# ASP.NET MVC 5 Fundamentals

The following is a list of sections in this document. You can use these as hyperlinks to navigate to any particular section.

|  |  |
| --- | --- |
| [Understanding the ASP.NET Framework Options](#_Understanding_the_ASP.NET) | [Creating a Blank Solution with a Start-up MVC Project](#_Creating_a_Blank) |
| [ASP.NET MVC Application – the Model in Model-View-Controller](#_ASP.NET_MVC_Application_5) | [ASP.NET MVC Application – the Controller and View in Model-View-Controller](#_ASP.NET_MVC_Application_1) |
| [ASP.NET MVC Application – the Model, View, & Controller Combined](#_ASP.NET_MVC_Application_2) | [ASP.NET MVC Application – Razor Page Introduction](#_ASP.NET_MVC_Application_3) |
| [ASP.NET MVC Application – Start-up](#_ASP.NET_MVC_Application_4) | [Defining Routes for ASP.Net](#_Defining_Routes_for) |
| [Configuring ASP.NET with web.config](#_Configuring_ASP.NET_with) | [Autofac Inversion of Control Container (for Dependency Injection)](#_Autofac_Inversion_of) |
| [The MVC Controller – Query String](#_The_MVC_Controller) | [MVC and API Controllers](#_MVC_and_API) |
| [Scaffolding a Restaurant API](#_Scaffolding_a_Restaurant) | [Scaffolding Restaurant Details](#_Scaffolding_Restaurant_Details) |
| [Working with Action Results](#_Working_with_Action) | [Setting up a Create View](#_Setting_up_a) |
| [Validating Models and Model State](#_Validating_Models_and) | [Validating Models with Data Annotations](#_Validating_Models_with) |
|  | [Following the POST-Redirect-GET Pattern](#_Following_the_POST-Redirect-GET_1) |
| [Implementing the Edit Action](#_Implementing_the_Edit) | [Working with SQL Server LocalDB](#_Working_with_SQL) |
| [Installing and Configuring the Entity Framework](#_Installing_and_Configuring) | [Defining a DbContext](#_Defining_a_DbContext) |
| [Creating a Data Access Service](#_Creating_a_Data) | [Configuring a DbContext](#_Configuring_a_DbContext) |
| [Exploring the New Database](#_Exploring_the_New) | [Implementing a Delete Feature](#_Implementing_a_Delete) |
| [Understanding Razor Syntax](#_Understanding_Razor_Syntax) | [Specifying Layout Views](#_Specifying_Layout_Views) |
| [Using the View Bag](#_Using_the_View) | [Adding Flexibility with Layout Sections](#_Adding_Flexibility_with) |
| [Reusing HTML with Partial Views](#_Reusing_HTML_with) | [Partial View Postscript – Using CSS for <div> Positioning](#_Partial_View_Postscript) |
| [Passing Data across Requests with TempData](#_Passing_Data_across) |  |
| [Managing Client Libraries](#_Managing_Client_Libraries) | [Bundling Client Assets](#_Bundling_Client_Assets) |
| [Enabling Client Validation](#_Enabling_Client_Validation) | [Creating Single Page Apps](#_Creating_Single_Page) |
| [Evaluating the Deployment Options](#_Evaluating_the_Deployment) | [Installing Web Server Software](#_Installing_Web_Server) |
| [Publishing Apps with Visual Studio](#_Publishing_Apps_with) | [Creating an IIS Website](#_Creating_an_IIS) |
| [Configuring ASP.NET for Production](#_Configuring_ASP.NET_for) | [Configuring a SQL Server Login](#_Configuring_a_SQL) |
| [Running in Production](#_Running_in_Production) |  |

Refer to solution **ASP\_NET\_MVC\_5** (on the Windows 7 computer) or **ASP\_NET\_MVC\_5\_W10** (on the Windows 10 computer) for the VisualStudio sample code.

## Understanding the ASP.NET Framework Options

This section begins with guidelines on installing Visual Studio 2019. This introductory portion continues with a statement dispelling some possible misunderstandings.

The ASP.NET version numbers may be a little bit confusing. The framework studied in this course is “ASP.NET MVC 5”, which is part of the “ASP.NET version 4” framework. By contrast there also exists “ASP.NET Core”. Refer to [Choose between ASP.NET 4.x and ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/choose-aspnet-framework?view=aspnetcore-3.1). The instructor makes the point that ASP.NET Core is well suited to developers who need to develop on macOS or Linux, whereas ASP.NET version 4.x provides the full .NET framework which includes Windows Communication Foundation (WCF), WebAPI, Web Pages, and others.

## Creating a Blank Solution with a Start-up MVC Project

This course starts with **Blank Solution** (NOT Empty Project), and adds a project whose template is .NET Framework. Give the solution a name that contains words suggesting asp.net or mvc 5.

I have discovered from experimenting that it is best **not** to create the repository folder – outside of Visual Studio. Instead, when Visual Studio creates a blank solution, it creates the folder and the solution together. (When you create the repository folder before creating the solution, you get 2 nested folders with similar or identical names.)

The next step is to add a project to the solution. Right-click the solution name in the Solution Explorer. Click **Add > New Project...** in the pop-up menu. VisualStudio displays a pop-up dialog titled **Add a New Project.** Choose **ASP.NET Web Application (.NET Framework)** with qualifiers C#, Web, and Windows. Click **Next.** VisualStudio displays a pop-up dialog titled **Configure your new Project.**

* Type a name for the project. (I followed the instructor’s example and named the project “OdeToFood.Web”.) It is good practice to place the project under the folder created for the solution. Also, click the down arrow to the right of the **Framework** box, and choose the latest version (4.7.2 in this case). Then click the **Create** command button.
* Visual Studio displays a page titled **Create a new ASP.NET Web Application**, which invites the programmer to select the type of web application; select **MVC**. In the right-hand panel, click the **Configure for HTPPS** check box. Then click the **Create** command button. Visual studio creates the new project; the components of the project are displayed in the **Solution Explorer**, and **Overview** displays in the center of Visual Studio’s page.
* Use the **Build** pull-down menu to build the project.
* Use **Debug > Start Without Debugging** to see the result of executing the template code.

## ASP.NET MVC Application – the Model in Model-View-Controller

The next step is to move toward a web page that will display restaurant data. The details about restaurant data will entail creating a C# class, and the instructor has chosen to put this into a separate project. The separate project allows for the possibility of code sharing at some future time.

* Right-click the solution name. Then select **Add > New Project…** Visual Studio displays the familiar pop-up dialog titled **Add a new project**.
* Type “library” into the **Search for template** box at the top of the screen. Visual Studio displays additional templates in the right-hand pane.
* Scroll down (if necessary) until you see “Class Library (.NET Framework)” with “C#” and “Windows” below it. Click this entry. Click the **Next** button. VisualStudio displays a pop-up dialog titled **Configure your new project**.
* Type a name for the class library. (I followed the instructor’s example and named the project “OdeToFood.Data”.) As before place it in the same folder as the folder containing the solution, and select the latest version of .NET Framework. Click the **Create** button. Visual Studio adds the library to the solution – with template code.
* Visual studio has added “Class1” to the library. We don’t need it. Delete it either in the code or in the solution explorer.

Of course, it’s not necessary to separate the classes into different categories, but with the benefit of 20-20 foresight, the instructor is doing this.   (1) He anticipates classes that define the shape of the data. (2) He anticipates classes that perform the services that provide access to the data. These two categories will reside in different subfolders within the solution. Therefore, right-click the class-name node, and select **Add > New Folder**, and give it an appropriate name (e.g. “Models”).

The C#-class source files that we will add to “Models” will provide the “Model” component of Model-View-Controller. These classes will show the structure of data about a typical restaurant.

* We want to insert a C#-class source file into the folder; right click the folder name and select **Add > Class…**; VisualStudio displays a pop-up dialog titled **Add New Item**.
* If the type of item – in the right-hand pane - is not already displayed as “Visual C# Items An empty class definition”, find it in the list in the center pane and click it. Type the name of the class (e.g. “Restaurant”) in the **Name** text box (at the bottom of the dialog). Click the **Add** button. Visual Studio shows the new C#-class source file in the Solutions Explorer, and it displays the code for this “empty” **class** in the center of the screen.
* This **class** will need to be externally accessible; therefore, add the prefix **public** to **class**.
* A fast way to write **class** properties is to type the snippet “prop” followed by 2 successive **tab** keystrokes. Use this to add the identifier as an integer property; this will refer to the primary key of a database table, and the conventional name is “Id”.
* Similarly add the property “Name” as a string. This is the name of a restaurant.
* Although there are – in reality – several properties of a restaurant that one could include, for the sake of brevity, we add only one additional property – the category of cuisine. One way to do this is to make this property a reference to an enumerator class.  The very next step creates the class of this enumerator.  After that we’ll return to this Restaurant  class, and add the property by referring to the enumerator class.
* Add another **class** into the “Models” folder, and give it the name “CuisineType”. In the C# code replace **class** with **public enum**. Add a list starting with “None”, followed by typical names of cuisines (e.g. Italian, Indian, French).
* Return to the “Restaurant” **class**, and complete it by adding the property “Cuisine” of “CuisineType”; at this point intellisense will help you by suggesting the name “CuisineType”.

We will start with in-memory data – to be used to simulate data access to an SQL-server database; later we will switch over to a real SQL-server database. For this to be successful, it is good practice to hide the physical form of the data while we are working on the visual side; a good way to do this is to use an **interface** between the visual side and the either (1) the simulated data or (2) the actual data.

* Add a second folder; call it “Services”.
* Add an item – of type **interface** to “Services”.
* Add the prefix **public** to **interface**, because this **interface** will need to be externally accessible.
* Type the name of the **interface** (e.g. “IRestaurantData”) in the **Name** text box (at the bottom of the dialog). C# programmers have adopted the convention of using “I” as the first character for the name of an **interface**.
* There are several actions we want to perform. We start with a method in this **interface** that returns a list of restaurants available. Therefore, the return type is “IEnumerable<Restaurant>”. Notice that while you are typing this, intellisense suggests the **class** “Restaurant”. Notice, also, that after having typed “IEnumerable<Restaurant>”, VisualStudio has alertly added the necessary “using” statements at the top of the source file.
* Complete the declaration with the function signature “GetAll()”.

At this point the instructor points out a couple of VisualStudio shortcuts that make coding easy. We know that we will need a **class** that implements the **interface** IRestaurantData. For starters, this will be the in-memory implementation. The instructor recommends typing the code for this **class** immediately below the **interface** itself; he knows (from previous usage of VisualStudio) that he can quickly move this **class** to the appropriate place within the **solution**.

1. Type

**public class** InMemoryRestaurantData : IRestaurantData

**{**

}

(I followed the instructor’s example and used “InMemoryRestaurantData” for the **class** name.)

At this point VisualStudio has painted a squiggly red underscore beneath IRestaurantData, because this new class does not (yet) contain an implementation of GetAll(). Hover the mouse over IRestaurantData, and type <ctrl>. ; VisualStudio responds by offering to supply a default interface. Accept the default, and VisualStudio supplies the default implementation of GetAll().

1. By convention the code for public **class**es are maintained in separate **.cs** files. Hover the mouse over InMemoryRestaurantData, and type <ctrl>. ; VisualStudio responds by offering to move the code to a file titled InMemoryRestaurantData.cs. Accept the default, and VisualStudio performs the file gymnastics.

We are now faced with the task of implementing in-memory restaurants – i.e. replacing “NotImplementedException()” (default supplied by VisualStudio) with appropriate c# code for InMemoryRestaurantData.GetAll(). This is very likely throw-away code, because we expect that it will no longer be useful when the SQL-server version becomes implemented. Therefore, there is no need to be too fancy.

One way to maintain the data is in an elastic list; see the private declaration of “restaurants” in the VisualStudio solution ‘ASP\_NET\_MVC\_5’, in **class** OdeToFood.Data/Services.InMemoryRestaurantData. See also, in the same class the constructor where “restaurants” is populated with sample data.

Having access to the restaurant data in

List<Restaurant> restaurants;

the body of InMemoryRestaurantData.GetAll() is simply

return restaurants.OrderBy(r => r.Name);

.OrderBy() is method in the **Linq** extension of C#. When you type .OrderBy(), Visual Studio might automatically supply “using System.Linq;”. If not, hover the mouse cursor over the .OrderBy clause, and type <ctrl>. to instruct Visual Studio to add the **using** statement. r => r.Name indicates that the list of restaurants be provided in ascending (alphabetic) name order.

## ASP.NET MVC Application – the Controller and View in Model-View-Controller

A web application typically recognizes requests and delivers an appropriate response. It is the responsibility of a **controller** to decode the nature of the request and to initiate the process that delivers the response. Run the application that we started building in the previous section. The screen that this web application displays has three tabs labeled *Home*, *About*, and *Contact*. When the user changes the tab selection, the web application changes the text that is displayed. More specifically a **controller** (the HomeController) decodes the request and starts the process of displaying the appropriate text.

Examine the code in Controllers/HomeController.cs, more specifically the function Index(). This is the code that executes when the application starts. To prove this, temporarily change Index() to

public string Index()

{

return “Hello, world.”;

}

and restart the web application. This temporary change has masked the function View(), which plays a key role in MVC.

Examine the files in Views/Home; when the MVC framework encounters View() within the context of HomeController.cs, it will extract a combination of C# and HTML mark-up (a **razor** view) from an appropriate file; the extension .cshtml indicates a **razor** view. More specifically . . .

* if the instance of View() is in Index(), the **razor** view code is in Index.cshtml.
* if the instance of View() is in About(), the **razor** view code is in About.cshtml.
* if the instance of View() is in Contact(), the **razor** view code is in Contact.cshtml.

A comment in a **razor** page begins with “@\*” and ends with “\*@”.

Try making wholesale changes to Index.cshtml.

* Delete <div class =”row”> . . . </div>
* Replace the last two lines inside of <div class =”jumbotron”> . . . </div> with

<div>Hello, world.</div>

Rebuild and test the web application.

## ASP.NET MVC Application – the Model, View, & Controller Combined

Before invoking the view (the statement “return View();) the **controller** reaches out to the **model** to obtain the data – in this example the list of restaurants.

* Add a private member (e.g. named “db”) to class HomeController of type IRestaurantData. There are two things that need to be done. (1) IRestaurantData is a recognized type in the OdeToFood.Data project, but not in the OdeToFood.Web project; we need a **using** statement. (2) We need to add a reference to link the two projects. Add the reference to OdeToFood.Web’s list of references. Then use the <ctrl>. shortcut (while the mouse is hovering over IRestaurantData) to add the **using** statement.
* Provide a constructor for the HomeController class in order to initialize db. At this point in the course the statement in the constructor is

db = new InMemoryRestaurantData();

(This is not entirely satisfactory, because the **controller** should not be aware of the implementation details in the **model**. Instead, it should be possible to allow for the possibility SQL-Server implementation, without the need to rewrite the above statement. The instructor has promised that later in the course he would demonstrate “dependency injection” to avoid having to use the **new** keyword.)

