**Agenda: Introduction**

* What is MVC Architecture
* What is ASP.NET MVC?
* Understanding Model
* Understanding View
* Understanding Controller
* Advantages of MVC based Web Application
* Features of ASP.NET MVC Framework

**What is MVC Pattern?**

MVC stands for Model - View – Controller. It is architecture for developing interactive applications where there would be a user interaction involved and event handling would occur.

**What is role of Model, View and Controller?**

* **Model**: It manages data basically state of application in memory. There is no fixed size or shape of modelobjects since what we intent to hold in memory in one application may not be the same as that in other application. It includes all of an application’s validation logic, business logic and data access logic. For example, an **Employee object (Model)** might retrieve information from a database, operate on it, validate it and then write updated information back to a Products table in database.
* **View**: It contains logic for rendering Graphical Representation / HTML output. Typically it creates UI withdata from Model. An example would be an edit view of a Products table that displays text boxes, drop-down lists, and check boxes based on the current state of a Products object.
* **Controller**: It contains control flow logic. It is the one which interacts with both models and views tocontrol the flow of application execution. The controller handles and responds to user input and interaction by creating and passing model to view.

**What is Framework?**

A Framework is a collection of classes which provide abstraction of a particular concept. Framework classes are re-usable since a Framework is like a Semi-complete application which has to be extended and made complete for developing a specific application.

Our code fits into the Framework through extension points.

What is ASP.NET MVC Framework?

• It’s an alternative framework to ASP.NET WebForm framework for developing Web based applications using ASP.NET Framework.

• ASP.NET MVC 1.0 was launched in first part of 2009.

• Every Web Application is interactive by nature and hence is a potential candidate is applying MVC architecture.

What is a Model?

1. MVC model is basically a C# or VB.NET class

2. A model is accessible by both controller and view

3. A model can be used to pass data from Controller to view

4. A view can use model to display data in page.

**What is a View?**

1. View is an ASPX/CSHTML page without having a code behind file.
2. All page specific HTML generation and formatting can be done inside view.
3. One can use Inline code (server tags ) to develop dynamic pages.
4. A request to **view** can be made only from a controller’s action method.

**What is a Controller?**

1. **Controller** is basically a C# or VB.NET class which inherits **System.Mvc.Controller**
2. **Controller** is a heart of the entire MVC architecture
3. Inside **Controller’s class** action methods can be implemented which are responsible for responding to browser OR calling views.
4. **Controller** can access and use **model** class to pass data to **views**
5. **Controller** uses **ViewData** to pass any data to **view**

**Advantages of an MVC-Based Web Application:**

* Its light weight because it does not use view state or server-based forms or server controls.
* It makes it easier to manage complexity by dividing an application into the model, the view, and the controller.
* Separating the view from rest of the application logic enables changing of view (technology) in future without affecting the rest of the application. For example you might have views in Silverlight or HTML 5.
* Each developer based on his expertise can work on different parts of the application without stepping on each other toes. For example, one developer can work on the view, a second developer can work on the controller logic, and a third developer can focus on the business logic in the model.
* RESTful / User friendly Url’s and this enables SEO.
* Clean HTML and easy integration with JavaScript and JQuery.
* It doesn’t have renaming of ID’s as in WebForm framework.
* It provides better support for test-driven development (TDD). This is because we can focus on one aspect at a time i.e. we can focus on view without worrying about business logic.
* It’s created to support pattern based software development.
* It works well for Web applications that are supported by large teams of developers and Web designers who need a high degree of control over the application behavior. Easy to maintain since we can clearly pin point what code to open in case there are any changes to be done.

**Advantages of ASP.NET Web Forms:**

* It supports an event model that preserves state over HTTP, which benefits line-of-business Web application development. The Web Forms-based application provides dozens of events that are supported in hundreds of server controls.
* It uses a Page Controller pattern that adds functionality to individual pages.
* It uses view state on server-based forms, which can make managing state information easier.
* It works well for small teams of Web developers and designers who want to take advantage of the large number of components available for rapid application development.
* In general, it is less complex for application development, because the components (the **Page** class, controls, and so on) are tightly integrated and usually require less code than the MVC model.
* ASP.NET Web Forms offers a much greater and more robust toolbox (web controls) whereas MVC offers a more primitive control set relying more on rich client-side controls via jQuery (Javascript).
* It's been around since 2002 and there is an abundance of information with regards to questions, problems, etc. Offers more third-party control - need to consider your existing toolkits

**Features of the ASP.NET MVC Framework:**

* Separation of application tasks (input logic, business logic, and UI logic) and test-driven development (TDD) by default. All core contracts in the MVC framework are interface-based and can be tested by using mock objects, which are simulated objects that have the behavior of actual objects in the application. We can test the application without having to run the code in an ASP.NET process, which makes unit testing fast and flexible.
* An extensible and pluggable framework.
* Extensive support for ASP.NET routing, a powerful URL-mapping component that lets you build applications that have comprehensible and searchable URLs.
* It is integrated with all existing ASP.NET features, such as Master Pages, Session Management,

Security, and Authentication.

**Agenda: First MVC Application**

* Software Requirements
* First MVC Application
* Rendering HTML output
* Life Cycle of ASP.MVC Request.

**Software Requirements:**

To build an ASP.NET MVC 3.0 application, you need either of the following:

* If you have nothing, please visit <http://www.asp.net/mvc>and download and install Visual Web Developer Express 2010.
* If you already have Visual Studio 2010, please visit <http://www.microsoft.com/en-us/download/details.aspx?id=1491>and download the standalone installation for ASP.NET MVC.

To build as ASP.NET 4.0 application you just need to install VS.NET 2012.

**Creating a New ASP.NET MVC Project:**

1. Open Visual Studio and go to **File**  **New**  **Project** **Select ASP.NET Empty MVC 4 Web Application Name the** **project as MyFirstMVCApp** Click OK, Visual Studio will set up a default project structure.
2. Now try to run the application by pressing F5 **(or by selecting debug** **** **Start Debugging)**.
3. Since we used the empty project template, the application doesn’t yet contain any controllers, so it will simply return a 404 Not Found error.
4. Following is the list of folders created
   1. Content
   2. Controllers
   3. Models
   4. Scripts
   5. Views
   6. Views/Shared
5. Global.asax
6. Web.Config
7. Packages.config: This file is managed by the NuGet infrastructure. It's used to track installed

packages with their respective versions. It uses NuGet to track packages such as jQuery, EntityFramework, WebApi, Modernizr etc.

j.

