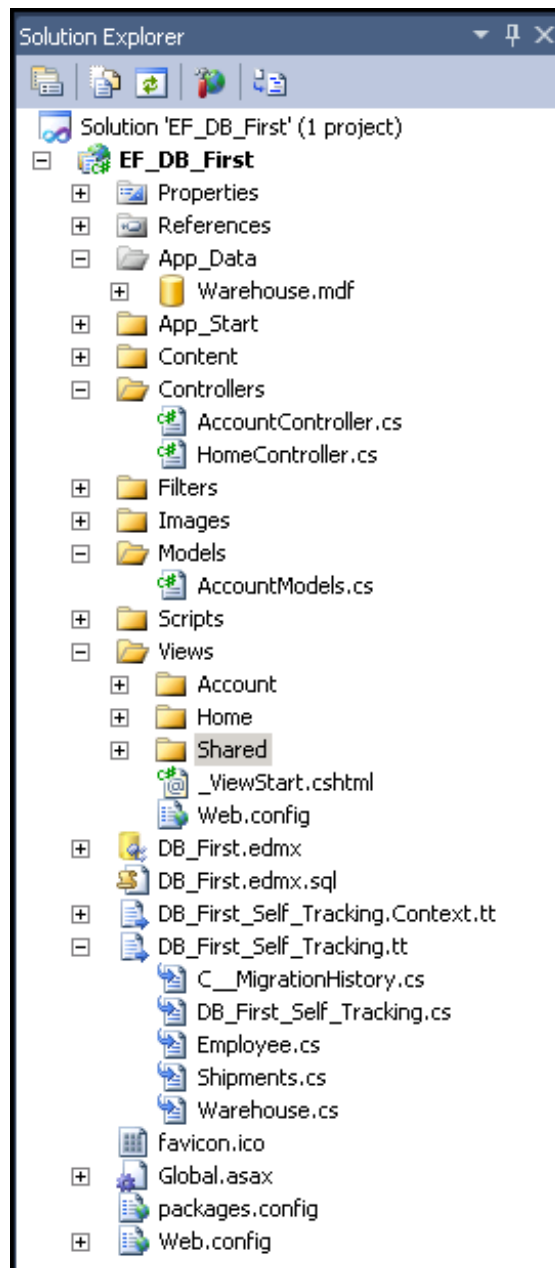


Click **Add**. You may see a Security Warning like this:



In this case, it's alright to click **OK**.

Notice that the framework has built and added some new files for us under the folders **DB_First_Self_Tracking.tt** and **DB_First_Self_Tracking.Context.tt**:



Now that we have all of our code, let's add a controller so we can begin to use our newly created self-tracking entities. Right-click the **Controllers** folder and add a new controller named **WarehouseController**, with the Model Class set to **Warehouse** and the Context Class set to **WarehouseEntities** (our models are located in the **DB_First_Self_Tracking.tt** folder).

Add Controller

Controller name:
WarehouseController

Scaffolding options

Template:
MVC controller with read/write actions and views, using Entity Framework

Model class:
Warehouse (EF_DB_First)

Data context class:
WarehouseEntities (EF_DB_First)

Views:
Razor (CSHTML)

Advanced Options...

Add Cancel

The code that is generated automatically, is similar to some of our Controllers from the Code First sections of this lesson:

OBSERVE: WarehouseController.cs

```
.
.
.
public ActionResult Details(int id = 0)
{
    Warehouse warehouse = db.Warehouses.Single(w => w.Id == id);
    if (warehouse == null)
    {
        return HttpNotFound();
    }
    return View(warehouse);
}

//
// GET: /Warehouse/Create

public ActionResult Create()
{
    return View();
}


//
// POST: /Warehouse/Create

[HttpPost]
public ActionResult Create(Warehouse warehouse)
{
    if (ModelState.IsValid)
    {
        db.Warehouses.AddObject(warehouse);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(warehouse);
}
```

.
.
.



and  and navigate to the Warehouse Index page:

your logo here

[Register](#) [Log in](#)

[Home](#) [About](#) [Contact](#)

Index

[Create New](#)

Name	WarehouseAddress_Country	WarehouseAddress_State	WarehouseAddress_City	WarehouseAddress_Street	WarehouseAddress_Zip	WarehouseAddress_HousingNumber	
Warehouse 1	Japan	Province B	Xiangzhou	Warehouse Way	65147	1000	Edit Details Delete
Warehouse 2	United States	LA	Tabasco	Warehouse Way	91542	2000	Edit Details Delete
Warehouse 3	Brazil	Small	Beach Town	Warehouse Way	89245	3000	Edit Details Delete

© 2013 - My ASP.NET MVC Application

The database-first approach allows us to develop rapidly for an existing system, and makes code maintenance straightforward. When your code is capable of evolving with an application, you'll be a much more efficient programmer.

Tip You can find more information about ADO.NET [here](#).

Before you move on to the next lesson, as always, do your homework! See you soon...

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Interfaces and Extensions

Lesson Objectives

In this lesson you will:

- implement interfaces to increase the ability of collaborative development and create applications that require no outside, back-end code modifications.
- design robust functions that remove the requirement of database query knowledge.
- use extension methods to effectively abstract applications to create an ad-hoc type API.
- create custom extensions to extend already existing functionality of ASP.NET and Entity Framework.

After you complete this lesson, you will have designed and implemented a fully functioning web application API with data abstraction interfaces and user interface extensions, and created data extensions that further a web application API and abstraction.

Interfaces and MVC

IQueryable Interface

We'll start by creating a new ASP.NET MVC 4 Application named **Interfaces_Extensions** and choosing the **Internet Application** project type.

Before we begin to create a custom IQueryable interface, we need a database with tables in it. Open **Web.config** and add a connection string as shown:

CODE TO TYPE: Adding a Connection String To /Web.config

```
.  
.   
.   
    </configSections>  
    <connectionStrings>  
        <add name="DefaultConnection" connectionString="Data Source=.\\SQLEXPRESS;Initial Catalog=aspnet-Interfaces_Extensions-20130624172244;Integrated Security=SSPI" providerName="System.Data.SqlClient" />  
  
        <add name="InterfacesContext"  
            connectionString="Data Source=.\\SQLEXPRESS;Initial Catalog=Interface;AttachDBFilename=|DataDirectory|\\Interface.mdf;Integrated Security=SSPI;User Instance=true"  
            providerName="System.Data.SqlClient"  
        />  
    </connectionStrings>  
.   
.   
. 
```

We'll need a couple of Models so we can create some entities. Right-click the **Models** folder and add a new class named **Students** and another named **Courses**.

Add some properties to **/Models/Students.cs** and **/Models/Courses.cs** as shown:

CODE TO TYPE: Properties For /Models/Students.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Interfaces_Extensions.Models
{
    public class Students
    {
        [Key]
        [Display(Name = "Student ID")]
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Range(10000, 32000, ErrorMessage = "Invalid {0}. Must be between {2} and {1} digits in length.")]
        public Int16 Id { get; set; }

        [Display(Name="Student Name")]
        [StringLength(100, ErrorMessage="Invalid length for {0}. Must be between {2} and {1} characters.", MinimumLength = 10)]
        public string Name { get; set; }

        public virtual ICollection<Courses> courses { get; set; }
    }
}
```

CODE TO TYPE: Properties For Courses.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Interfaces_Extensions.Models
{
    public class Courses
    {
        [Key]
        [Display(Name="Course Number")]
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Range(100, 999, ErrorMessage = "Invalid {0}, Courses must be numbered between {1} and {2}")]
        public int Id { get; set; }

        [Display(Name="Course")]
        [StringLength(50, ErrorMessage="Invalid {0}, {0} Must be between {2} and {1} in length", MinimumLength=5)]
        public string CourseName { get; set; }

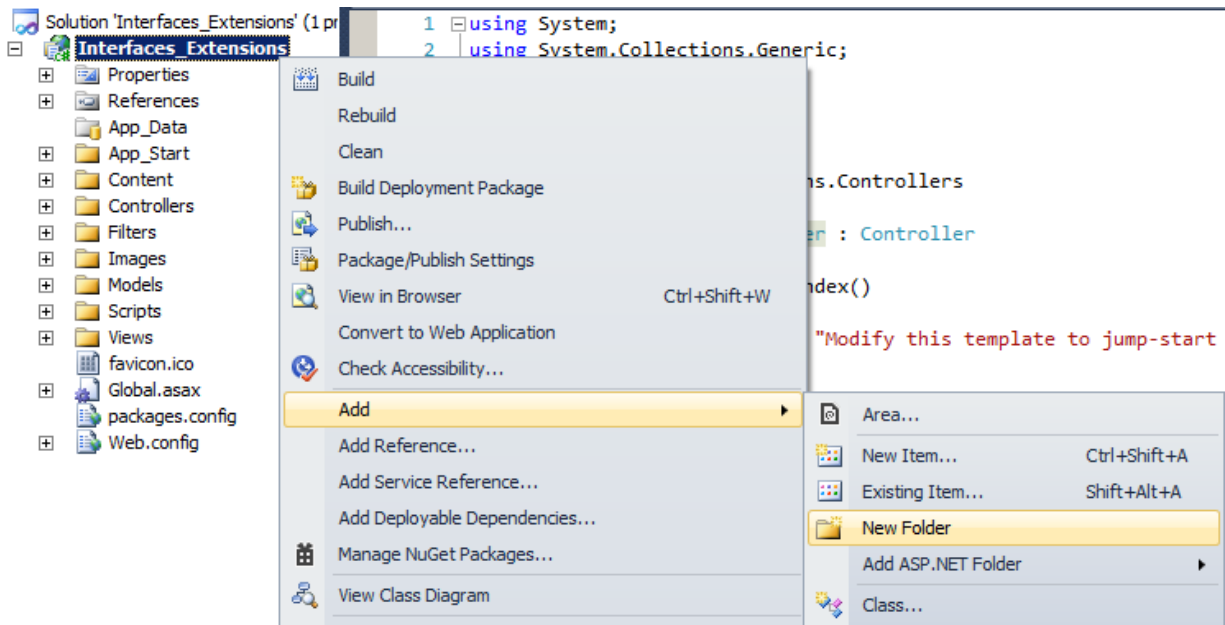
        [Display(Name="Credit Hours")]
        [Required(ErrorMessage="{0} is required for all new classes.")]
        [Range(0, 4, ErrorMessage = "The {0} has a range of {2} and {1}.")]
        public int Credits { get; set; }

        public virtual ICollection<Students> students { get; set; }
    }
}
```

Students can have many Courses and Courses have many Students, so we created a many-to-many relationship between our entities.

Now add a folder to hold our Database Context and Initializer. Right-click the **Interfaces_Extensions**

project and select **Add | New Folder** and name it **IE_Context**.



Add a new class named **InterfacesContext** to this folder.

Now set up our context. Modify **/IE_Context/InterfacesContext.cs** as shown:

CODE TO TYPE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using Interfaces_Extensions.Models;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace Interfaces_Extensions.IE_Context
{
    public class InterfacesContext : DbContext
    {
        public DbSet<Courses> Courses { get; set; }
        public DbSet<Students> Students { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

Create a new file to seed our entities. Add a class to the **IE_Context** folder named **InterfacesInitializer**.

Modify **/IE_Context/InterfacesInitializer.cs** file to seed our entities as shown (you can copy and past this large code block if you like):

CODE TO TYPE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using Interfaces_Extensions.Models;
using Interfaces_Extensions.IE_Context;

namespace Interfaces_Extensions.IE_Context
{
    public class InterfacesInitializer : DropCreateDatabaseAlways<InterfacesContext>
    {
        protected override void Seed(InterfacesContext context)
        {
            var _courses = new List<Courses>()
            {
                new Courses { Id = 222, CourseName = "Physics I", Credits = 3, students = new List<Students>() },
                new Courses { Id = 101, CourseName = "Psychology", Credits = 3, students = new List<Students>() },
                new Courses { Id = 144, CourseName = "Business Philosophy", Credits = 3, students = new List<Students>() },
                new Courses { Id = 296, CourseName = "Calculus I", Credits = 4, students = new List<Students>() },
                new Courses { Id = 442, CourseName = "Physics II", Credits = 3, students = new List<Students>() },
                new Courses { Id = 111, CourseName = "English I", Credits = 3, students = new List<Students>() },
                new Courses { Id = 235, CourseName = "Artistic Expression", Credits = 2, students = new List<Students>() },
                new Courses { Id = 599, CourseName = "CSE Ind. Study", Credits = 1, students = new List<Students>() },
                new Courses { Id = 485, CourseName = "Structural Dynamics", Credits = 4, students = new List<Students>() },
                new Courses { Id = 386, CourseName = "Bioinformatics", Credits = 4, students = new List<Students>() }
            };
            _courses.ForEach(x => context.Courses.Add(x));
            context.SaveChanges();

            var _students = new List<Students>()
            {
                new Students { Id = 10025, Name = "Michael Hadsworth", courses = new List<Courses>() },
                new Students { Id = 10435, Name = "Arielle Highsmith", courses = new List<Courses>() },
                new Students { Id = 11324, Name = "Daniel Hartfords", courses = new List<Courses>() },
                new Students { Id = 30982, Name = "Crystal Hingerfield", courses = new List<Courses>() },
                new Students { Id = 29051, Name = "Shawndra Kiljoy", courses = new List<Courses>() },
                new Students { Id = 20058, Name = "Yuri Goferfeldt", courses = new List<Courses>() },
                new Students { Id = 10598, Name = "Andrew Bandelsy", courses = new List<Courses>() },
                new Students { Id = 14568, Name = "Randolph Nomenture", courses = new List<Courses>() },
                new Students { Id = 31258, Name = "Barry Mantilok", courses = new List<Courses>() },
                new Students { Id = 10852, Name = "Gabriel Jones", courses = new List<Courses>() },
                new Students { Id = 30125, Name = "Caesar Juneseth", courses = new List<Courses>() },
                new Students { Id = 31215, Name = "Rebecca Vantry", courses = new List<Courses>() }
            };
            _students.ForEach(x => context.Students.Add(x));
            context.SaveChanges();
        }
    }
}
```

```

w List<Courses>() }
    };
    _students.ForEach(x => context.Students.Add(x));
    context.SaveChanges();

    _courses[0].students.Add(_students[0]);
    _courses[3].students.Add(_students[0]);
    _courses[2].students.Add(_students[0]);
    _courses[8].students.Add(_students[0]);

    _courses[9].students.Add(_students[1]);
    _courses[5].students.Add(_students[1]);
    _courses[7].students.Add(_students[1]);
    _courses[4].students.Add(_students[1]);

    _courses[8].students.Add(_students[2]);
    _courses[1].students.Add(_students[2]);
    _courses[6].students.Add(_students[2]);
    _courses[3].students.Add(_students[2]);

    _courses[0].students.Add(_students[3]);
    _courses[9].students.Add(_students[3]);
    _courses[5].students.Add(_students[3]);
    _courses[4].students.Add(_students[3]);

