# 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.

|  |  |
| --- | --- |
| [ASP.NET MVC Application – the Model in Model-View-Controller](#_ASP.NET_MVC_Application) | [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) | [Following the POST-Redirect-GET Pattern](#_Following_the_POST-Redirect-GET) |
| [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) | [Passing Data across Requests with TempData](#_Passing_Data_across) |
| [Managing Client Libraries](#_Managing_Client_Libraries) |  |
|  |  |
|  |  |

Refer to solution **ASP\_NET\_MVC\_5**.

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

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.

This course chose a blank solution, and added a project whose template is .NET Framework.

The next step is to add a project to the solution. Visual studio 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 built-in code (from the template).

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 a subfolder of 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. 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 **enum**.
* 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.

I did not understand (and still do not understand) the arguments in the .OrderBy clause in the **return** statement of GetAll();. The instructor indicated that Visual Studio suggested this code from <ctrl>. applied to “OrderBy”; but at my version of VisualStudio this did not happen. I simply copied “r => r.Name” as suggested by the instructor.

## 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 JavaScript 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, there 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 Ode.ToFood.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”.
* The defaults in this dialog – most importantly the **View name** “index” – are exactly what we need. 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 “Hello!” 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 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/Interface1.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 resides 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 is 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” />

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. 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) the Autofac IoC container.

* Add a 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

var builder = new ContainerBuilder(); (VisualStudio supplies the appropriate “using”

statement automatically)

builder.registercontrollers (when you spell the method incorrectly VisualStudio prompts with

the correct spelling, and supplies the appropriate “using”

statement)

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

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

(.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));

## 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.
* 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 via the “??” (the null coalescing operator), and supply the alternative “no name ”.

## 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. In particular note that it important that

GlobalConfiguration.Configure(WebApiConfig.Register);

be added in a place where it is executed before

RouteConfig.RegisterRoutes(RouteTable.Routes);

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 necessary **using** statement.
* Click “<ctrl>.” after placing the mouse cursor in “db” to induce the creation and initialization of a private readonly field named “db”.
* 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. Change the return type of the function appropriately.

Now the code compiles successfully, but you will get an error when it runs. The error states that ‘RestaurantsController’ needs a parameterless public constructor. We had a similar error with MVC-5, and he 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 explantion 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 is obtained from the calling-sequence argument of RegisterContainer(). (The class HttpConfiguration is defined, and the appropriate **using** statement will be added automatically by VisualStudio.) Change the function statement

RegisterContainer()

To

RegisterContainer(HttpConfiguration httpConfiguration)

Place the mouse cursor in “HttpConfiguration”, and type “<ctrl>.” to remind 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(). 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.

* 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.
* Immediately before invoking “View()”, build the **model** as a property of db.

var model = db.GetAll();

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,.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

It is easy to infer from the error message (preceding paragraph) that the null object reference is in the “model” argument that is passed 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. 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.

## 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 <div class=”from-group”> . . . </div>, and place the copy immediately below this <div>.
* In the copy, replace “Model.Name” with “Model.Cuisine” (in 3 places).
* Save/build 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 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 in the “restaurants” controller (OdeToFoods.Web/Controllers/RestaurantsController.cs). Precede this action with a validate-anti-forgery-toke attribute. Insert the code

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

* Update the interface (OdeToFood.Data/Services/Interface1.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 you 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 **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

var restaurantName = restaurant.Name.Trim(); // handles the case when the name contains

// only white space

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

But this is a low-level approach. MVC provides an easier way.

## Following the POST-Redirect-GET Pattern

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 add “[Required]” immediately before the code that specifies Restaurant’s “Name” property in OdeToFood.Data/Models/Restaurant.cs. (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”.

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/Interface1.cs) to include signature of the new method.
* Add the new method to the in-memory implementation (OdeToFood.Data/Services/InMemoryRestaurantData.cs).

Refer to solution **ASP\_NET\_MVC\_5** 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, I expanded the tree by clicking the triangle, which showed me “MSSQLLocalDB”, “ProjectsV13”, and “SQLExpress” (which I have already installed). Right now we are focused on LocalDB. I expanded the “MSSQLLocalDB” node, and **VisualStudio** displayed a list of databases that I have used in an earlier Pluralsight course. We will ignore these for now, and 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.

## 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 a database. An appropriate place to create this is 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. At this point we have only 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”.) In the database “Restaurants” is the name of the table.

* 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 column 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: (1) we could create a new instance in a constructor of SqlRestaurantData; then copy that instance into a private field; (2) ask the environment to pass an instance of OdeToFoodDbContext as an argument, and then copy that instance into a private field; this 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 to the database; finally one writes a statement that commits these actions – in the form of a “SaveChanges()” statement.

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

db.Restaurants.Add(restaurant);

This refers to the “Restaurants” table, and the calling-sequence argument “restaurant”, which is to be added to the table. 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 will write the value of Id into restaurant.Id.

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, being smart enough to note that there is a change to the restaurant with Id = restaurant.Id, 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 the Name field, 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 wrote what 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.

* In OdeToFood.Web/App\_Start/ContainerConfig.cs we registered RegisterType<InMemoryRestaurantData>. We are replacing this with our 2 new types <SqlRestaurantData> and <OdeToFoodDbContext>. Refer to ContainerConfig.cs in the Visual Studio solution for details.
* 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.

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

## Exploring the New Database

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.

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.

## 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 **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.
* The restaurant’s cuisine has not automatically been provided. The Pluralsight instructor explained (earlier) 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.
* Add the 2nd ([HttpPost] action item.

[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 function is identical to the signature of the 1st Delete() function in this class. (The argument is otherwise ignored.)

* The statement “db.Delete(id);” in the preceding step refers to a method that does not yet exist. Modify the model’s interface (OdeToFood.Data/Services/Interface1.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 the “Mango Grove” restaurant. Note that it is deleted from the “Index” view. Also, check the database in Visual Studio’s object explorer, and observe that now there remains only one restaurant.

## Understanding Razor Syntax

This module 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, RazorDemo, for the purpose of shadowing his instruction. I am creating RazorDemo using steps similar to the Greeting controller and view. But before building RazorDemo’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 “RazorDemo”. These hyperlinks are coded in OdeToFood.Web/View/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 ( "RazorDemo", "Index", "RazorDemo" )</li>

The 1st argument, “RazorDemo”, 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, “RazorDemo”, refers to the RazorDemoController.

Of course, we don’t yet have a RazorDemo 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 RazorDemo 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. Use

Layout = null;

Notice the file OdeToFood.Web/Views/Shared/\_ViewStart.cshtml. It contains a single C# statement.

Layout = “~/Shared/\_Layout.cshtml”;

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** 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. Secondly it 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 providing information (the specific title) from an ordinary **view** (e.g.Index.cshtml) to the Layout **view**. 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/RazorDemoController.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 javascript 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 javascript, 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. The list-of-restaurants **view** now contains the customized footer.

## 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.

## 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. At this point we want to send some text to the restaurant’s “details” **view** – specifically Views/Restaurants/Details.cshtml. And we will catch and display this text from this **view**.

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

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

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

}

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 the details page subsequently.

## Managing Client Libraries

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

Many of these are JQuery scripts. 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 librarie; instead I followed the example set by the instructor to continue using the versions provided by the template.