|  |  |
| --- | --- |
|  |  |
| Due Date: | November 18, 9AM |
| Percentage of ASP.NET 1 module mark: | 10% |
| Submission Location: | / ASPDotNET/Dropbox/ |
| Submission Details: | All code must be submitted in one document. |
| Late Penalty: | 20% deducted each day this assignment is late. |

Contents

[MVC Introduction 1](#_Toc466467955)

[Adding a Home Controller 3](#_Toc466467956)

[Handling Parameters in Your Controller 5](#_Toc466467957)

[Adding a Weakly Typed View 6](#_Toc466467958)

[Razor Syntax 7](#_Toc466467959)

[ViewData 10](#_Toc466467960)

[ViewBag 11](#_Toc466467961)

[Layout Pages 13](#_Toc466467962)

[Models 13](#_Toc466467963)

[Adding a Detail View 17](#_Toc466467964)

[Adding a List View 20](#_Toc466467965)

[Get and Post 23](#_Toc466467966)

[Referencing an Entity Data Model 26](#_Toc466467967)

# MVC Introduction

The MVC architecture is driven by these three components:

**M**odel: Classes for manipulating data.

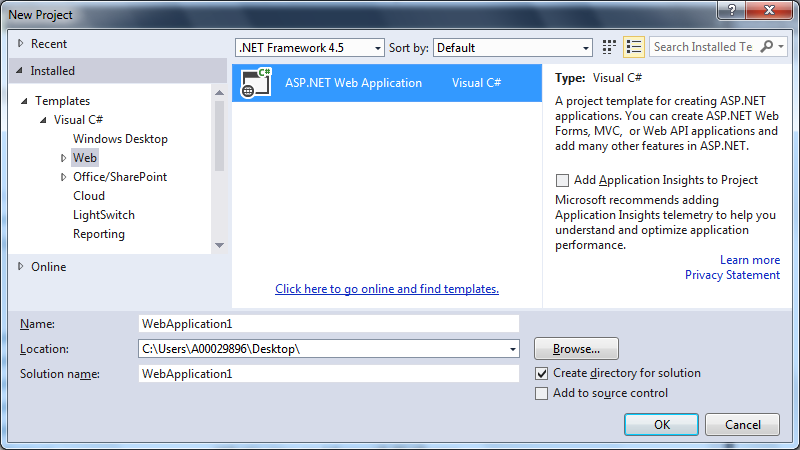
**V**iew**:** Output generated from HTML templates.

**C**ontrollers**:** Dispatcher which receives page inputs and issues outputs.

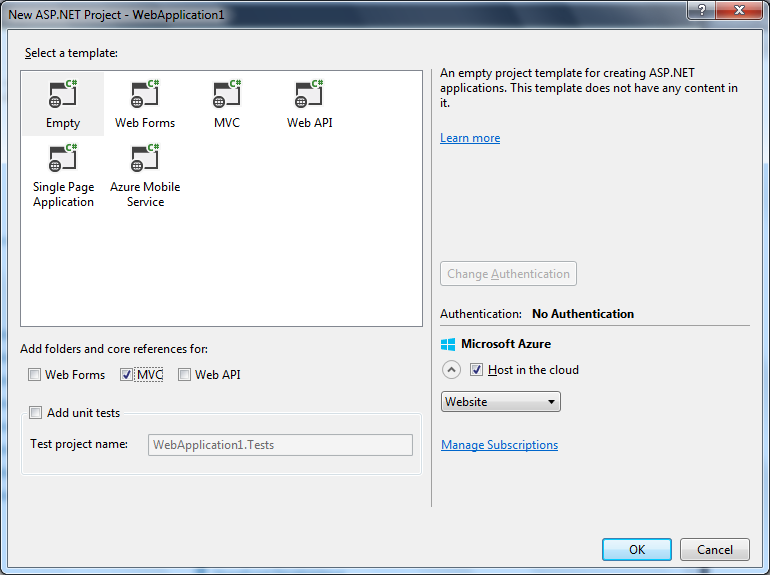
To understand this architecture better, let’s look at some examples.

Example : Controller Introduction

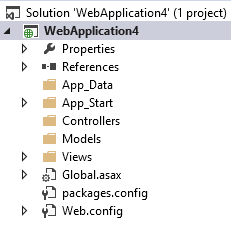
🞑 In this example, we will create an MVC application that uses a controller to generate output. To begin, create an ASP.NET MVC Web application.



Then, create an empty MVC application.

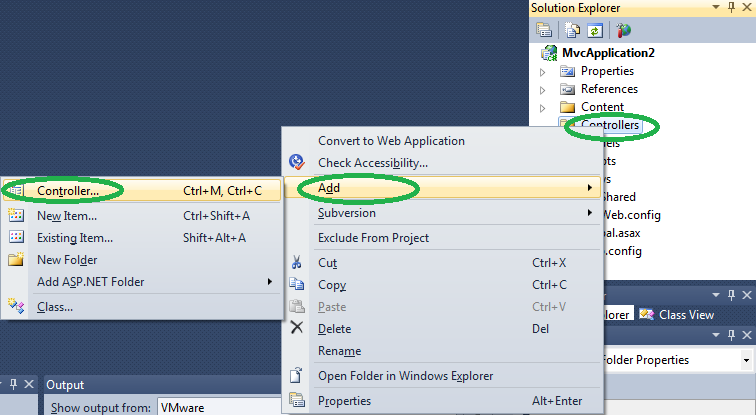


When the empty project is generated, the folder structure needed is built for you.