* In function Index(), use db – with its GetAll() method - to get the list of all restaurants, and pass that list as an argument to function View(). I am following the example set by the instructor -- using “model” as the argument, which delivers the list of restaurants.

## ASP.NET MVC Application – Razor Page Introduction

As we saw earlier View() invoked from Index() passes the information to index.cshtml, which contains **razor** view code. To understand how this happens, we need to gain at least an introduction to **razor**. Navigate to index.cshtml. Note the C# code enclosed in braces ({}) immediately following “@”.

* In a **razor** page “@” introduces a C# expression; the result of evaluating this expression is inserted into the mark-up at this position. If the C# code is merely an expression (not an assignment statement nor a declaration), it does not require a terminating (;). For example

@DateTime.Now.Year

Introduces “2020” into the markup.

* After @ one can use an expression that contains a property of the **view**; in particular **Model** represents the argument that was provided when View() was invoked, specifically

View(model);

For example, one could use

<div>The number of restaurants is @Enumerable.Count(Model) </div>

* @ also introduces directives. For example, one might declare that the View() function’s argument must be an Enumerable referring to restaurants.

@**model** IEnumerable<OdeToFood.Data.Models.Restaurant>

The instructor suggests placing this at the top of the **razor** page. (**model** is a keyword here. It is merely a coincidence that we used “model” as the argument provided to View().) With this directive in place, one can now use the more familiar way of obtaining the number of restaurants.

<div>The number of restaurants is @Model.Count() </div>

* One can mix C# and HTML with the confidence that **razor** will recognize that HTML directives, and will guide the page to leave C# mode and transition back to HTML mode. Also, local variables that are introduced remain recognizable. For example

<ul>

@foreach (var restaurant in Model)

{

<li>@restaurant.Name</li>

}

</ul>

<ul> . . . </ul> introduces an unordered list (with a bullet for the prefix instead of an integer). @foreach introduces a C# loop – looping through the members of the **IEnumerable** Model. <li> … </li> - executed for each element of Model – reverts to HTML by producing bulleted text. But the text is the value of @restaurant.Name. Notice that restaurant was introduced as a local variable in the **foreach** statement, and that @restaurant takes us back to C# mode.

## ASP.NET MVC Application – Start-up

Every ASP.NET application has a file called Global.asax. This file defines what happens at application start-up. Right-click OdeToFood.Web.Global.asax, and click **View Markup** in the pop-up menu. VisualStudio displays the file; it is very brief and contains only “Application” followed by 3 clauses.

Look at Global.asax’s code-behind file, Global.asax.cs. This file specifies one class, MvcApplication, which, in turn, contains one or more event handlers pertaining to the web application’s life cycle. In particular look at the event handler, Application\_Start(), which invokes 4 “register” methods for MVC-5, and most of this code is in the OdeToFood.Web.App\_Start folder.

The class FilterConfig is defined in App\_Start/FilterConfig.cs.

The class RouteConfig is defined in App\_Start/RouteConfig.cs.

The class BundleConfig is defined in App\_Start/BundleConfig.cs.

All 3 of these classes contain static methods that are invoked by statements in Application\_Start().

App\_Start.BundleConfig.RegisterBundles() bundles java-script and CSS files. The purpose is to minify scripts to reduce download time. (See <http://minifycode.com/>for a definition of “minify”.) Specifically it combines OdeToFood.Web.Content.bootstrap. css and OdeToFood.Web.Content.Site.css, and it attempts to minify the combination.

## Defining Routes for ASP.Net

When the web application is directed to navigate to a new URL, MVC-5 uses “routing” to find the appropriate software needed to deliver the new web page. When the application is run from VisualStudio, and when the “About” tab is selected, VisualStudio displays the URL at the top of the page:

https://localhost:44372/Home/About

The left-hand part, “https://localhost:44372” is not pertinent to routing. Instead routing deals with the right-hand part, “/Home/About”. Therefore, the term “URL” in the following explanation means the right-hand part of the URL.

Application\_Start() begins the initialization of routing with the statement

RouteConfig.RegisterRoutes(RouteTable.Routes);

You will find RegisterRoutes() in App\_Start.RouteConfig.cs. The argument RouteTable.Routes (in the statement above) is a collection of rules; MVC-5 uses the collection of rules to map the new URL to the appropriate software. Initially – before RegisterRoutes() is invoked - RouteTable.Routes contains 0 rules. Open and examine App\_Start.RouteConfig.cs; note that there are two executable statements in RegisterRoutes. Each of these statements adds 1 ruleto RouteTable.Routes; as a result, the number of rules after initialization is 2.

The first statement in RegisterRoutes() is

routes.IgnoreRoute ( “{resource}.axd/{\*pathInfo}” );

This statement is telling MVC-5 to ignore URL’s that match the template “{resource}.axd/{\*pathInfo}”. For example

/trace.axd/1/2/3/4

matches the template, because “trace” matches the placeholder {resource}, and the “\*” in {\*pathInfo} indicates that “.axd/” might be followed by one or more characters. Since the example URL matches the template, it should be ignored.

The second statement in RegisterRoutes() is

routes.MapRoute (

name: “Default”,

url: “{controller}/{action}/{id}”,

defaults: new { controller = “Home”, action = “Index”, id = UrlParameter.Optional }

);

The template (the function’s argument) is very pertinent to our usage of MVC. What it tells us is as follows:

* “Default” is the name of the route.
* The URL can be parsed into 3 parts with the placeholder names {controller}, {action}, and {id}.
* The 3rd row tells us that if the URL is incomplete, treat it as if the 1st part contained “Home” and as if the 2nd part contained “Index”; moreover, the 3rd part is a URL parameter, which is optional.

For example, consider

/home/contact/1

MVC-5 interprets this as a controller whose name is “home” and whose action is “contact”; the URL parameter is “1”. Examine OdeToFood.Web.Controllers.HomeController.cs; yes, this module does contain a class named HomeController, and it does contain a member function named Contact(). Assuming that we are not case sensitive, we have a match, and MVC-5 is directed to that member function. I am guessing that we would pass “1” as the query string.

Try it. Edit the URL in the displayed page; replace “/home/about” with “/home/contact/1”. It works.

On the other hand try a URL such as “/greeting” that does not match the template. It yields a “resource cannot be found” run-time error, because MVC-5 did not find a controller named GreetingController. Let’s resolve the problem.

* Right-click Ode.ToFood.Web.Controllers. VisualStudio displays a pop-up menu.
* Click **Add > Controller…** VisualStudio displays a pop-up dialog titled **Add New Scaffolded Item**.
* Select **MVC 5 Controller - Empty**; then click the **Add** command button. VisualStudio displays another pop-up dialog titled **Add Controller**.
* Change the name to “GreetingController”; then click the **Add** command button. VisualStudio dismisses the dialog, and adds GreetingController.cs to Ode.ToFood.Web.Controllers, adds a folder “Greeting” to OdeToFood.Web/Views, and displays the source code of GreetingController.cs in the center of the screen.
* The body of class GreetingController is exactly what we want, because (1) “/greeting” is not followed by an {action} suffix, (2) the rule says to consider the action to be “Index” by default, and (3) the name of the ActionResult method in the body of GreetingController is Index().
* View() in this context doesn’t exist, yet. We have to create it. Right-click “View()”. VisualStudio displays a pop-up dialog titled “Add View”.
* If the **View name** text box in the dialog does not contain “Index”, type “Index” into this text box. Click the **Add** command button. VisualStudio adds Index.cshtml to Ode.ToFood.Web.Views, and displays source code of Index.cshtml in the center of the screen.
* Add <p<Hello!</p> at the end of this source code. Rebuild the project, and refresh the output displayed by the web app. Success; “Hello!” displays in the web page https://localhost:44372/Greeting.

## Configuring ASP.NET with web.config

The exercise at the end of the previous section demonstrated the process of displaying a greeting on a new web page. In the spirit of MVC we should separate the process of displaying the text from the content of the text; and one way to do this is to use Web.config as the repository of the text. This section provides the details, but it also reiterates the MVC style.

The following highlights the parallel between *restaurants* and *greetings*. Anything enclosed in square brackets represents development that will be introduced in this section.

|  |  |
| --- | --- |
| RESTAURANT | GREETING |
| model: OdeToFood.**Data**/Models/**CuisineType**.cs and OdeToFood.**Data**/Models/**Restaurant**.cs | model: [OdeToFood.**Web**/Models/**GreetingViewModel**.cs] |
| controller: OdeToFood.Web/Controllers/**HomeController**.cs | controller: OdeToFood.Web/Controllers/**GreetingController**.cs |
| retrieval\_flexibility: OdeToFood.Data/Services/InMemoryRestaurantData.cs and OdeToFood.Data/Services/IRestaurantData.cs | [retrieval\_flexibility: Web.config] |
| view: OdeToFood.Web/Views/**Home**/Index.cshtml | view: OdeToFood.Web/Views/**Greeting**/Index.cshtml |

The models for both Restaurant and Greeting are similar. They both use class definitions to describe the structure of the data. There does not seem to be much significance to the fact that the Restaurant model classes reside under OdeToFood.Data, whereas the Greeting class definition (to be defined below) will reside under OdeToFood.Web. The Restaurant class declarations are separate, because the project may be reusable in another context; the Greeting class declaration arguably pertains to Web.config in the main web project.

The controllers and views for both Restaurant and Greeting are similar.

I introduced “retrieval\_flexibility” to express the retrieval of the data in contrast to the data’s structure. The point has been made that code reuse is enhanced when data retrieval is separated from other processing. For Restaurant the pertinent modules are (1) the interface declaration (2) the 1st of 2 or more implementations (in-memory). For Greeting the counterpart (to be introduced below) will be Web.Config.

Web.Config provides a section where options can be declared. “key” is an identifier of the option, and “value” is the data.

<configuration>

<appSettings>

<!-- options expressed via key-value pairs -->

</appSettings>

</configuration>

In this demo the line of text is

<add key=”message” value=”Have a great day” />

(Use <ctrl>s to save these changes.)

Suppose that the owner of the web application felt that the above was an appropriate greeting for US users, but he wanted a different greeting for United Kingdom users, and a 3rd greeting for India. We can do this – without rebuilding the web app - by shipping a different Web.config file into the 3 deployment areas.

The expression

ConfigurationManager.AppSettings["message"]

in C# code - provides the string of characters from the “message” <appSettings> element, i.e. “Have a great day”.

Following the MVC pattern . . .

* Create a “model” class for the greeting. The instructor chose to introduce this into the folder OdeToFood.Web.Models. See OdeToFood.Web.Models.GreetingViewModel.cs; there is nothing remarkable about this class definition.
* Modify the greeting controller, OdeToFood.Web.Controllers.GreetingController.cs; (1) instantiate the model that was declared in the preceding statement, (2) retrieve the greeting text from Web.Config using the code cited above, and (3) pass the completed model instance as an argument when invoking “View(model)”.
* Modify the greeting view, OdeToFood.Web.Views.**Greeting**.Index.cshtml; (1) introduce the @model directive at the start of the razor page, and (2) replace the hard-coded “Hello” with Model.Message. (Model was sent by the controller; .Message is the text property.)

## Autofac Inversion of Control Container (for Dependency Injection)

We want to do some code clean-up. Refer to [ASP.NET MVC Application – the Model,View, & Controller Combined](#_ASP.NET_MVC_Application_2) , specifically the statement

db = new InMemoryRestaurantData();

and the parenthesized remarks that follow. A solution to this problem is to introduce an **Inversion of Control (IoC)** container, and to inject it as a calling-sequence argument of the constructor of the HomeController class. Specifically (1) we delete the statement (above) that relies on InMemoryRestaurantData(), (2) we expect the value of db to be supplied as an argument to the contstructor -- change

public HomeController()

to

public HomeController(IRestaurantData db)

and (3) assign the value of db from the constructor’s calling sequence to our private value of db – the statement

this.db = db;

in the body of the constructor.

When we try to execute this code, we get the run-time error “No parameterless constructor defined for this object.”. (“object” in this run-time error is the instance of HomeController, and the MVC framework is trying to instantiate it with the expectation of a 0-parameter constructor.) As mentioned above, one way to get around this problem is to use, instead, an **IoC** controller to instantiate the HomeController and to supply the appropriate value for its parameter.

Autofac is one supplier of the IoC container, and we can use **Nuget** to install it.

* Right-click OdeToFood.Web.References. Visual Studio displays a pop-up menu. Select **ManageNuGet Packages…** Visual Studio displays a list of the project’s NuGet packages.
* Click the **Browse** tab in the upper-left corner. Then type “Autofac” in the **Search** text box. Visual Studio displays a list of Autofac packages.
* Select a package titled “Autofac.Mvc5”. Then click the **Install** command button in the upper-right corner. VisualStudio installs the package.

Now we need to create and configure (initialize) both the Autofac IoC container and the MVC framework (start-up code). (Unlike configuration by modifying Web.Config, these configurations involve changes to C# code.)

* Add a C# class to OdeToFood.Web/App\_Start named ContainerConfig; when you specify “ContainerConfig.cs”, Visual Studio automatically names the class correctly.
* Notice that the other App\_Start configuration files omit “App\_Start” from the **namespace** statement; therefore, to stay consistent change the **namespace** statement in the C# file to “OdeToFood.Web”.
* Invoke this new class from **Global.asax**. Add the statement

RegisterContainer();

* RegisterContainer() does not yet exist, but VisualStudio will create it if you place the mouse cursor on “RegisterContainer”, type <ctrl>., and follow the prompts.
* Navigate back to ContainerConfig.cs, and remove the placeholder statement “throw new NotImplementedException();”. Replace it with the code that uses Autofac methods.

var builder = new ContainerBuilder();

(Applying <ctrl>. To “ContainerBuilder” causes VisualStudio to supply the appropriate “using” statement.)

builder.RegisterControllers(typeof(MvcApplication).Assembly);

(Applying <ctrl>. To “RegisterControllers” causes VisualStudio to supply the appropriate “using” statement. The class name “MvcApplication” is defined in Global.asax.cs.)

builder.RegisterType<InMemoryRestaurantData>().As<IRestaurantData>().SingleInstance();

(Applying <ctrl>. To “InMemoryRestaurantData” causes VisualStudio to supply the appropriate “using” statement. .SingleInstance() is only temporary; it would not work with multiple users, who are reading/writing restaurant data concurrently; we look forward to a multi-user solution later in the course.)

var container = builder.Build();

DependencyResolver.SetResolver(new AutofacDependencyResolver(container));

(Applying <ctrl>. To “DependencyResolver” causes VisualStudio to supply the appropriate “using” statement.)