    _courses[8].students.Add(_students[4]);
    _courses[4].students.Add(_students[4]);
    _courses[7].students.Add(_students[4]);
    _courses[2].students.Add(_students[4]);

    _courses[9].students.Add(_students[5]);
    _courses[3].students.Add(_students[5]);
    _courses[1].students.Add(_students[5]);
    _courses[6].students.Add(_students[5]);

    _courses[0].students.Add(_students[6]);
    _courses[2].students.Add(_students[6]);
    _courses[4].students.Add(_students[6]);
    _courses[6].students.Add(_students[6]);

    _courses[8].students.Add(_students[7]);
    _courses[0].students.Add(_students[7]);
    _courses[1].students.Add(_students[7]);
    _courses[3].students.Add(_students[7]);

    _courses[5].students.Add(_students[8]);
    _courses[7].students.Add(_students[8]);
    _courses[9].students.Add(_students[8]);
    _courses[1].students.Add(_students[8]);

    _courses[0].students.Add(_students[9]);
    _courses[2].students.Add(_students[9]);
    _courses[4].students.Add(_students[9]);
    _courses[6].students.Add(_students[9]);

    _courses[8].students.Add(_students[10]);
    _courses[0].students.Add(_students[10]);
    _courses[1].students.Add(_students[10]);
    _courses[3].students.Add(_students[10]);

    _courses[5].students.Add(_students[11]);
    _courses[7].students.Add(_students[11]);
    _courses[9].students.Add(_students[11]);
    _courses[0].students.Add(_students[11]);
    context.SaveChanges();
}
}

```

```
}
```

We want to make sure that our project can see our entity context and initializer. Modify **/Global.asax** and add **using** statements and a new method to the **Application_Start** function as shown:

CODE TO TYPE:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Interfaces_Extensions.IE_Context;
using System.Data.Entity;

namespace Interfaces_Extensions
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            AuthConfig.RegisterAuth();
            Database.SetInitializer<InterfacesContext>(new InterfacesInitializer
());
        }
    }
}
```

Now add a controller for the Courses entity. Right-click the **Controllers** folder, select **Add | Controller**. Set the Name to **CoursesController**; the Template to **MVC controller with read/write actions and views, using Entity Framework**; the Model class to **Courses (Interfaces_Extensions.Models)**; the Data context class to **InterfacesContext (Interfaces_Extensions.IE_Context)**; and the Views to **Razor (CSHTML)**:

Add Controller

Controller name:
CoursesController

Scaffolding options

Template:
MVC controller with read/write actions and views, using Entity Framework

Model class:
Courses (Interfaces_Extensions.Models)

Data context class:
InterfacesContext (Interfaces_Extensions.IE_Context)

Views:
Razor (CSHTML)

Advanced Options...

Add Cancel

Create the **StudentsController** using the same options as before, but instead of setting the Model class to **Courses**, set it to **Students**:

Add Controller

Controller name:
StudentsController

Scaffolding options

Template:
MVC controller with read/write actions and views, using Entity Framework

Model class:
Students (Interfaces_Extensions.Models)

Data context class:
InterfacesContext (Interfaces_Extensions.IE_Context)

Views:
Razor (CSHTML)

Advanced Options...

Add Cancel

Now, let's add a couple of links to **/Views/Shared/_Layout.cshtml**:

CODE TO TYPE:

```
.  
.   
.   
<div class="float-right">  
  <section id="login">  
    @Html.Partial("_LoginPartial")  
  </section>  
  <nav>  
    <ul id="menu">  
      <li>@Html.ActionLink("Home", "Index", "Home")</li>  
      <li>@Html.ActionLink("About", "About", "Home")</li>  
      <li>@Html.ActionLink("Contact", "Contact", "Home")</li>  
      <li>@Html.ActionLink("Courses", "Index", "Courses")</li>  
      <li>@Html.ActionLink("Students", "Index", "Students")</li>  
    </ul>  
  </nav>  
</div>  
.   
.   
. 
```

Note We won't be using the About or Contact views.

Let's build and run our project.



and and navigate to the Courses and Students pages to verify that our database was properly seeded. They'll look like this:

your logo here

Index

[Create New](#)

Course	Credit Hours	
Psychology	3	Edit Details Delete
English I	3	Edit Details Delete
Business Philosophy	3	Edit Details Delete
Physics I	3	Edit Details Delete
Artistic Expression	2	Edit Details Delete
Calculus I	4	Edit Details Delete
Bioinformatics	4	Edit Details Delete
Physics II	3	Edit Details Delete
Structural Dynamics	4	Edit Details Delete
CSE Ind. Study	1	Edit Details Delete

your logo here

Index

[Create New](#)

Student Name

Michael Hadsworth	Edit Details Delete
Arielle Highsmith	Edit Details Delete
Andrew Bandelsy	Edit Details Delete
Gabriel Jones	Edit Details Delete
Daniel Hartfords	Edit Details Delete
Randolph Nomenture	Edit Details Delete
Yuri Goferfeldt	Edit Details Delete
Shawndra Kiljoy	Edit Details Delete
Caesar Juneseth	Edit Details Delete
Crystal Hingerfield	Edit Details Delete
Rebecca Vantry	Edit Details Delete
Barry Mantilok	Edit Details Delete

© 2013 - My ASP.NET MVC Application

Now add a folder for our custom interfaces. Right-click the **Interfaces_Extensions** project and choose **Add | New Folder**, and name it **Interfaces**.

In the **/Interfaces** folder, add a new Class named **ICourseRepository**. This class will be our Course Repository interface.

Modify **/Interfaces/ICourseRepository.cs** to define some methods for our new repository:

CODE TO TYPE: Methods for ICourseRepository.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Linq.Expressions;
using Interfaces_Extensions.Models;
using Interfaces_Extensions.IE_Context;

namespace Interfaces_Extensions.Interfaces
{
    public class interface ICourseRepository : IDisposable
    {
        IEnumerable<Courses> GetAllCourses();
        IQueryable<Courses> SearchCourses(Expression<Func<Courses, bool>> query)
        ;
        IQueryable<Courses> CheckIfCourseExists(Expression<Func<Courses, bool>>
query, string courseValue);
        IQueryable<Courses> GetCourseRange(int val_1, int val_2);
        void AddCourse(Courses course);
        void EditCourse(Courses course);
        void DeleteCourse(int id);
        Courses GetByCourseNumber(int id);
        void SaveChanges();
    }
}
```

The **interface** keyword defines a class in which all methods must be overridden by an inheriting class, in this case, the CourseRepository.

We use the return type of IQueryable<Courses> on some of our interface methods so that when we return our objects we can iterate over the collection of items returned. This will become clearer when we implement these methods in our controllers.

The parameters of our IQueryable return types is a Lambda expression, so we can pass in a query using the Method syntax. This will also become clearer when we implement these methods in our controllers.

You've seen the remaining methods in earlier lessons. We take a Courses object and perform a basic operation on it. We either add a new item to the table, delete an item, or retrieve a single item.

Let's implement our Courses repository so we can use our Courses interface. Right-click the main project directory and add a new folder named **Repositories**, and in that folder, add a new class named **CoursesRepository**.

Now we'll implement our interface and its methods, discussing each piece as it's implemented:

CODE TO TYPE: Defining IQueryable<Courses> SearchCourses in /Repositories/CoursesRepository.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data;
using System.Linq.Expressions;
using Interfaces_Extensions.IE_Context;
using Interfaces_Extensions.Models;
using Interfaces_Extensions.Interfaces;

namespace Interfaces_Extensions.Repositories
{
    public class CoursesRepository : ICourseRepository, IDisposable
    {
        private InterfacesContext context;

        public CoursesRepository(InterfacesContext context)
        {
            this.context = context;
        }

        public IEnumerable<Courses> GetAllCourses()
        {
            return context.Courses.ToList();
        }

        public IQueryable<Courses> SearchCourses(Expression<Func<Courses, bool>>
query)
        {
            return context.Courses.Where(query);
        }
    }
}
```

Here, we declare the method public and accept an expression that uses a lambda function that returns a boolean value. We name the parameter **query**, and **query** actually holds the method syntax that we'll use in the function calls.

When we're able to implement them in our controller you'll get a better idea of how these work. Modify the code as shown:

CODE TO TYPE: Defining IQueryable<Courses> CheckIfCourseExists in CoursesRepository.cs

```
.
.
.
public IQueryable<Courses> SearchCourses(Expression<Func<Courses, bool>> query)
{
    return context.Courses.Where(query);
}

public IQueryable<Courses> CheckIfCourseExists(Expression<Func<Courses, bool>> q
query, string courseValue)
{
    return context.Courses.Where(obj => obj.CourseName.Contains(courseValue));
}
.
.
.
```

In this method, we define another Expression, but this time we pass a string as a second parameter. We return all the Courses with names that contain the string courseValue. Modify your code as shown:

CODE TO TYPE: Defining IQueryable<Courses> GetCourseRange in CoursesRepository.cs

```
.  
.br/>.br/>public IQueryable<Courses> CheckIfCourseExists(Expression<Func<Courses, bool>> query, string courseValue)  
{  
    return context.Courses.Where(obj => obj.CourseName.Contains(courseValue));  
}  
  
public IQueryable<Courses> GetCourseRange(int val_1, int val_2)  
{  
    return context.Courses.Where(obj => (obj.Id > val_1 && obj.Id < val_2)).AsQueryable();  
}  
.br/>.br/.
```


This method is a little different from those we've used previously. We use the int value that is passed in to our expression and the Courses object to perform a query. We also pass in a second integer val_2 to perform a search between two values. Upon retrieving the values from our query, we cast it as queryable so we receive an enumerator with our return type:

CODE TO TYPE: Defining remaining methods in CoursesRepository.cs

```
.  
.br/>.br/>  
    public void AddCourse(Courses course)  
    {  
        context.Courses.Add(course);  
    }  
  
    public void EditCourse(Courses course)  
    {  
        context.Entry(course).State = EntityState.Modified;  
    }  
  
    public void DeleteCourse(int id = 0)  
    {  
        Courses course = context.Courses.Find(id);  
        context.Courses.Remove(course);  
    }  
  
    public Courses GetByCourseNumber(int id = 0)  
    {  
        Courses course = context.Courses.Find(id);  
        return course;  
    }  
  
    public void SaveChanges()  
    {  
        context.SaveChanges();  
    }  
  
    private bool ContextDisposed = false;  
  
    protected virtual void Dispose(bool disposal)  
    {  
        if (!this.ContextDisposed)  
        {  
            if (disposal)  
            {  
                context.Dispose();  
            }  
        }  
        this.ContextDisposed = true;  
    }  
  
    public void Dispose()  
    {  
        Dispose(true);  
        GC.SuppressFinalize(this);  
    }  
  
    }  
}
```

Now that we have our interface and repository designed using the IQueryable interface, let's run the application again.



and . If we navigate to the Courses index page, the application works the same way as it did before because we haven't implemented the repository in the controller:

your logo here

Index

[Create New](#)

Course	Credit Hours	
Psychology	3	Edit Details Delete
English I	3	Edit Details Delete
Business Philosophy	3	Edit Details Delete
Physics I	3	Edit Details Delete
Artistic Expression	2	Edit Details Delete
Calculus I	4	Edit Details Delete
Bioinformatics	4	Edit Details Delete
Physics II	3	Edit Details Delete
Structural Dynamics	4	Edit Details Delete
CSE Ind. Study	1	Edit Details Delete

© 2013 - My ASP.NET MVC Application

In order to be able to use the repository, modify **/Controllers/CourseController.cs** as shown (we'll introduce the functions one at a time and explain them as we go):

CODE TO TYPE: Switching Index action method of CourseController.cs over to use the Courses Repository

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Interfaces_Extensions.Models;
using Interfaces_Extensions.IE_Context;
using Interfaces_Extensions.Repositories;
using Interfaces_Extensions.Interfaces;

namespace Interfaces_Extensions.Controllers
{
    public class CoursesController : Controller
    {
        private InterfacesContext context = new InterfacesContext();
        private ICourseRepository courseRepo;

        public CoursesController()
        {
            this.courseRepo = new CoursesRepository(new InterfacesContext());
        }

        public CoursesController(ICourseRepository courseRepo)
        {
            this.courseRepo = courseRepo;
        }

        //
        // GET: /Courses/

        public ActionResult Index()
        {
            var courses = courseRepo.GetAllCourses();
            return View(coursesdb.Courses.ToList());
        }
    }
}
```

We have replaced the original Index action method with the Courses Repository to retrieve Courses entries:

CODE TO TYPE: Switching Details action method of CourseController.cs over to use the Courses Repository

```
.
.
.
public ActionResult Details(int id = 0)
{
    Courses courses = courseRepo.GetByCourseNumber(id)db.Courses.Find(id);
    if (courses == null)
    {
        return HttpNotFound();
    }
    return View(courses);
}
.
.
.
```

We replaced the default code with the repository equivalent of the `db.Courses.Find(id)`:

CODE TO TYPE: Switching Create action method of CourseController.cs over to use the Courses Repository

```
.  
.   
.   
//  
// GET: /Courses/Create  
  
public ActionResult Create()  
{  
    return View();  
}  
  
//  
// POST: /Courses/Create  
  
[HttpPost]  
public ActionResult Create(Courses courses)  
{  
    if (ModelState.IsValid)  
    {  
        db.Courses.Add(courses)courseRepo.AddCourse(courses);  
        db.SaveChanges()courseRepo.SaveChanges();  
        return RedirectToAction("Index");  
    }  
  
    return View(courses);  
}  
.   
.   
. 
```

We replaced the default code with our new repository-based code:

CODE TO TYPE: Switching Edit action method of CourseController.cs over to use the Courses Repository

```
.  
.   
.   
//   
// GET: /Courses/Edit/5  
  
public ActionResult Edit(int id = 0)  
{  
    Courses courses = db.Courses.Find(id)courseRepo.GetByCourseNumber(id);  
    if (courses == null)  
    {  
        return HttpNotFound();  
    }  
    return View(courses);  
}  
  
//   
// POST: /Courses/Edit/5  
  
[HttpPost]  
public ActionResult Edit(Courses courses)  
{  
    if (ModelState.IsValid)  
    {  
        db.Entry(courses).State = EntityState.ModifiedcourseRepo.EditCourse(courses);  
        db.SaveChanges()courseRepo.SaveChanges();  
        return RedirectToAction("Index");  
    }  
    return View(courses);  
}  
.   
.   
.
```

We replaced the default code with our repository-based code in the Edit action method:

CODE TO TYPE: Switching Delete action method of CourseController.cs over to use the Courses Repository

```
.
.
.
//
// GET: /Courses/Delete/5

public ActionResult Delete(int id = 0)
{
    Courses courses = db.Courses.Find(id)courseRepo.GetByCourseNumber(id);
    if (courses == null)
    {
        return HttpNotFound();
    }
    return View(courses);
}