1. **Adding the First Controller:** The default project template includes a folder called Controllers. Right-click theControllers folder and choose **Add** **** **Controller**. Enter the name **HomeController** and then click Add.
2. Replace the whole HomeController class with this
3. public class HomeController : Controller

{

public string Index()

{

return "Hello, ASP.NET MVC";

}

}

**Controller:** In ASP.NET MVC incoming requests are handled by ***controllers***.

* Controllers are just simple classes inherited from **System.Web.Mvc.Controller**
* Each public method on a controller is known as an ***Action Method***, which means you can invoke it from the Web via some URL.
* Now run the application by pressing ctrl + F5 and you can see output as a plain text

**Hello ASP.NET MVC.**

**Note:** ASP.NET MVC applications use the*routing system*. This infrastructure decides how URLs map onto particularcontrollers and actions. Under the default routing configuration, you could request any of the following URLs and it would be handled by the Index action on HomeController:

* /
* /Home
* /Home/Index

So, when a browser requests http://*yoursite*/ or http://*yoursite*/Home, it gets back the output from Home Controller’s Index method. The output right now is string

**Hello ASP.NETMVC.**

**Part 2: Producing some HTML output:**

1. **Creating and Rendering a View:**

To render a **view** from Index() method, first rewrite the method as follows:

public ActionResult Index()

{

return View();

}

1. **To Add view:** Right-click the action method (either on the Index() method name or somewhere inside the methodbody) and then choose Add View. This will lead to the pop-up window.
2. Click Add. For the action method a new view file is created at **~/Views/Home/Index.cshtml.**
3. Replace generated markup content with below html <body>

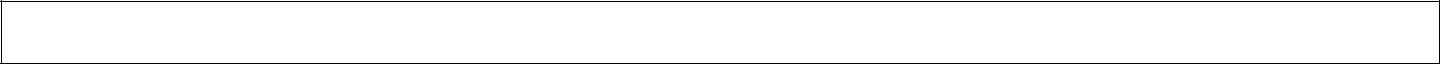
Hello, <b>ASP.NET MVC </b>(from the view)! </body>

1. Press CTRL + F5 to run the application again, and you should see your view template at work.

Now, though, we are returning an object of type **ViewResult**, which instructs the MVC Framework to render a view. If we do not specify a view name, it picks the conventional one for this action method (i.e., ~/Views/Home/Index.aspx)

**Agenda – Exploring Controllers**

* Controllers Overview
* Passing data from Controller to View
* Comparing ViewData, ViewBag and TempData
* Types of Action Methods
* Action Method Parameters
* Action Selectors
* Action Filters Overview
* Building Custom Action Filters



**Controller Overview**

The ASP.NET MVC framework maps URLs to classes that are referred to as controllers.

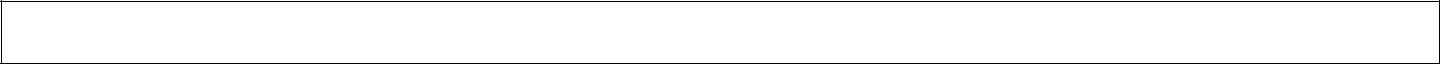
Controllers

* Process incoming requests
* Handle user input and interactions and
* Execute appropriate application logic.

A controller class typically calls a separate view component to generate the HTML markup for the request.

The Controller class inherits from **ControllerBase** and is the default implementation of a controller. The Controller class is responsible for the following processing stages:

1. Locating the appropriate action method to call and validating that it can be called.
2. Getting the values to use as the action method's arguments.
3. Handling all errors that might occur during the execution of the action method.
4. Providing the View for rendering ASP.NET pages to browser.



**Passing data from Controller to View**

**ViewBag vs. ViewData vs. TempData**

All three are properties

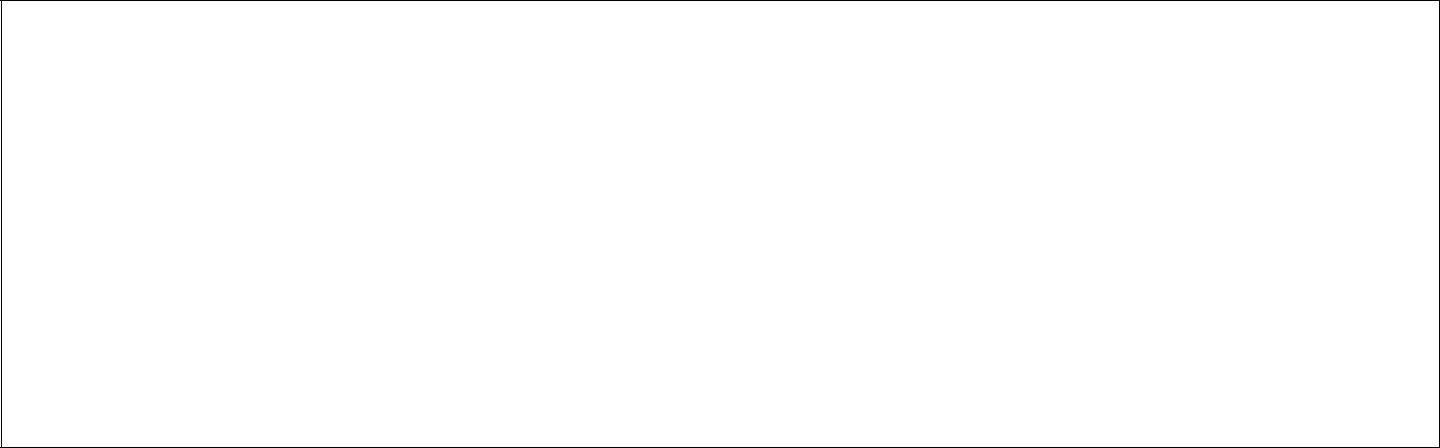
* **ViewData** is a dictionary object that you put data into, which then becomes available to the view. ViewData isa derivative of the ViewDataDictionary class, so you can access by the familiar "key/value" syntax.
* **ViewBag** object is a wrapper around the ViewData object that allows you to create dynamic properties for theViewBag.
* Both the ViewData and ViewBag objects are great for accessing extra data (i.e., outside the data model), between the controller and view.

1

Exploring Controller



* **TempData** has additional advantage of Passing data between the current and next HTTP requests.



public ViewResult Index()

{

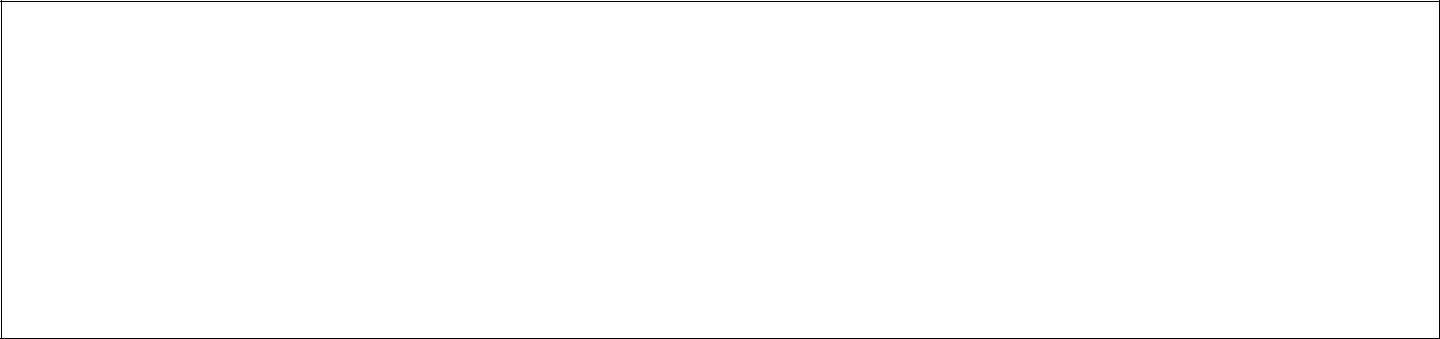
int hour = DateTime.Now.Hour;