Build and test the application. The “parameterless constructor” error has disappeared, because the ContainerConfig class has instructed the MVC framework to pass the data - from InMemoryRestaurantData - to the HomeController’s contructor.

## The MVC Controller – Query String

Let us revisit the “Greeting” URL. When invoked with a query string the URL might be

/greeting?name=Scott

The traditional way of parsing the query string would use C# code such as

var name = HttpContext.Request.QueryString["name"];

But with MVC this is unnecessary. Instead, add the argument “name” of type **string** to the method OdeToFood.Web/Controllers/GreetingController.ActionResult(). In response the MVC controller will search the URL for any occurrence of “name” in the URL, and it will supply its value when it invokes ActionResult(). To be able to harvest the value . . .

* Make a change to the GreetingViewModel class; add “Name” as a property.
* Make changes to the GreetingController class.
  + add the argument “(string name)”. MVC automatically checks the query string, and when it finds “name” in the query string, it supplies the value.
  + Assign the value of name to the Name property of the model. If name happens not to be present, it will have the value **null**; we can recognize **null** via the “??” (the null coalescing operator), and supply the alternative “no name ”. The statement that does this is

model.Name = name ?? “no name”;

* Use the “Name” property in Views/Greeting/Index.cshtml. Replace

@Model.Message

with

@Model.Message, @Model.Name

Build and test the web application.

## MVC and API Controllers

In MVC-5 the Controller gets an HTTP request, and delivers HTML via the View. Additionally in ASP.NET, but not in MVC-5, there is a 2nd type of Controller; this is called an API Controller. It similarly gets an HTTP request, but it delivers an XML or JSON (Java Script Object Notation) response.

To create an API controller . . .

* Add a new folder under OdeToFood.Web titled “Api”. (The management decision is to keep API controllers in a folder titled “Api” – separating them from MVC-5 controllers that reside in OdeToFood.Web/Controllers.)
* In the folder OdeToFood.Web.Api, right-click, select **Add > New Scaffolded Item…**, select **Web API 2 Controller – Empty**, click the **Add** command button, give the API controller the name “RestaurantsController”, and click the **Add** command button (again). VisualStudio retrieves the appropriate NuGet package, creates a template for RestaurantsController.cs (in OdeToFood.Web/Api), and displays a read-me file. VisualStudio has also added WebApiConfig.cs to OdeToFood.Web/AppStart*.*
* Follow the instructions displayed in the read-me file. The sentence at the top of the read-me file suggests that ApplicationStart() is in GlobalAsax.cs. (In our web application ApplicationStart() is already present.) In particular note that it is important that

GlobalConfiguration.Configure(WebApiConfig.Register);

be added **at the beginning of ApplicationStart()**, because it must be executed before

RouteConfig.RegisterRoutes(RouteTable.Routes);

Display RouteConfig.cs and WebApiConfig.cs side-by-side in Visual Studio’s center pane. (Use the pull-down menu **Window > New Vertical Document Group**.) Recall that we discussed RouteConfig.cs in a previous section - [Defining Routes for ASP.Net](#_Defining_Routes_for); WebApiConfig.cs is a newly added file. Notice the similarity between RouteConfig.cs and WebApiConfig.cs; one could interpret the routing rules similarly. The URL that invokes API action begins with “/api”; to direct routing to the new controller (RestaurantsController.cs), “/restaurants” would be appended to the end of “/api”. Build and run the application. Type “/api/restaurants” after “localhost44nnn”. An error message is displayed, but at least it is displayed in an XML format.

The error message is “No action was found on the controller ‘Restaurants’ that matches the request.” Unlike MVC-5 controllers, the convention with regard to actions are the verbs GET, PUT, POST, and DELETE. If none of these verbs are provided in the URL, the default is GET. Therefore, /api/restaurants instructs the router to look for a method named “Get” in the RestaurantsController class (in RestaurantsController.cs). Into the RestaurantsController class, insert the method

public string Get()

{

return "Hello, world!";

}

Then rebuild the web application, and run it with “/web/restaurants” at the end of the URL. You should see XML enclosing the string “Hello, world!”.

Instead of “Hello, world!”, we want the API controller to return a list of restaurants. This will entail invoking InMemoryRestaurantData(), which probably gets us back into the issues of IoC, Dependency Injection, and configuration; the changes needed in class RestaurantsController (Api/RestaurantsContoller.cs) will probably be similar to the changes that were needed in class HomeController (Controllers/HomeController.cs).

* In RestaurantsController.cs add a constructor. Provide a parameter for the constructor.

(IRestaurantData db)

* Click “<ctrl>.” after placing the mouse cursor in “IRestaurantData” to induce the statement.

**using** OdeToFood.Data.Services;

* Click “<ctrl>.” after placing the mouse cursor in “db” to induce the creation and initialization of a private readonly field named “db”. In response Visual Studio inserts

**private readonly** IRestaurantData db;

and it inserts

this.db = db;

into the body of the constructor.

* Replace

return “Hello, world!”;

with

var model = db.GetAll();

return model;

The Web API framework is smart enough to figure out how to serialize the data to produce the list of restaurants in XML or JSON form. But notice that the return value is no longer a string; instead it is a list (IEnumerable) of restaurants, i.e. IEnumerable<Restaurant>. (We know this because of the return type of GetAll() in OdeToFood.Data/Services/IRestaurantData.cs.) Change the return type of the function appropriately. Also apply <ctrl>. To <Restaurant> to get the appropriate **using** statement.

Now the code compiles successfully. Try to run it as before with “api/restaurants” at the end of the URL. You get an error message that states “‘RestaurantsController’ needs a parameterless public constructor”. We had a similar error with MVC-5, and we resolved it by using the the Autofac IoC container, which we configured in a new class, ContainerConfig (in OdeToFood.Web/App\_Start). The last statement in ContainerConfig.RegisterContainer()

DependencyResolver.SetResolver(new AutofacDependencyResolver(container));

completes the IoC (Inversion of Control) for MVC-5. We need a similar statement for Web API. One way to do this is to add the statement to the end of RegisterContainer().

httpConfiguration.DependencyResolver =

new AutofacWebApiDependencyResolver ( container );

The last two statements are similar, but using a different style. The Pluralsight instructor’s explanation is that Web API and MVC-5 evolved differently.

The statement that was just added has some problems: (1) httpConfiguration is not initialized (and also undefined). (2) AutofacWebApiDependencyResolver is also undefined. These issues are resolved as follows.

With regard to httpConfiguration . . .

* httpConfiguration is of type HttpConfiguration, and it will be provided by RegisterContainer()’s calling-sequence. Change the function statement

RegisterContainer()

To

RegisterContainer(HttpConfiguration httpConfiguration)

Place the mouse cursor in “HttpConfiguration”, and type “<ctrl>.” to induce VisualStudio to provide “**using** System.Web.Http”.

* RegisterContainer() is called by a statement in ApplicationStart() (in Global.asax.cs). We can use GlobalConfiguration.Configuration as the value that ApplicationStart passes to that new argument of RegisterContainer(). (Recall that GlobalConfiguration was initialized in ApplicationStart() by code suggested in the read-me that was provided when we installed the web-api NuGet package.) Change

ContainerConfig.RegisterContainer();

to

ContainerConfig.RegisterContainer(GlobalConfiguration.Configuration);

Hover the mouse over “.Configuration” that you just typed to confirm that this is of type HttpConfiguration.

With regard to AutofacWebApiDependencyResolver. . .

* This class does not exist in this project. Instead, it exists in yet another NuGet package that has to be installed. However, there is an easy resolution.
* Place the mouse cursor in “AutofacWebApiDependencyResolver”, and type “<ctrl>.” VisualStudio is smart enough to know about this particular type, and it suggests the package Autofac.WebApi2. Select that option, and choose to install the latest version.

There remains only one more minor change. The statement

builder.RegisterControllers(typeof(MvcApplication).Assembly);

(also in RegisterContainer()) registers all MVC-5 controllers in the project, but it does not register the Web API controllers. This is resolved by adding the statement

builder.RegisterApiControllers(typeof(MvcApplication).Assembly);

Finally build the project and test with the URL “/api/restaurants”; the response is an XML file containing the list of restaurants. Evidently the Chrome web browser chooses XML as the default format for the Web API response. You can use Fiddler to see the result in JSON format. See <https://www.tutorialsteacher.com/webapi/test-web-api>.

## Scaffolding a Restaurant API

The plan for this section is to modify the structure of the Web application to let the user navigate to a page dedicated to restaurants – where he could add, modify, delete, etc. in addition to retrieving the list of all restaurants.

The following steps are very similar to the ones that we performed in setting up the Greeting controller and view (refer to the end of [Defining Routes for ASP.Net](#_Defining_Routes_for)).

* Right-click OdeToFood.Web/Controllers, and select **Add > New Scaffolded Item…** VisualStudio displays a pop-up dialog titled “Add New Scaffolded Item”.
* Select **MVC 5 Controller – Empty**, and then click the **Add** command button. VisualStudio displays a 2nd dialog that allows you to name the controller. Name it “RestaurantsController”, and click the **Add** command button. VisualStudio responds by producing a skeletal form of RestaurantsController.cs and a new folder titled “Restaurants” under OdeToFood.Web/Views.
* The “view” counterpart to the new RestaurantsController.cs would be a file Index.cshtml in the new folder “Restaurants”. To create this easily, right-click anywhere in the body of RestaurantsController.Index(), and then select **AddView…**; VisualStudio pops up a dialog titled “AddView”; accept the defaults, and click the **Add** command button. VisualStudio creates and opens Index.cshtml.
* Change “<h2>Index</h2>” to “<h2>Hello</h2>”.
* Run the web application with the URL /Restaurants, and the result is the page displaying “Hello”.

Since the list of restaurants is going to be displayed on a page dedicated to restaurants, we no longer need to display the list of restaurants in the view OdeToFood.Web/Views/Home/Index.cshtml. Therefore, delete the code from this **razor** page between <ul> and </ul>. (Remember that you can use @\* . . . \*@ to comment out code from a **razor** page.)

All of the pages in this web application have a tabbed menu across the top with hyperlinks “Home”, “About”, and “Contact”. We need to add a new hyperlink tab “Restaurants”; and when the user clicks that link, it would induce navigation to the RestaurantsController that we built earlier in this section. These hyperlinks are coded in OdeToFood.Web/View/Shared/\_Layout.cshtml. Examine this razor page. Note the lines of code – between <ul> and </ul> that refer to “Home”, “About”, and “Contact”. We can insert the new Restaurants hyperlink by adding a similar line of code – immediately after the “Contact” hyperlink:

<li>@Html.ActionLink ( "Restaurants", "Index", "Restaurants" )</li>

The 1st argument, “Restaurants”, is the text displayed in the hyperlink.

The 2nd argument, “Index”, is the name of the method to invoke in the controller.

The 3rd argument, “Restaurants”, refers to the RestaurantsController.

To display a list of restaurants, of course we will have to have the appropriate **view** (**razor** page). But first, we need to modify OdeToFood.Web/Controllers/RestaurantsController.cs - to build a model and to relay that model to OdeToFood.Web/Views/Restaurants /Index.cshtml. This is similar to what we did when the list of restaurants was managed in OdeToFood.Web/Controllers/HomeController.cs; see [ASP.NET MVC Application – the Model, View, & Controller Combined](#_ASP.NET_MVC_Application_2).

* Add a constructor for the RestaurantsController class. The constructor takes one argument.

public RestaurantsController ( IRestaurantData db )

{

}

* Type <ctrl>. with the cursor on IRestaurantData to induce the appropriate **using** statement.
* Type <ctrl>. with the cursor on db to create and initialize the field db.
* In the Index() method, immediately before the statement

return “View();

build the **model** using a method of db.

var model = db.GetAll();

and supply “model” as the argument of “View”.

This time the **view** is somewhat more sophisticated. Fortunately VisualStudio makes this easy by supplying the code for the **razor** page automatically.

* We no longer need the **view** for restaurants that displays “Hello”; delete OdeToFood.Web/Views/Restaurants/Index.cshtml.
* As before, to start **view** creation open OdeToFood.Web/Controllers/RestaurantsController.cs, right-click anywhere in the body of RestaurantsController.Index(), and then select **AddView…**; VisualStudio pops up a dialog titled “AddView”.
* This time, instead of accepting defaults, expand the **Template:** drop-down list, and select **List**.
* Also, expand the **Model class:** drop-down list, and select “Restaurant (OdeToFood.Data/Models).
* Click the **Add** command button. VisualStudio creates and displays an excellent starting point for the **razor** page.

The @model directive at the top of the new **razor** might look familiar; it the same as what we supplied when building index.cshtml for the **view** that corresponds to the “Home” controller. We will discuss the “Create” **ActionLink** later in the course; it can be used to create a new restaurant. An HTML table is used to display the list of restaurants. The 1st row of the table contains code for the column headers. The **foreach** loop displays the table details. The page contains DisplayNameFor() and DisplayFor() helper functions that will be discussed later.

Build the project, and refresh the page. The more sophisticated view of restaurant data is displayed.

## Scaffolding Restaurant Details

Run the web application to display the “Restaurants” page. Hover the mouse over the “Details” hyperlink to the right of the “Scott’s Pizza” restaurant, and examine the text in the lower-left corner of the browser window; it shows that this hyperlink will instruct the web browser to navigate to /restaurants/detail/1; (Scott’s Pizza is the restaurant with id = 1). But navigating to that URL results in a 404-not-found error. Click the browser’s “back” button, and let’s fix that error.

* Open the controller, OdeToFood.Web/Controllers/RestaurantsController.cs. We need to add an ActionResult method to the class RestaurantsController

public ActionResult Details(int id)

{

}

The argument (int id) asks the MVC framework to look in the query string (URL) anything identified by “id”, which happens to be the 3rd part of “restaurants/detail/1”. Recall that in OdeToFood.Web/App\_Start/RouteConfig.cs, the 3rd part of the URL was named “id”.

* We need a way to get a restaurant from the data based on its id. Let’s increase the requirements for the interface of restaurants in OdeToFood.Data/Services/IRestaurantData.cs. Add - immediately after the statement containing GetAll() - the method

Restaurant Get ( int id );

Additionally we have to add the implementation of this method in OdeToFood.Data/Services/InMemoryRestaurantData.cs.

* + VisualStudio has already recognized that implementation is needed, and it denotes this by the red squiggly line under IRestaurantData.
  + Type <ctrl>. while the cursor is in IRestaurantData, and select **Implement interface**. VisualStudio adds a skeletal version of “ public Restaurant Get( int id )”.
  + Replace the “throw” statement with

return restaurants.FirstOrDefault ( r => r.Id == id );

The above is an expression in a C# extension titled “Linq”. The translation is “return the first restaurant (in the list of restaurants) where r.Id is identical to id (Get’s calling-sequence argument); if no such restaurant is found, return the default – which happens to **null** for this class.