//
// POST: /Courses/Delete/5

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    Courses courses = db.Courses.Find(id);
    db.Courses.Remove(courses)courseRepo.DeleteCourse(id);
    db.SaveChanges()courseRepo.SaveChanges();
    return RedirectToAction("Index");
}

protected override void Dispose(bool disposing)
{
    db.Dispose()courseRepo.Dispose();
    base.Dispose(disposing);
}
```

Before we go forward, modify one view item to fulfill our Course entities' requirements. In the **/Views/Courses/** folder, add some code to the **Edit.cshtml** and **Create.cshtml** files:

CODE TO TYPE: Adding Course Number input for /Views/Courses/Edit.cshtml Courses view

```
.
.
.
<fieldset>
    <legend>Courses</legend>
    <div class="editor-label">
        @Html.LabelFor(model => model.Id)
    </div>
    <div class="editor-field">
        @Html.TextBoxFor(model => model.Id)
    </div>
    @Html.HiddenFor(model => model.Id)

    <div class="editor-label">
        @Html.LabelFor(model => model.CourseName)
    </div>
.
.
.
```



and



. Now the Course Number appears in the Edit view:

your logo here

Edit

Course Number

Course

Credit Hours

[Back to List](#)

© 2013 - My ASP.NET MVC Application

Enter the course number manually, as defined in our entity properties by modifying **/Views/Courses/Create.cshtml** view:

CODE TO TYPE: Adding Course Number input for Create.cshtml Courses view

```
.  
.   
.   
<legend>Courses</legend>  
  
<div class="editor-label">  
    @Html.LabelFor(model => model.Id)  
    @Html.ValidationMessageFor(model => model.id)  
</div>  
<div class="editor-field">  
    @Html.TextBoxFor(model => model.Id)  
</div>  
<div class="editor-label">  
    @Html.LabelFor(model => model.CourseName)  
</div>  
.   
.   
.
```



and



your logo here

Create

Course Number

Course

Credit Hours

Create

[Back to List](#)

© 2013 - My ASP.NET MVC Application

Now we can use our IQueryable objects. Open Courses controller and create a new action method named **QueryCourses**:

CODE TO TYPE: Implementing IQueryable method QueryCourses

```
.
.
.
// POST: /Courses/Delete/5

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    courseRepo.DeleteCourse(id);
    courseRepo.SaveChanges();
    return RedirectToAction("Index");
}

public ActionResult QueryCourses(string queryString = "")
{
    if (!String.IsNullOrEmpty(queryString))
    {
        var courseSearch = courseRepo.SearchCourses(x => x.CourseName.ToUpper()
== queryString.ToUpper());
        return View("Index", courseSearch);
    }
    else
    {
        var courses = courseRepo.GetAllCourses();
        return View("Index", courses);
    }
}
.
.
.
```

courseRepo.SearchCourses(x => x.CourseName.ToUpper() == queryString.ToUpper()) sends a LINQ to Entity query to the SearchCourses IQueryable function that we defined in our ICoursesRepository and CoursesRepository. This function returns an IQueryable Courses object.

Add an area to **/Views/Courses/Index.cshtml** where we can search for a course:

CODE TO TYPE: Adding search box to Courses Index.cshtml

```
@model IEnumerable<Interfaces_Extensions.Models.Courses>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
<div>
    @using(Html.BeginForm("QueryCourses", "Courses", FormMethod.Post))
    {
        @Html.TextBox("queryString", null, new { placeholder = "Enter Course Search" })
        <input type="submit" value="Search" />
    }
</div>
<p>
    @Html.ActionLink("Return to Index", "Index")
    @Html.ActionLink("Create New", "Create")
</p>
<table>
```



and



and perform a search to see if our new action method works.

Index

[Return to Index](#)
[Create New](#)

Course	Credit Hours	
Psychology	3	Edit Details Delete

© 2013 - My ASP.NET MVC Application

This search method is a bit strict. It requires a user to enter the exact course name. Our other IQueryable function offers more flexibility. Let's create a **LooseQuery** action method in **/Controllers/CoursesController**:

CODE TO TYPE:

```

.
.
.
public ActionResult QueryCourses(string queryString = "")
{
    if (!String.IsNullOrEmpty(queryString))
    {
        var courseSearch = courseRepo.SearchCourses(x => x.CourseName.ToUpper()
== queryString.ToUpper());
        return View("Index", courseSearch);
    }
    else
    {
        var courses = courseRepo.GetAllCourses();
        return View("Index", courses);
    }
}

public ActionResult LooseQuery(string queryString = "")
{
    if (!String.IsNullOrEmpty(queryString))
    {
        var courseSearch = courseRepo.CheckIfCourseExists(x => x.CourseName.ToUp
per() == queryString.ToUpper(), queryString);
        return View("Index", courseSearch);
    }
    else
    {
        var courses = courseRepo.GetAllCourses();
        return View("Index", courses);
    }
}
.
.
.

```

courseRepo.CheckIfCourseExists(x => x.CourseName.ToUpper() == queryString.ToUpper(), queryString) implements the same query as before, but we pass queryString to the function call as defined in our ICourseRepository and CoursesRepository. Now if a course contains any part of the string we

submitted as the secondary parameter, it will be displayed.

OBSERVE: ICourseRepository and CoursesRepository definition of CheckIfCourseExists

```
IQueryable<Courses> CheckIfCourseExists(Expression<Func<Courses, bool>> query, string courseValue);  
.  
.  
.  
public IQueryable<Courses> CheckIfCourseExists(Expression<Func<Courses, bool>> query, string courseValue)  
{  
    return context.Courses.Where(obj => obj.CourseName.Contains(courseValue));  
}
```

Update the search box in **/Views/Courses/Index.cshtml** to reference our new, more robust search query:

CODE TO TYPE:

```
.  
.  
.  
<h2>Index</h2>  
<div>  
    @using(Html.BeginForm("QueryCoursesLooseQuery", "Courses", FormMethod.Post))  
    {  
        @Html.TextBox("queryString", null, new { placeholder = "Enter Course Search" })  
        <input type="submit" value="Search" />  
    }  
</div>  
.  
.  
.
```



and



and try the new search function:

Index

[Return to Index](#) [Create New](#)

Course	Credit Hours	
Business Philosophy	3	Edit Details Delete
Bioinformatics	4	Edit Details Delete

© 2013 - My ASP.NET MVC Application

Our new query returns courses that contain a full or partial match of the string entered in the search box.

Now implement the **GetCourseRange** function by passing in a query and a secondary integer to compare. In **/Controllers/CoursesController.cs**, create a new action method named **GetRange** as shown:

CODE TO TYPE: Implementing GetCourseRange in the GetRange() action method

```
.  
. .  
.  
public ActionResult GetRange(string val_1 = "", string val_2 = "")  
{  
    int val_a, val_b;  
    bool tryParse = int.TryParse(val_1, out val_a);  
    if (tryParse && val_a > 99 && val_a < 1000)  
    {  
        tryParse = int.TryParse(val_2, out val_b);  
        if (tryParse && val_b > 99 && val_b < 1000)  
        {  
            if (val_a > val_b)  
            {  
                int temp = val_a;  
                val_a = val_b;  
                val_b = temp;  
            }  
            var CourseRange = courseRepo.GetCourseRange(val_a, val_b).ToList();  
            return View("Index", CourseRange);  
        }  
        else  
        {  
            ViewBag.Error = "Second input value is invalid. Must be an integer:  
" + val_2;  
            return View("Index", courseRepo.GetAllCourses());  
        }  
    }  
    else  
    {  
        ViewBag.Error = "First input value is invalid. Must be a valid integer:  
" + val_1;  
        return View("Index", courseRepo.GetAllCourses());  
    }  
}  
. .  
.
```

OBSERVE:

```
courseRepo.GetCourseRange(val_a, val_b).ToList();
```

courseRepo.GetCourseRange takes two integers rather than a query. This method lets us have more control over the way a user interacts with our interfaces. Instead of letting users create their own syntax, they are restricted to passing only the values required to complete the operation. Then we call **ToList()** to create an item over which we can iterate.

Modify **/Views/Courses/Index.cshtml** again so we can pass in a range of values to our action method.

Note We pass in strings and then parse them there for improved error checking and greater flexibility.


Modify your code as shown:

CODE TO TYPE: Adding search boxes to Index.cshhtml to allow a user to search a course range

```
<h2>Index</h2>

<div class="error">
    @if (!String.IsNullOrEmpty(ViewBag.Error))
    {
        @ViewBag.Error
    }
</div>
<div>
    @using(Html.BeginForm("LooseQuery", "Courses", FormMethod.Post))
    {
        @Html.TextBox("queryString", null, new { placeholder = "Enter Course Search" })
        <input type="submit" value="Search" />
    }
</div>
<div>
    @using (Html.BeginForm("GetRange", "Courses", FormMethod.Post))
    {
        <text><b>Search by course range: </b></text>@Html.TextBox("val_1", null,
        new { @style = "width:30px;" })<text> <b>T0</b>
        </text>@Html.TextBox("val_2", null, new { @style = "width:30px;" })
        <input type="submit" value="Search Courses" />
    }
</div>
.
```



and  and enter both valid and invalid values to verify that our error checking and query work as expected:

Index

Search by course range:

TO

[Return to Index](#) [Create New](#)

Course	Credit Hours	
Psychology	3	Edit Details Delete
English I	3	Edit Details Delete
Business Philosophy	3	Edit Details Delete
Physics I	3	Edit Details Delete
Artistic Expression	2	Edit Details Delete
Calculus I	4	Edit Details Delete
Bioinformatics	4	Edit Details Delete
Physics II	3	Edit Details Delete
Structural Dynamics	4	Edit Details Delete
CSE Ind. Study	1	Edit Details Delete

Invalid input in box 1:

Index

First input value is invalid. Must be a valid integer: 10

Search by course range: TO

[Return to Index](#) [Create New](#)

Course	Credit Hours	
Psychology	3	Edit Details Delete
English I	3	Edit Details Delete
Business Philosophy	3	Edit Details Delete
Physics I	3	Edit Details Delete
Artistic Expression	2	Edit Details Delete
Calculus I	4	Edit Details Delete
Bioinformatics	4	Edit Details Delete
Physics II	3	Edit Details Delete
Structural Dynamics	4	Edit Details Delete
CSE Ind. Study	1	Edit Details Delete

Invalid input in box 2:

Index

Second input value is invalid. Must be an integer: 1000

Search by course range: TO

[Return to Index](#) [Create New](#)

Course	Credit Hours	
Psychology	3	Edit Details Delete
English I	3	Edit Details Delete
Business Philosophy	3	Edit Details Delete
Physics I	3	Edit Details Delete
Artistic Expression	2	Edit Details Delete
Calculus I	4	Edit Details Delete
Bioinformatics	4	Edit Details Delete
Physics II	3	Edit Details Delete
Structural Dynamics	4	Edit Details Delete
CSE Ind. Study	1	Edit Details Delete

Valid input results:

Index

Search by course range:

TO

[Return to Index](#) [Create New](#)

Course	Credit Hours	
Psychology	3	Edit Details Delete
English I	3	Edit Details Delete
Business Philosophy	3	Edit Details Delete

© 2013 - My ASP.NET MVC Application

Other Queryable interfaces can be used the same way. The Interface queryable type objects can all be extended. They have been created to be flexible.

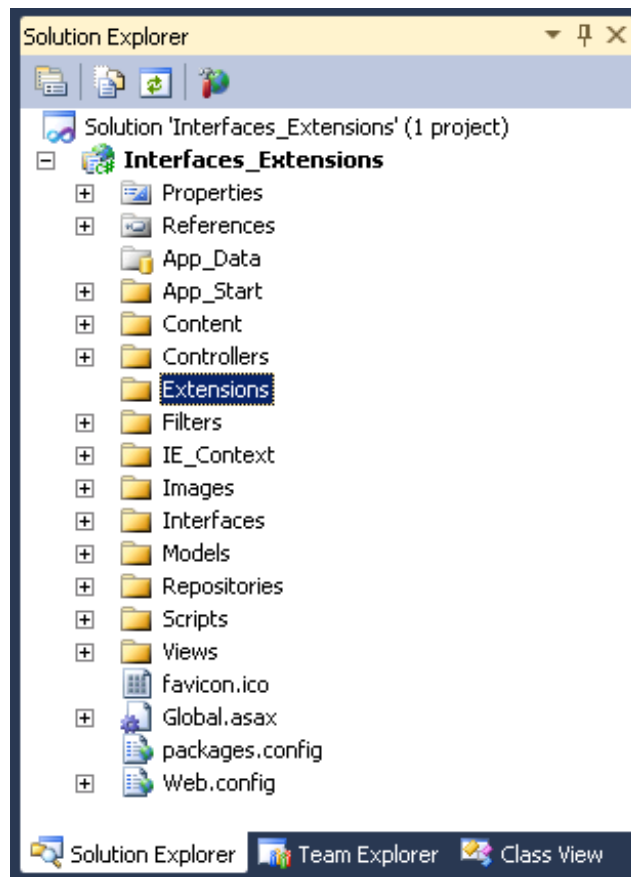
Tip For a listing of queryable methods and their arguments, see [MSDN Queryable Methods](#).

Extension Methods

Extension methods give us the freedom to create functions and override different elements of ASP.NET C#'s built-in objects. We can create our own HTML Helper methods, Expressions, Queries, and so on.

Create a new folder named **Extensions** to hold the extension methods we want to build.

Our Solution Explorer looks like this now:



Create an HtmlHelper extension. We'll create an ActionLink that displays links that are highlighted for emphasis.

Move the HTML extensions into a separate folder to keep them organized. In the **/Extensions** folder, add a new folder named **HtmlExtensions**.

In the **/HtmlExtensions** folder, add a class named **HtmlHelperExtensions**.

Now, add the required **using** statements and an HtmlHelper extension method that will outline the current page in the navigation menu. Modify **/Extensions/HTMLExtensions/HtmlHelperExtensions.cs** as shown:

CODE TO TYPE: Required using statements for /Extensions/HTMLExtensions/HtmlHelperExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Linq.Expressions;
using System.Web.Mvc;
using System.Web.Mvc.Html;
using System.Web.Routing;

namespace Interfaces_Extensions.Extensions.HtmlExtensions
{
    public static class HtmlHelperExtensions
    {
        public static MvcHtmlString isCurrentPageActionLink(this HtmlHelper html, string text, string action, string controller)
        {
            var _action_ = html.ViewContext.RouteData.GetRequiredString("action");
            var _controller_ = html.ViewContext.RouteData.GetRequiredString("controller");

            if (action == _action_ && controller == _controller_)
            {
                TagBuilder anchorTag = new TagBuilder("a");
                anchorTag.Attributes["href"] = "#";
                anchorTag.AddCssClass("current-link");
                anchorTag.SetInnerText(text);
                return MvcHtmlString.Create(anchorTag.ToString(TagRenderMode.Normal));
            }
            else
            {
                return html.ActionLink(text, action, controller);
            }
        }
    }
}
```

In our **isCurrentPageActionLink** extension method, we pass in arguments just as if we were creating an actual **Html.ActionLink**, because the ActionLink method is a function of **MvcHtmlString**.