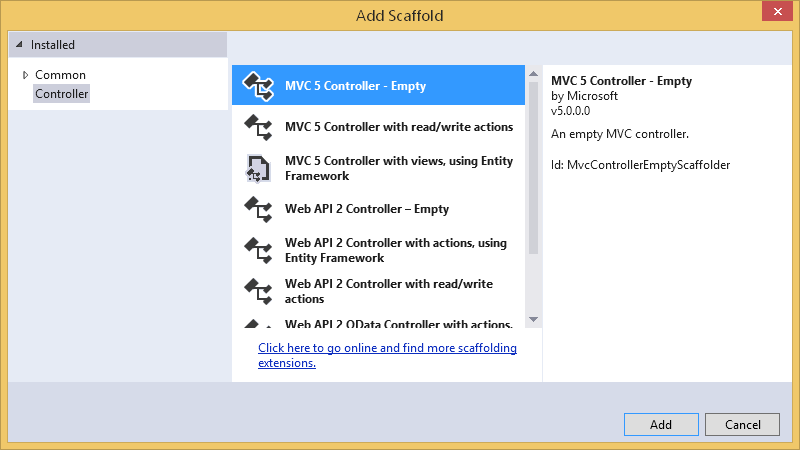


## Adding a Home Controller

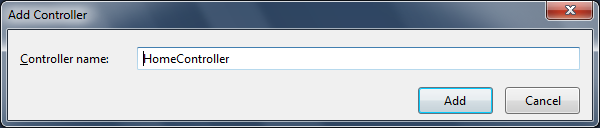
Now we will add a controller to the empty project. The **Home** controller is the default controller that gets selected when none is specified. The **Home** controller manages page calls to and from the project’s root directory. To create a **Home** controller, right click the **Controllers** folder, choose **Add**, and select **Controller**.



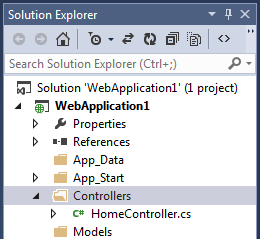
Then, when prompted choose to add an Empty controller.



Then, in the dialog that appears enter HomeController in the Controller Name text box, make sure that the Empty MVC controller template is selected, and choose Add:



Once you do this, you will notice that a **HomeController.cs** file is generated for you.



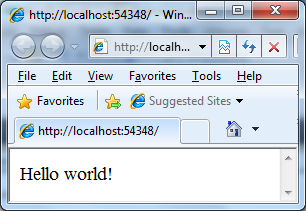
The auto-generated controller in this file is:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  namespace WebApplication1.Controllers {  public class HomeController : Controller {  // GET: Home  public ActionResult Index() {  return View();  }  }  } |

Try replacing the Index() method with this revised version.

|  |
| --- |
| public string Index(){  return "<html><body>Hello world!</body></html>";  } |

When you run your project, you will see the following output in the browser;



Exercise (1 mark)

🖍 View the page source from solution. In the box below, paste the HTML code that is generated by the browser when you run the project:

|  |
| --- |
|  |

Exercise (1 mark)

🖍 In , what is the difference between the return type of the original *Index()* method that was generated when adding a controller with the new *Index()* method that you implemented?

|  |
| --- |
|  |

## Handling Parameters in Your Controller

ASP.NET MVC controllers enable and enforce strongly typed URL parameters.

Example : Passing Parameters to the Controller

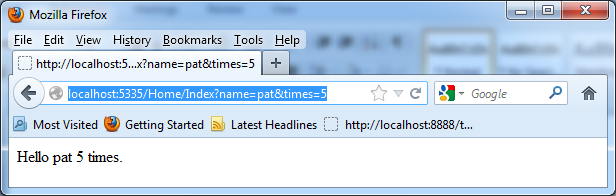
🞑 This example demonstrates simple parameter handling in the controller. To start, create an MVC 5 application that is empty. Then add a Home controller and replace the Index() method with this revision that accepts parameters:

|  |
| --- |
| public string Index(string name, int times = 1){  string message = "<html><body>Hello " + name  + " " + times + " times.</html>";  return message;  } |

Then, navigate to [http://localhost:**<yourdevserverportnumber>**/Home/Index?name=pat&times=5](http://localhost:%3cyourdevserverportnumber%3e/Home/Index?name=pat&times=5)

**Note:** Your development server port number will appear in the URL when you run your project.

If successful, you should see your parameters appear in the output.

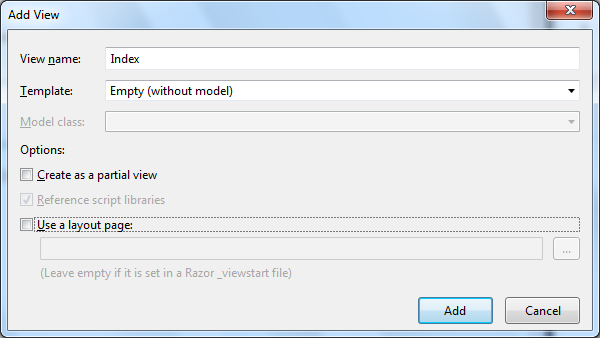


## Adding a Weakly Typed View

So far, we have manually generated a view but we have been outputting HTML through a manually built string. ASP.NET MVC can automate the process of generating a view through a template.

Example : Adding a Weakly Typed View

🞑 In this example, we will see how to add a weakly typed view. To begin, create a new MVC 5 project that is empty. Add a Home controller. Then, right click the *Index()* action method header and choose Add View. Enter the values that are shown below in the **Add View** dialog and click **Add**.

****

When you click Add, an Index.cshtml file in the Views/Home directory. Next, add the following html in between the div tags inside the body of the view that was just created.

<b>Hello world!</b>

Exercise (1 mark)

🖍 In the box below, paste the HTML code that is generated in your browser when you run the application.

|  |
| --- |
|  |

# Razor Syntax

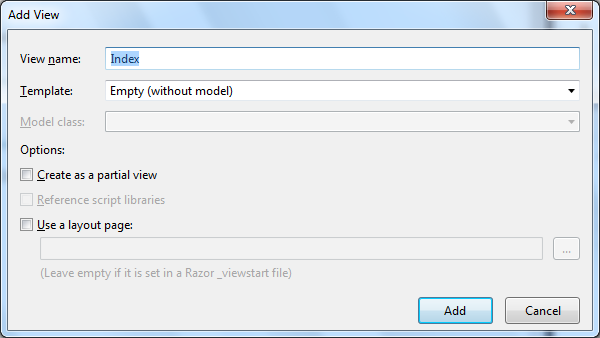
Razor syntax allows you to nest programmatic code inside your HTML. It was introduced in January 2011. Here is a summary of razor syntax rules taken from <http://www.w3schools.com/aspnet/razor_syntax.asp>

* Razor code blocks are enclosed in @{ ... }
* Inline expressions (variables and functions) start with @
* Code statements end with semicolon
* Variables are declared with the var keyword
* Strings are enclosed with quotation marks
* C# code is case sensitive

Implementing large amounts of razor syntax is not recommended because this logic generally can only be used once. For complex logic, other reusable alternatives are preferred.