* Now we complete the body the ActionResult method that we started above invoking Get(), which we have just defined . . .

var model = db.Get(id);

return View(model);

* This code will still not work until we have a **view** that recognizes the “Details” method. As before right-click anywhere in the body of RestaurantsController.Details(), and then select **AddView…**; VisualStudio pops up a dialog titled “AddView”.
* Expand the **Template:** drop-down list, and select **Details**.
* Expand the **Model class:** drop-down list, and select “Restaurant (OdeToFood.Data/Models).
* Click the **Add** command button. VisualStudio creates and displays a starting point for the restaurant-details **razor** page.
* Change

<h2>Details</h2>

to

<h2>Details for @Html.DisplayFor(model => model.Name)</h2>

* The restaurant’s cuisine has not automatically been provided. The Pluralsight instructor explained that the scaffolding engine does not deal with **enum** types well. But we can fix this by copying and modifying the <dt>…</dt> - <dd>…</dd> pair – replacing “.Name” with “.Cuisine” in the copy.

Build the project, and refresh the page. Try selecting the “Details” hyperlink for the different restaurants, and examine what the page displays. Observe that – on the details page- the “Edit” hyperlink does not work (not yet implemented), but that the “Back to List” hyperlink is already working. Also, try changing the id number in the URL; instead of 1, 2, or 3, try – for example – 7. The response is the error message “Object reference not set to an instance of an object”. We’ll try to fix that now.

## Working with Action Results

The error message (preceding paragraph) suggests that the null object reference is in the “model” argument that is passed from the controller to details.cshtml. Going back to the controller (OdeToFoods.Web/Controllers/RestaurantsController.cs), we recall that “model” was obtained from the Get() method of the class that implements IRestaurantData. That does not necessarily mean that the Get() method has a bug merely because it is returning a null value; on the contrary, we deliberately instructed it to provide a null value if the id does not refer to any restaurant in its list of restaurants. (Refer to “restaurants.FirstOrDefault()” in OdeToFood.Data.Services/InMemoryRestaurantData/Get().) The pertinent question is where to take remedial action and what to do.

* In OdeToFood.Web/Views/Restaurants/Index.cshtml we might use

@if ( Model == null )

{

// some kind of error message

}

else

{

// non-error code that we have now

}

The Pluralsight instructor advised using this kind of logic in the **view**, and said it would be preferable to implement this kind of decision making in the **controller**.

* Alternatively in OdeToFoods.Web/Controllers/RestaurantsController.cs, in the function Details(), we might test whether model contains null; if so, avoid invoking View(model).

if ( model == null )

{

return *something\_else*

}

* One possibility is that *something\_else* could be “RedirectToAction(“Index”), which means go back to the “Index” action for this same controller, or (with a 2nd argument) redirect to “Index” of a different controller.
* Another possibility is that *something\_else* could be “View(*name\_of\_view*). This is the remedy we will adopt. Insert the code

if ( model == null )

{

return View ( “NotFound” );

}

* Now we need to add the “NotFound” **view** -in OdeToFood.Web/Views/Restaurants/ - an empty view. (Refer to notes above to see how to add a view.) Write “Sorry!” between <h2> and </h2>. Follow that with “<div>That restaurant is not found!</div>”. Follow that with the hyperlink that lets the user revert to the list of restaurants. (Copy it from /Restaurants/details.cshtml.)

Build the web application, and test it again; change the id number in the URL. Observe that the web application displays the friendlier error page.

Take another look at /AppStart/RouteConfig.cs. Notice that the URL element “id” is specifically included as the 3rd name in the

url: "{controller}/{action}/{id}"

Notice, also, that “id” is used specifically in /Views/Restaurants/ index.cshtml in the 3 @html.ActionLink() statements (argument “new {id=item.Id}”) near the end of the file. What would happen if we chose to change the routing, i.e. replacing “id” with some other name, e.g. “key”, in /AppStart/RouteConfig.cs - both in the “url:” argument, and in the (subsequent) “defaults:” argument? Would we get an error when navigating from the restaurant list to the details for a chosen restaurant? Try making the change. The application continues to work without error. Since “id” is no longer available as a name of a URL element, MVC is smart enough to provide it as a query-string addendum - “id=*number”* at the end of the URL.

Undo the “key” for “id” substitution. {id} is more conventional for routing.

## Setting up a Create View

The view of our list of restaurants (OdeToFood.Web/Views/Restaurants /Index.cshtml) contains a hyperlink that the user might invoke to create a new restaurant. Until now it is not implemented.

To implement “create a new restaurant” we need to add a “Create” action in the “restaurants” controller (OdeToFoods.Web/Controllers/RestaurantsController.cs). Insert the code

public ActionResult Create()

{

return View();

}

As before, to create the view, right-click in this method, click **Add View…** in the pop-up menu; in the **Add View** pop-up dialog, select “Create” from the **Template:** drop-down list, select “Restaurant (OdeToFood.Data/Models)” from the **Model class:** drop-down list, and click the **Add** command button. Visual studio creates and displays the **view** (OdeToFood.Web/Views/Restaurants /Create.cshtml). Build the web application, and run it to see what we have.

The Pluralsight instructor made two comments about the form that Create.cshtml displays.

1. The cuisine-type of the new restaurant is missing.
2. The form does not actually create the restaurant in the data base; instead, it merely collects the information from the user. The **controller** will need to use that data – in another action item – to add the new restaurant to the data base.

The instructor also had some pertinent comments about the HTML helpers in Create.cshtml. Revisit this section of the course to get the benefit of those comments.

Now we add an input to Create.cshtml that allows the user to specify the cuisine.

* We start by copying the code in the first <div class=”form-group”> . . . </div> (code that is immediately below @Html.ValidationSummary()), and place the copy immediately below this <div>.
* In the copy, replace “Model.Name” with “Model.Cuisine” (in 3 places).
* Save/build/test the web application.

The form does permit the user to provide a cuisine type for the new restaurant. But we would prefer that it provide a list, from which the user could select the cuisine. What we need to do is to replace (in the 2nd copy) “.EditorFor” with another HTML helper.

* Erase “.EditorFor”. Then type “.” Immediately after “@Html”. VisualStudio displays a list of HTML helpers.
* Select – from the list – “EnumDropDownListFor”. This should create a drop-down list from the enum that we used earlier while creating the “Restaurant” model.
* Save/build/test the web application.

The form displays the drop-down list successfully.

Now we have to tackle the 2nd problem, that the form only returns the user input; we need more code to perform the create-restaurant action.

Add a 2nd “create” action method in the “restaurants” controller (OdeToFoods.Web/Controllers/RestaurantsController.cs). Precede this method with two attributes – as shown below.

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Create( Restaurant restaurant )

{

db.Add( restaurant );

return View();

}

The attribute [HttpPost] instructs the router to choose this action method in response to a “Post” request. The other action methods in RestaurantsController.cs have the attribute [HttpGet] by default. Some programmers believe that it good practice to set that attribute explicitly; therefore, I have added “[HttpGet]” at the beginning of all of the other action methods in this **controller**.

The attribute [ValidateAntiForgeryToken] discourages cross-site request forgery attacks. A cross-site request forgery is an attack in which a harmful script element, malicious command, or code is sent from the browser of a trusted user.

The db.Add() method is not yet defined.

* Update the interface (OdeToFood.Data/Services/IRestaurantData.cs). In the body of “interface IRestaurantData” add the statement

void Add ( Restaurant restaurant );

* Update the in-memory implementation (OdeToFood.Data/Services/InMemoryRestaurantData.cs). Insert

public void Add ( Restaurant restaurant )

{

restaurants.Add ( restaurant );

restaurant.Id = restaurants.Max ( r => r.Id ) + 1;

}

* Save/build the web application.

Test the web application by attempting to add a 4th restaurant. The information displayed at the end of the “Create” operation does not display any confirmation that the “Create” action was successful. To see this one would have to click the hyperlink to return to the list of restaurants. This “minor” user-interface flaw will be remedied later.

## Validating Models and Model State

The need for validation on the **Create** page is demonstrated by a simple example. As our web application currently exists, one could ignore the name box, select a cuisine, and click the **Create** command button. The result would be an added restaurant with no name.

The **razor** page OdeToFood.Web/Views/Restaurants /Create.cshtml contains a couple of HTML helpers that could be exploited to test and diagnose inappropriate user input.

One approach to validation is to use a data structure **ModelState** inside the [HttpPost] action method in the **controller**. For example to prevent the user from creating a restaurant with a null or blank name, change the body of this action method to

// The following does not handle the case when restaurant.Name contains only white space

// (spaces, tabs, etc.). It would not be difficult to (1) check for Null, (2) trim restaurant.Name,

// and finally (3) check for Empty.

if ( String.IsNullOrEmpty ( restaurant.Name ) )

{

ModelState.AddModelError ( nameof ( restaurant.Name ), “The name is required.” );

}

if ( ModelState.IsValid )

{

db.Add ( restaurant );

return View(); // to be replaced later with an indication of success

}

return View(); // displays the diagnostic

Save/build/test the web application. The diagnostic displays when the name is omitted. But this is a low-level approach. MVC provides an easier way.

## Validating Models with Data Annotations

The “better” way of doing validation is the use of Data Annotation. Remove (comment out) the 1st half of the code that was added in the preceding section; the part remaining would be the code beginning with “if ( ModelState.IsValid )”. Data Annotation employs an attribute in the **model**. In this case in OdeToFood.Data/Models/Restaurant.cs, add “[Required]” immediately before the code that specifies Restaurant’s “Name” property. (Use <ctrl>. with the cursor on “Required” to bring in the **using** statement for System.ComponentModel.DataAnnotations.) This is all that is needed to check for a null or empty restaurant name. Other useful Data Annotaions are

[RegularExpression(“*expression text*”)]

[Range(*minimum,maximum*)]

[DisplayFormat (DataFormatString = “*format strings for numbers or dates*”)]

[DisplayFormat (NullDisplayText = “*what to display when the value is null*”) ]

[DataType(DataType.Html)]

[DataType(DataType.Password)]

[DataType(DataType.Url)]

[DataType(DataType.EmailAddress)]

[Display(Name = "*identifying label*")]

[MaxLength(*integer indicating maximum number of characters permitted*)]

If multiple Data Annotations apply to the same item, you can place them all inside the same pair of square brackets separated by commas, or you can stack them on separate lines.

For this example I used “[Required, MaxLength(20)]” for the “Name” property, and I used “[Display (Name = “Type of Food”)] for the “Cuisine” property.

Build and test the web application. Note that the app displays an error message if the name is blank, or if you type more than 20 characters for the name. Notice that the label “Cuisine” has been replaced with “Type of food”.

## Following the POST-Redirect-GET Pattern

The other annoying problem – mentioned in earlier sections, is the fact that this web application ends the “Create” process by redisplaying the **razor** page OdeToFood.Web/Views/Restaurants /Create.cshtml. This is bad practice for 2 reasons.

1. It does not give the user a confirmation that he successfully created the new restaurant.
2. It is confusing; the user might click the **Create** button a 2nd time, which causes a second copy of the same restaurant to be created.

The problem (as we observed earlier) is in OdeToFood.Web/Controllers/RestaurantsController.cs, in the [HttpPost] version of the Crate() method, at the 1st “return View()” statement. One could redirect to the “Index” action (which displays the list of restaurants). This is satisfactory, but arguably a more useful place to go would be the page that displays the details for the newly created restaurant. This can be done by replacing the “return View()” statement with

return RedirectToAction ( “Details”, new { id = restaurant.Id } );

new { id = restaurant.Id } creates an object with an anonymous type, and with an id property whose value is the Id of the restaurant that we just created.

Build and test the web application. Observe what is displayed after creating a new restaurant.

## Implementing the Edit Action

The code to implement the **edit** hyperlink is very similar to what we used for the **create** hyperlilnk ([Setting up a Create View](#_Setting_up_a)). The steps are as follows.

* Add an [HttpGet] “Edit” action in OdeToFoods.Web/Controllers/RestaurantsController.cs. The code for obtaining the model is similar to the code in the Details() action method.
* Right-click in the “Edit” action method to induce VisualStudio to build the **view** (OdeToFood.Web/Views/Restaurants/Edit.cshtml).
* Fix the “Create” **view** to fix the missing “Cuisine” logic – in the form of an **enum** drop-down list.
* Add an [HttpPost] “Edit” action in OdeToFoods.Web/Controllers/RestaurantsController.cs. This will include a function call to a new method in the **model**, db.Update().
* Modify the model’s interface (OdeToFood.Data/Services/IRestaurantData.cs) to include signature of the new method.
* Add the new – db.Update() - method to the in-memory implementation (OdeToFood.Data/Services/InMemoryRestaurantData.cs).

Refer to solution **ASP\_NET\_MVC\_5** or **ASP\_NET\_MVC\_5\_W10** to see the detailed results of performing the steps above. Save, build, and test the web application.

## Working with SQL Server LocalDB

To test whether LocalDB has been installed in a version of VisualStudio 2019, one can use a command prompt. There is a command-prompt program that is delivered with VisualStudio 2019; to start it . . .

* Click the **Windows** start button in the lower-left corner of the screen. **Windows** pops up a list of programs. Below this list you will see a **Search** text box.
* Type “developer” in the **Search** text box. Before you finish typing “developer”, **Windows** will display the program name “Developer Command Prompt for VS 2019”; click the name, and **Windows** will launch a command-prompt window titled “Visual Studio 2019 Developer Command Prompt”.

At the command prompt type the command “sqllocaldb”. If LocalDB is installed, **Windows** displays the help information for this command. Next type “sqllocaldb i”. This asks for the instances of LocalDB on this computer. The list that **Windows** displays in response will probably include “MSSQLLocalDB” and “ProjectsV13”. The instance that we want to use is “MSSQLLocalDB”. To get more information about this instance, type “sqllocaldb i MSSQLLocalDB”.

In VisualStudio click the **View** menu. **VisualStudio** displays a pull-down menu, and one of the items on this menu is **SQL Server Object Explorer**. Click **SQL Server Object Explorer**. **VisualStudio** a tree view titled **SQL Server Object Explorer**. If the small triangle to the left of the **SQL Server** node is missing, you may have to add the server; if so, use “Windows Authentication”. I found that the triangle was present, and I expanded the tree by clicking the triangle. On the Windows 7 computer the expanded view showed “MSSQLLocalDB”, “ProjectsV13”, and “SQLExpress” (which I have already installed). On the Windows 10 computer (which thus far has not been used as much with Visual Studio) the expanded view showed only “MSSQLLocalDB”. This is no problem, because at this point we are focused on LocalDB. I next expanded the “MSSQLLocalDB” node. On the Windows 7 computer the expanded view showed a list of databases that I have used in earlier Pluralsight courses. On the Windows 10 computer the expanded view showed only the **System Databases** node. Again, this is no problem, because we will work with a new (to be created) database, “OdeToFood”.