Let's discuss our method:

OBSERVE:

```
public static MvcHtmlString isCurrentPageActionLink(this HtmlHelper html, string text, string action, string controller)
{
    var _action_ = html.ViewContext.RouteData.GetRequiredString("action");
    var _controller_ = html.ViewContext.RouteData.GetRequiredString("controller");

    if (action == _action_ && controller == _controller_)
    {
        TagBuilder anchorTag = new TagBuilder("a");
        anchorTag.Attributes["href"] = "#";
        anchorTag.AddCssClass("current-link");
        anchorTag.SetInnerText(text);
        return MvcHtmlString.Create(anchorTag.ToString(TagRenderMode.Normal));
    }
    else
    {
        return html.ActionLink(text, action, controller);
    }
}
```

- **MvcHtmlString** returns an HTML encoded string that should not be encoded again.
- **html.ViewContext.RouteData.GetRequiredString(string)** retrieves a string from a set of route data from the ViewContext class. (ViewContext is responsible for encapsulating information related to views).
- **TagBuilder anchorTag = new TagBuilder("a");** creates an anchor tag for our links so they are encoded properly. We use TagBuilder to create HTML tags.
- **anchorTag.Attributes["href"] = "#"** associates the **href** attribute (or any link property) with our anchorTag, so instead of allowing it to link back to itself, we render it as an empty link so we can't click the link, rerender the view, and reload all relevant data.
- **anchorTag.AddCssClass("current-link");** assigns a CSS class to the anchorTag element so that its style is different from that of other links.
- **MvcHtmlString.Create(anchorTag.ToString(TagRenderMode.Normal))** creates an MvcHtmlString object and sets the **TagRenderMode** to normal. TagRenderMode is an enumerator type object.

Note

Here's a list of TagRenderMode options and values:
Normal renders "<tag></tag>" (with attributes if defined)
StartTag renders "<tag>" (with attributes if defined)
EndTag renders "</tag>"
SelfClosing renders "<tag />"

Now that we have an HtmlExtension, we'll add a bit of CSS so our **current-link** property does something.

To add CSS, modify **/Content/site.css** as shown:

CODE TO TYPE: Adding definition of current-link to /Content/site.css

```
/* Custom CSS for HtmlHelperExtension */
a.current-link
{
    border:1px solid Red;
}

/*_____End CSS for HtmlHelperExtension_____*/

html {
    background-color: #e2e2e2;
    margin: 0;
    padding: 0;
}
.
.
.
```

We add a red border to the link for the view we are currently using. Now we need to change the ActionLinks in the **_Layout.cshtml** file so we know where we are:

CODE TO TYPE: Adding our isCurrentPageActionLink() to _Layout.cshtml

```
.
.
.
<nav>
    <ul id="menu">
        <li>@Html.isCurrentPageActionLink("Home", "Index", "Home")</li>
        <li>@Html.isCurrentPageActionLink("Courses", "Index", "Courses")</li>
        <li>@Html.isCurrentPageActionLink("Students", "Index", "Students")</li>
    </ul>
</nav>
.
.
.
```


Finally, add the HtmlHelperExtension namespace to **Web.config**—not the main Web.config—located in the Views folder:

CODE TO TYPE: Adding the namespace to /Views/Web.config

```
.  
.   
.   
<system.web.webPages.razor>  
  <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version=4  
.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />  
  <pages pageBaseType="System.Web.Mvc.WebViewPage">  
    <namespaces>  
      <add namespace="System.Web.Mvc" />  
      <add namespace="System.Web.Mvc.Ajax" />  
      <add namespace="System.Web.Mvc.Html" />  
      <add namespace="System.Web.Optimization"/>  
      <add namespace="System.Web.Routing" />  
      <add namespace="Interfaces_Extensions.Extensions.HtmlExtensions" />  
    </namespaces>  
  </pages>  
</system.web.webPages.razor>  
.   
.   
.
```

Run the application to see the new addition.



and , then navigate the main links and observe how they differ from previous runs.

Tip If the changes do not take effect, restart the sandbox environment.

your logo here

Register Log in

Home Courses Students

Home Page. Modify this template to jump-start your ASP.NET MVC application.

To learn more about ASP.NET MVC visit <http://asp.net/mvc>. The page features videos, tutorials, and samples to help you get the most from ASP.NET MVC. If you have any questions about ASP.NET MVC visit [our forums](#).

Note MSDN has a listing of all methods that have extensions and are available to the Entity Framework.

Create a new folder in the **Extensions** folder named **DisplayNameForExtensions** and add a new class to it named **DisplayNameForExtensionHelper**.

Modify this new file as shown (create methods named **DisplayNameForExtension**, **GetDisplayName**, and **getPropertyName**):

CODE TO TYPE: Defining the three methods necessary for the DisplayNameForExtension method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Linq.Expressions;
using System.Web.Mvc;
using System.Web.Mvc.Html;
using System.Web.Routing;

namespace Interfaces_Extensions.Extensions.DisplayNameForExtensions
{
    public static class DisplayNameForExtensionHelper
    {
        public static MvcHtmlString DisplayNameForExtension<ModelType, PropertyType>(this HtmlHelper<IEnumerable<ModelType>> html, Expression<Func<ModelType, PropertyType>> query)
        {
            var expression = GetDisplayName(html, query);
            return getPropertyName<ModelType>(expression);
        }

        public static string GetDisplayName<ModelType, ClassType, PropertyType>(HtmlHelper<ModelType> html, Expression<Func<ClassType, PropertyType>> query)
        {
            var expression = ExpressionHelper.GetExpressionText(query);
            expression = html.ViewContext.ViewData.TemplateInfo.GetFullHtmlFieldName(expression);
            return expression;
        }

        public static MvcHtmlString getPropertyName<ClassType>(string property)
        {
            var metadata = ModelMetadataProviders.Current.GetMetadataForProperty(() => Activator.CreateInstance<ClassType>(), typeof(ClassType), property);
            return new MvcHtmlString(metadata.DisplayName ?? typeof(ClassType).Name);
        }
    }
}
```

Our new extension method performs the method calls like this:

1. DisplayNameForExtension()
2. GetDisplayName()
3. getPropertyName()

getPropertyName() returns either metadata.DisplayName or typeof(ClassType).Name. Let's discuss this new syntax:

OBSERVE:

```
public static string GetDisplayName<ModelType, ClassType, PropertyType>(HtmlHelper<ModelType> html, Expression<Func<ClassType, PropertyType>> query)
{
    var expression = ExpressionHelper.GetExpressionText(query);
    expression = html.ViewContext.ViewData.TemplateInfo.GetFullHtmlFieldName(expression);
    return expression;
}

public static MvcHtmlString getPropertyName<ClassType>(string property)
{
    var metadata = ModelMetadataProviders.Current.GetMetadataForProperty(() => Activator.CreateInstance<ClassType>(), typeof(ClassType), property);
    return new MvcHtmlString(metadata.DisplayName ?? typeof(ClassType).Name);
}
```

- **ExpressionHelper.GetExpressionText(query)** returns the name of the model from the lambda expression that is passed into the method.
- **html.ViewContext.ViewData.TemplateInfo.GetFullHtmlFieldName(expression)** retrieves a fully qualified name from a field using an HTML attribute.
- **ModelMetadataProviders.Current.GetMetadataForProperty(() => Activator.CreateInstance<ClassType>(), typeof(ClassType), property)** retrieves the metadata for our ClassType property and returns the value in **property**. It requires a modelAccessor **((() => Activator.CreateInstance<ClassType>()))**, a type of container **(typeof(ClassType))**, and the name of a property **(property)**.

Let's put our new extension method to work in the **/Views/Students/Index.cshtml** file. Open that file and replace the old **DisplayNameFor** with our method **DisplayNameForExtension**:

CODE TO TYPE: Replacing DisplayNameFor with DisplayNameForExtension in /Views/Students/Index.cshtml

```
.
.
.
    <tr>
        <th>
            @Html.DisplayNameForExtension(model => model.Name)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
.
.
.
}
```

Add the namespace to **/Views/Web.config**:

CODE TO TYPE: Adding namespace for DisplayNameForExtension to /Views/Web.config

```
.
.
.
<pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
        <add namespace="System.Web.Mvc" />
        <add namespace="System.Web.Mvc.Ajax" />
        <add namespace="System.Web.Mvc.Html" />
        <add namespace="System.Web.Optimization"/>
        <add namespace="System.Web.Routing" />
        <add namespace="Interfaces_Extensions.Extensions.HtmlExtensions" />
        <add namespace="Interfaces_Extensions.Extensions.DisplayNameForExtensions" />
    </namespaces>
</pages>
.
.
.
```

Run the application to see the result of all of your hard work.



and



and navigate to the Students index page.

your logo here

Index

[Create New](#)

Student Name

Michael Hadsworth	Edit Details Delete
Arielle Highsmith	Edit Details Delete
Andrew Bandelsy	Edit Details Delete
Gabriel Jones	Edit Details Delete
Daniel Hartfords	Edit Details Delete
Randolph Nomenture	Edit Details Delete
Yuri Goferfeldt	Edit Details Delete
Shawndra Kiljoy	Edit Details Delete
Caesar Juneseth	Edit Details Delete
Crystal Hingerfield	Edit Details Delete
Rebecca Vantry	Edit Details Delete
Barry Mantilok	Edit Details Delete

© 2013 - My ASP.NET MVC Application

It works just as planned. It displays all the students' names and nothing has changed on the front end.

Filtering Using Extension Methods

Now implement an Interface and repository for the Students entity. Create a new class in the **/Interfaces** folder named **IStudentsRepository**.

Modify the file as shown to implement the repository with the interface already defined:

CODE TO TYPE: Defining /Interfaces/StudentsRepository.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Linq.Expressions;
using Interfaces_Extensions.Models;
using Interfaces_Extensions.IE_Context;
namespace Interfaces_Extensions.Interfaces
{
    public classinterface IStudentsRepository : IDisposable
    {
        IEnumerable<Students> getStudents();
        IQueryable<Students> OrderByName(char letter);
        IQueryable<Students> Filter(string filterParams = "");
    }
}
```

We have an Interface, now we need to create a repository in the **/Repositories** folder named **StudentsRepository**.

To begin to implement our interface for the repository and define our interface methods, modify **/Repositories/StudentsRepository.cs** as shown:

CODE TO TYPE: Defining /Repositories/StudentRepository Interface methods

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data;
using System.Linq.Expressions;
using Interfaces_Extensions.IE_Context;
using Interfaces_Extensions.Models;
using Interfaces_Extensions.Interfaces;

namespace Interfaces_Extensions.Repositories
{
    public class StudentsRepository : IStudentsRepository, IDisposable
    {
        private InterfacesContext context;

        public StudentsRepository(InterfacesContext context)
        {
            this.context = context;
        }

        public IEnumerable<Students> getStudents()
        {
            return context.Students.ToList();
        }

        public IQueryable<Students> OrderByName(char letter)
        {
            var list = getStudents();
            return list.OrderByDescending(x => x.Name).AsQueryable();
        }

        public IQueryable<Students> Filter(string filterParams = "")
        {
            var list = getStudents();
            return context.Students.Where(x => x.Name.Contains(filterParams));
        }

        private bool ContextDisposed = false;

        protected virtual void Dispose(bool disposal)
        {
            if (!this.ContextDisposed)
            {
                if (disposal)
                {
                    context.Dispose();
                }
            }
            this.ContextDisposed = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
}
```

OBSERVE: getStudents() method

```
public IEnumerable<Students> getStudents()
{
    return context.Students.ToList();
}

public IQueryable<Students> OrderByName(char letter)
{
    var list = getStudents();
    return list.OrderByDescending(x => x.Name[0] == letter).AsQueryable();
}

public IQueryable<Students> Filter(string filterParams = "")
{
    return context.Students.Where(x => x.Name.Contains(filterParams));
}
```

The **getStudents()** method returns a list of students from our Students table.

For the **OrderByName()** method we are required to pass a letter from our controller that will be used to move any student name that start with that letter.

Our **Filter()** implementation accepts a string as a parameter and will filter the names depending on whether the name contains the string passed in to the function.

Now that we have set up our Student repository and interface, add a search box and a drop-down list so we can use the ordering and filtering functions we just created. Modify **/Views/Students/Index.cshtml** as shown:

CODE TO TYPE: Students Index.cshtml adding dropdown list and search box

```
@model IEnumerable<Interfaces_Extensions.Models.Students>
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>

<p>
@Html.ActionLink("Create New", "Create")

    @using (Html.BeginForm("Filters", "Students", FormMethod.Post))
    {
        @Html.TextBox("filterParams", null, new{ placeholder = "Enter filter terms" })
        <input type="submit" value="Filter!" />
    }
    @using (Html.BeginForm("OrderByName", "Students", FormMethod.Post))
    {
        <div>
            Sort by student First Names <select name="letter">
                @for (char c = 'A'; c <= 'Z'; c++ )
                {
                    <option>@c</option>
                }
            </select>
            <input type="submit" value="Sort" />
        </div>
    }
</p>
<table>
<tr>
<th>
        @Html.DisplayNameForExtension(model => model.Name)
    </th>
<th></th>
</tr>

@foreach (var item in Model)
{
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
@Html.ActionLink("Edit", "Edit", new { id = item.Id }) +
@Html.ActionLink("Details", "Details", new { id = item.Id }) +
@Html.ActionLink("Delete", "Delete", new { id = item.Id })
        </td>
    </tr>
}
</table>
```

Let's discuss the dropdown menu code:

OBSERVE: Dropdown List for Students Index.cshtml

```
@using (Html.BeginForm("OrderByName", "Students", FormMethod.Post))
{
    <div>
        Sort by student First Names <select name="letter">
            @for (char c = 'A'; c <= 'Z'; c++ )
            {
                <option>@c</option>
            }
        </select>
        <input type="submit" value="Sort" />
    </div>
}
```

Tip Dynamic dropdown lists are available using the Razor syntax. See [DropDownListFor\(\) - MSDN](#).

Here, we use basic HTML to create a **<select></select>** item; then we populate our dropdown list explicitly, using a **for loop and characters as the range**.

Now we need to modify the **StudentsController** to implement our repository. Since we won't be using any of the Entity Framework-created action methods (Add, Edit, and Delete), we'll delete them. We'll keep only the Index and Dispose methods:

CODE TO TYPE: Adding/Removing code from StudentsController.cs to implement repository

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Interfaces_Extensions.Models;
using Interfaces_Extensions.IE_Context;
using Interfaces_Extensions.Repositories;
using Interfaces_Extensions.Interfaces;

namespace Interfaces_Extensions.Controllers
{
    public class StudentsController : Controller
    {
        private InterfacesContext db = new InterfacesContext();
        private IStudentsRepository studRepo;

        public StudentsController()
        {
            this.studRepo = new StudentsRepository(new InterfacesContext());
        }

        public StudentsController(IStudentsRepository studRepo)
        {
            this.studRepo = studRepo;
        }

        public ActionResult Filters(string filterParams = "")
        {
            return View(studRepo.Filter(filterParams));
        }

        public ActionResult OrderByName(char letter)
        {
            ViewBag.letter = letter;
            return View(studRepo.OrderByName(letter));
        }

        //
        // GET: /Students/

        public ActionResult Index()
        {
            return View(db.Students.ToList()studRepo.getStudents());
        }

        //
        // GET: /Students/Details/5

        public ActionResult Details(short id = 0)
        {
            Students students = db.Students.Find(id);
            if (students == null)
            {
                return HttpNotFound();
            }
            return View(students);
        }

        //
        // GET: /Students/Create

        public ActionResult Create()
        {