Example : Razor Syntax Variable Declaration, Initialization, and Implementation

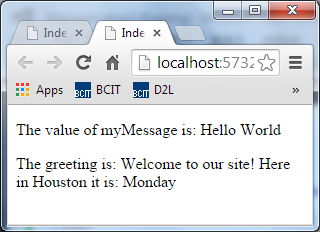
🞑 Here is a brief sample from <http://www.w3schools.com/aspnet/razor_syntax.asp>. To begin, create a new **MVC 5 Web Application** and select an **Empty** template with Razor syntax enabled to create your project. Then, right click the Controllers directory and select **Add | Controller**. When prompted, add a controller named **HomeController** and select the **Empty MVC Controller** template. Next, to add the view, right click the Index() action method that is generated in the controller and select **Add View** in the dialog that appears. Your settings should resemble the following:



Then, in the div tags within the body tags of the view template that is generated, add the following

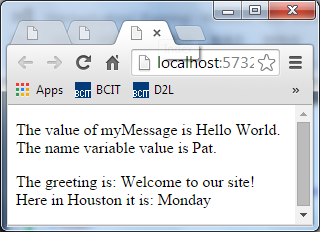
|  |
| --- |
| **<!-- Single statement block -->**  @{ var myMessage = "Hello World"; }  **<!-- Inline expression or variable -->**  <p>The value of myMessage is: @myMessage</p>  **<!-- Multi-statement block -->**  @{  var greeting = "Welcome to our site!";  var weekDay = DateTime.Now.DayOfWeek;  var greetingMessage = greeting + " Here in Houston it is: " + weekDay;  }  **<!-- Output -->**  <p>The greeting is: @greetingMessage</p> |

When you run the application, the output is:



Exercise (1 mark)

🖍 Starting with the solution from Example 4, in razor syntax, declare a variable named *myName* and assign your name to it. Then, adjust the output so your name displays your *myName* variable contents in the output like in the dialog below (but with your name).



Show your revised view code here:

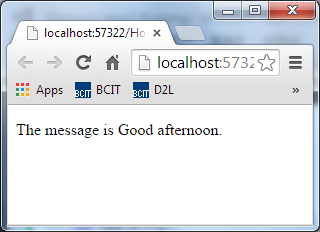
|  |
| --- |
|  |

Example : Implementing Logic in Razor Syntax

🞑 Aside from declaring and manipulating variables, it is possible to perform complex logic in ASP.NET. To examine this, create an empty MVC 5 application with a Home controller. Add a view for the Index action method and replace all code in the view with the following:

|  |
| --- |
| <!DOCTYPE html>  @{  var txt = "";  if (DateTime.Now.Hour > 12)  { txt = "Good afternoon."; }  else  { txt = "Good morning."; }  }  <html>  <body>  <p>The message is @txt</p>  </body>  </html> |

The output that is generated displays the message ‘Good Morning’ or ‘Good Evening’ is displayed on when you run it.



Exercise

🖍 Why is use of large amounts razor code not recommended?

|  |
| --- |
|  |

# ViewData

ViewData is a global dictionary that allows you to store data in the controller with name / value pairs. The ViewData class has been available sinceASP.NET MVC 1. ViewData can be stored in the controller with syntax like:

ViewData["my\_message"] = "Hello from the ViewData dictionary";

ViewData["my\_number"] = 4.5f;

ViewData can also store lists and arrays:

List<string> GetStringList() {

List<string> names = new List<string>();

string[] namesArray = {"John", "Paul", "Ringo", "George" };

names.InsertRange(0, namesArray);

return names;

}

ViewData["my\_list"] = GetStringList();

# ViewBag

The ViewBag class allows you to define arbitrary properties for the ViewData global dictionary within the controller. Data is assigned to the ViewBag properties inside the controller. Data can be retrieved from the ViewBag in the view. The ViewBag is new as of MVC3 and it is preferred over ViewData for storing and retrieving arbitrary data in controllers and views.

This tiny sample shows data being stored in the ViewBag from inside a controller:

**// Store data using ViewBag properties.**

ViewBag.MyVariableProperty = "ViewBag is a weird name!";

ViewBag.MyViewBagList = GetStringList();

This sample shows how to retrieve the data inside the view:

@Html.Raw(ViewBag.MyVariableProperty)<br />

@{

var names = ViewBag.MyViewBagList;

@Html.Raw("<br />");

foreach(string name in names) {

<text>@Html.Raw(name)</text><br />

}

}

**Warning:**

Some developers suggest that the ViewBag and ViewData dictionary never be used in favour of well-designed models and view model classes. While I agree on the need for smart class selection, sometimes a ViewBag is very convenient.