**ViewData["Greetings"]** = (hour < 12 ? "Good Morning" : "Good Afternoon");

**ViewBag.Greetings** = (hour < 12 ? "Good Morning" : "Good Afternoon"); **TempData["Greetings"]** = (hour < 12 ? "Good Morning" : "Good Afternoon");return View();

}

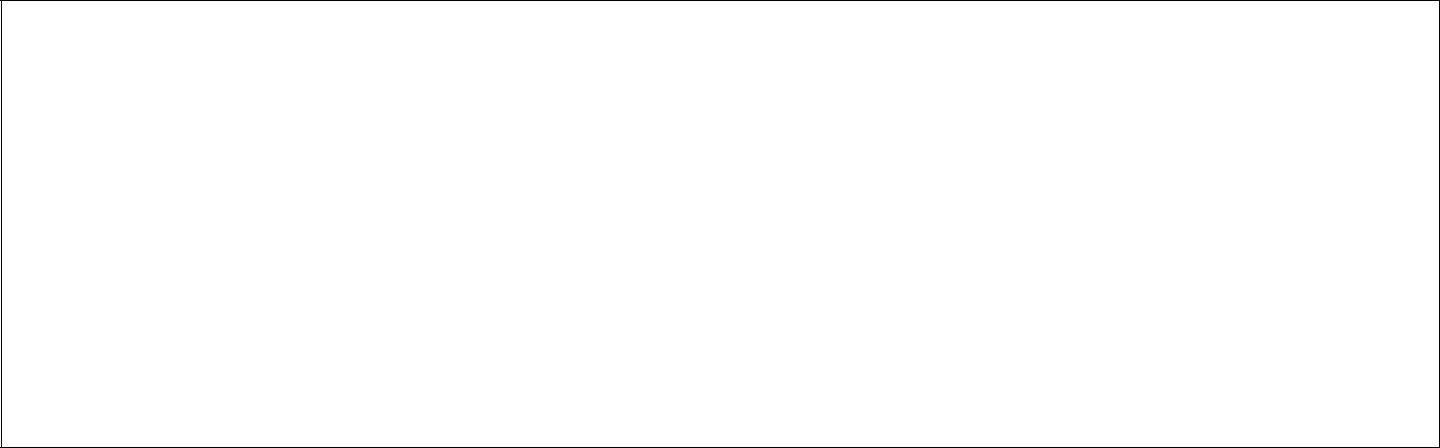
Update your **Index.aspx** view template to display it as follows:



<body>

* ViewData["Greetings"], Yawar (from the view)!
* ViewBag.Greetings, Yawar (from the view)!
* ViewData["Greetings"], Yawar (from the view)!
* **Html.ActionLink("Postback For TempData", "TempDataDemoView")**

</body>



public ViewResult TempDataDemoView()

{

var vd = **ViewData["Greetings"]**;

var vb = **ViewBag.Greetings**;

var td = **TempData["Greetings"]**;

return View();

}

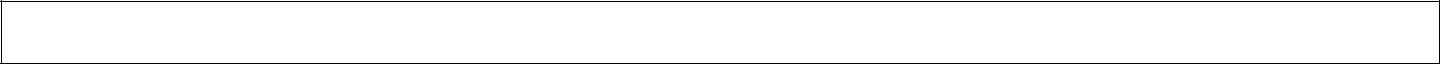
**Put a break point and observe that vd, vb are null but td is not.**

**TempData.Keep(); - To retain key in TempData in subsequent round trip**

Note: TempData uses HttpSession and can be demonstrated by disabling session in web.config <sessionState mode="Off"></sessionState>

2

Exploring Controller



**Action Methods**

ASP.NET MVC application is organized around controllers and action methods. The controller defines action methods. Controllers can include as many action methods as needed.

* Action methods typically have a one-to-one mapping with user interactions. Examples of user interactions include entering a URL into the browser, clicking a link, and submitting a form. Each of these user interactions causes a request to be sent to the server. In each case, the URL of the request includes information that the MVC framework uses to invoke an action method.
* Most action methods return an instance of a class that derives from [**ActionResult**.](http://msdn.microsoft.com/en-us/library/system.web.mvc.actionresult%28v=VS.98%29.aspx) The ActionResult class is the base for all action results. However, there are different action result types, depending on the task that the action method is performing. For example, the most common action is to call the [**View**](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.view%28v=VS.98%29.aspx) method. The View method returns an instance of the [**ViewResult**](http://msdn.microsoft.com/en-us/library/system.web.mvc.viewresult%28v=VS.98%29.aspx) class, which is derived from ActionResult.
* We can create action methods that return an object of any type, such as a string, an integer or a Boolean value. These return types are wrapped in an appropriate ActionResult type before they are rendered to the response stream.

o In Controller

public string SayHello(string name)

{

return "Hello " + name;

}

1. In View

@Html.ActionLink("Say Hello", "SayHello", new {name="Sandeep"})

The following table shows the built-in action result types and the action helper methods that return them:

|  |  |  |
| --- | --- | --- |
| **ActionResult Inherited Classes** | **Action Method** | **Description** |
|  |  |  |
| [ViewResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.viewresult%28v=VS.98%29.aspx) | [View](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.view%28v=VS.98%29.aspx) | Renders a view as a Web page. |
|  |  |  |
| [PartialViewResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.partialviewresult%28v=VS.98%29.aspx) | [PartialView](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.partialview%28v=VS.98%29.aspx) | Renders a partial view, which defines a section of a |
|  |  | view that can be rendered inside another view. |
|  |  |  |
| [RedirectResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.redirectresult%28v=VS.98%29.aspx) | [Redirect](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.redirect%28v=VS.98%29.aspx) | Redirects to another action method by using its URL. |
|  |  |  |
| [RedirectToRouteResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.redirecttorouteresult%28v=VS.98%29.aspx) | [RedirectToAction](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.redirecttoaction%28v=VS.98%29.aspx) | Redirects to another action method. |
|  | [RedirectToRoute](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.redirecttoroute%28v=VS.98%29.aspx) |  |
|  |  |  |
| [ContentResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.contentresult%28v=VS.98%29.aspx) | [Content](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.content%28v=VS.98%29.aspx) | Returns a user-defined content type. |
|  |  |  |
| [FileResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.fileresult%28v=VS.98%29.aspx) | [File](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.file%28v=VS.98%29.aspx) | Returns binary output to write to the response. |
|  |  |  |
| [JsonResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.jsonresult%28v=VS.98%29.aspx) | [Json](http://msdn.microsoft.com/en-us/library/system.web.mvc.controller.json%28v=VS.98%29.aspx) | Returns a serialized JSON object. |
|  |  |  |
| [EmptyResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.emptyresult%28v=VS.98%29.aspx) | Null | Returns Empty Page. |
|  |  |  |