```

```

return View();
}

//
// POST: /Students/Create

[HttpPost]
public ActionResult Create(Students students)
{
    if (ModelState.IsValid)
    {
        db.Students.Add(students);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

return View(students);
}

//
// GET: /Students/Edit/5

public ActionResult Edit(short id = 0)
{
    Students students = db.Students.Find(id);
    if (students == null)
    {
        return HttpNotFound();
    }
return View(students);
}

//
// POST: /Students/Edit/5

[HttpPost]
public ActionResult Edit(Students students)
{
    if (ModelState.IsValid)
    {
        db.Entry(students).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
return View(students);
}

//
// GET: /Students/Delete/5

public ActionResult Delete(short id = 0)
{
    Students students = db.Students.Find(id);
    if (students == null)
    {
        return HttpNotFound();
    }
return View(students);
}

//
// POST: /Students/Delete/5

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(short id)
{
    Students students = db.Students.Find(id);
    db.Students.Remove(students);

```

```

db.SaveChanges();
return RedirectToAction("Index");
+

protected override void Dispose(bool disposing)
{
    dbstudRepo.Dispose();
    base.Dispose(disposing);
}
}

```

Add a View for our **Filters** and **OrderByName** action methods. Right-click the action methods and select **Add | View**:

Add View

View name:
Filters

View engine:
Razor (CSHTML)

☒ Create a strongly-typed view

Model class:
Students (Interfaces_Extensions.Models)

Scaffold template:
List

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceholder ID:
MainContent

Add Cancel

Modify **/Views/Students/Filters.cshtml** as shown:

CODE TO TYPE: /Views/Students/Filters.cshtml

```
@model IEnumerable<Interfaces_Extensions.Models.Students>

@{
    ViewBag.Title = "Filters";
}

<h2>Filters</h2>

<p>
    @Html.ActionLink("Create New", "Create", "Students")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
        <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.Id }) +
                @Html.ActionLink("Details", "Details", new { id=item.Id }) +
                @Html.ActionLink("Delete", "Delete", new { id=item.Id })
            </td>
        </tr>
    }
</table>
```

Add View

View name:
OrderByName

View engine:
Razor (CSHTML)

☒ Create a strongly-typed view

Model class:
Students (Interfaces_Extensions.Models)

Scaffold template:
List

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:
MainContent

Add Cancel

Perform the same changes to **/Views/Students/OrderByName.cshtml**:

CODE TO TYPE: Views/Students/OrderByName

```
@model IEnumerable<Interfaces_Extensions.Models.Students>

@{
    ViewBag.Title = "OrderByName";
}

<h2>OrderByName</h2>

<p>
    @Html.ActionLink("Create New", "Create", "Students")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
        <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @if (item.Name[0] == ViewBag.letter)
                {
                    <td style="color:#03c108;">
                        @Html.DisplayFor(modelItem => item.Name)
                    </td>
                }
                else
                {
                    <td>
                        @Html.DisplayFor(modelItem => item.Name)
                    </td>
                }
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.Id }) +
                @Html.ActionLink("Details", "Details", new { id=item.Id }) +
                @Html.ActionLink("Delete", "Delete", new { id=item.Id })
            </td>
        </tr>
    }
</table>
```

Now that we have our controller assembled, let's see it in action:



and



and navigate to the Students index page and test our filtering extensions:

Index - My ASP.NET MVC Application - Windows Internet Explorer

http://localhost:53045/Students

your logo here

Register Log in

Home Courses **Students**

Index

[Create New](#)

Filter!

Sort by student First Names **A** **Sort**

Student Name

- Michael Hadsworth
- Arielle Highsmith
- Andrew Bandelsy
- Gabriel Jones
- Daniel Hartfords
- Randolph Nomenture
- Yuri Goferfeldt
- Shawndra Kiljoy
- Caesar Juneseth
- Crystal Hingerfield
- Rebecca Vantry
- Barry Mantilok

© 2013 - My ASP.NET MVC Application

Index - My ASP.NET MVC Application - Windows Internet Explorer

http://localhost:53045/Students

your logo here

Register Log in

Home Courses **Students**

Index

[Create New](#)

and **Filter!**

Sort by student First Names **A** **Sort**

Student Name

- Michael Hadsworth
- Arielle Highsmith
- Andrew Bandelsy
- Gabriel Jones
- Daniel Hartfords
- Randolph Nomenture
- Yuri Goferfeldt
- Shawndra Kiljoy
- Caesar Juneseth
- Crystal Hingerfield
- Rebecca Vantry
- Barry Mantilok

© 2013 - My ASP.NET MVC Application

Filters

[Back To Index](#)

Student Name

Andrew Bandelsy	Edit Details Delete
Randolph Nomenture	Edit Details Delete

Index

[Create New](#)

Filter!

Sort by student First Names

A ▾

Sort

Student Name

Michael Hadsworth
Arielle Highsmith
Andrew Bandelsy
Gabriel Jones
Daniel Hartfords
Randolph Nomenture
Yuri Goferfeldt
Shawndra Kiljoy
Caesar Juneseth
Crystal Hingerfield
Rebecca Vantry
Barry Mantilok

- A ▾
- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M
- N
- O
- P
- Q
- R
- S
- T
- U
- V
- W
- X
- Y
- Z

OrderByName

[Back to Index](#)

Student Name

Randolph Nomenture

Rebecca Vantry

Michael Hadsworth

Arielle Highsmith

Andrew Bandelsy

Gabriel Jones

Daniel Hartfords

Yuri Goferfeldt

Shawndra Kiljoy

Caesar Juneseth

Crystal Hingerfield

Barry Mantilok

© 2013 - My ASP.NET MVC Application

Now let's make our repository use an extension method so that ordering and filtering are independent of an interface.

Create a new folder in **/Extensions** named **IQueryableExtensions**. Then, in **/Extensions/IQueryableExtensions**, create a class named **IQueryableFilters**.

Modify the new filter file as shown:

CODE TO TYPE: Adding code to /Extensions/IQueryableExtensions/IQueryableFilters.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Linq.Expressions;
using System.Web.Mvc;
using System.Web.Mvc.Html;
using System.Web.Routing;
using System.Reflection;
using Interfaces_Extensions.Models;

namespace Interfaces_Extensions.Extensions.IQueryableExtensions
{
    public static class IQueryableFilter
    {
        public static IQueryable<Students> filterByName(this IQueryable<Students> student, string name)
        {
            return student.Where(x => x.Name.Contains(name));
        }

        public static IQueryable<Students> filterByLetter(this IQueryable<Students> student, char letter)
        {
            return student.OrderByDescending(x => x.Name[0] == letter);
        }

        public static IQueryable<Courses> checkIfCourseExists(this IQueryable<Courses> course, string courseName)
        {
            return course.Where(x => x.CourseName.ToUpper().Contains(courseName.ToUpper()));
        }

        public static IQueryable<Courses> courseRange(this IQueryable<Courses> course, int val_a, int val_b)
        {
            return course.Where(x => (x.Id > val_a && x.Id < val_b));
        }
    }
}
```

OBSERVE:

```
public static IQueryable<Courses> courseRange(this IQueryable<Courses> course, int val_a, int val_b)
```

The **this** keyword refers to the instance of the current class context.

Note The **this** keyword is used to access properties of a class within a Constructor, Instance Methods, and Instance Accessors.

The code is similar to the CoursesRepository and StudentsRepository files we implemented earlier:

OBSERVE: /Repositories/CoursesRepository

```
.  
.br/>.br/>public IQueryable<Courses> CheckIfCourseExists(Expression<Func<Courses, bool>> query, string courseValue)  
{  
    return context.Courses.Where(obj => obj.CourseName.Contains(courseValue));  
}  
  
public IQueryable<Courses> GetCourseRange(int val_1, int val_2)  
{  
    return context.Courses.Where(obj => (obj.Id > val_1 && obj.Id < val_2)).AsQueryable();  
}  
.br/>.br/.
```

OBSERVE: /Repositories/StudentsRepository

```
.br/>.br/>.br/>public IQueryable<Students> OrderByName(char letter)  
{  
    var list = getStudents();  
    return list.OrderByDescending(x => x.Name[0] == letter).AsQueryable();  
}  
  
public IQueryable<Students> Filter(string filterParams = "")  
{  
    var list = getStudents();  
    return context.Students.Where(x => x.Name.Contains(filterParams));*/  
}  
.br/>.br/.
```

Now, open **/Views/Web.config** and add the new namespace to it:

CODE TO TYPE: Adding IQueryableExtensions namespace to /Views/Web.config

```
.  
.   
.   
  <system.web.webPages.razor>  
    <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version=4  
.0.0.0, Culture=neutral, PublicKeyToken=XXXXXXXXXXXX" />  
    <pages pageBaseType="System.Web.Mvc.WebViewPage">  
      <namespaces>  
        <add namespace="System.Web.Mvc" />  
        <add namespace="System.Web.Mvc.Ajax" />  
        <add namespace="System.Web.Mvc.Html" />  
        <add namespace="System.Web.Optimization"/>  
        <add namespace="System.Web.Routing" />  
        <add namespace="Interfaces_Extensions.Extensions.DisplayNameForExtensions" />  
        <add namespace="Interfaces_Extensions.Extensions.HtmlExtensions"/>  
        <add namespace="Interfaces_Extensions.Extensions.IQueryableExtensions"/>  
      </namespaces>  
    </pages>  
  </system.web.webPages.razor>  
.   
.   
.   

```

Add the **filterByName** and **filterByLetter** methods to **/Repositories/StudentsRepository.cs**:

CODE TO TYPE: adding IQueryableExtension methods to /Repositories/StudentsRepository.cs

```
.  
.   
.   
using System.Linq.Expressions;  
using Interfaces_Extensions.IE_Context;  
using Interfaces_Extensions.Models;  
using Interfaces_Extensions.Interfaces;  
using Interfaces_Extensions.Extensions.IQueryableExtensions;  
.   
.   
.   
public IQueryable<Students> OrderByName(char letter)  
{  
    var list = getStudents().AsQueryable();  
    return list.OrderByDescending(x => x.Name[0] == letter).AsQueryable();list.filterBy  
Letter(letter);  
}  
  
public IQueryable<Students> Filter(string filterParams = "")  
{  
    var list = getStudents().AsQueryable();  
    return context.Students.Where(x => x.Name.Contains(filterParams));list.filterByName  
(filterParams);  
}
```


Add **checkIfCourseExists** and **courseRange** methods to **/Repositories/CoursesRepository.cs** file:

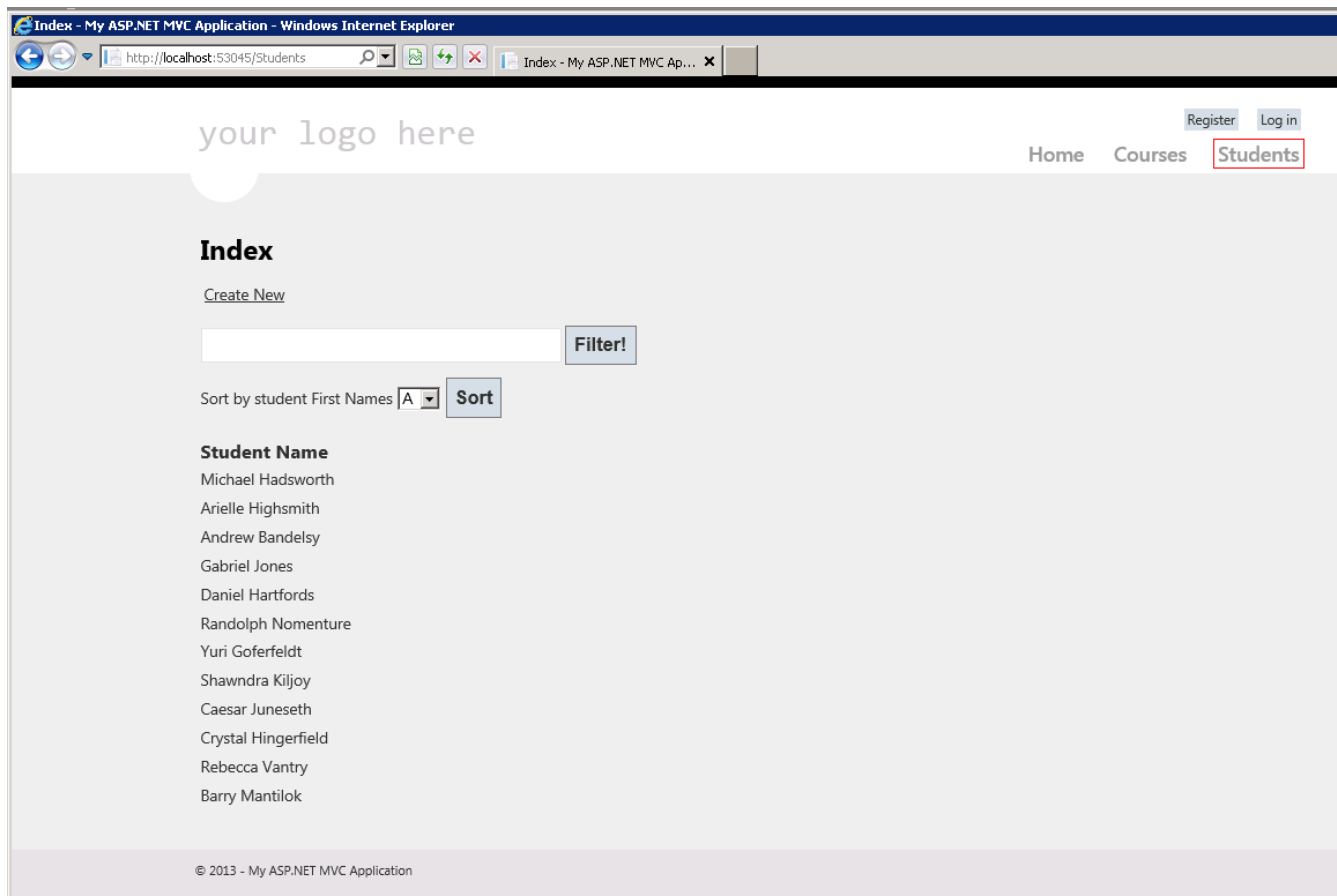
CODE TO TYPE: adding IQueryableExtension methods to /Repositories/CoursesRepository.cs