Example : Razor Syntax, View Data, and ViewBag

🞑 This example demonstrates how to add and retrieve data to and from a ViewData dictionary and also to and from a ViewBag with properties. In a new empty MVC 5 application, create a Home controller. Then replace the existing Index action method with the following code:

|  |
| --- |
| List<string> GetStringList() {  List<string> names = new List<string>();  string[] namesArray = {"John", "Paul", "Ringo", "George" };  names.InsertRange(0, namesArray);  return names;  }  public ActionResult Index(){  **// Store data with ViewData dictionary.**  ViewData["my\_message"] = "Hello from the ViewData dictionary";  ViewData["my\_number"] = 4.5f;  **ViewData["my\_list"] = GetStringList();**  **// Store data using ViewBag properties.**  ViewBag.MyVariableProperty = "ViewBag is a weird name!";  ViewBag.MyViewBagList = GetStringList();  return View();  } |

Next, add a weakly typed view for the Index action method. Inside the div tags place the following code:

@Html.Raw(ViewData["my\_message"])<br />

@Html.Raw(ViewData["my\_number"])<br />

@{

foreach (var firstName in (ViewData["my\_list"] as IEnumerable<string>)) {

@Html.Raw(firstName);<br />

}

}

<br /><br />

**<!-- Show ViewBag properties -->**

@Html.Raw(ViewBag.MyVariableProperty)<br />

@{

var names = ViewBag.MyViewBagList;

@Html.Raw("<br />");

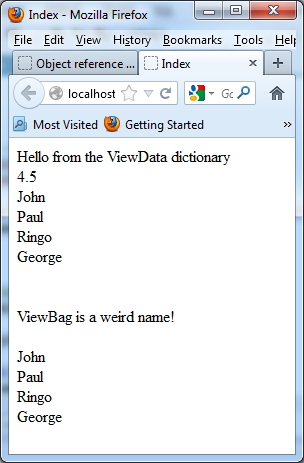
foreach(string name in names) {

<text>@Html.Raw(name)</text><br />

}

}

When running the project, our output looks like:



Example (3 marks)

🞑 Using the List class, create a list of integers containing the values 1, 11, and 111. Store the list in the ViewData dictionary and also in the ViewBag. Then using Razor syntax in the view, calculate and display the sum of the integers that are contained in the ViewData dictionary and display the sum. Do the same for the ViewBag. Show your controller code here:

|  |
| --- |
|  |

Show your view code here:

|  |
| --- |
|  |

# Layout Pages

A layout page is a re-usable application template that can be used to encase nested views. The layout may also reference CSS which applies formatting to the layout and nested view.

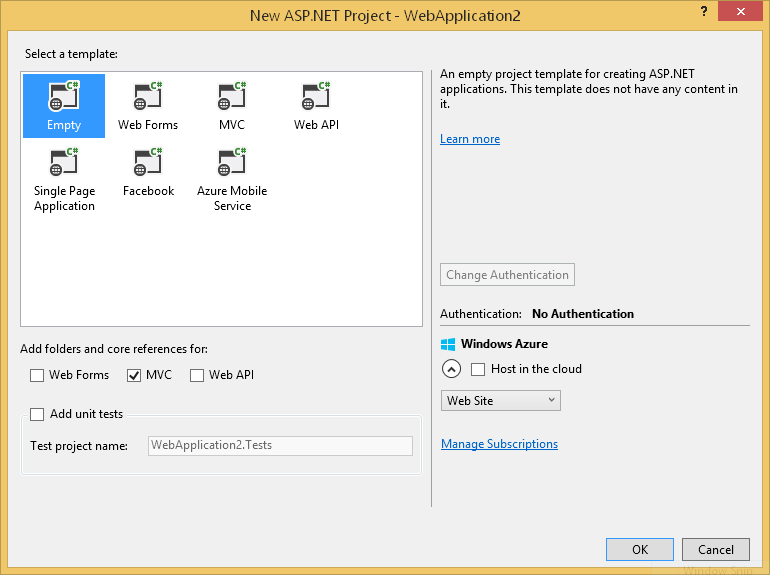
# Models

**Models** are classes you can use to manage data in your MVC application. A great feature of MVC development is the ability to auto-generate code with a model.

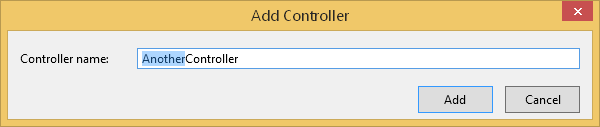
Example : Adding a Detail View with a Layout Page

🞑 This example demonstrates how to use ASP.NET MVC with models to auto-generate web content. To obtain the default master page, create a new project. Begin by creating a new empty MVC application. To do this click **File | New | Project** and then on the Web node select **ASP.NET Web Application**. Then click OK.

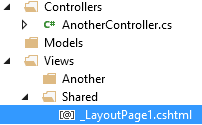
On the New ASP.NET Project dialog click the Empty icon and check MVC. Then click OK.



Right click the Controllers folder and choose Add | Controller | MVC 5 Controller Empty. In the Add Controller dialog enter **AnotherController** and click **Add**.



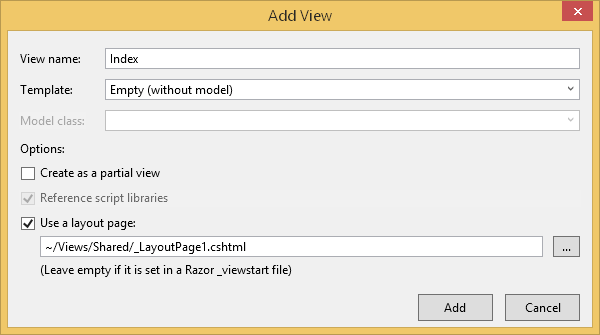
Then, right click the Views folder and add a new folder called **Shared**. Then, right click the Shared folder and click Add | MVC 5 Layout Page (Razor), leave the default name that appears in the Specify Name for Item dialog and click OK. At this point, the folders and files in your project should resemble the ones shown below in this screenshot from the Solution Explorer.



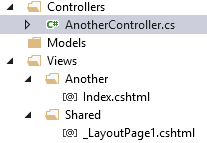
Modify the layout page to include information that you would want to appear as a header on all pages. For this example, add the two header elements (h1 and h2) at shown below.



With the cursor placed on the Index() method header, right click it and choose **Add View**. Before clicking Add from the Add View dialog, browse to and select the existing layout page in the project Shared folder. Then click Add.