3

Exploring Controller



**Example of ViewResult**

By default, the Controller actions will return the ActionResult object. We can return various types of results as ActionResult, which will decide how the output needs to render on the browser. Example of **ViewResult**

public ActionResult About()

{

return View();

}

**Example of ContentResult.**

public class SampleController : Controller

{

public ActionResult Index()

{

return Content("Hello from Index action in Sample Controller");

}

public ActionResult RenderXML()

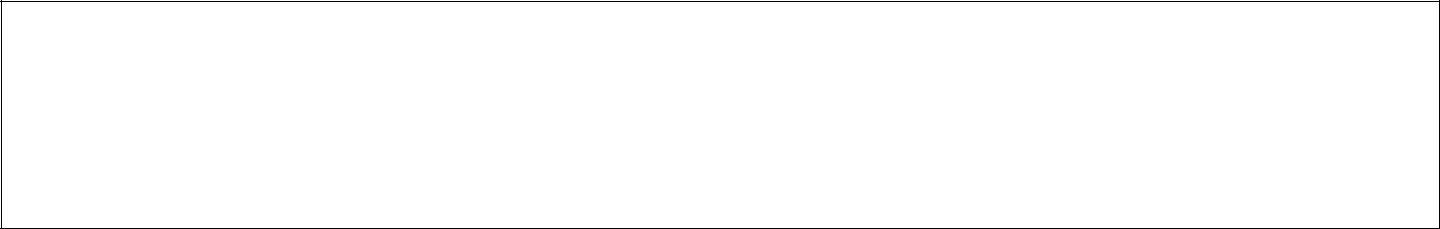
{

return Content("<Demo>This is Test</Demo>","text/xml",System.Text.Encoding.Unicode);

}

}

**Example of Redirect**

****

public ActionResult SayHello(string name)

{

return Redirect("~/Home/RenderXml");

}

**Example of RedirectToAction:** Depending on the input values, we can redirect to another Action.

public ActionResult Index()

{

* Following Redirect‘s to Verify action inside the Sample Controller return RedirectToAction("Verify", "Sample");

}

**Example of RedirectToRoute**

4

Exploring Controller



When we need to redirect to a route defined in ***Global.asax*,** we will use the **RedirectToRoute** object.

In Global.asax:

routes.MapRoute("RenderXml", // Route name "Home/RenderXml");//, // URL with parameters

public ActionResult Index()

{

return RedirectToRoute("RenderXml");

}

**Example of File**

File is used to return the content of a file to the browser.

public ActionResult Index()

{

return File(Server.MapPath("~/Demo.xml"), "text/xml");

}

Note: FileResult loads the file and renders the content to the browser without actually redirecting to the URL of mentioned file.

**Example of JSON**

We can render the text to the result page or can send it as a file to the client using JSON notation.

public ActionResult Index()

{

return Json("{"name": "html", "years": "5"}", "text/html", JsonRequestBehavior.AllowGet);

}

**Non-Action Methods:**

By default, the MVC framework treats all public methods of a controller class as action methods. If your controller class contains a public method and you do not want it to be an action method, you must mark that method with the [NonActionAttribute](http://msdn.microsoft.com/en-us/library/system.web.mvc.nonactionattribute%28v=VS.98%29.aspx) attribute.

**Example:**

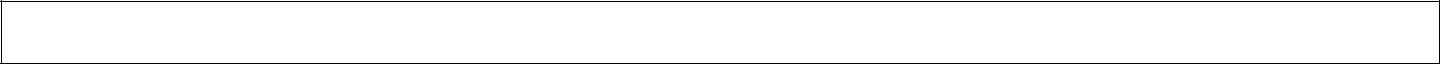
[NonAction]

private void DoSomething()

{ // Method logic.}

5

Exploring Controller



**Action Method Parameters (Overview)**

By default, the values for action method parameters are retrieved from the request's data collection. The data collection includes name/values pairs for form data, query string values, and cookie values.

**Example:**

public void Detail()

{

int id = Convert.ToInt32(Request["id"]);

}

The ASP.NET MVC framework can automatically map URL parameter values to parameter values for action methods. By default, if an action method takes a parameter, the MVC framework examines incoming request data and determines whether the request contains an HTTP request value with the same name. If so, the request value is automatically passed to the action method.

**Example:**

public ResultAction Detail(int id)

{

ViewData["DetailInfo"] = id;

return View();

}

You can also embed parameter values as part of the URL instead of as query-string values. For example, instead of using the URL with a query string such as **/Products/Detail?id=3**, you can use a URL like **/Products/Detail/3**. The default route-mapping rule has the format /{controller}/{action}/{id}. If there is a URL sub-path after the controller and action names in the URL, it is treated as a parameter named id, and is automatically passed to the action method as a parameter value.

The MVC framework also supports optional arguments for action methods. Optional parameters in the MVC framework are handled by using nullable-type arguments for controller action methods. For example, if a method can take a date as part of the query string but you want to the default to be today's date if the query string parameter is missing,

public ActionResult Index(int? id,string p1,string p2)

{

if (id.HasValue)

return Content(id.Value + p1 + "-" + p2);

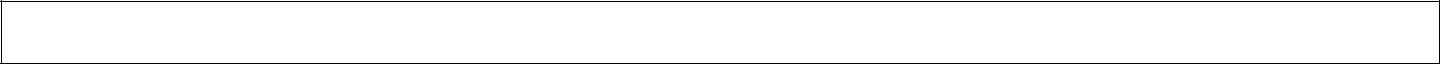
else

return Content("Id is not provided-" + p1 + "-" + p2);

}

6

Exploring Controller



**Action Selectors**

When the MVC Framework is selecting one of the controller public methods to invoke as an action, it will use any action selector attribute that might be present to define the correct action to invoke.

1. ActionName
2. AcceptVerbs
   * HttpPost
   * HttpGet

**ActionName:** When we apply this to a controller action it will specify the action name for that method.

For the Index method we have bellow we no longer reach this method as action name “Index”. We have to reach this method as “**Start**”. (http://localhost:123/Home/Start

[ActionName("Start")]

public ActionResult Index()

{

ViewBag.Message = "Welcome to Yawar!";

return View();

}

Note: A view by name “Start” must be added. Index view will not work.

**AcceptVerbs:** Http verbs allow us to reach a particular action we can say action method is reachable with:

1. [HttpGet] or [AcceptVerbs(HttpVerbs.Get)]
2. [HttpPost] or [AcceptVerbs(HttpVerbs.Post)]
3. **[RequireHttps]** –Forces the Http request to be resent over Https.