```
.  
.br/>.br/>using System.Linq.Expressions;  
using Interfaces_Extensions.IE_Context;  
using Interfaces_Extensions.Models;  
using Interfaces_Extensions.Interfaces;  
using Interfaces_Extensions.Extensions.IQueryableExtensions;  
.br/>.br/>.br/>public IQueryable<Courses> CheckIfCourseExists(Expression<Func<Courses, bool>> query, s  
tring courseValue)  
{  
    return context.Courses.Where(obj => obj.CourseName.Contains(courseValue)); context.C  
ourses.checkIfCourseExists(courseValue);  
}  
  
public IQueryable<Courses> GetCourseRange(int val_1, int val_2)  
{  
    return context.Courses.Where(obj => (obj.Id > val_1 && obj.Id < val_2)).AsQueryable  
(()); context.Courses.courseRange(val_1, val_2);  
}  
.br/>.br/.
```

Make similar changes to the calls to the CheckIfCourseExists method in **CoursesController.cs** and **ICourseRepository.cs**.



and  and navigate to the Courses and Students pages to verify that it works as it did before:



Index - My ASP.NET MVC Application - Windows Internet Explorer

http://localhost:53045/Students

your logo here

Register Log in

Home Courses **Students**

Index

[Create New](#)

and

Sort by student First Names

Student Name

- Michael Hadsworth
- Arielle Highsmith
- Andrew Bandelsy
- Gabriel Jones
- Daniel Hartfords
- Randolph Nomenture
- Yuri Goferfeldt
- Shawndra Kiljoy
- Caesar Juneseth
- Crystal Hingerfield
- Rebecca Vantry
- Barry Mantilok

© 2013 - My ASP.NET MVC Application

Filters

[Back To Index](#)

Student Name

Andrew Bandelsy	Edit Details Delete
Randolph Nomenture	Edit Details Delete

© 2013 - My ASP.NET MVC Application

Index

[Create New](#)

Filter!

Sort by student First Names

Sort

Student Name

Michael Hadsworth

Arielle Highsmith

Andrew Bandelsy

Gabriel Jones

Daniel Hartfords

Randolph Nomenture

Yuri Goferfeldt

Shawndra Kiljoy

Caesar Juneseth

Crystal Hingerfield

Rebecca Vantry

Barry Mantilok

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z

OrderByName

[Back to Index](#)

Student Name

Randolph Nomenture

Rebecca Vantry

Michael Hadsworth

Arielle Highsmith

Andrew Bandelsy

Gabriel Jones

Daniel Hartfords

Yuri Goferfeldt

Shawndra Kiljoy

Caesar Juneseth

Crystal Hingerfield

Barry Mantilok

© 2013 - My ASP.NET MVC Application

We've created an element of abstraction by implementing our IQueryable filtering extensions. Rather than require our repositories to implement the interfaces, we can create a separate section where our filtering and searching is self-contained.

By using Interfaces and Extension methods, an API (Application Programmer Interface) can interact with other programs.

Tip

MSDN provides excellent documentation on the [IQueryable](#) generalized interface that can be modified to fit specific your needs.

Phew! That was a long lesson. I'm glad you're sticking with it. Practice what you've learned here in your homework before you move on.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Web Services

Lesson Objectives

In this lesson you will:

- find and use web services with your C# applications.

What is a Web Service?

A web service is software written and deployed to act as a software service using the internet. As you learn to create a web service, you'll encounter a wide variety of standards, protocols, styles, and so on. Essentially, web services are separated into two camps: SOAP and REST. We will cover each of these standards in depth, but first we'll create a web service to facilitate our discussion.

Any software that runs over a network using any communication protocol and standard, could be considered a web service. For our purposes, a *web service* is a software service accessible through the internet using a standard protocol such as HTTP, that exchanges information using a standard format such as JSON or XML. Web services are programming language-independent, but we'll be using C# and MVC.

Web services can be complete applications, but more commonly they are software components that serve a specific role. Many browser-based software applications (web applications), interact with one or more individual web services, although many desktop, mobile, tablet, and other application types also use web services.

A web service is almost always self-contained and self-describing. In other words, you can access the full range of features of an individual web service without requiring other web services, and you can determine which features are available by querying the web service. In fact, an entire group of vendors exists using Service Oriented Architecture (SOA) as a guiding concept for their products. Google has most of their products created using the SOA concept, and have exposed numerous web services for anyone to use. You can find free web services by searching online for **free web services**.

Okay, enough talk. Let's create a C# MVC web service project!

Our First Web Service

Select **File | New Project**, and in the New Project dialog box, select **Visual C# | Web** under Installed Templates, then **ASP.NET MVC 4 Web Application**. Change the project Name to **MVCStudentWebAPI**, click **OK**, select **Web API** from the New ASP.NET MVC 4 Project dialog box, and click **OK**. Run the project, and you'll see this page:

ASP.NET Web API

Welcome to ASP.NET Web API! Modify the code in this template to jump-start your ASP.NET Web API development.

ASP.NET Web API allows you to expose your applications, data and services to the web directly over HTTP.

To learn more about ASP.NET Web API visit <http://asp.net/web-api>. The page features **videos, tutorials, and samples** to help you get the most from ASP.NET Web API. If you have any questions about ASP.NET Web API, visit [our forums](#).

We suggest the following steps:

- 1 Getting Started**
ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework. [Learn more...](#)
- 2 Add NuGet packages and jump-start your coding**
NuGet makes it easy to install and update free libraries and tools. [Learn more...](#)
- 3 Find Web Hosting**
You can easily find a web hosting company that offers the right mix of features and price for your applications. [Learn more...](#)

That screen looks like a typical MVC web application—what's different? When we examine the Controllers folder in the

Solution Explorer, we find two controllers: HomeController and ValuesController. When we open the HomeController.cs, we find a typical controller that returns a view, undoubtedly where the default page came from. When we open the ValuesController, we see methods that are designed for use as a web service. The methods listed below are provided by default, although your listing may be slightly different:

OBSERVE: ValuesController.cs

```
public class ValuesController : ApiController
{
    // GET api/values
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/values/5
    public string Get(int id)
    {
        return "value";
    }

    // POST api/values
    public void Post([FromBody]string value)
    {
    }

    // PUT api/values/5
    public void Put(int id, [FromBody]string value)
    {
    }

    // DELETE api/values/5
    public void Delete(int id)
    {
    }
}
```

These methods may appear similar to normal MVC controller and view methods; the controller inherits from **ApiController**. Classes derived from this class do not return views. Familiar syntax comments preceding each method. This API syntax not only describes the browser path, but the required HTTP verbs.

- **GET** is used to send information as part of the URI and should be repeatable without any side effects.
- **POST** is used to send individual groups of information that does have side effects. Each POST results in a separate server response. POST is typically used with HTML form control submissions.
- **DELETE** is used to request the removal of data and is not repeatable.
- **PUT** is used to request modification or insertion of data, typically to a single object, and is repeatable.

Note

Did you notice we used the acronym URI rather than URL? URI is the acronym for Uniform Resource Identifier. Only when the complete specification is included to identify the resource does a URI become a URL, which requires the protocol and the domain. Without this information, a URL path is really just a URI.

When we worked with JavaScript and Ajax, we were able to specify the HTTP verb, which makes using Ajax with web services convenient.

Let's try the GET verb and URI pattern. Return to the browser or rerun the application, then add **/api/values/** to the URL.

Depending on your browser, you will either see output, or be prompted to download the results as a file. If you were prompted to save as a file, save the HTTP GET results as a text file, and then open the file using a text editor like Notepad:

OBSERVE: GET URL With Numerical

```
<ArrayOfstring><string>value1</string><string>value2</string></ArrayOfstring>
```

How did our MVC application use the route to resolve to the correct method? We've seen routing in previous lessons, and our MVC Web API application has the same default routing that we saw earlier in the **/App_Start/RouteConfig.cs** file, but for Web API routing, MVC uses the **/App_Start/WebApiConfig.cs** file routing instead as shown:

OBSERVE: WebApiConfig.cs

```
config.Routes.MapHttpRoute(  
    name: "DefaultApi",  
    routeTemplate: "api/{controller}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
);
```

Each HTTP request is routed to a controller by consulting the routing table. From the routing declaration in **WebApiConfig.cs**, we can see that the route is **api/{controller}/{id}**.

You can use the **{controller}** and **{id}** placeholders to match against the incoming URI and **{id}** if you like. These are the general routing rules used in an MVC Web API project:

- **{controller}** is matched against the controller name.
- The HTTP verb (also known as HTTP request method) is matched against all or the first part of the controller method names.
- **{id}**, if present, is matched to a method parameter named **id**.
- Query parameters are matched to method parameter names where possible.

Note Query parameters typically follow the question mark (?) symbol in a URI.

Method names do not have to be an exact match; the MVC routing will attempt to find the closest match.

Go ahead and try the GET verb followed by a number, by adding **/api/values/5** to the URL.

Your results will look something like this:

OBSERVE: GET URL With Numerical

```
<string>value</string>
```

Not bad for a first web service project. We'll create another custom web project shortly, but first let's discuss web services in general, and the different methods available.

Web Services: SOAP and REST

Historically, the most common web service has been SOAP (Simple Object Access Protocol). Today most online web service providers continue to offer this web service specification, and many others offer only this option. Since the inception of SOAP (around 1998), another web service architecture has emerged known as REST (Representational State Transfer). Each of these web service protocols offers unique features, but discussion of those is beyond the scope of this lesson. For now, we'll summarize a couple of points about each protocol:

SOAP:

- uses a variety of communication protocols, but HTTP has become more common.
- requires SOAP standards, where a SOAP message must include an XML-based envelope for identification, and an XML-based body that specifies service and response specifics.
- requires and provides an XML-based WSDL (Web Services Description Language) construct that describes the services offered and mechanism for usage.
- typically uses XML-based response format, although other formats are supported.

REST:

- typically uses HTTP, but many other protocols are possible.
- uses a URL syntax for complete access to the web service.
- the URI syntax is intended to be self-explanatory.
- typically uses JSON as a data transport, although XML is also heavily used, and can support any valid HTTP-compatible media type.
- syntax used by MVC Web API.

The sample MVC Web API project employ the REST web service protocol, so let's modify it to add capabilities.

Adding REST Methods

In the MVCStudentWebAPI project, add a student model, a new controller to manage a web services API for students with appropriate messages, a "hard-coded" array to represent our "database," and then we'll query our database using Ajax.

Right-click on the **/Models** folder in the Solution Explorer, select **Add | Class**, change the class Name to **Student**, and click **Add**. Modify **/Models/Student.cs** as shown:

CODE TO TYPE: /Models/Student.cs

```
.  
. .  
namespace MVCStudentWebAPI.Models  
{  
    public class Student  
    {  
        public string Name { get; set; }  
        public string ID { get; set; }  
    }  
}
```

Now add the controller.

Right-click the **Controllers** folder, select **Add | Controller**, change the controller name to **StudentController**, select the **Empty API Controller** template, and click **Add**. Modify **/Controllers/StudentController.cs** as shown:

CODE TO TYPE: /Controllers/StudentController.cs

```

.
.
.
using MVCStudentWebAPI.Models;
.
.
.
public class StudentController : ApiController
{
    Student[] students = new Student[]
    {
        new Student { Name = "Tom Thumb", ID="ID001" },
        new Student { Name = "Bob Robertson", ID="ID002" },
        new Student { Name = "Alice Wonders", ID="ID003" },
        new Student { Name = "Sarah Smith", ID="ID004" },
        new Student { Name = "April Showers", ID="ID005" },
    };

    // GET /api/student
    public IEnumerable<Student> GetAllStudents()
    {
        return students;
    }

    // GET /api/student/id
    public HttpResponseMessage GetStudentByID(string id)
    {
        HttpResponseMessage response;
        var student = students.FirstOrDefault(s => s.ID == id);
        if (student == null)
        {
            response = new HttpResponseMessage(HttpStatusCode.NotFound)
            {
                Content = new StringContent(string.Format("No student with ID = {0}", id)),
                ReasonPhrase = "Student ID Not Found"
            };
        }
        else
        {
            response = Request.CreateResponse(HttpStatusCode.OK, student);
            response.Content.Headers.Expires = new DateTimeOffset(DateTime.Now.AddSeconds(300));
        }
        return response;
    }
}
.
.
.

```



and . Augment the URL like before, but follow the API URI syntax as given in the comments. Adding **/api/student** returns all of the students, while **/api/student/ID001** returns an individual student. A request for an invalid ID will return an error message.

The MVC Web API by default will return XML, but you can specify which type of data you want, just as we did in the JavaScript lesson when using jQuery and Ajax. Let's implement code in our view that will return JSON data instead of XML. We'll use the shortcut jQuery Ajax getJSON call.

Modify **/Views/Home/Index.cshtml**. Delete all of the Razor and HTML code in the file, and retype the code as shown:

CODE TO TYPE: /Views/Home/Index.cshtml

```
<header>
    <div class="content-wrapper">
        <div class="float-left">
            <p class="site-title">
                <a href="/">ASP.NET MVC Web API</a></p>
            </div>
        </div>
    </header>
<div id="body">
    <section class="content-wrapper main-content clear-fix">
        <div>
            <h2>All Students</h2>
            <ul id="students" />
        </div>
        <div>
            <h2>Search by Student ID</h2>
            <input type="text" id="studentID" size="5" />
            <input type="submit" value="Search" onclick="studentSearch();" />
            <p id="student" />
        </div>
    </section>
</div>


@section scripts {
    <script type="text/javascript">
        var apiURI = "api/student";

        $(document).ready(function () {
            // Send an Ajax request
            $.getJSON(apiURI)
                .done(function (data) {
                    // On success, 'data' contains a list of students
                    $.each(data, function (key, student) {
                        // Add a list item for the student.
                        $('<li>', { text: formatStudent(student) }).appendTo($('#students')
                    );
                });
        });

        function formatStudent(student) {
            return student.Name + ': ' + student.ID;
        }

        function studentSearch() {
            var id = $('#studentID').val();
            $.getJSON(apiURI + '/' + id)
                .done(function (data) {
                    $('#student').text(formatStudent(data));
                })
                .fail(function (jqXHR, textStatus, err) {
                    $('#student').text('Error: ' + jqXHR.responseText);
                });
        }
    </script>
}
```



and . You see this web page:

ASP.NET MVC Web API

All Students

- Tom Thumb: ID001
- Bob Robertson: ID002
- Alice Wonders: ID003
- Sarah Smith: ID004
- April Showers: ID005

Search by Student ID

After the page loads, the first call to getJSON gets all of the students. Using the **Search** text field and button, you can search for a valid student ID, or generate an error message by searching for an invalid student ID.

Accessing Public Web Services

We can access public web services, but any service available for free online could disappear, so we need to select one we think will be around for awhile. Let's try a web service from Google that returns time zone information.

One of the benefits of a web service is that often you can access the service completely through a web browser. Enter the URL **<https://maps.googleapis.com/maps/api/timezone/json?location=39.6034810,-119.6822510×tamp=1331161200&sensor=true>** in a web browser, and you'll see output similar to this:

OBSERVE: Google TimeZone URL and JSON Result