## Installing and Configuring the Entity Framework

We need to install an entity-framework NuGet package into both projects – OdeToFood.Data and OdeToFood.Web. To install the package into more than one project . . .

* Right-click the solution. In the pop-up menu click **Manage Packages for Solution…** VisualStudio pops up a dialog titled **Manage Packages for Solution**. (The right-hand panel of this dialog is initially blank.
* Click **Browse** (upper-left corner of dialog), and type “entity framework” in the text box below **Browse**. VisualStudio displays a list of packages pertaining to Entity Framework.
* Click the package titled **EntityFramework**. VisualStudio displays – in the right-hand panel – (1) a list of projects and (2) **Options**. The Pluralsight instructor installed version 6.2.0 – listed as the latest stable version when he built the course. I decided to try using the most recent stable version – 6.4.4.
* Click the check boxes next to the 2 projects, and click the **Install** command button. VisualStudio displays a dialog titled **Preview Changes**.
* Click the click the **OK** command button. VisualStudio displays a dialog titled **License Acceptance**.
* click the **I Acept** command button. VisualStudio installs the packages.

One can verify that **Entity Framework** has been installed.

* OdeToFood.Web/Global.asax.packages.config contains an element whose id = “EntityFramework”.
* OdeToFood.Web/Web.Config has also been changed.
  + There is a <section> in <configSections> whose name = “entityFramework”.
  + There is an <entityFramework> . . . </entityFramework> section – near the end of Web.Config.

Unlike the experience of the Pluralsight instructor the <entityFramework> . . . </entityFramework> section in Web.Config does not contain a <defaultConnectionFactory> section. If I drop back to install version 6.2.0 of Entity Framework, the install process does insert a <defaultConnectionFactory> section, but even then it does not contain a <parameters> subsection. Let’s carry on, trusting that “improvements” made by Microsoft will not affect the outcome.

## Defining a DbContext

DbContext is the name of a base class. A class – defined in a VisualStudio project – that is derived from DbContext provides the gateway for obtaining access to any database. An appropriate place to create this class is in OdeToFood.Data/Services.

* Right-click OdeToFood.Data/Services. Select **Class…** from the pop-up menu. VisualStudio displays a pop-up dialog titled **Add New Item**.
* Select **Class** from the list of items, and provide a name in the **Name:** text box: OdeToFoodDbContext.cs. Click the **Add** command button. VisualStudio creates and displays the skeletal version of the new class.
* Add the prefix “public” and the suffix “: DbContext” to the class name. VisualStudio automatically provides the appropriate **using** statement for the namespace.
* The body of the class OdeToFoodDbContext could contain specifications for multiple tables in the database. On the other hand, when a database contains a very large number of tables, it is possible to use multiple Db-Context classes that refer to different parts this database. This approach is considered preferable when following principles of good software management.
* At this point we have only one table, a table of restaurants. The syntax for expressing this table is

public DbSet<Restaurant> Restaurants { get; set; }

(Apply <ctrl>. to “Restaurant” to induce VisualStudio to provide “using OdeToFood.Data.Models”.) “Restaurants” is the name of the table In the database.

* Refer to OdeToFood.Data.Models.Restaurant. The properties Id, Name, and CuisineType are the columns of the table. Data annotations for these properties instruct Entity Framework about attributes. “[Required]” translates to “column not nullable” in SQL Server. “[MaxLength(20)]” translates to the maximum number of characters that are permitted for the name.

## Creating a Data Access Service

Now that we have a class that defines the structure of the database, we use this class to define the counterpart to the InMemoryRestaurantData. Define a new class OdeToFood.Data/Services/SqlRestaurantData.

* Create and name the class using instructions similar to what we did earlier in this section to create OdeToFoodDbContext.
* Add the prefix “public” and the suffix “: IRestaurantData”, signifying that this class implements IRestaurantData.
* Apply <ctrl>. to “IRestaurantData” induce VisualStudio to provide “using OdeToFood.Data.Services”, and to induce VisualStudio to provide the skeletal version of the methods in this class.

To specify the definitions of the methods (Add(), Get(), GetAll(), and Update()) we need an instance of our gateway, OdeToFoodDbContext. There are two viable approaches, both of which employ a constructor for our SqlRestaurantData class: (1) we could create a new instance in the constructor of SqlRestaurantData; then copy that instance into a private field; (2) we could ask the environment to pass an instance of OdeToFoodDbContext as an argument of the constructor, and then copy that instance into a private field (similar to what we did in OdeToFood.Web/Conrollers/RestaurantsController.cs). The 2nd approach seems to be preferred, because we have already set up Autofac and dependency injection. Insert the following lines of code at the start of “class SqlRestaurantData”.

private readonly OdeToFoodDbContext db;

public SqlRestaurantData(OdeToFoodDbContext db)

{

this.db = db;

}

Now that we have the instance (db) it can be used to specify the actions.

The “unit-of-work design pattern” is a concept in Entity Framework: the unit-of-work consists of one or more statements that specify what is to be done when one makes changes to the database. To commit the actions specified in the unit-of-work, append the statement

SaveChanges();

With regard to the “Add” method, the unit-of-work statement is one statement, simply

db.Restaurants.Add(restaurant);

This statement says “add ‘restaurant’ (calling-sequence argument) to the’Restaurants’ table of the db context. There are several defaults of Entity Framework behavior that we are implicitly using.

* EntityFramework notices the “Id” property (i.e. column), and “Id” in Entity Framework parlance is the primary key of a table.
* This primary key is (by default) of type “int”.
* This column (by default) is an “identity” column, which means “increment the value by 1 each time you add a new record”.
* After adding the restaurant to the database, Entity Framework writes the value of Id into restaurant.Id (which would make this value available to our software).

With regard to the “Get” method, use a statement similar to what we used in InMemoryRestaurantData.

return db.Restaurants.FirstOrDefault(r => r.Id == id);

With regard to the “GetAll” method, if you are not concerned with the sort order, the statement is simply

return db.Restaurants;

On the other hand, if you want to return the restaurants sorted by Name, use

return from r in db.Restaurants

orderby r.Name

select r;

With regard to the “Update” method, this is slightly more complicated, because there is a risk of a conflict if more than one user is trying to update the same restaurant. The simple approach is to select the restaurant based on its Id, specify the changes, and then SaveChanges(). For example . . .

var r = Get(restaurant.Id);

if (r.Name != restaurant.Name)

{

r.Name = restaurant.Name;

}

if (r.Cuisine != restaurant.Cuisine)

{

r.Cuisine = restaurant.Cuisine;

}

db.SaveChanges();

Entity Framework, is smart enough to note that there is a change to the restaurant whose Id equals restaurant.Id. Therefore, Entity Framework commits those changes to the database in response to SaveChanges(). When this is being done by more than one user concurrently, and when there are different changes to one or more fields in the table, the change made by the last user to click the **Save** button will persist. If this is not satisfactory, one might use a technique called “optimistic concurrency”. This entails setting a flag when the user reads the fields of a restaurant. Later when the user is ready to make the changes, Entity Framework will tell him whether the restaurants fields have changed. The Pluralsight instructor gave a brief introduction about how to code for optimistic concurrency. What he suggested was

var entry = db.Entry(restaurant);

entry.State = EntityState.Modified;

db.SaveChanges();

But the instructor did not go into details of how to use that code with other code that makes the changes. In the Visual Studio example that I am maintaining, I used the 1st set of code (above), which I believe would be the correct code for non optimistic concurrency.

## Configuring a DbContext

There are some configuration changes. Recall that configuration comes in two flavors: (1) c# code – specifically OdeToFood.Web/App\_Start/ContainerConfig.cs – and (2) XML – OdeToFood.Web/Web.Config. We will need to make changes in both of these configuration files.

* In OdeToFood.Web/App\_Start/ContainerConfig.cs we registered RegisterType<InMemoryRestaurantData>. Specifically the existing statement is

builder.RegisterType<InMemoryRestaurantData>().As<IRestaurantData>().

SingleInstance();

We want to use our new class <SqlRestaurantData> instead of <InMemoryRestaurantData>. Also, we don’t want SingleInstance(), because we need a separate instances for multiple concurrent users. Substitute InstancePerRequest() in place of SingleInstance(). (InstancePerRequest() causes a new instance of DbContext to be created for each HTTP request; and this context is discarded when the request has completed.) As a result, the replaced statement should be

builder.RegisterType<SqlRestaurantData>().As<IRestaurantData>(). InstancePerRequest();

* At the same time (for each HTTP request) we need to register the type of the DbContext, specifically <OdeToFoodDbContext>. The statement that does this is

builder.RegisterType< OdeToFoodDbContext>().InstancePerRequest();

* In OdeToFood.Web/Web.Config we added a Connection String. Refer to the <connectionStrings> section in Web.Config in the Visual Studio solution, specifically the connection string whose name = OdeToFoodDbContext.
* We now have to specify the database that the DbContext converses with. For this purpose we use OdeToFood.Web/Web.Config. The Pluralsight instructor suggested adding a new section

<connectionStrings> . . . </connectionStrings>

between <appSettings> and <system.web>. We need one statement in the <connectionStrings> section, which adds a connection string for the OdeToFood database. The statement is

<add name=”OdeToFoodDbContext” connectionString=

”Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=OdeToFoodMVC;Integrated Security=SSPI” providerName=”System.Data.SqlClient” />

The name= value is a string that matches the name of the class derived from the DbContext. In Data Source= the value for Catalog= is a descriptive name for the catalog in the local DB. The ProviderName= value depends upon the type of database being used; the value shown above is appropriate for SQL Server.

Build and test the Web Application. It should run without errors, but it will show 0 restaurants.

## Exploring the New Database

Run the application, and click “Restaurants” at the top of the screen. Note that the list is empty, which is what we would expect for a new database with 0 content.

Look again at the Object Explorer in Visual Studio. Drill down SQL Server/(localdb)MSSQLLocalDB. Notice that Entity Framework is smart enough to create the OdeToFoodMVC database when it did not find this database. (If you do not see LocalDB initially, exit Visual Studio, and open it up again.)

Try using this Web Application to populate the database. Create a restaurant. Use “Scott’s Pizza” for the name, and “Italian” for the cuisine. Create a 2nd restaurant. Use “Mango Grove” for the name, and “Indian” for the cuisine. Navigate to the List of Restaurants.

Return to the Object Explorer. Drill down in OdeToFoodMVC > Tables. Right click dbo.Restaurants, and select **View Data**. VisualStudio displays the two restaurants plus their Id and Cuisine values.

If you want to change the content of a particular cell in a table, (1) click the mouse at that cell, (2) click the F2 function key, and (3) make changes in the same manner as you would in Excel or Word.

## Implementing a Delete Feature

When the user clicks the “Delete” hyperlink, it is good practice to ask for confirmation – in case the click was an accident. Therefore, we will need a pair of action methods in OdeToFood.Web/Controllers/RestaurantsController.cs – [HttpGet] followed by [HttpPost]. The 1st of these will display the data in the restaurant selected for deletion, and ask the user whether he wants to proceed with the deletion. The 2nd will check for a “yes” response; if so, it will perform the deletion.

* Add the following code to OdeToFood.Web/Controllers/RestaurantsController.cs.

[HttpGet]

public ActionResult Delete ( int id )

{

var model = db.Get(id);

if ( model == null )

{

return View (“NotFound”);

}

return View(model);

}

* This code will still not work until we have a **view** that recognizes the “Delete” method. As before right-click anywhere in the body of RestaurantsController.Delete(), and then select **AddView…**; VisualStudio pops up a dialog titled “AddView”.
* Expand the **Template:** drop-down list, and select **Delete**.
* Expand the **Model class:** drop-down list, and select “Restaurant (OdeToFood.Data/Models).
* Click the **Add** command button. VisualStudio creates and displays the template of a delete-restaurant **razor** page.

Examine this page. The first part of the page expresses the “are you sure?” question, and displays the name of the restaurant that it believes you want to delete. The bottom of the page displays two hyperlinks separated by a “|”. If you click the 1st of these, you are redirected to the [HttpPost] “Delete” action item. If you click the 2nd of these (labeled “Back to List”), you are redirected to RestaurantsController/Index(), which is the way you would say “no, I do not want to delete this restaurant”.

* Add the [HttpPost] “Delete” action item on the restaurants controller page.

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Delete(int id, FormCollection form)

{

db.Delete(id);

return RedirectToAction(“Index”);

}

The 2nd argument, FormCollection form, is provided only to prevent the C# compiler from complaining that the signature of this overloaded function is identical to the signature of the [HttpGet] Delete() function in this class. (The argument is otherwise ignored.)

* The statement “db.Delete(id);” in [HpptPost] Delete() refers to a method that does not yet exist. Modify the model’s interface (OdeToFood.Data/Services/IRestaurantData.cs) to include signature of this new method.

void Delete(int id);

* Add the new method to the in-memory implementation,

OdeToFood.Data/Services/InMemoryRestaurantData.cs.

public void Delete(int id)

{

var restaurant = Get(id);

if (restaurant != null)

{

restaurants.Remove(restaurant);

}

}

* Add the new method to the SQL Server implementation, OdeToFood.Data/Services/SqlRestaurantData.cs.

public void Delete(int id)

{

var restaurant = db.Restaurants.Find(id);

db.Restaurants.Remove(restaurant);

db.SaveChanges();

}

Build and test the web application. Delete a restaurant. Note that it is deleted from the “Index” view. Also, check the database in Visual Studio’s object explorer, and observe that the table contains one less restaurant.

## Understanding Razor Syntax

This section goes into detail about **razor** pages. At the outset the Pluralsight instructor mentions many of the elements of **razor** syntax. But since he does not demonstrate specific changes to our VisualStudio project (Ode to Food), I have decided to add a page, RazorSandbox, for the purpose of shadowing his instruction. I am creating RazorSandbox using steps similar to the Greeting controller and view. But before building RazorSandbox’s controller and view, I will modify the horizontal list of tabs to facilitate invoking this web page.

Earlier in the section [Scaffolding a Restaurant API](#_Scaffolding_a_Restaurant) we added a hyperlink tab “Restaurants”. Let’s use a similar change to create the hyperlink tab “RazorSandbox”. These hyperlinks are coded in OdeToFood.Web/Views/Shared/\_Layout.cshtml - between <ul> and </ul> that refer to “Home”, “About”, “Contact”, and “Restaurants”. Insert the new line of code immediately after the line that specifies the “Restaurants” hyperlink.