**HttpGet:** Only get request will be served by action method.

[HttpGet]

public ActionResult LogOn()

{

return View();

}

**HttpPost:** Only post request will be served by the action method.

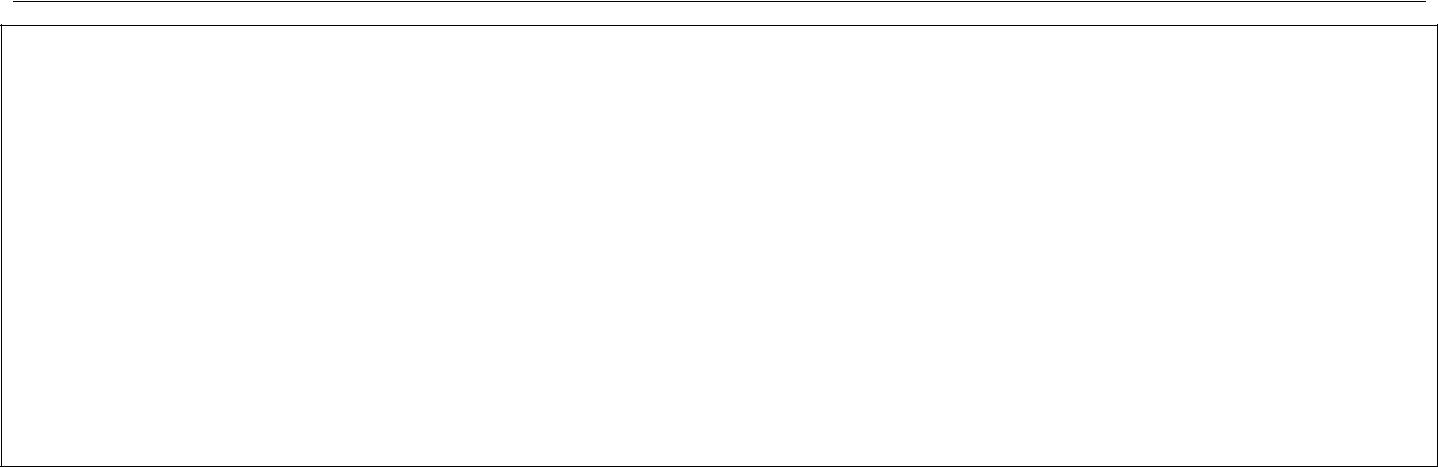
In View:

@{ Html.BeginForm(); }

<input type="submit" value="Submit" />

7

Exploring Controller



@{ Html.EndForm(); }

In Controller:

[**HttpPost**()]

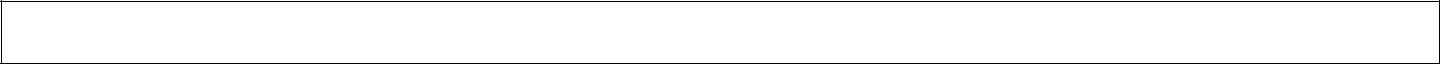
public ActionResult Index(FormCollection col)

{

ViewBag.Greetings1 = "Posted";

return View();

}



**Action Filters**

Action filters will apply pre or post Processing logic to a controller action and it result. Action filters are the component we want to use to apply cross cutting logic to our application i.e. the logic that we must execute across multiple controller actions but we do not want to duplicate the logic inside of individual controller.

|  |  |
| --- | --- |
| **Name** | **Description** |
|  |  |
| OutputCache | Cache the output of an Action method. |
|  |  |
| Authorize | Restrict an action or controller to authorize user or role. |
|  |  |
| ValidateInput | Turn on/off the request validation. |
|  |  |
| ValidateAntiForgeryToken | Helps prevent cross site request forgeries. |
|  |  |
| HandelError | Can specify a view to render in the event of an |
|  | unhandled exception. |
|  |  |

**OutputCache:** tells the runtime that it allows to cache the final output of the action and use the cache result toserve the future request.

[OutputCache(Duration=10, VaryByParam="t1")]

**Authorize:** Authorize attribute allow us it ensure that user is login before action/controller allow to process therequest.

**[Authorize]** // To access the Feedback Action Method User have to login.

public ActionResult Feedback()

{

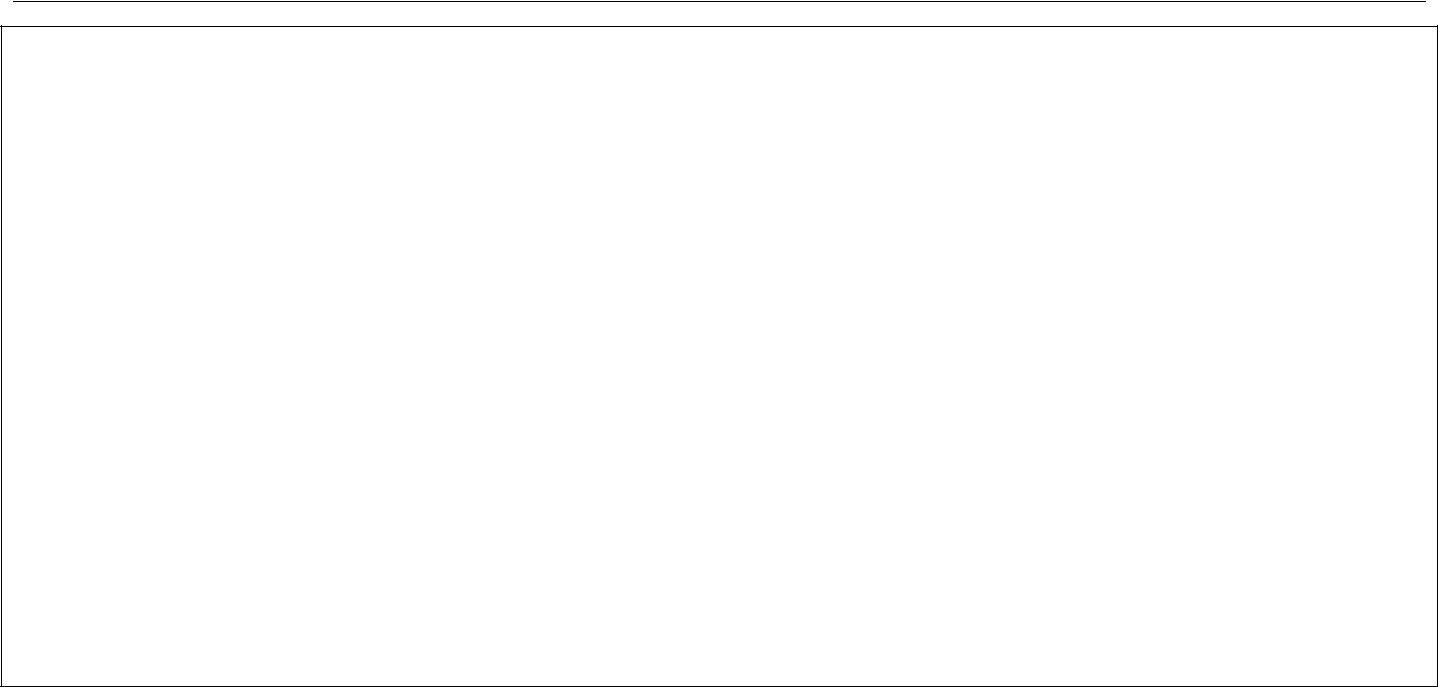
return View();