```
{
  "dstOffset" : 0.0,
  "rawOffset" : -28800.0,
  "status" : "OK",
  "timeZoneId" : "America/Los_Angeles",
  "timeZoneName" : "Pacific Standard Time"
}
```

Just as we were able to interact with the web services we created ourselves, we can make calls to other web services, as long as we understand how to use them. For the Google TimeZone API, you can refer to the support page at [The Google Time Zone API](#). Let's modify our student project to display the timeZoneld from a call to this web service. Modify **/Views/Home/Index.cshtml** as shown:

CODE TO TYPE: /Views/Home/Index.cshtml

```
.  
.   
.   
<div>  
    <h2>All Students</h2>  
    <p id="tz" />  
    <ul id="students" />  
</div>  
.   
.   
.   
$(document).ready(function () {  
    // Send an Ajax request  
    $.getJSON(apiURI)  
    .done(function (data) {  
        // On success, 'data' contains a list of students  
        $.each(data, function (key, student) {  
            // Add a list item for the student.  
            $('<li>', { text: formatStudent(student) }).appendTo($('#students'));  
        });  
    });  
    $.getJSON('https://maps.googleapis.com/maps/api/timezone/json?location=39.6034810,-  
119.6822510&timestamp=1331161200&sensor=true')  
    .done(function (data) {  
        $('#tz').text(data.timeZoneId);  
    });  
});  
.   
.   
.
```



and . You see the time zone information **America/Los_Angeles** for the location.

You can also make the web service call from within C# code and not jQuery. When making a web service call from code, you have to consider the type of web service you want to call and adhere to the requirements of that web service. For a REST call using GET (which is what the Google Time Zone call uses), we need to do a web request from within C#.

Modify **/Controllers/HomeController.cs** as shown:

CODE TO TYPE: /Controllers/HomeController.cs

```
.
.
.
using System.Net;
using System.Xml;
.
.
.
public ActionResult Index()
{
    string url = "https://maps.googleapis.com/maps/api/timezone/xml?location=39.6034810,-119.6822510&timestamp=1331161200&sensor=true";
    HttpWebRequest request = WebRequest.Create(url) as HttpWebRequest;
    using (HttpWebResponse response = request.GetResponse() as HttpWebResponse)
    using (XmlReader xmlReader = XmlReader.Create(response.GetResponseStream()))
    {
        xmlReader.ReadToFollowing("time_zone_id");
        ViewBag.TimeZone = xmlReader.ReadElementContentAsString();
    }

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

Now, modify **/Views/Home/Index.cshtml** as shown:

CODE TO TYPE: /Views/Home/Index.cshtml

```
<div>
    <h2>All Students</h2>
    Internal web service read time zone id: @ViewBag.TimeZone
    <p id="tz" />
    <ul id="students" />
</div>
```



and . You now see a new line that shows the time zone ID as retrieved in code and passed to the view using ViewBag.

Web Service Legacy and Windows Communication Foundation (WCF)

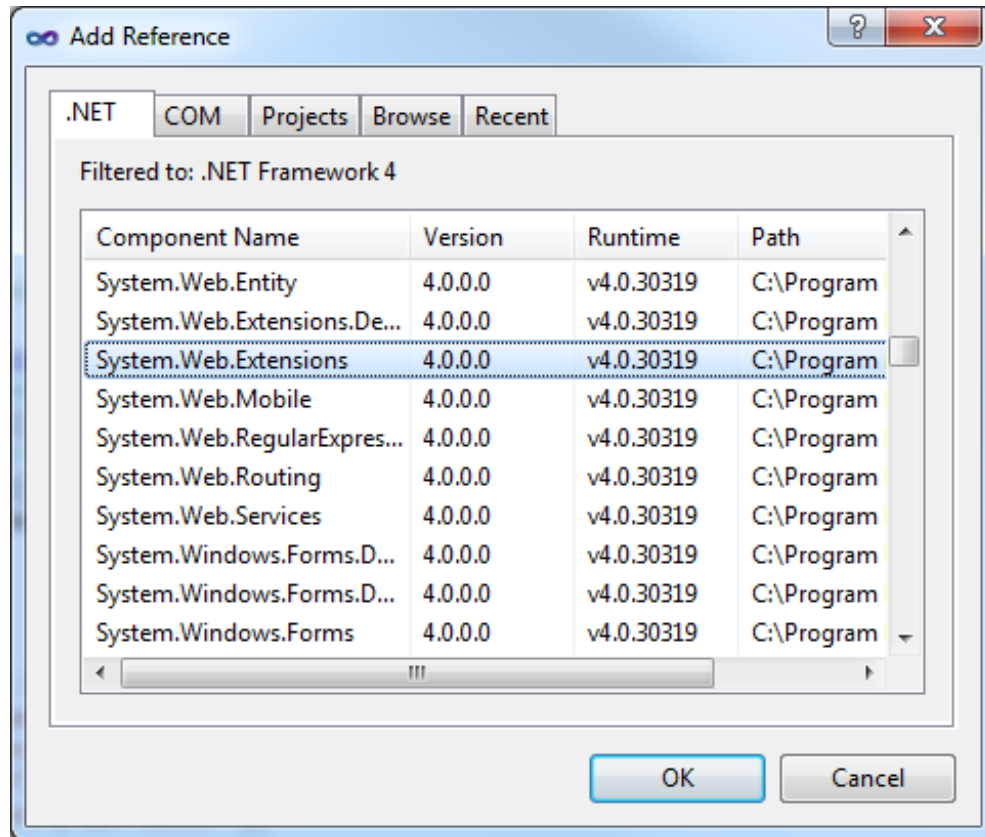
To be complete, we would like to demonstrate using pure SOAP from C#, but if you search online for SOAP-based web services, most of them are no longer available. This is a testament to the growing popularity of REST and the decline of SOAP. Despite the prevalence of REST-based web services, SOAP still exists, but finding an available SOAP service that will continue to be available after these lessons are published might be problematic, so we will create our own SOAP service.

The Microsoft Visual Studio with .NET arrived in 2002; the features of that product have undergone a number of changes, including web services. Previously, Studio supported creating an ASP.NET Web Service, but this template is no longer available as a primary template. However, we can still create these components by adding them directly to our project. We do that because we want to create a quick SOAP-compatible component we can use to demonstrate WCF. WCF is a Microsoft framework that can be used to create SOA style of software applications, and supports the SOAP WSDL as well as REST; data can be exchanged in a variety of formats, including XML and JSON. We could create a SOAP service using WCF, but the complexity of such an implementation is beyond the scope of this lesson. The older Web Service components are still completely valid, and often used when an implementation does not need the complexity of WCF.

Right-click the **MVCStudentWebAPI** project name in the Solution Explorer, select **Add | New Item**, select **Web** for the Template category, find and select **Web Service** in the list, change the web service Name to **StudentWebService.asmx**, and click **Add**. The Studio Code Editor will open with the StudentWebService.asmx code.

Now add a reference to our project to support the web service.

Right-click the **/References** folder in the Solution Explorer, select **Add Reference**, and in the Add Reference dialog box, select the **.NET** tab, locate **System.Web.Extensions** in the list, as shown:



Click **OK**. Modify **StudentWebService.asmx** as shown:

CODE TO TYPE: StudentWebService.asmx

```

.
.
.
using System.Web.Script.Services;
using System.Xml;
using System.Xml.Serialization;
using System.IO;
using System.Text;
using MVCStudentWebAPI.Models;
.
.
.
///[System.Web.Script.Services.ScriptService]
public class StudentWebService : System.Web.Services.WebService
{
    Student[] students = new Student[]
    {
        new Student { Name = "Tom Thumb", ID="ID001" },
        new Student { Name = "Bob Robertson", ID="ID002" },
        new Student { Name = "Alice Wonders", ID="ID003" },
        new Student { Name = "Sarah Smith", ID="ID004" },
        new Student { Name = "April Showers", ID="ID005" },
    };

    [WebMethod]
    [ScriptMethod(ResponseFormat = ResponseFormat.Json, UseHttpGet = true)]
    public string HelloWorld()
    {
        return "Hello World";
    }

    [WebMethod(Description="Returns all students")]
    public string GetAllStudents()
    {
        return ObjectToXML(students);
    }

    private string ObjectToXML(object obj)
    {
        StringBuilder strXML = new StringBuilder();
        try
        {
            Type objectType = obj.GetType();
            XmlSerializer xmlSerializer = new XmlSerializer(objectType);
            MemoryStream memoryStream = new MemoryStream();
            try
            {
                using (XmlTextWriter xmlTextWriter = new XmlTextWriter(memoryStream, En
coding.UTF8) { Formatting = Formatting.Indented })
                {
                    xmlSerializer.Serialize(xmlTextWriter, obj);
                    memoryStream = (MemoryStream)xmlTextWriter.BaseStream;
                    strXML.Append(new UTF8Encoding().GetString(memoryStream.ToArray()))
;
                }
            }
            finally
            {
                memoryStream.Dispose();
            }
        }
        catch (Exception ex)
        {
            strXML.Clear();
            strXML.Append("<Error>" + ex.Message + "</Error>\n");
        }
    }
}

```

```

        return strXML.ToString();
    }
}
.
.
.

```

The changes we made allow our code to support calling the methods and serializing the student data as an XML string.

Now, modify **/Views/Home/Index.cshtml** to demonstrate calling the StudentWebService.asmx HelloWorld method:

CODE TO TYPE: /Views/Home/Index.cshtml


```

.
.
.
<div>
    <h2>Web Service</h2>
    <p id="webServiceHelloWorld" />
</div>
.
.
.
$(document).ready(function () {
    // Send an Ajax request
    $.getJSON(apiURI)
    .done(function (data) {
        // On success, 'data' contains a list of students
        $.each(data, function (key, student) {
            // Add a list item for the student.
            $('<li>', { text: formatStudent(student) }).appendTo($('#students')
        });
    });
    $.getJSON('https://maps.googleapis.com/maps/api/timezone/json?location=39.6
034810,-119.6822510&timestamp=1331161200&sensor=true')
    .done(function (data) {
        $('#tz').text(data.timeZoneId);
    });
});

$.ajax({
    type: "GET",
    contentType: "application/json; charset=utf-8",
    url: "StudentWebService.asmx/HelloWorld",
    data: '{ }',
    dataType: "json",
    success: function (data) {
        $('#webServiceHelloWorld').text(data.d);
    },
    error: function (data) {
        $('#webServiceHelloWorld').text(data.d);
    }
});

```



and ; you see the text **Hello World**.

Remember, we created the web service to demonstrate SOAP using WCF. Now let's demonstrate that the web service does actually support SOAP. Run the project again, but this time, add **/StudentWebService.asmx** to the URL.

You see something like this:

StudentWebService

The following operations are supported. For a formal definition, please review the [Service Description](#).

- [GetAllStudents](#)
Returns all students
- [HelloWorld](#)

This web service is using <http://tempuri.org/> as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

Each XML Web service needs a unique namespace in order for client applications to distinguish it from other services on the Web. <http://tempuri.org/> is available for XML Web services that are under development, but published XML Web services should use a more permanent namespace.

Your XML Web service should be identified by a namespace that you control. For example, you can use your company's Internet domain name as part of the namespace. Although many XML Web service namespaces look like URLs, they need not point to actual resources on the Web. (XML Web service namespaces are URIs.)

For XML Web services creating using ASP.NET, the default namespace can be changed using the WebService attribute's Namespace property. The WebService attribute is an attribute applied to the class that contains the XML Web service methods. Below is a code example that sets the namespace to "<http://microsoft.com/webservices/>":

C#

```
[WebService(Namespace="http://microsoft.com/webservices/")]
public class MyWebService {
    // implementation
}
```

Visual Basic

```
<WebService(Namespace:="http://microsoft.com/webservices/")> Public Class MyWebService
    ' implementation
End Class
```

C++

```
[WebService(Namespace="http://microsoft.com/webservices/")]
public ref class MyWebService {
    // implementation
};
```

For more details on XML namespaces, see the W3C recommendation on [Namespaces in XML](#).

For more details on WSDL, see the [WSDL Specification](#).

For more details on URIs, see [RFC 2396](#).

ASP.NET Web Services generate these pages to help test your web service. Read through the text, then click on either of the web service methods. You'll see a page that discusses the SOAP request and response for these web services; you can execute, or "invoke" them:

StudentWebService

Click [here](#) for a complete list of operations.

HelloWorld

Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Invoke

SOAP 1.1

The following is a sample SOAP 1.1 request and response. The [placeholders](#) shown need to be replaced with actual values.

```
POST /StudentWebService.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/HelloWorld"


<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <HelloWorld xmlns="http://tempuri.org/" />
  </soap:Body>
</soap:Envelope>

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <HelloWorldResponse xmlns="http://tempuri.org/">
      <HelloWorldResult>string</HelloWorldResult>
    </HelloWorldResponse>
  </soap:Body>
</soap:Envelope>
```

View the WSDL by appending **?wsdl** to the URL, resulting in an XML listing that describes each available method.

Finally, create a way to call to work with a SOAP web service using WCF. Although there are a variety of techniques that might be used to interact with a SOAP web service using WCF, the technique we'll use is to add a *Service Reference*, effectively creating a strongly typed object we can use in our code.

Right-click the **/References** folder in the Solution Explorer and select **Add Service Reference**. In the Add Service Reference dialog, select the **Discover | Services in Solution** dropdown to have Studio find the StudentWebService within the project. Once it is discovered, you can use the Services section of the dialog box to examine the available services. Change the Namespace to **StudentWebServiceReference** and click **OK**. .

When you add a service reference, a new folder named **Service References** is added to your project with the StudentWebServiceReference. Changes are made to the /Web.Config file to support this reference, so you can use this reference to access our SOAP-supporting web service.

The process of adding a Service Reference to wrap a SOAP web service is referred to as creating a proxy.

You can remove the service reference by deleting it from the project. Deleting the service reference will also remove the /Web.Config changes.

Modify **/Controllers/StudentController.cs** as shown:

CODE TO TYPE: /Controllers/StudentController.cs

```
.  
.   
.   
// POST /api/student  
[HttpPost]  
public string GetWCFStudents()  
{  
    StudentWebServiceReference.StudentWebServiceSoapClient ws = new StudentWebServiceRe  
ference.StudentWebServiceSoapClient();  
    return ws.GetAllStudents();  
}  
.   
.   
.
```

This code may look familiar, except that we've decorated the method call with the `HttpPost` decoration to ensure this method can only be called when using the POST verb. Why? Because MVC will apply routing to our URI syntax, and we already have one GET method, so to avoid complicating this project by changing the routing tables, we can specify the unused POST verb. Using the newly created `StudentWebServiceReference`, we're able to call our web service using SOAP without worrying about the SOAP details.

To complete the process and test our changes, modify **/Views/Home/Index.cshtml** as shown:

CODE TO TYPE: /Views/Home/Index.cshtml

```
.  
.   
.   
<div>  
    <h2>WCF Service</h2>  
    <textarea rows="20" cols="80" id="wcfStudents"></textarea>  
</div>  
.   
.   
.   
$.ajax({  
    type: "POST",  
    contentType: "application/json; charset=utf-8",  
    url: apiURI,  
    data: '{ }',  
    dataType: "text",  
    success: function (data) {  
        document.getElementById('wcfStudents').value = data;  
    },  
    error: function (data) {  
        document.getElementById('wcfStudents').value = data;  
    }  
});  
.   
.   
.
```



and , and now you'll see the returned XML from the WCF mapped web service using SOAP.