<li>@Html.ActionLink ( "RazorSandbox", "Index", "RazorSandbox" )</li>

The 1st argument, “RazorSandbox”, is the text displayed in the hyperlink.

The 2nd argument, “Index”, is the name of the method to invoke in the controller.

The 3rd argument, “RazorSandbox”, refers to the RazorSandboxController.

Of course, we don’t yet have a RazorSandbox controller; we will create it soon. Build and test the web application. The new hyperlink shows. But when we click it, we get a 404-not-found error message.

Earlier - in the last part of the section [Defining Routes for ASP.Net](#_Defining_Routes_for) - we created a controller and view for the “Greeting” page. Use similar steps to do the same for the RazorSandbox page. Build and test the web application.

Comments by the Pluralsight Instructor

Many of the view pages (.cshtml) begin with the directive **@model** (lower-case m). This is used to specify the type of object passed from the controller. For example, in OdeToFood.Web/Views/Restaurants/Index.cshtml, the **model** is of type

IEnumerable<OdeToFood.Data.Models.Restaurant>

which delivers the list of all restaurants. Subsequently – in the body of the .cshtml file, “Model” (with an upper-case M) refers to the instance of this argument.

‘@’ indicates that the following text is C# code. Curly braces ‘{‘ and ‘}‘ continue to enclose more C# code. If you want a string of characters explicitly delivered (as HTML), precede this text with “@:”.

‘@’ may need to be rendered explicitly either (1) as part of an email address, or (2) in some other context. Razor automatically recognizes the syntax of an e-mail address, and the embedded ‘@’ does not cause it to switch into C# mode. But to write

@abcde

into the output stream, you would need to code this as

@@abcde

## Specifying Layout Views

(This section of the document summarizes the Pluralsight sections **Managing Page Structure with Layout Views** and **Settings Layout Defaults with \_ViewStart**.)

Most of the **view**s are specifically requested by some **controller**; the Pluralsight instructor refers to these as “content” **view**s. These content views appear to be imbedded in some larger view, a view that provides the horizontal list of navigation hyperlinks at the top of the page and the footer. To attain this behavior there needs to be a setting that specifies the “Layout” property for each content view. The Layout property is automatically set globally (explained later in this section); but one can set this property manually in the content **view** to override the global setting. A good place to do this would be immediately after the

ViewBag.Title = . . .

statement. For example, one could use the statement

Layout = “\_TestLayout.cshtml”;

but only if you create \_TestLayout.cshtml. (OdeToFood.Web/Views/Shared/\_Layout.cshtml contains an example of a Layout view.)

Another possibility is that you may want to override the automatic Layout view to a no-layout-view setting. (Note that OdeToFood.Web/Views/Shared/\_Layout.cshtml does exist, but the browser was searching for \_TestLayout.cshtml in OdeToFood.Web/Views/Restaurants.) When you remove the “Layout =” statement, and retest the application, we can see that \_Layout.cshtml is being set by default for Restaurants/Index.cshtml. In fact, if you override that default by specifying a no-layout-view setting - -

Layout = null;

there is an obvious indication of missing features (e.g. no tabs across the top of the page). Using “Layout = null;” override can be useful as a debugging tool to assist in focusing on the data provided by the controller and view. Remove the “Layout = null;” statement after observing the web page’s appearance in the absence of a Layout property.

Where is the magic that connects \_Layout.cshtml to the various content **view**s? Notice that OdeToFood.Web/Views/Shared/\_ViewStart.cshtml contains a single C# statement.

Layout = “~/Shared/\_Layout.cshtml”;

This specifies the default for the page’s layout view. In summary, when rendering a content view, then

If the view contains a “Layout = . . .” statement, MVC-5 will execute the code (if any – i.e. when not null) in the specified Layout view. Otherwise, it executes the code in \_ViewStart.cshtml, which refers execution, in turn, to \_Layout.cshtml.

Later – in [Adding Flexibility with Layout Sections](#_Adding_Flexibility_with) – there will be more detail how flow of control transfers between the content **view**s and the Layout **view**.

## Using the View Bag

Examine the Layout view, \_Layout.cshtml, specifically the settings between <head> and </head>. The 1st two of those statements provide information that appears in all web pages produced by Visual Studio. The last two of those statements contain contains references to style-sheet files and java-script files. The remaining statement - identified by “<title>” - is somewhat curious. The <title> statement specifies the text to be displayed on the tab of the web browser. In this case the text is the property value of ViewBag.Title followed by “ – My ASP.NET Application”.

When you hover the mouse over “ViewBag”, you will note the prefix **dynamic** before its class name. **dynamic** means that its properties can change at run time. This makes it possible to use ViewBag as a dictionary with a collection of code/value pairs. To begin with ViewBag serves as a communication link to pass information (in this case the specific title) from an ordinary **view** (in this case Restaurants/Index.cshtml) to the Layout **view**. Following our example, note that the statement

ViewBag.Title = “Index”;

near the beginning of Restaurants/Index.cshtml sets the value of ViewBag.Title. Secondly ViewBag is elastic; the number of code/value pairs can increase during the life of the web application.

More commonly ViewBag serves as a communication link from a controller to the corresponding view. For example, examine OdeToFood.Web/Controllers/HomeController.cs – near the end of the code; specifically refer to the actions About() and Contact(). Both of these set a value for the property ViewBag.Message. An examination of OdeToFood.Web/Views/Home/About.cshtml and OdeToFood.Web/Views/Home/Contact.cshtml illustrate how these **view**s retrieve the ViewBag.Messageproperty. You can introduce a new ViewBag property name merely by assigning a value. For example, I tried

ViewBag.SNCole = “Stephen says hello.”;

In OdeToFood.Web/Controllers/RazorSandboxController.cs immediately before the “return View()” statement. It works.

## Adding Flexibility with Layout Sections

We observed in [Specifying Layout Views](#_Specifying_Layout_Views) that the Layout **view** is invoked either automatically (when the content **view** does not contain a “Layout =”) or explicitly by the name given the “Layout =” statement. But how is MVC-5 instructed to revert to the content **view**? Note the

RenderBody()

statement in OdeToFood.Web/Views/Shared/\_Layout.cshtml. This instructs MVC-5 to follow instructions in the content **view** to build HTML and substitute it in place of the RenderBody() statement.

In addition to the body of the content **view**, it might be convenient to insert other lines of HTML – perhaps in the header or perhaps in the footer. In fact \_Layout.cshtml has already made accommodation for java-script statements at the end of the HTML; refer to the statement – at the end of \_Layout.cshtml –

@RenderSection ( “scripts”, required: false )

The 1st argument, “scripts” is the name of the section; the section contains markup or java-script, and this would be inserted in place of the RenderSection() directive. The 2nd argument, required: false, indicates the active content **view** is not required to have a section named “scripts”, that it is not an error if “scripts” is missing from that **view**.

Let’s try an example of customizing the footer for the Restaurants “index” **view**. Insert into \_Layout.cshtml – immediately before </footer>

@RenderSection ( “footer”, required: false )

In OdeToFood.Web/Views/Restaurants/Index.cshtml insert the following code (the end of the markup - right after </table> - is a convenient insert point):

@section footer {

<div>This is a custom footer</div>

}

Rebuild and test the web application. Now the list-of-restaurants **view** contains the customized footer. Observe that “index” **view** is the only content view that contains “@section footer”; the other content views will not be penalized for not containing “@section footer”, because of “required: false”.

## Reusing HTML with Partial Views

Layout **view**s promote code sharing. The content **view**s in Ode To Food that invoke (by default) \_Layout.cshtml are exploiting the directives in <head> . . . </head>, the C#/HTML in the header, and the C#/HTML in the footer. MVC affords yet another way to attain code sharing. Pluralsight illustrates this by example.

Refer to the <table> in OdeToFood.Web/Views/Restaurants /Index.cshtml. This portion of the code displays the list of restaurants; it also allows the user to edit, display (drill down), and delete any restaurant in the list. Suppose we want to reuse this code in other places in the web application. We do this by employing a **partial view** named “\_RestaurantSummary”. Replace the code in <table> . . . </table> with

@foreach (var item in Model)

{

@Html.Partial ( “\_RestaurantSummary”, item )

}

Recall that – in this context – the Model is the list of restaurants; therefore, item refers to a single restaurant. The underscore in “\_RestaurantSummary” is not required; but it is a useful naming convention to distinguish between content **view**s and special **view**s. The “item” object in the code above parameterizes the markup provided by the **partial view** in a manner analogous to the way the “model” object parameterizes the content **view**.

You have the option of creating the **partial view** \_RestaurantSummary in the Views/Shared folder or in the Views/Restaurants. Placing it in the Views/Shared folder makes it available to be used by any content **view** in Ode to Food; placing it in the Views/Restaurants folder limits its scope to content **view**s that pertain to restaurants. Since its subject matter is “restaurants”, the latter is more appropriate.

* Right-click Views/Restaurants in the **Solution Explorer**; select **Add > View . . .** VisualStudio displays the pop-up dialog titled **Add View**.
* Put \_RestaurantSummary in the **View name** text box. Leave the default “Empty (without model)” in the **Template:** text box unchanged. Click the check box **Create a partial view**. Click the **Add** command button. Visual Studio creates the **partial view**’s .cshtmlfile, and displays it as a blank razor page.

In the manner analogous to a content **view** the **partial view** optionally begins with an @model directive; in this case the **model** for this **view** is an individual restaurant. On the 1st line of \_RestaurantSummary.cshtml type

@model OdeToFood.Data.Models.Restaurant

In addition to factoring out the code into the **partial view**, we will be using a series of “bootstrap panels” instead of <table>, <tr>, and <td>. (In another course a Pluralsight instructor said – emphatically – to use HTML table directives only for creating tables, and not for positioning.) I am uncertain what “bootstrap panels” means in this context, but evidently it is a judicious combination of <div>’s and CSS.

Next add (to \_RestaurantSummary.cshtml)

<div class=”panel panel-default” >

<div class=“panel-heading”>

<h3 class=”panel-title”>@Model.Name</h3>

</div>

<div class=”panel-body”>

@Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |

@Html.ActionLink("Details", "Details", new { id = Model.Id }) |

@Html.ActionLink("Delete", "Delete", new { id = Model.Id })

</div>

</div>

Build and test the web application.

## Partial View Postscript – Using CSS for <div> Positioning

Our Restaurants-Index view is not really satisfactory. What we had before the introduction of the **partial view** was one row for each restaurant – the name followed on the same row by the Edit-Details-Delete hyperlink trio. Now the hyperlink trio follows on the row below the name. This is happening, because – by default - <div> induces a new row. This can be overridden by inserting

style="float: left"

Into both the <div> that introduces the name and into the <div> that introduces the hyperlink trio; we also need

<div style="clear: both" />

After the hyperlink trio. This doesn’t quite work, because the names are of different lengths, and this prevents the hyperlink trios from lining up vertically. To solve this problem we can use an outer <div> that contains the list of restaurant names and a second outer <div> that contains the list of hyperlink trios. Inside the 1st outer <div> we code a foreach loop that repeatedly invokes \_RestaurantSummaryHeading, and Inside the 2nd outer <div> we code a foreach loop that repeatedly invokes \_RestaurantSummaryDetails. There remains one more problem; the hyperlink trios consume more vertical space than the titles; this prevents some of the restaurant names from lining up with their respective hyperlink trios. My solution for this is to force the names and hyperlink trios to use the same amount of vertical space by inserting

height: 30px

In summary the code in Index.cshtml is

<div style="float:left">

@foreach (var item in Model)

{

@Html.Partial("\_RestaurantSummaryHeading", item)

}

</div>

<div style="float:left">

@foreach (var item in Model)

{

@Html.Partial("\_RestaurantSummaryDetails", item)

}

</div>

<div style="clear:both" />

The code for \_RestaurantSummaryHeading is

@model OdeToFood.Data.Models.Restaurant

<div class=”panel panel-default”>

<div class="panel-heading" style="float: left; height: 30px">

<h3 class="panel-title">@Model.Name</h3>

</div>

</div>

The code for \_RestaurantSummaryDetails is

@model OdeToFood.Data.Models.Restaurant

<div class=”panel panel-default”>

<div class="panel-body" style="float: left; height: 30px">

@Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |

@Html.ActionLink("Details", "Details", new { id = Model.Id }) |

@Html.ActionLink("Delete", "Delete", new { id = Model.Id })

</div>

</div>

## Passing Data across Requests with TempData

Another useful data structure pertaining to **view**s is called **TempData.** **TempData** allows the web-app code to store server-side data, that persists for one server request before it is removed automatically. Except for the automatic removal, it is very much like **Session State** storage.

The following is an example of using **TempData**. (Incidentally it is also an example of how to produce a server-side alert.) In this example the user has finished editing a restaurant; recall that if the changes that he made are valid, control passes to the [HttpPost] Edit action method in RestaurantsController.cs. When we find that the changes made by the user are valid, we update the database via the statement

Db.Update(restaurant);

But before relaying control to the “Details” **view** for the restaurant, we want to send some text that confirms that the Edit action was successful. The **TempData** facility is well suited for this purpose. And the code that renders the “Details” **view** will retrieve and display this text.

* Navigate to OdeToFood.Web/Controllers/RestaurantsController.cs, action method [HttpPost] Edit ( Restaurant restaurant ).
* Insert – between the db.Update() and RedirectToAction() statements

TempData[“Message”] = “You have saved the restaurant changes.”;

* We have the option of retrieving and displaying this message in the content view (Details.cshtml) or in its **Layout view** - (OdeToFood.Web/Views/Shared/\_Layout.cshtml). The **Layout view** is preferable, because then it could be used in a context other than editing a restaurant.
* If you place the retrieve/display code immediately before the @RenderBody() statement in the **Layout view**, then it will appear at the top of the page. Insert the following code before @RenderBody().

@if ( TempData[“Message”] != null )

{

<div class=”alert alert-info”>

@TempData[“Message”]

</div>

}

<div class=”alert alert-info”> introduces a server-side alert, more specifically an informational alert.

Build and test the web application. Verify that the message displays after a change to the restaurant, but it is no longer displayed when one displays any page subsequently.

***The following 2 sections, Managing Client Libraries and Bundling Client Assets are probably introduced here to introduce some of the ingredients of an MVC project. I suspect that this background will be pertinent to web-application deployment, which follows shortly.***

## Managing Client Libraries

The term “Client Side Libraries” refers to the files in OdeToFood.Web/Scripts.

Many of these are Java-script files in JQuery format. When the script’s file name contains “.min”, that means that the script is minified, i.e. crunched down to make it very small – primarily to maximize down-loading speed; the minifier removes unnecessary white space, and it renames variables to reduce the number of characters in the names of the variables.