}

**[Authorize]** // To access the FeedbackController User have to login.

8

Exploring Controller



public class FeedbackController : Controller

{

public ActionResult Feedback()

{

return View();

}

[HttpPost]

public ActionResult Feedback(Feedback f)

{

return View("FeedbackDisplay", f);

}

}

**ValidateInput:** To allow or disallow submission of tags and other potentially dangerous content.



[HttpPost()]

[ValidateInput(false)]

public ActionResult Index(FormCollection col)

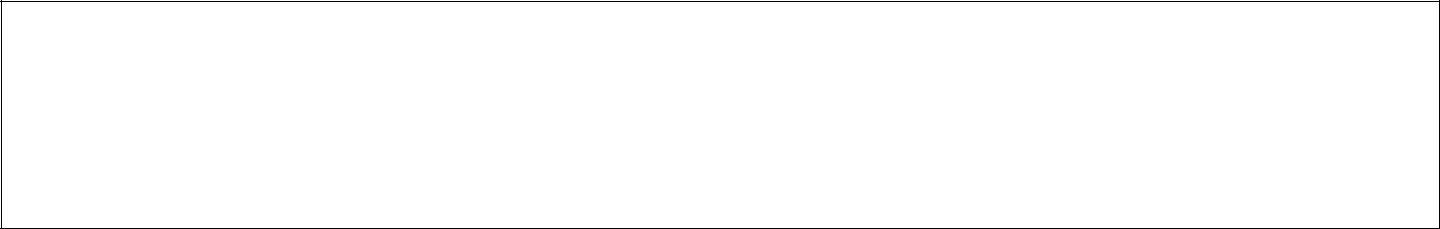
{

ViewBag.Greetings1 = DateTime.Now;

return View();

}

**ValidateAntiForgeryToken**: If this attribute is provided then cross site submission will not be allowed.



[HttpPost(), ValidateAntiForgeryToken()]

public ActionResult Index(FormCollection col)

{return View();

}

**HandleError:** attribute is used to display friendly error pages to the user when something goes wrong.

[HandelError]

public ActionResult Feedback()

{

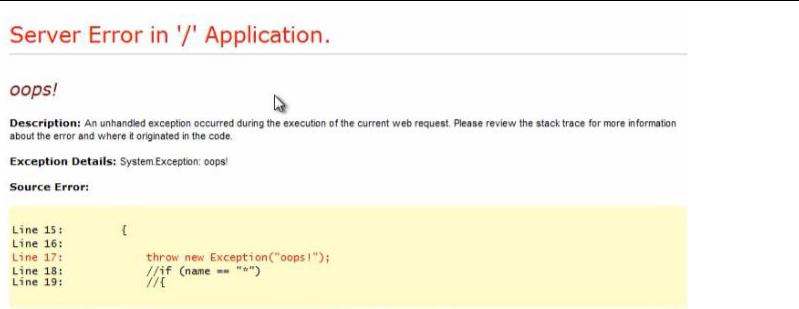
throw new ApplicationException("Ooops!!!");

return View();

}

9

Exploring Controller



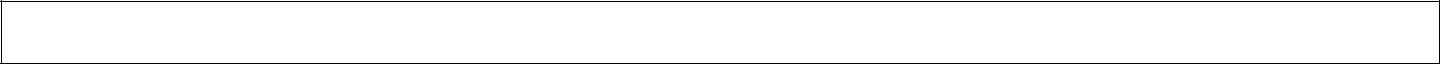
Edit Web.config as below:

<system.web>

<customErrors mode="On "></customErrors>

</system.web>

If exception occurs then it will render the error page which is in the views folder and under the shared folder and is called **Error.cshtml** file.



**Filter Types**

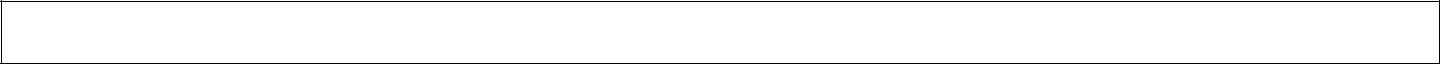
ASP.NET MVC supports the following types of action filters:

1. **Authorization Filter:**
   * Class Implementing IAuthorizationFilter
   * Eg: AuthorizeAttribute and RequireHttpAttribute
   * Can override OnAuthorization method
2. **Action Filter:**
   * Class Implementing IActionFilter
   * Eg: ActionFilterAttribute – Used for writing Custom Attributes
   * Can override OnActionExecuting and OnActionExecuted
3. **Result Filter:**
   * Class Implementing IResultFilter
   * Eg: OutputCacheAttribute.
   * ActionFilterAttribute implements both IResultFilter and IActionFilter.
   * Can override OnResultExecuting and OnResultExecuted
4. **Exception Filter:**
   * Class Implementing IExceptionFilter
   * Eg: HandleErrorAttribute

Note: Controller class Implements all of the above and we can override in it all the methods mentioned.

10

Exploring Controller



**Building Custom Action Filters**

An action filter is implemented as an attribute class that inherits from ActionFilterAttribute.

The base **ActionFilterAttribute** class has the following methods that you can override:

* **OnActionExecuting** –This method is called **before** a controller action method is executed.
* **OnActionExecuted** –This method is called **after** a controller action method is executed.
* **OnResultExecuting** –This method is called before the **ActionResult** instance that is returned by youraction is invoked.
* **OnResultExecuted** –This method is called after the **ActionResult** instance that is returned by your actionis invoked.