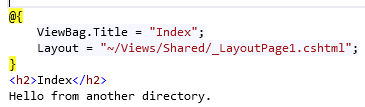


Your project set up should show the index view, the AnotherController.cs file, and the /Views/Shared/\_LayoutPage1.cshtml files as displayed below:

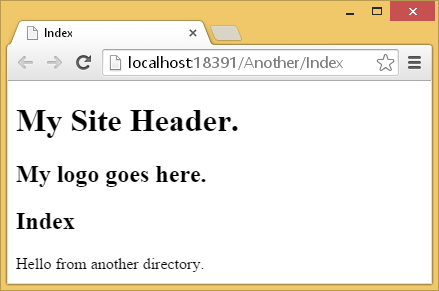


In your View, add the following text:

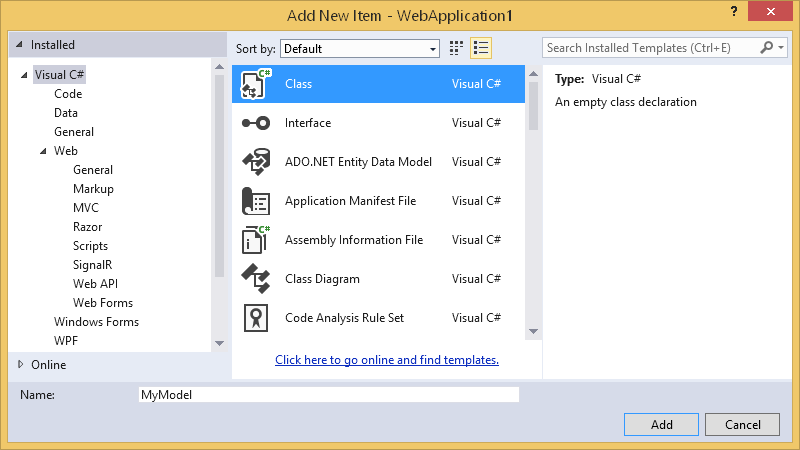
“Hello from another directory” so the view appears as below:



Run your application. The browser may open at the project root. If so append “Another” to the url to see your newly created view.



Next, we will add a model class to define a data set for storage and retrieval. To add the model, right click the Models folder and choose **Add | New Item**. In the Add New Item dialog, choose the Class template and name it MyModel. Then click Add.



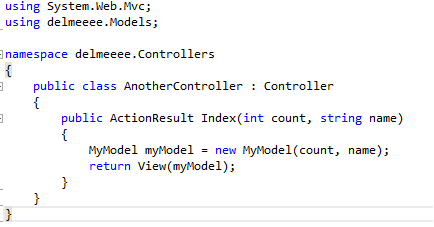
Replace the auto-generated class code with the following:

|  |
| --- |
| public class MyModel{  public int Count { get; set; }  public string Name { get; set; }  public MyModel(int count, string name){  Count = count;  Name = name;  }  } |

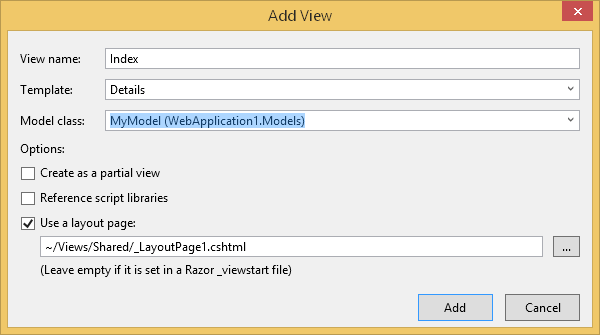
Next, build your project to generate a reference to the model. You can do this by right-clicking the project name and choosing **Build**.

## Adding a Detail View

Now we will modify our controller to use the model we just created. In **AnotherController.cs** file, **reference the namespace of the model.** Adjust the header of the Index() method so it receives (int) **count** and (string) **name** parameters. Inside Index(), create an object of MyModel that is initialized with both of these parameters. Return the MyModel object as a parameter of View() when exiting the method.



Once you finish editing the Index() method, replace the existing view. First delete the existing view. Then right click this Index action method header and choose Add View. This time though, select **Create a strongly-typed view** and then select **Details** in the View content drop-down. Be sure to select MyModel as the data class. **If the model does not appear you will need to build your project to once to make the reference appear in the list.** Also, select the default master page that is included in the project. Then click Add.



Once you click Add, in the Another folder, an Index.cshtml page is generated.



The Model class exposes the Count and Name properties for you in this view. If successful, you should be able to run your project and navigate to a url similar to **http://localhost:<portNum>/Another?name=bob&Count=5**

The parameters that you enter in the url should then get passed through the controller to your view from your model.



## Adding a List View

In the last example, we just saw one object displayed in a **Detail view**. However, you may want to display many objects in a **List view** to display a collection of objects.

Example

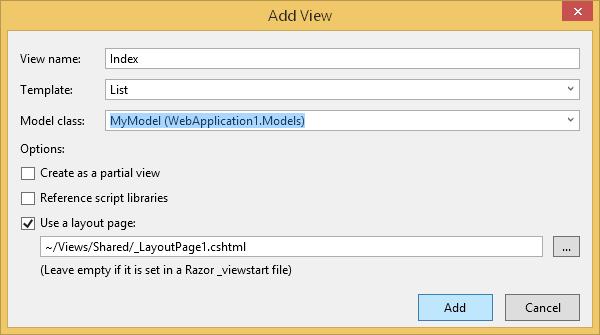
🞑 This next part of the demonstration builds on to generate a list of objects. Create a new class inside the models folder called *MyModelRepository*. To create a list of objects, you can create a method that returns a strongly typed **IEnumerable List** of *MyModel* objects. The code below shows how to do this.

|  |
| --- |
| public IEnumerable<MyModel> GetAllValues() {  const int MAX\_COUNT = 5;  List<MyModel> myModel = new List<MyModel>();  for (int i = 0; i < MAX\_COUNT; i++) {  const string BCIT = " BCIT: ";  myModel.Add(new MyModel(i + 1, BCIT + i.ToString()));  }  return myModel;  } |

In your *AnotherController*, replace the existing *Index()* method with this code to return many objects to the view. You may need to build your project so the new class is recognized by compiler.