You know quite a bit about web services. Practice using that knowledge and I'll see you soon!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Final Project

Lesson Objectives

In this final lesson and project, you will:

- combine many of the techniques you've learned throughout this course and previous C# courses to create a comprehensive dynamic MVC web application that will include unit tests and a database.
 - create a website that includes public and private webpage content, and employ ASP.NET security.
 - incorporate UI Design, Class Diagram, and Use Cases.
 - plan a complete project and include the development of Use Cases.
-

Project Functional Specifications

This list summarizes the functional specifications and requirements for your final course project:

- Create a web application called **OSTWebBlog**.
- The web application will allow multiple users to add blog posts to a blog website.
- Each blog user ("blogger") will register with the website, selecting their username and password.
- Only registered users may add blogs.
- Each blog post will consist of a title, main content, and a time/date stamp that records when the blog was created.
- A main page will list each blogger by their username, will allow selection to view an individual blog, and display the current total of blog postings per blogger.
- Blog postings are only managed, including editing and deletion, by the registered blogger who added the content.
- Display of blog postings will be in reverse chronological order by the last posted time/date stamp.
- Non-registered readers should be able to add comments to individual blog postings.
- For a blog posting, the title and main content are required, so you must use C# model attribute validation.
- The application must be implemented using MVC and Razor, and include Unit Testing.
- You must use the Entity Framework to create your class models. You may use either the Database First, Model First, or Code First workflow.

In addition to the required features, you may consider adding any of these features:

- Add a flag to allow blog posts to be private or public.
- Administrator review and flagging of questionable postings and comments.
- Categorizing blog postings.
- Multiple sort views.

Planning the Project

As with other projects, you need to plan how you will complete each of the necessary tasks for this project, which includes Use Cases, UI Design, the Entity Data Model and database Data Model, Coding, Unit Tests, and Testing. We've provided the Functional Specifications, listing out the requirements of the application, and included a few optional features, but you can add other features so long as the minimum functionality is provided. Proceed through each of the categories below, producing the needed content as part of your final project.

Use Cases

Consider how your website application will be used, and then list each of the Use Cases. You may find this task easier if you create a list of each of the MVC Views, then list the capabilities or functionality provided by each View. For example:

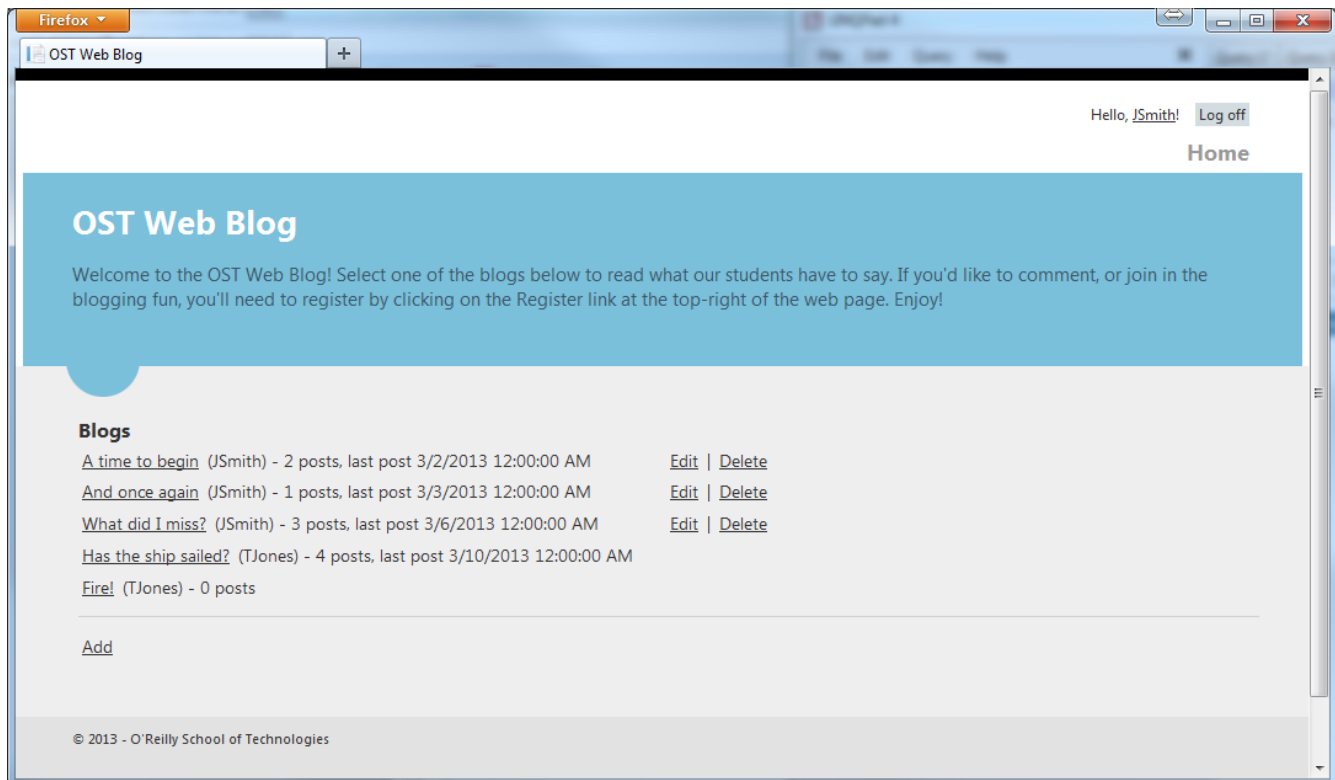
Index View

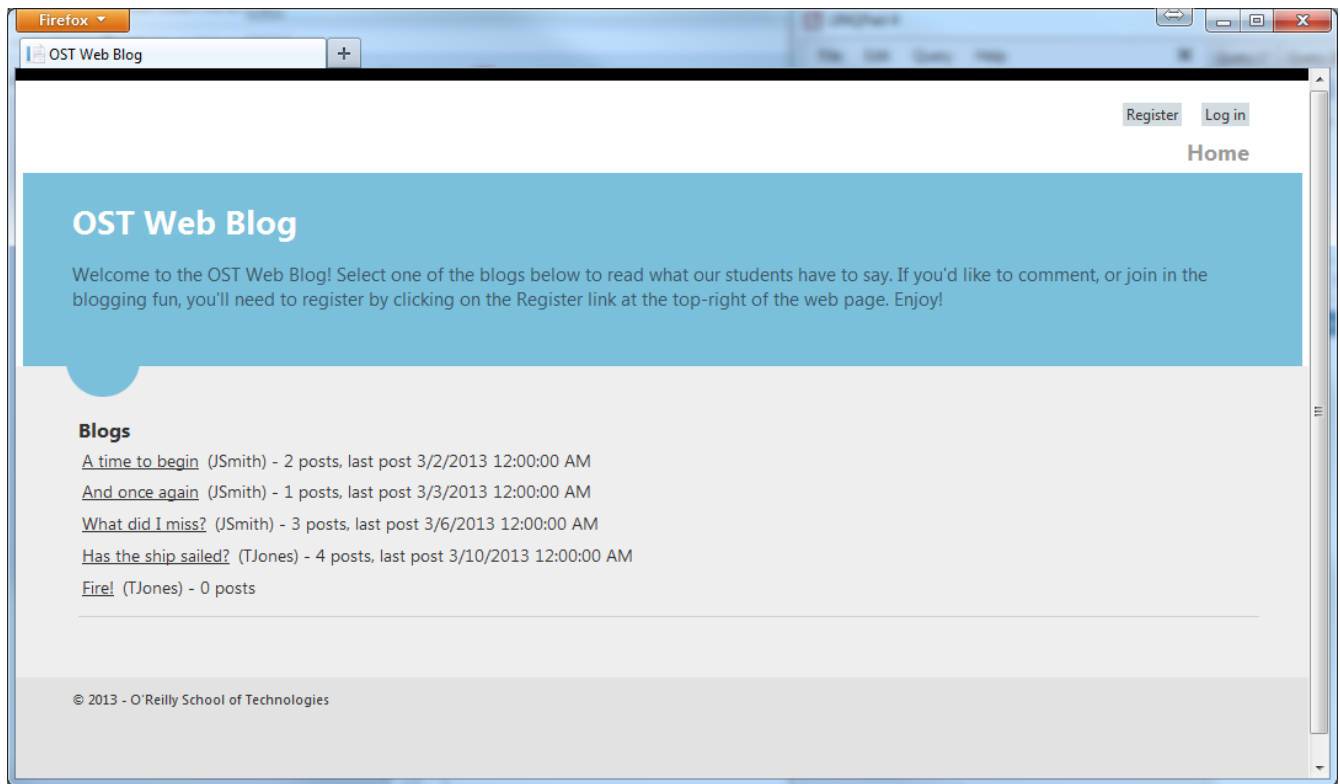
- Blog selection from complete list of bloggers.
- Registration using blog registration link.
- Blogger login using blog login entry boxes.
- Navigation to other sections of website.

UI Design

A major component of any application is the user interface (UI). For development, creating an initial or prototype UI will help you organize your application, and provide a visual reminder of the features you need to implement. You must provide an initial UI Design for your project, however, with MVC the UI is often constructed using class data models, so you will need to iterate through this step, and the Entity Data Model step, to complete the initial UI Design.

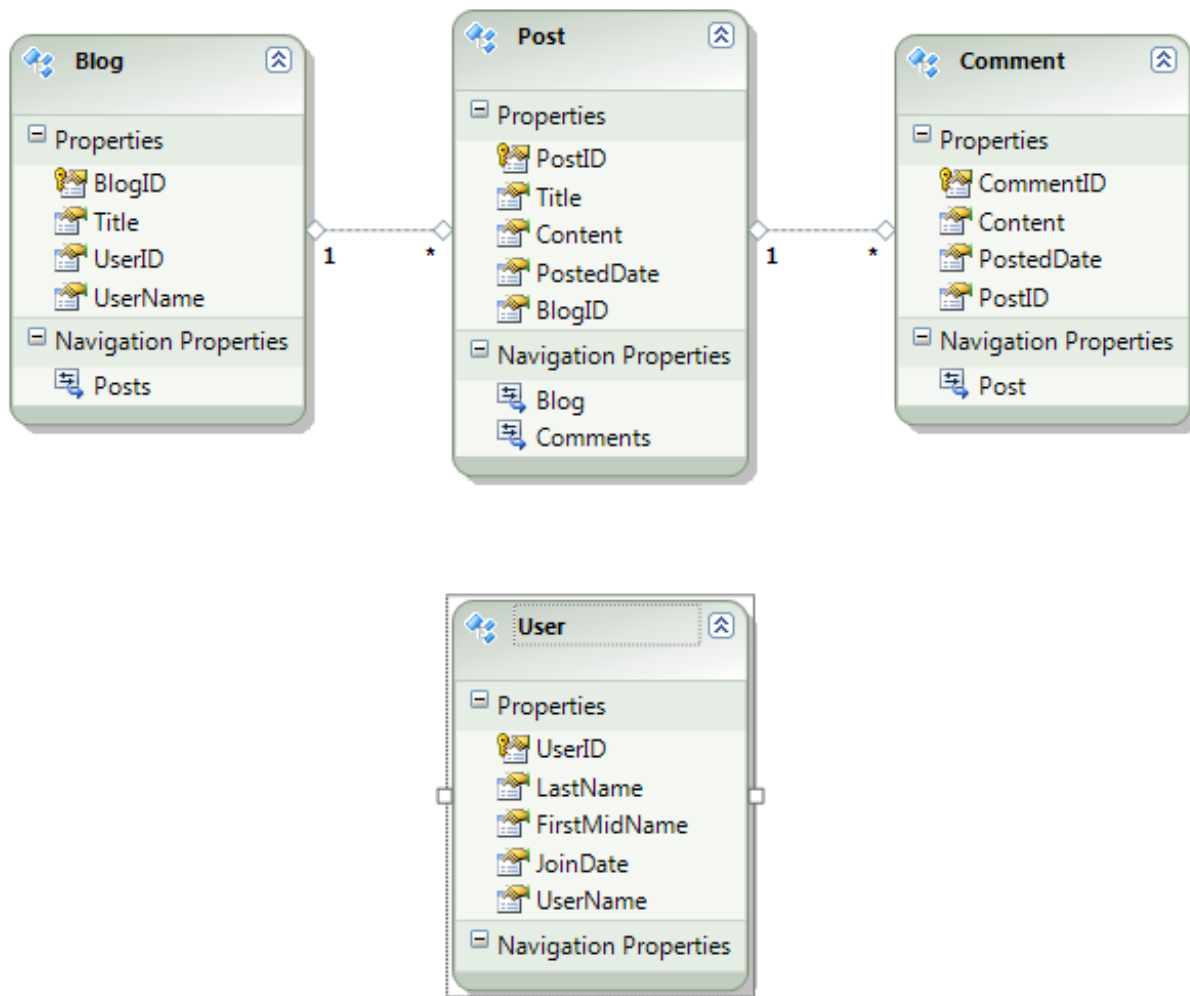
There is a sample UI Design for the Index View below (user logged in, and user logged out). Although the layout and colors are based on the MVC Internet Application template, you may set up your project layout and colors however you'd like. We elected to use the Internet Application template to include ASP.NET registration and security.





Data Modeling: Entity Data Model

When creating MVC applications, classes serve as the models to hold our application data, back up our UI controls, and enable MVC and the Entity Framework to provide data-typed constructs that simplify the development process. You must use one of the Entity Framework workflows. Your final data model must include appropriate associations between your classes. Here's one possible data model for this application:



Data Modeling: Database Tables

An advantage of MVC is its ability to persist models to permanent storage, such as a database. A variety of techniques are available to create database tables. If you elect to follow the Model First workflow, you may want to use the Entity Framework Entity Data Model Designer to generate your tables.

Software Development and Unit Test Development

After you've created your Entity Data Model and generated the individual data model classes, you will proceed to add code and View content. We strongly recommend that you code a set of use cases and View content, and add the necessary functionality to your class models. As you add such functionality, add the unit tests that are required to test your class functionality thoroughly. Unit tests are required for the project, and before submission, all of your unit tests should pass with no errors.

Testing and Completion

As part of the project development process, you should iterate through use case, View, and functionality, adding unit tests as appropriate. After you've finished adding the functionality for a use case, and as you add additional use cases, perform integrated testing to ensure that your application works as designed. Do not wait to perform integrated testing until the end of all use cases, but instead perform testing incrementally as the application is developed.

Make sure your final project compiles without errors, meets all of the functional requirements and specifications, and includes unit tests for all of the class functionality.

Once you've completed this final project, you've completed the course! Congratulations!

*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*