The following example shows a simple action filter that logs trace messages:

public class LogAttribute : System.Web.Mvc.ActionFilterAttribute

{

public override void OnActionExecuting(ActionExecutingContext filterContext)

{

Log("OnActionExecuting", filterContext.RouteData);

}

public override void OnActionExecuted(ActionExecutedContext filterContext)

{

Log("OnActionExecuted", filterContext.RouteData);

}

public override void OnResultExecuting(ResultExecutingContext filterContext)

{

Log("OnResultExecuting", filterContext.RouteData);

}

public override void OnResultExecuted(ResultExecutedContext filterContext)

{

Log("OnResultExecuted", filterContext.RouteData);

}

private void Log(string methodName, RouteData routeData)

{

var controllerName = routeData.Values["controller"];

var actionName = routeData.Values["action"];

var message = String.Format("{0} controller:{1} action:{2}", methodName, controllerName, actionName); Debug.WriteLine(message, "Action Filter Log");

11

Exploring Controller



}

}

**Applying Custom Action filter to a controller or Action method:**

**[Log]**

public ActionResult Index()

{

return View();

}

Note: We can also attach the attribute to Controller class

**Global Registering Action filters:**

In **App\_Start\FilterConfig.cs**

public static void **RegisterGlobalFilters**(GlobalFilterCollection filters)

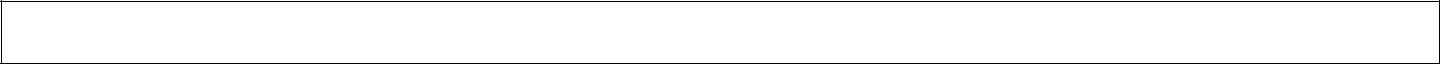
{

filters.Add(**new** LogAttribute());

}

12

Exploring Controller



**Example2: UserAgent ActionFilter**

public class UserAgentResult : ActionFilterAttribute

{

public override void OnActionExecuting(ActionExecutingContext filterContext)

{

if (filterContext.HttpContext.Request.UserAgent.ToLower().Contains("firefox"))

{

ContentResult cr = new ContentResult();

cr.Content = "Firefox cannot be used";

filterContext.Result = cr; // new HttpStatusCodeResult(403);

}

else if (filterContext.HttpContext.Request.UserAgent.ToLower().Contains("ie"))

{

ViewResult vr = new ViewResult();

vr.ViewName = "IEView";

filterContext.Result = vr;

}

}

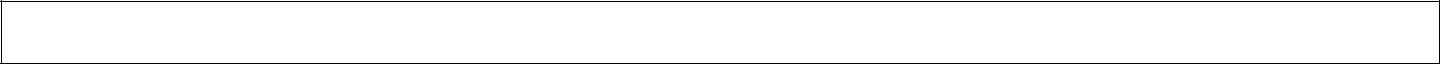
public override void OnActionExecuted(ActionExecutedContext filterContext)

{

//If filterContext.Result != null – This method and action method will not execute

}

}



**Order of Action Filter Execution**

Filters run in the following order:

1. Authorization filters
2. Action filters
3. Result filters
4. Exception filters

**Further the order is controlled by “Order” property:**

1. Unless the **Order** property is set explicitly, an action filter has an implied order of -1.
2. If the **Order** property of multiple action filters are explicitly set, the filter with the lowest value executes before those with greater values, as shown in the following example:

**[Filter1(Order = 2)]**

13

Exploring Controller



**[Filter2(Order = 3)]**

**[Filter3(Order = 1)]**

public void Index()

{

View("Index");

}

In this example, action filters would execute in the following order: Filter3, Filter1, and then Filter2

1. If two action filters have the same **Order** property value, and if one action filter is defined on a type and the other action filter is defined on a method, the action filter defined on the type executes first. The following example shows two an action filter defined on the type and another defined on a method.

**[FilterType(Order = 1)]** public class MyController

{

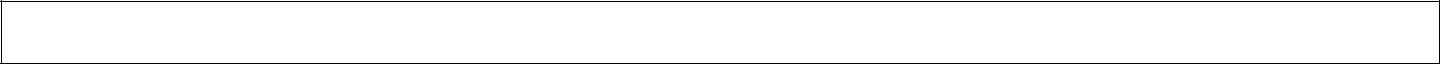
**[FilterMethod(Order = 1)]** public void Index()

{

View("Index");

}

}



**Canceling Filter Execution**

1. You can cancel filter execution in the **OnActionExecuting** methods by setting the **Result** property to a **non-null** value.
   * Any pending OnActionExecuted and OnActionExecuting filters will not be invoked
   * Invoker will not call the OnActionExecuted method for the canceled filter or for pending filters.
   * The OnActionExecuted filter for previously run filters will run.
   * All of the OnResultExecuting and OnResultExecuted filters will run.
2. You can cancel filter execution in the **OnResultExecuting** methods by setting the **Cancel** property true.
   * Any pending OnResultExecuted and OnResultExecuting are cancelled and blank o/p is rendered to browser.
   * The OnResultExecuted filter for previously run filters will run

**Agenda – Exploring Razor Views**

* Introducing Razor View
* Razor Syntax
* Layout view Razor (Master Pages)
* Significance of **\_ViewStart.cshtml**
* Working with Sections
* Working with Partial Views
* Types of Views

**Introduction to Razor View**

Popular ASP.NET MVC view engines used today include **ASPX, Spark, Nhaml and Razor.**

1. Razor View is a file with .cshtml/.vbhtml extension.
2. These files are not included in the build of MVC application (DLL)
3. Files that cannot be shown by direct requests (master pages, partial views etc) have underscore (\_) in the beginning of their names.
4. The Razor View engine allows us to use razor templates to produce HTML.

These are **templates** consisting of HTML and code expressions that place data in to the markup.

**Advantages of Razor View**

1. **Razor minimizes the number of characters and keystrokes** required when writing a view template, andenable a fast coding workflow. Unlike most template syntaxes, you do not need to interrupt your coding to explicitly denote the start and end of server blocks within your HTML. The Razor parser is smart enough to infer this from your code.

<p>Single line If – braces are compulsory</p>

@if (result != "") {

<p>Result: @result</p>

}

1. **Is not a new language:** Existing C# / VB (or other) language skills can be used with Razor, and deliver atemplate markup syntax that enables an awesome HTML construction workflow with the language of choice.
2. **Has great Intellisense**: While Razor has been designed to not require a specific tool or code editor, it hasawesome statement completion support within Visual Studio.
3. **Works with any Text Editor**: Razor doesn’t require a specific tool and enables you to be productive in anyplain old text editor (notepad works great).

**Razor Syntax**

**Code Expressions:**

Razor allows you to output a variable value, or a sub-property on a variable that is referenced via “dot” notation.

You can also use “dot” notation to access sub-properties of multiple levels:

UserName: **@Model.user.UserName**

Razor is smart enough to implicitly identify a lot of code nugget scenarios. But there are times when you want/need to be more explicit in how you scope the code nugget expression. The @(expression) syntax allows you to do that:

@{

int var = 10;

}

@var/2

**Output: 10/2**

@{

int var = 10;

}

@(var/2)

**Output**: **5**

**Using “@” in string:**

Email: [user@Model.user.UserName](mailto:user@Model.user.UserName)

**Output: userUser1**

user@@gmail.com **or** [user@gmail.com](mailto:user@gmail.com)

**Output:** [user@gmail.com](mailto:user@gmail.com)

**Appending/Inserting Dynamic values in Static text:**

You can use the explicit expression syntax to append static text at the end of a code nugget without having to

worry about it being incorrectly parsed as code:

<img src="/Images/**@(Model.EmployeeName)**.jpg

Above we have embedded a code nugget within an <img> element’s src attribute. It allows us to link to images with URLs like “/Images/E1.jpg”. Without the explicit parenthesis, Razor would have looked for a “.jpg” property on the EmployeeName (and raised an error). By being explicit we can clearly denote where the code ends and the text begins.