|  |
| --- |
| public ActionResult Index() {  MyModelRepository manyObjects = new MyModelRepository();  return View(manyObjects.GetAllValues());  } |

You will have to rebuild the project to recognize the new class. Next, delete the existing view for the AnotherController’s *Index()* method. Then, right click the *Index()* method header and choose Add View. Select the parameters as shown below:



Exercise (1 mark)

🖍 Why is List selected for the current example instead of Detail like in the previous example?

|  |
| --- |
|  |

Exercise (1 mark)

🖍 We want to show results from the *MyModelRepository* class so why is *MyModel* chosen as the type for the **Model class** in the Add View dialog?

|  |
| --- |
|  |

The auto generated code for the view looks similar to:



Exercise (1 mark)

🖍 How do the model references in the **List** view differ from the **Detail** view?

|  |
| --- |
|  |

If you run the project the following output is generated for you.



Obviously the output needs some customization but the bulk of the work is done automatically. You can customize the rest of the presentation manually.

Exercise (1 mark)

🖍 Edit the view above to remove the links for: **edit | details | delete**. Test your revised markup to be sure it works and show your view tags here:

|  |
| --- |
|  |

## Get and Post

This example demonstrates common techniques for retrieving from both a post action and from url parameters.

To start:

* Create an empty MVC project.
* Add the following model . After adding the model, build the project so the *FoodChoice* model appears in the references for the Views. You can build the project by right clicking it in the Solution Explorer and choosing **Build**.

|  |
| --- |
| public class FoodChoice {  public string FruitType { get; set; }  public int Qty { get; set; }  public string Allergy { get; set; }  public FoodChoice(string fruitType, int qty, string allergy){  FruitType = fruitType;  Qty = qty;  Allergy = allergy;  }  } |

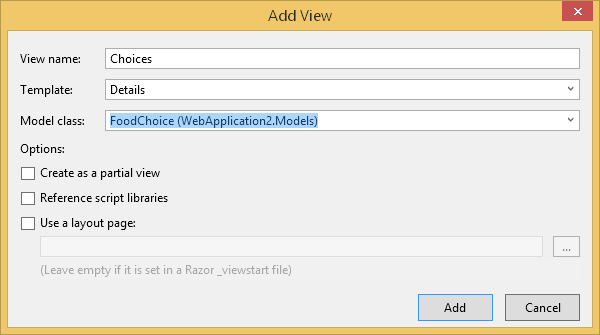
* Add a **Home** Controller.
* Right click the **Index()** action method in the Home controller to create a weakly bound view.
* Place the following HTML code inside the view.

|  |
| --- |
| @{  Layout = null;  }  <!DOCTYPE html>  <html>  <head>  <meta name="viewport" content="width=device-width" />  <title>Index</title>  </head>  <body>  <div>  <!—Controller/Action-->  @using(Html.BeginForm("Choices", "Home", FormMethod.Post)) {  <!--text box-->  <label>Name:</label> <input type="text" name="txtName" /><br /><br />  <!--radio button list-->  <input type="radio" name="radioFruit" value="Apple"/><label>Apple</label><br />  <input type="radio" name="radioFruit" value="Peach"/><label>Peach</label><br />  <input type="radio" name="radioFruit" value="Pear"/><label>Pear</label><br /><br />  <!--drop down list-->  <label>Quantity: </label>  <select name="ddlQuantity">  <option>1</option>  <option>2</option>  <option selected="selected">3</option>  </select><br /><br />  <!--checkbox-->  <input type="checkbox" name="chkAllergy" value="yes" />  <label>Dairy allergy</label><br /><br />  <!--submit button-->  <input type="submit" value="Submit" /><br /><br />  }  <!--hyperlink-->  @Html.ActionLink("This uses GET: ", // text  "Choices", // action  // route values (Action name and parameters)  new {  Action = "Choices",  ddlQuantity = 1,  chkAllergy = "",  radioFruit = "Apple"  })  </div>  </body>  </html> |

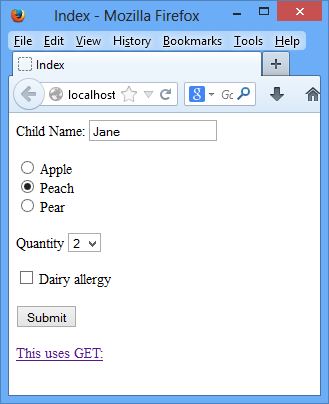
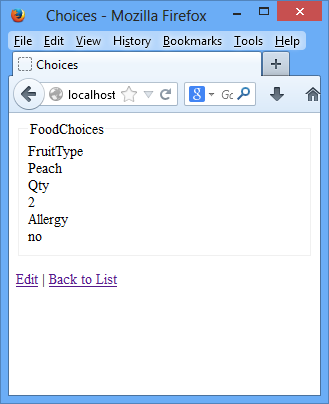
Next, add the following methods to your Home controller. You will need to ensure that you have a reference to the Models namespace.

|  |
| --- |
| string IsAllergic(string allergy){  if (allergy == "")  return "no";  return "yes";  }  [AcceptVerbs(HttpVerbs.Get)]  **// radioFruit (default is "" if none selected). Must go at end of parameter list.**  public ActionResult Choices(int ddlQuantity, string chkAllergy="",  string radioFruit = ""){  FoodChoice foodChoices = new FoodChoice(radioFruit, ddlQuantity,  IsAllergic(chkAllergy));  return View(foodChoices);  }  [AcceptVerbs( HttpVerbs.Post)]  public ActionResult Choices(FormCollection formCollection){  int ddlQuantity = 0; string chkAllergy=""; string radioFruit = "";  foreach (var key in formCollection.Keys){  var value = formCollection[key.ToString()];  switch (key.ToString()){  case "ddlQuantity":  ddlQuantity = Convert.ToInt32(value);  break;  case "chkAllergy":  chkAllergy = IsAllergic(value);  break;  case "radioFruit":  radioFruit = value;  break;  }  }  FoodChoice foodChoices = new FoodChoice(radioFruit, ddlQuantity,  IsAllergic(chkAllergy));  return View(foodChoices);  } |

Next, create a strongly-typed **Details** view using the FoodChoice model. You can do this by right clicking one of the *Choices()* action methods and entering the properties that are shown in the screenshot below:



Run and test your project. You should be able to either post selections you enter or get default parameters from the hyperlink.