Some of the jquery script’s filenames contain “.validate”, and these scripts participate in client-side validation.

The script bootstrap.js contains some interactive components such as a rotating picture. This script works in conjunction with the bootstrap CSS files, which reside in OdeToFood.Web/Content.

The script whose name begins with “modernizr”. It deals with older web browsers; this allows clients with older browsers to use the web application.

One of the difficult requirements is keeping these scripts up-to-date. There are several possible approaches, and the instructor describes these briefly. The approach preferred by the instructor is facility built into Visual Studio.

* Right-click **OdeToFood.Web**. Then click **Manage Client-Side Libraries…** > **Add**  > **Client-Side Library…** VisualStudio responds by displaying a pop-up dialog titled “Add Client-Side Library”.
* We can use this dialog to update library files. For example, if we want to update jquery files, we would type “jquery” in the **Library:** text box followed by the **Enter** key. (As of today’s date, VisualStudio responds by appending “3.5.1”.) If I click the **Install** command button, VisualStudio would (1) install updated versions into a new folder **OdeToFood.Web/Lib**, and (2) create a new file **libman.json**.

**libman.json** maintains a record of which updates were installed, and it shows the destination folders of these installed new library versions. If you follow this approach, **libman.json** should be included among the files tracked by Source Control (e.g. **Git**).

When I was shadowing this course, I did not actually update the libraries; instead, I followed the example set by the instructor to continue using the versions provided by the template.

## Bundling Client Assets

The scripts and CSS files described in the previous section were introduced into the web-app solution, when we selected a template at the start of development. This template also set up “bundling” for client-side assets. This section defines and describes “bundling”.

Bundling – as the name suggests – involves combining several client-asset elements into a single bundle. It does this, because it can speed up the process of downloading these assets to the client’s computer. (This becomes less important if the client’s browser supports HTTP/2; if the browser is still at HTTP/1 or HTTP/1.1, bundling will be worthwhile.)

Refer to OdeToFood.Web/Views/Shared/\_Layout.cshtml. Note 2 statements in <head> . . . </head>:

@Styles.Render ( "~/Content/css" )

@Scripts.Render ( "~/bundles/modernizr" )

Also, note 2 statements at the end of <body> . . . </body>:

@Scripts.Render ( "~/bundles/jquery" )

@Scripts.Render ( "~/bundles/bootstrap" )

These are instructions to download and minify scripts and CSS files. (Refer to [Managing Client Libraries](#_Managing_Client_Libraries), which contains a brief explanation of “minify”).

It can be somewhat mysterious if you search the solution explorer for ~/Content/css or for ~/bundles/*some\_name*; you will not find them. Instead, the Pluralsight instructor describes them as “an abstraction”. However, look at OdeToFood.Web/App\_Start/BundleConfig.cs. This code (evidently executed at app start-up), governs the contents of these bundles. The element **{version}** in BundleConfig.cs represents some version number.

The bundler can detect (from Web.Config) whether you are running in **debug** or **release** mode. If you are running in **release** mode, it will use the minified versions of the client-side assets.

Try typing “,“ immediately before the closing parenthesis of any **Include()** in BundleConfig.cs. The pop-up help text is telling you that **Include()** will accept any number of arguments. An example of this is at the end of BundleConfig.cs, where **Include()** is receiving 2 string arguments, each of which refers to a different style-sheet file.

## Enabling Client Validation

In the OdeToFood sample web application we are doing validation on 2 web pages – create-restaurant and edit-restaurant. However, this is only server-side validation, not client-side validation. Server-side validation is by no means inappropriate; however, if we can also do client-side validation, then our web application is enhanced.

For example, when the user supplies a blank name for the restaurant, and then clicks the *Create* command button (on the create-restaurant page) or the *Save* command button (on the edit-restaurant) page, the validator displays the message

The name field is required.

Then if you had a client-side validator, that message should disappear when the user corrects the error by starting to type into the name text box.

With only a minor code change, the OdeToFood web application would have client-side validation in addition to the existing server-side validation.

To understand what is happening, . . .

* Run the web application, and navigate to the edit-restaurant web page.
* Right-click the *Name* text box.
* Click **Inspect** in the pop-up menu.

The browser displays part of the mark-up of the web page, and it highlights the mark-up pertaining to the *Name* text box. Read the highlighted text. Observe that the metadata for client-side validation are already in place - -

**data-val-maxlength**

**data-val-required**

the text to display when validation fails

the maximum number of characters permitted

Where did VisualStudio get this metadata? Look at the place where this restaurant-name input field is declared - in OdeToFood.Web/Views/Restaurants/Create.cshtml and OdeToFood.Web/Views/Restaurants/Edit.cshtml; both of these pages contain the statement

@Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })

This statement says create an input field (text box) to receive data of type model.Name; recall that “model”, in turn, is of type OdeToFood.Data.Models.Restaurant (see the 1st line of the .cshtml file). OdeToFood.Data.Models.Restaurant is declared in OdeToFood.Data/Models/Restaurant.cs, and the “Name” property is specified by the statement

public string Name { get; set; }

The metadata is provided by data annotation on the preceding line . . .

[Required, MaxLength = *maximum\_length\_value*]

With the validation metadata in place, the next ingredient we need is some java-script to perform the validation. This java-script is provided by the Script library files

odeToFood.Web/Scripts/jquery-*version\_number*.js

odeToFood.Web/Scripts/jquery.validate.js

odeToFood.Web/Scripts/jquery.validate.unobtrusive.js

The only ingredient left for client-side validation is some way to trigger this validation.

Recall – earlier in the course how VisualStudio was induced to provide starting points for the create-restaurant view (Create.cshtml) and the edit-restaurant view (Edit.cshtml).

* In OdeToFood.Web/Controllers/RestaurantsController.cs right-click in the Create() or Edit() Action method. VisualStudio displays a pop-up menu.
* Click **AddView…** in the pop-up menu. VisualStudio display the **Add View** dialog.
* Write the appropriate name – “Create” or “Edit” in the **View Name:** text box. Select **Create** or **Edit** from the **Template:** drop-down list. Select the restaurant’s model class “Restaurant (OdeToFood.Data.Models). Uncheck **Create a partial view**, and check **User a layout page:**.

It is significant that the **Reference script libraries** check box was not checked. If it had been done, VisualStudio would have inserted additional scripts into the razor view; it does this by adding 3 lines of code at the end of the .cshtmlfile:

@section Scripts {

@Scripts.Render("~/bundles/jqueryval")

}

The 1st additional line of code, “@section Scripts”. defines the “Scripts” section. Notice – at the end of OdeToFood.Web/Views/Shared/\_Layout.cshtml – the statement ‘@RenderSection(“scripts”, required: false)’, which says that if the content view (.cshtml file) has a “Scripts” section, place it here – at the end of the composite razor page. The order is important in \_Layout.cshtml; the ‘@RenderSection(“scripts”, required: false)’ statement must be placed after the ‘@Scripts.Render(“~/bundles/jquery”)’ statement.

The 2nd additional line of code, ‘@Scripts.Render("~/bundles/jqueryval")’, specifies which jquery scripts to download to the client. Where is ~/bundles/jqueryval defined? Look at OdeToFood.Web/App\_Start/BundleConfig.cs. Its 2nd bundles.Add statement says that ~/bundles/jqueryval consists of the library elements ~/Scripts/jquery.validate\*. The wild-card suffix (\*) gives additional qualification, i.e. jquery.validate.js plus jquery.validate.unobtrusive.js.

Let us demo the client-side validation by copying the 3 lines (above) to the end of Script.cshtml and Edit.cshtml. Build the web application. Then restart the test from scratch. Try to create a new restaurant; but click the “Save” command button while the “Name” text box is blank. The application displays a validation-error diagnostic – as before. As soon as you start typing text in the “Name” text box, the validation-error diagnostic disappears. This is evidence of client-side validation. Erase the text in the “Name” text box, and the validation-error diagnostic reappears. This is additional evidence of client-side validation.

What if the validation is more complicated? The answer is “don’t try to enforce error checking on the client side for complicated validation, but make certain that you have server-side checking for all of the possible validation errors.” The danger is that a hacker can circumvent client-side validation more easily than server-side validation.

## Creating Single Page Apps

When I shadowed the Pluralsight demo, I built a new web application, **ASP\_NET\_MVC\_5\_SNGL\_PAGE**.

The features of the single-page MVC web application are

Includes a framework to manage user identities; when you run the application it immediately asks the user to create a user-id and password.

The app makes API (Application Program Interface) calls to get data – in XML or JSON form - which it displays using one or more Java Script facilities.

* At the VisualStudio start-up page, under the **Get Started** column, click the **Create a new project** icon.
* Again – under the **Search for Template** combo box, we want language = **C#**, platform = **Windows**, and project type = **Web**. Next select **ASP.NET Framwork (.NET Framework)**. Click **Next**. VisualStudio brings up the **Configure your new project** dialog.
* In the **Project name** text box, type ASP\_NET\_MVC\_5\_SNGL\_PAGE. Select an appropriate folder for the solution and project (e.g. under GitRepositories\VSAndOtherPlatformProjects). Click **Create**. VisualStudio brings up the **Create a new ASP.NET Web Application** dialog.
* Here is where we deviate from what we did in [Creating a Blank Solution with a Start-up MVC Project](#_Creating_a_Blank); click **Single Page Application**. Click **Create**. VisualStudio scaffolds the web application, and presents it with the **Solution Explorer** in the right-hand pane.

Build and run the web application. You will note that the web app asks the user to provide credentials, or to register as a new user. Click the *Register as a new user* hyperlink. The web application responded by displaying a web page titled *Register. Create a new account.* My responses on that page were

*Email:* sncole@comcast.net

*Hometown:* Atlanta

*Password:* pw\_MVC\_5\_sngl\_page

The web application responded by opening the “Home” page. Notice – under *Your information* – the statement “Your Hometown is : Atlanta”. Notice also – in the upper-right corner – the statement “Hello sncole@comcast.net” followed by a hyperlink labeled “Log off”.

Click the *API* hyperlink in the upper-left corner to navigate to the API page. “GET api/me” indicates an API that is available in this web application. Click “GET api/me” for more details. This page, which seems to be informational suggests, that this single-page web application is making an API function call, and that the return value is my home town. The single-page web application is not retrieving HTML (razor views); instead, the Java Script on the client side (Jquery, Angular, Knockout, or . . .) will make HTTP calls to Get, Post, or Delete information from/to (presumably) another MVC web application that is negotiating with a database.

Click the *Home* hyperlink to return to the Home page. Almost all of this text is ordinary HTML (see Views/Home/\_Home.cshtml). But the little bit that displays “Your Hometown is : *hometown*” is placed via an Ajax call; see Scripts/app/home.viewmodel.js. (Much of this code is beyond my current Java Script experience, and it is probably beyond the scope of this course.)

Many of the ingredients that make up this project are very similar to what we encountered earlier in this Pluralsight course .

* The controllers in the **Controllers** folder are familiar. The *ManageController* class in Controllers/ManageController.cs contains code that deals with authentication (user ID, password, etc.) The *MeController* class in Controllers/MeController.cs is derived from *ApiController*; it can be invoked from JavaScript function, and it returns information either in XML or JSON format; the data are obtained via Get() – defined at the end of MeController.cs.
* Next turn your attention to the client side (Views). Look at Views/Home/Index.cshtml. @Html.Partial("\_Home") renders a Partial view named \_Home. (\_Home.cshtml - as we observed above - contains the markup for our home page.) Secondly Index.cshtml renders some scripts – a bundle for **knockout** and then a bundle for *app* (this application). You can find out what those bundles contain by looking at AppStart/BundleConfig.cs.
  + **knockout** is a JavaScript framework that offers client-side data binding and change detection, which makes it easier to build a single-page application.
  + The *app* bundle contains the most of the .js files in Scripts/app. (This includes Scripts/app/home.viewmodel.js mentioned briefly above.)

## Evaluating the Deployment Options

This section discusses various deployment options.

* Azure DevOps. This is covered in Scott Allen’s Pluralsight course “Azure for .NET Developers”.
* Use VisualStudio to publish to IIS – Local, Remote, or Virtual machine. (IIS is an abbreviation for Internet Information Services.) “Remote” means a computer somewhere else on the same network where VisualStudio is running. “Virtual” means a computer running in the Cloud.
* Use VisualStudio to publish to Azure App Services. This is covered in the Pluralsight course “Getting Started with Azure”.
* Use Visual Studio to publish to a File System. The result is a folder on the file system that contains all of the files needed for production. These files (perhaps zipped) can be copied to a place on the network to be shared with an administrator.

The last of these options (File System) is going to be covered in this course. After the application has been uploaded to a file system, the Pluralsight instructor will show how to point IIS to this collection of files.

## Installing Web Server Software

Software needed for web deployment to home computer:

* Another version of SQL Server. LocalDb (installed with Visual Studio) is difficult to use in working with IIS. There are two free versions of SQL Server – “Developer” and “Express”. The Pluralsight instructor will be demonstrating with SQL Server Developer; I have installed SQL Server Express on the Windows-7 computer.
* IIS installed on both home computers.

We can verify that SQL Server [Developer or Express] is working by . . .

* Launch VisualStudio.
* Click **View > SQL Server Object Explorer**. Visual Studio displays **SQL Server Object Explorer** (probably in the left pane) in tree-view format.
* If SQL Server [Developer or Express] is not shown in the tree . . .
  + Right-click the **SQL Server** node. VisualStudio displays a pop-up dialog titled **Connect**.
  + Expand the node **Local** (in the upper pane of the pop-up dialog). VisualStudio displays a list of database servers. On my Windows-10 computer one of these is SNCOLE0-PC (the name of this computer).
  + Click SNCOLE0-PC. VisualStudio copies this name to the **Server Name:** text box (in the lower pane).
  + The Pluralsight instructor is using his authentication as the administrator of his computer. The default offered to me is **Windows Authentication**, and I am using that option.
  + Click Connect. VisualStudio erases that pop-up dialog, and shows SNCOLE0-PC in the list of database servers.

I installed IIS on my Windows-7 computer while shadowing an earlier Pluralsight course – “Introduction to ASP.NET 4 Web Forms”, specifically the suite of sections under “Web Deployment”. My notes about this Pluralsight training are at

. . . \WebAppCourseNotes\IntroductionToASP\_NET\PracticesAndFeatures.docx

specifically in the sections titled “Configuring IIS Locally” and “Deploying Web Application by Hand”.

To install IIS on the Windows-10 computer, I followed the instructions in this Pluralsight course.