Exercise (2 marks)

🖍 Revise your *FoodChoice* model to include a read and write *PersonName* property. Also, modify the *FoodChoice* constructor to initialize the *PersonName* value. Show your revised *FoodChoice* class:

|  |
| --- |
|  |

Exercise (4 marks)

🖍 Revise both *Choices()* action methods in the controller to also receive the value that is submitted from the *txtName* input. The parameter of the GET action method will need to be adjusted and the value will need to also be extracted in the POST action method. Also, in both action methods submit the *txtName* value to the FoodChoice model’s constructor along with the other parameter values. Show your revised code for both *Choices()* action methods here:

|  |
| --- |
|  |

Exercise (2 marks)

🖍 Note that if the name is sent as part of the url you will have to add the txtName parameter to the url. Revise your *Choices()* view to also show the name that was submitted in the text box. Show your revised view code here:

|  |
| --- |
|  |

## Referencing an Entity Data Model

If you add an entity data model to your project, you are really just adding a data model to your project.

Example : Adding and Referencing an Entity Data Model in your Project

🞑 This example shows how easy it is to view and manipulate data through MVC. To begin, create an empty MVC5 web application. Then, add an edmx reference to the following *SuperMarket* database.

|  |
| --- |
| DROP TABLE ProductInvoice;  DROP TABLE Invoice;  DROP TABLE Product;  DROP TABLE Store;  GO  CREATE TABLE Product(  productID INTEGER NOT NULL,  name VARCHAR(25),  vendor VARCHAR(25),  price MONEY,  CHECK (Price>0),  PRIMARY KEY(productID)  )  GO  INSERT INTO Product VALUES(1, 'Cake Mix', 'Ducan Hines', 2.99);  INSERT INTO Product VALUES(2, 'Cookie Dough', 'Ducan Hines', 1.25);  INSERT INTO Product VALUES(3, 'Orange Juice', 'Florida Orange', 4.25);  INSERT INTO Product VALUES(4, 'Cookie Dough', 'Pilsbury', 1.45);  INSERT INTO Product VALUES(5, 'Carrots', 'Hot House', 1.01);  GO  CREATE TABLE Store(  branch VARCHAR(25),  PRIMARY KEY (branch)  )  GO  INSERT INTO Store VALUES('Vancouver');  INSERT INTO Store VALUES('Kamloops');  INSERT INTO Store VALUES('Richmond');  INSERT INTO Store VALUES('Mission');  GO  CREATE TABLE Invoice(  invoiceNum INTEGER,  branch VARCHAR(25),  PRIMARY KEY (invoiceNum),  FOREIGN KEY (branch) REFERENCES Store(branch)  )  GO  INSERT INTO Invoice VALUES(1001, 'Vancouver');  INSERT INTO Invoice VALUES(1002, 'Vancouver');  INSERT INTO Invoice VALUES(1003, 'Kamloops');  INSERT INTO Invoice VALUES(1004, 'Mission');  GO  CREATE TABLE ProductInvoice(  productID INTEGER,  invoiceNum INTEGER,  PRIMARY KEY (productID, invoiceNum),  FOREIGN KEY (productID) REFERENCES Product(productID),  FOREIGN KEY (invoiceNum) REFERENCES Invoice(invoiceNum)  )  GO  INSERT INTO ProductInvoice VALUES(1, 1001);  INSERT INTO ProductInvoice VALUES(2, 1001);  INSERT INTO ProductInvoice VALUES(3, 1001);  INSERT INTO ProductInvoice VALUES(4, 1002);  INSERT INTO ProductInvoice VALUES(2, 1003);  INSERT INTO ProductInvoice VALUES(3, 1003);  INSERT INTO ProductInvoice VALUES(1, 1004);  INSERT INTO ProductInvoice VALUES(2, 1004);  INSERT INTO ProductInvoice VALUES(3, 1004); |

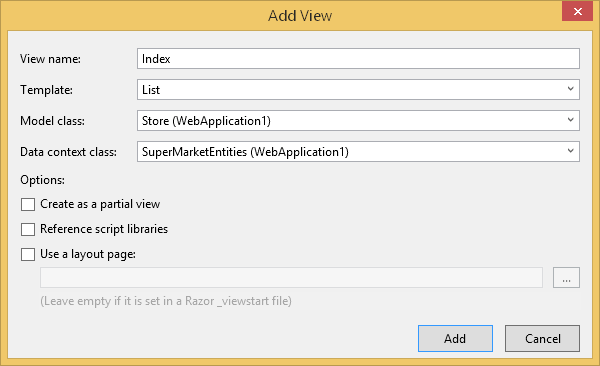
Next, add a model named *StoreRepository* and use this code for your class:

|  |
| --- |
| public class StoreRepository{  public IEnumerable<Store> ShowAllStores(){  SuperMarketEntities context = new SuperMarketEntities();  return context.Stores.ToList();  }    public Store GetStore(string storeName){  SuperMarketEntities context = new SuperMarketEntities();  Store store  = context.Stores.Where(s => s.branch == storeName).FirstOrDefault();  return store;  }  public void AddStore(string storeName){  SuperMarketEntities context = new SuperMarketEntities();  Store store = new Store();  store.branch = storeName;  context.Stores.Add(store);  context.SaveChanges();  }  } |

Once you add your model, be sure to build your project so you can reference it later. Then, create a Home controller and place this code inside it. (You will also need to reference the namespace for this model.)

|  |
| --- |
| [AcceptVerbs(HttpVerbs.Get)]  public ActionResult Index(){  StoreRepository myCustomModel = new StoreRepository();  return View(myCustomModel.ShowAllStores());  }  [AcceptVerbs(HttpVerbs.Post)]  public ActionResult NewStore(string txtStoreName) {  StoreRepository myCustomModel = new StoreRepository();  myCustomModel.AddStore(txtStoreName);  return View(myCustomModel.GetStore(txtStoreName));  } |

After, create a strongly typed List view for the *Index()* method using the Store class.



For some reason, which I have yet to discover, the scaffolding in MVC 5 will not show the primary key value so you have to add it in. To do this add the text that is highlighted in green to your view:

|  |
| --- |
| @foreach (var item in Model) {  <tr>  <td>  @Html.Raw(@item.branch)  @Html.ActionLink("Edit", "Edit", new { id = item.branch }) |  @Html.ActionLink("Details", "Details", new { id=item.branch }) |  @Html.ActionLink("Delete", "Delete", new { id=item.branch })  </td>  </tr>  } |

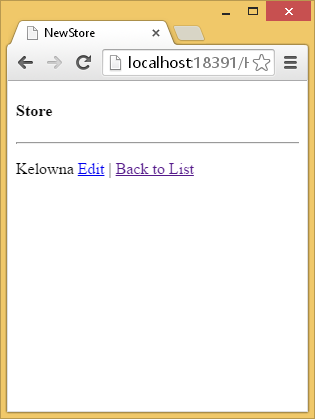
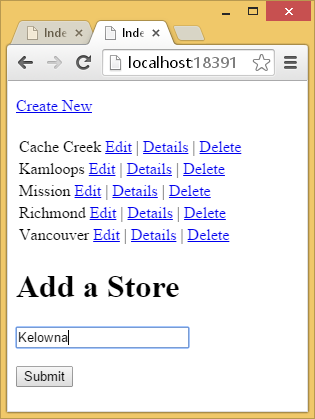
Just before the closing body tag in the new view add the following from generation code to post data to the NewStore action method:

|  |
| --- |
| @using (Html.BeginForm("NewStore", "Home", FormMethod.Post)) {  <h1>Add a Store</h1>  <!--store name textbox-->  <input type="text" name="txtStoreName" /><br /><br />  <!--submit button-->  <input type="submit" value="Submit" />  } |

Then, create a strongly typed detail view for the *NewStore()* method. To get the primary key column to appear you will have to manually add it to your view:

@Html.Raw(Model.branch)

Run your project. The home page shows all stores in the Store table. From there, you can add a new store which gets added on the *NewStore* page.



Exercise (3 marks)

🖍 Create a new empty MVC project that references your *SuperMarket* database as an EDM. Then, create a *ProductRepository* model that allows you list all products, get one product, and add a product. Show your model code here:

|  |
| --- |
|  |

Exercise (4 marks)

🖍 Show your Home controller code needed to use your *ProductRepository* model. *Index()* should enable the display of all products and also input a new product entry from Indexa. The *NewProduct()* method should allow you to display the NewProduct view.

|  |
| --- |
|  |

Exercise (2 marks)

🖍 Show the view code for Index

|  |
| --- |
|  |

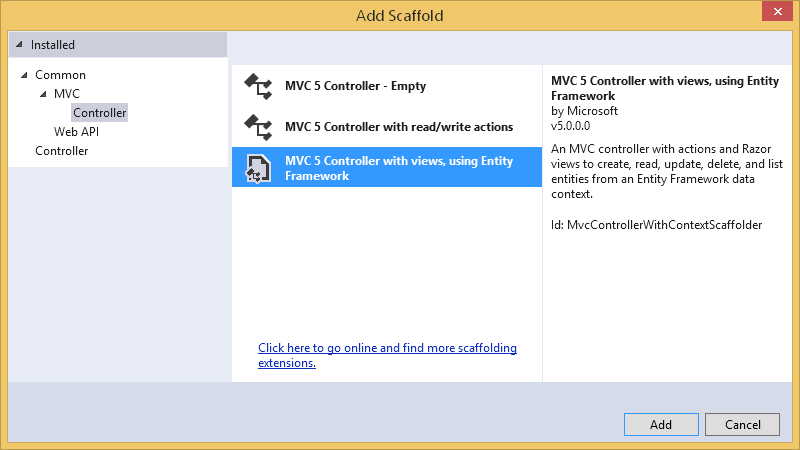
Exercise (2 marks)

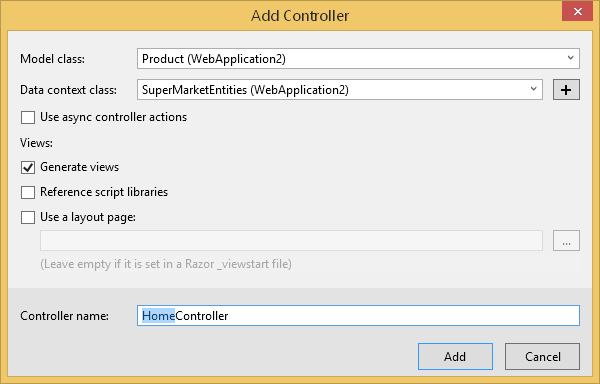
🖍 Show the view code used for NewProduct.

|  |
| --- |
|  |

Example : Fully Automated CRCUD

🞑 In a brand new empty project, add an entity data model for the *SuperMarket* database. Then, create a Home controller with the following options for the *Product* model class:





Build the application so the model is recognized by the rest of the project.

Then, remove the following code from the Edit page.

@section Scripts {

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

}

List, Create, Update, Delete, and Detail pages will be created for you so you will be able to run the project right away.

Exercise

🖍 When you run the project from you will only be able to delete carrots since this is the only product that is not invoiced. Fix your delete controller and view so a user friendly message appears to inform users that deletion of any product with an invoice is not permitted. Show your updated delete controller (3 marks):

|  |
| --- |
|  |

Show your updated view to display a user-friendly message that informs users that products which have corresponding invoices cannot be deleted. Choose any view you like to do this, or create one, but whatever you decide just keep your presentation looking professional: (3 marks)

|  |
| --- |
|  |