* Navigate to the Start Menu. Search for “features”; this is a quick way to find the Control Panel setting, “Turn Windows Features on or off”. Click this option in the pop-up menu; Windows displays a dialog titled “Windows Features”.
* Enable “Containers” and “Hyper-V”.
* Expand “Internet Information Services”.
  + Expand “Web Management Tools”; enable “IIS Management Console”.
  + Expand “World Wide Web Services”; expand “Common HTTP Features”; enable “Static Content”, “HTTP Errors”, “Directory Browsing”, and “Default Document”.
  + Expand “Application Development Features”; enable “.NET Extensibility 4.8” and “ASP.NET 4.8”.
* Click the OK command button.
* Reboot the computer, if instructed to do so.

We will be working extensively with the program “Internet information Services (IIS) Manager”. If it does not appear in the Start Menu, simply type the name in the Search box of the Start Menu. It will probably turn out to be convenient to pin Internet information Services (IIS) Manager to the Start Menu and to the task bar.

## Publishing Apps with Visual Studio

There are many different ways to publish a web application. We are going to use Visual Studio. An alternative would be to use the tools that Visual Studio uses (behind the scenes) – command-line MSBuild and WebDeploy; we will use this alternative when the time comes to automate the publishing process.

* Right-click the *OdeToFood.Web* project; then click **Publish…** VisualStudio displays a pop-up dialog titled **Publish**. This dialog contains 4 links – **new, edit**, **rename**, and **delete**.
* Click new. VisualStudio displays a second pop-up dialog titled **Publish**, which contains a variety of publish options.
* We could use **FTP/FTPS** (File Transfer Protocol); basically this means copying all of the needed files to the web server. We could publish directly to **Web Server (IIS)**; that would be too easy, and we would not learn much. The Pluralsight instructor has chosen to publish directly to a file folder. Then we will learn how to configure IIS to refer to this folder and run the application. Click **Folder**, then the **Next** command button. Visual Studio displays a dialog titled **Publish Provide the path to a local or network folder**.
* Following the example of the Pluralsight instructor, I chose to browse to a folder that is outside my Visual Studio development folder: Users\sncole\web\OdeToFood. Then click the **Finish** command button. VisualStudio reverts to the 1st dialog titled **Publish**, which now contains **FolderProfile** in the drop-down box.
* Click the **Publish** command button. VisualStudio builds the application (in release mode). Text should be displayed in **Output** showing a successful build and publish operation. Note that the result is in “Release” mode – which was the default setting.

Examine the publish file folder (Users\sncole\web\OdeToFood); observe that it is now fully populated with the files needed to run the web application. The \bin folder contains the assemblies used by the web application. The \Views folder contains the .cshtml files. What you will not find is pure C# source code (except for the C# code embedded in the .cshtml files).

## Creating an IIS Website

Navigate to the IIS Manager. The goal of this section is to instruct the IIS Manager to refer to our published web site Users\sncole\web\OdeToFood on port 80.

Make certain that **Default Web Site** – in the IIS Manager – is not configured to use port 80. (We could use the **Host Name** or the **IP Address** to make OdeToFood distinct from **Default Web Site**, but it is easier to use the **Port** number as a discriminant.) The steps for verifying or changing **Default Web Site**’s port number are as follows:

* Expand the root node and the **Sites** node, so that **Default Web Site** is exposed.
* Right-click **Default Web Site**, and click **Edit Bindings…** in the pop-up menu. IIS Manager displays the pop-up dialog **Site Bindings.**
* The **Host Name**, **Port** number, and **IP Address** are displayed in **Site Bindings**. Every web application managed by IIS Manager has a unique combination of these 3 components, and IIS Manager uses this combination to calculate how to route an incoming HTTP request.
* If the **Default Web Site** is using port 80, our approach is to change that port number for the **Default Web Site** and to adopt 80 for OdeToFood.
  + Click the row immediately below the column headings. IIS Manager highlights the row, and changes the **Edit…** command button to make it enabled.
  + Click the **Edit…** command button. IIS Manager displays the pop-up dialog **Edit** **Site Bindings.**
  + Type 81 (or another number in the **Port** text box. Then click the **OK** command button.
  + IIS Manager reverts to the **Site Bindings** dialog, and shows us that the port number is now 81. Click the **Close** command button to dismiss the **Site Bindings** dialog.

Now we will set up a new web in IIS Manager that points to the recently built folder for OdeToFood.

* Right-click the node **Sites** in the IIS Manager tree, and click **Add Web Site…** in the pop-up menu. IIS Manager displays the pop-up dialog **Add Web Site**.
* Type a friendly name such as OdeToFood into the **Site name:** text box. Use the browser **…** adjacent to **Physical path:** to navigate to the folder containing the published data for the web application. Make certain that **Port:** contains 80. We don’t need to change any of the remaining settings; leave the **Start Web site immediately** check-box checked. Before clicking **OK**, note that we will also be creating an application pool whose title (OdeToFood) has the same name as the name of the folder containing the published web-application data. (Presumably there might be a reason to use a different name in some other context.) One reason for noting the application pool name is that this name will be needed when we want to associate a database with this web application. Click the **OK** command button.

Observe that the friendly name now appears under the **Sites** node in IIS Manager.

Try to run the application by typing “Localhost” as the URL in an internet browser. It may take a while for the web application to respond, but eventually the application will display an error message:

## *A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 50 - Local Database Runtime error occurred. Cannot create an automatic instance. See the Windows Application event log for error details.)*

This error message is reminding us that our web application – at this point – is written to access local DB, and there are no instructions in the deployed web application that allow it to refer to local DB from IIS. Instead, we will need to change our deployed web application to contain instructions for referring to a database in the Developer version of SQL Server. (Although using IIS with local DB can be done, it is difficult, and the need for this combination is rare.)

## Configuring ASP.NET for Production

Web.Config contains the database connection string, and this is one of the files included in the published-files result, Users\sncole\web\OdeToFood. One could certainly edit Web.config in this folder to induce our web application to refer (correctly) to the Developer version of SQL Server. But what happens when the web application evolves, and someone publishes the new version of the web application? In doing so he would be overwriting the changes that we made to Web.config. We need a strategy that would automatically copy the appropriate database connection string during deployment to Users\sncole\web\OdeToFood. **Visual Studio** conveniently provides a facility for doing this.

Navigate to **Visual Studio** for this web application. In the **Solution Explorer** expand the node at OdeToFood.Web/Web.Config. In response **Visual Studio** displays two files – **Web.Debug.config** and **Web.Release.config**. These are known as Web.Config transformation files; they instruct **Visual Studio** how to transform Web.Config when it publishes Web.Config during deployment.

When the web application is published in “debug” mode, the transformation instructions in **Web.Debug.config** are applied.

When the web application is published in “release” mode, the transformation instructions in **Web.Release.config** are applied.

In particular if a specific database were appropriate for “release” mode deployment, one should put transformation instructions into **Web.Release.config** for inducing the application to refer to that database.

* Open **Web.Release.config**. The comment block at the top of the file pertains to database connection strings, and it provides an example.
* Move the --> part of this comment to the line immediately above <connectionStrings>. The <connectionStrings> block is now uncommented, and we will make modifications to the contents of this block that are pertinent to the **Developer** database.
  + To begin with the name “MyDB” needs to be changed to the name of our database. One can find that name in the <connectionStrings> block of Web.Config, i.e. “OdeToFoodDbContext”. Copy this name from Web.Config to overwrite “MyDB”.
  + In “connectionString” - “Data Source” (the name of the database engine) is the SQL Server installed on this computer; this is specified by using the name “.”.
  + In “connectionString” - “Initial Catalog” (the name of the database) can be any descriptive name; the Pluralsight instructor used “OdeTood”.
  + In “connectionString” - we could replace the “Integrated Security” with a user name and a password. The problem with doing this is that this configuration file will be maintained in source control (git), which could undermine security. Therefore, the Pluralsight instructor recommended retaining “Integrated Security=True”.
* The comment block at the bottom of **Web.Release.config** pertains to diagnostics displayed for unexpected errors.
* Move the --> part of this comment to the line immediately above <customErrors>. In release mode we would not want the user to see the very detailed diagnostics that are displayed when something has gone wrong. Instead, we intend to display an HTML page, whose page name is shown on the last line of the <customErrors> block. ***{Note from Stephen Cole} The Pluralsight course seems to be out-of-date with the version of Visual Studio delivered in 2020. (1) Web.Config (out of the box) does not contain a <customErrors> block, and this prevents Visual Studio from making the substitution of release-mode code from Web.Release.config to Web.Config. I corrected this omission by inserting <customErrors></customErrors> (at the end of the <system.web> block) into Web.Config. (2) We don’t need a new file, InternalError.htm, because the out-of-the-box file Views/Shared/Error.cshtml is an HTML file that should serve very well for a destination web page when a fatal error is encountered. I changed the name “InternalError.htm” to “Error.cshtml” in Web.Release.config in the last line of the <customErrors> block. I tested the changes above by examining Web.Config in the deployed block of files, and observed that the substitution had, indeed, occurred. However, I have not yet seen any instance where execution is redirected to Error.cshtml in the face of a fatal error.***

***{Note from Stephen Cole}***

Before resuming the Pluralsight course, it is instructive to see where we stand with regard to web deployment.

* Publish OdeToFood again: right-click the *OdeToFood.Web* project; then click **Publish…**; then click **Publish** command button in the pop-up dialog.
* Try to run OdeToFood: type Localhost:80 as the URL of a web browser. The result is an error diagnostic

## *Login failed for user 'IIS APPPOOL\OdeToFood'.*

The Pluralsight instructor warned us (above) that we would need additional configuration to allow our web application to Login to a non-local SQL Server when creating the new database. The exact words of the Pluralsight instructor were

But now, I need to make sure that SQL Server will accept the connection from the identity that my application is running with, and remember that my application is going to run in an app pool named ‘OdeToFood’. That’s what we saw configured inside of IIS. And so I need to make certain that SQL Server will accept this OdeToFood IIS identity, and will allow this identity to create a database. Because that’s what Entity Framework is going to do when it arrives on that server, and sees that there is no OdeToFood database. Of course, in a real production environment, if you’re working for a large organization and a large team, you can use tools with Entity Framework to generate a script, and use that script, and pass it through a DBA (database administrator) to create the database. But my goal in the next clip to configure SQL Server to allow my application to connect and create a database OdeToFood with all the tables it needs.

I am not certain that I fully understand what the Pluralsight instructor is saying. For example, if Entity Framework were not used to create the OdeToFood database, would it be necessary for Developer SQL Server to have log-in credentials to make data updates? But it is clear that the rationale for the next section is to show how to configure Developer SQL Server to allow my application to interact with it.

## Configuring a SQL Server Login

In this section we will configure Developer SQL Server to allow a connection with the OdeToFood web application running in IIS. We will need the name of the **App Pool** in which OdeToFood resides when we create the login. This **App Pool** happens to be named “OdeToFood”, also. But if you don’t know (or have forgotten) the name of the **App Pool**, you can retrieve this name from IIS Manager.

* Expand the top node of the **Connections** tree in the left pane of IIS Manager. IIS Manager displays the **Application Pools** and **Sites** nodes.
* Click the **Sites** node. IIS Manager displays a list of web sites in the center pane.
* Click the name of the web site that you are dealing with. IIS Manager modifies the right-hand pane (**Actions).**
* Click the **Basic Settings...** link in **Actions** pane. IIS Manager displays a pop-up dialog named **Edit Site**; this dialog contains the name of the **App Pool** in the center of the top row of this dialog.

Now we are ready to create and configure the SQL Server Login.

* Navigate to the SQL Server Object Explorer in Visual Studio. Visual Studio displays the information in tree format.
* Expand the node pertaining to Developer SQL Server. (On my Windows-10 computer this node is labeled SNCOLE0-PC . . .) Visual Studio displays 3 nodes: **Databases**, **Security**, and **Server Objects**.
* Expand the **Security** node. Visual Studio displays 8 nodes.
* Right-click the **Logins** node. Visual Studio displays a pop-up menu.
* Click the menu item labeled **Add New Login (SQL Server)…** Visual Studio opens a new page in the center pane titled **Login.sql**. This page contains default Transaction-SQL code for creating a user-name/password login.

**CREATE LOGIN [*some name*] WITH PASSWORD ‘*suggested password*’**

We need to modify this T-SQL code, because do not want a user-name/password login.

* Replace ***some name*** with

IIS APPPOOL\*app\_pool\_name*

In our case this is

IIS APPPOOL\OdeToFood

(Notice that we could have copied that syntax from the login-failure diagnostic – see above.)

* Replace **WITH PASSWORD ‘*suggested password*’** with

**FROM WINDOWS**

* The result is

**CREATE LOGIN [IIS APPPOOL\OdeToFood] FROM WINDOWS**

* Type **GO** on the subsequent line in this create-login script.
* Execute this script by (1) clicking **SQL** on the menu bar, and (2) clicking **Execute** in the pull-down menu.
* Next, we need to give this login permission to perform activities, such as allowing Entity Framework to create a database. Indeed, in a production environment, the DBA (database administrator) would be creating scripts for creating/modifying the database; and these scripts would be performed in a more controlled fashion. However, in this course it is sufficient to let our web application employ Entity Framework to create the database when it has not yet been created. To do this write a 2nd script below the create-login script above (in Visual Studio’s center pane).

*blank line*

**ALTER SERVER ROLE [dbcreator] ADD MEMBER [IIS APPPOOL\OdeToFood]**

**GO**

* Select the two lines of this 2nd script.
* Execute this 2nd script by (1) clicking **SQL** on the menu bar, and (2) clicking **Execute** in the pull-down menu.

For the purpose of record keeping I have stored a copy of login.sql in the Visual Studio solution **ASP\_NET\_MVC\_5\_W10** (on the Windows 10 computer) – in a new folder *OdeToFood.web/SQL Scripts*.

## Running in Production

In Visual Studio navigate back to the Publish dialog - right-click the *OdeToFood.Web* project; then click **Publish…** Observe that the settings for this **Publish** profile have been saved: (1) the name (by default) “FolderProfile” can be renamed via the **Rename** link; (2) the settings such as “Target location” and whether “Release” or “Debug” can be revised via the **Edit** link.

We could create new **Publish** profiles via the **New** link. One might have several **Publish** profiles for a given web application: for Q/A, for production, for testing, etc.

At this point the Pluralsight instruction suggests clicking the **Publish** command button to bring it up-to-date with regard to **Web.config** settings. If you followed the *{Note from Stephen Cole}* suggestion above to publish before creating the SQL Server Login, this would not be necessary, because we have not made any changes that would result in changes to the published results. But the redundant Publish operation can cause no harm.

Test the deployed OdeToFood web application via the web browser. Observe that there are no longer any displayed diagnostics, and that the web application has the same capabilities as before to create and update restaurants.

In Visual Studio’s SQL Server Object Explorer, drill down to and expand the **Databases** node. Observe that OdeToFood is listed as one of the available databases.