**Code Blocks/Multi-line Statements:**

‘@{ … }’ used for code blocks with multiple lines:

@{

ViewBag.Title = "EmployeeDetails";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

@{

var name = “John”;

var nameMessage = "Hello, my name is " + name;

}

@foreach (var item in Model) {

**Hello** // compiler error

<tr>

<td> … </td>

</tr>

}

**Hello** gives compiler error because here compiler thinks that **Hello** is C# code.

Whereas other content which in inside tags will not give error.

Use **<text>** tag to fix and <text> tag will not be rendered to the browser.

@for (int i = 0; i < 10; i++)

{

<text>This is Line @i <br /></text>

}

@{

var showToday = false;

if(showToday) {

@DateTime.Today;

}

else {

<text>Sorry!</text>

}

}

**foreach statement:**

@{

<h3>Team Members</h3>

string[] teamMembers = {"Matt", "Joanne"};

**foreach** (var person in teamMembers)

<p>@person</p>

}

**while statement:**

@{

var countNum = 0;

**while** (countNum < 50) {

countNum += 1;

<p>Line #@countNum: </p>

}

}

**Comments:**

Comments in Razor are delimited by **@\*\*@.** If you are inside a C# code block, you can also use // and /\* \*/ comment delimiters.

@\*

A Razor Comment

\*@

@{

//A C# comment

/\* A Multi

line C# comment

\*/

}

**Layout with Razor (Master Pages)**

We typically want to maintain a consistent look and feel across all of the pages within your web-site/application. ASP.NET introduced the concept of “master pages” which helps enable this when using .aspx based pages or templates. Razor also supports this concept with a feature called “layouts” – which allow you to define a common site template, and then inherit its look and feel across all the views/pages on your site.

Every application has common UI structure like navigation header and footer etc. to provide a structure we can use Layout view in ASP.NET MVC.

* Layout views are “Master Pages” for razor.
* Uses inherited methods to specify content areas.
  + **RenderBody**
  + **RenderSection**

**Layout View (Shared Folder):**

Go to \Views\**Shared**\\_Layout.cshtml” file.

As \_Layout.cshtml is file in Shared folder, this is shared by all the views.

<!DOCTYPE html>

<html>

<head>

<title>@**ViewBag.Title**</title>

<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>

<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript"></script>

</head>

<body>

@**RenderBody**()

</body>

</html>

* We are calling the **@RenderBody()** method within the layout file above to indicate where we want the views based on this layout to “fill in” their core content at that location in the HTML.

5

Exploring Razor Views



* We are outputting the “ViewBag.Title” property within the <title> element of our <head> section
* We did not need to wrap our main body content within a tag or element – by default Razor will automatically treat the content of .cshtml as the “body” section of the layout page. We *can* optionally define “named sections” if our layout has multiple replaceable regions.
* We are programmatically setting the **ViewBag.Title** in our .cshtml file. The code within our .cshtml file will run ***before*** the \_Layout.cshtml code runs – and so we can write view code that programmatically sets values we want to pass to our layout to render. This is particularly useful for things like setting the page’s title, as well as <meta> elements within the <head> for search engine optimization (SEO).
* At the moment, we are programmatically setting the Layout template to use within our EmployeeDetails.cshtml page. We can do this by setting the Layout property on the View

**How can we have an independent view without \_Layout.cshtml as Master**

In view do the following

@{

Layout = null;

}

**Significance of \_ViewStart.cshtml**

Currently we are programmatically setting the layout file to use at the top of our **.cshtml** file. This is fine for cases where we have some view-specific logic where the layout file will vary depending on the specific view. But setting it this way can end up being redundant and duplicative for most web applications – where either all of the views use the same layout, the logic for which layout to pick is common across all of the views.

Open **\Views\\_ViewStart.cshtml** underneath the folder of your project:

It can be used to define **common view code** that you want to execute at the start of each View’s rendering. For example, we could write code to programmatically set the Layout property for each View to be the \_Layout.cshtml file by default:

@{

**Layout = "~/Views/Shared/\_Layout.cshtml";**

}



**Working with Sections**

Razor also supports the ability to add additional "named sections” to layout templates as well. These sections can be defined anywhere in the layout file (including within the <head> section of the HTML), and allow you to render dynamic output content to multiple, non-contiguous, regions of the final response.

**Defining the “Footer” section in our Layout**

6

Exploring Razor Views



Let’s update our Layout template to define an additional “footer” section of content that will be rendered within the <div id=”footer”> region of our HTML. We can do this by calling the **RenderSection(string** **sectionName, bool required)** helper method within our Layout.cshtml file like below:

<body>

@RenderBody() – Here default content of content page will be placed.

<div id="footer">

@RenderSection("footer", required:false) – This is Place holder for section in content page. </div>

</body>

</html>

The first parameter to the “RenderSection()” helper method specifies the name of the section we want to render at that location in the layout template.

The second parameter is optional, and allows us to define whether the section we are rendering is required or not. If a section is “required”, then Razor will throw an error at runtime if that section is not implemented within a view template that is based on the layout file (which can make it easier to track down content errors). If a section is not required, then its presence within a view template is optional, and the above RenderSection() code will render nothing at runtime if it isn’t defined.

**Implementing the “footer” Section in our View Template:**

@{

ViewBag.Title = "EmployeeDetails";

…

}

@section footer

{

<center><b>This is footer of page </b></center>

}

We could have put our footer, @section declaration anywhere within the view template. It looks cleaner when defined at bottom of the file – but that is simply personal preference.

**Conditionally detecting if a Layout Section has been implemented**

Razor provides the ability for you to conditionally check (from within a layout file) whether a section has been defined within a view template, and enables you to render an alternate response in the event that the section has not been defined. This provides a convenient way to specify default UI for optional layout sections.

<body>

@RenderBody()

<div id="footer">

**@if (IsSectionDefined("footer"))**

7

Exploring Razor Views

**{**

**@RenderSection("footer", required: false)**

**}**

**else**

**{**

**<b>This is default footer of page </b></center>**

**}**

</div>

</body>

</html>

**Partial Views:**

A partial view allows us to put HTML and C# code in to a file so that we can reuse it in several different places.

Create a partial view named RenderTable.cshtml

<table>

@for (int i = 1; i <= 10; i++)

{

<tr>

<td>@ViewBag.Number</td>

<td>\*</td>

<td>@i</td>

<td>=</td>

<td>@(ViewBag.Number\*i)</td>

</tr>

}

</table>

In Index.cshtml add the following

@{

ViewBag.Number = 10;

}

@Html.Partial("RenderTable")

**OR**

@{

Html.RenderPartial("RenderTable"); //Can be used in code block

}



**Types of View**

1. **Loosely Typed Views**, View is not tightly coupled to Model and same view can use any Model
2. **Strongly Typed Views**: View is tightly coupled to Model.

**Example:**

1. Add Employee class in Models Folder of the application.

8

Exploring Razor Views



**Agenda – Digging into HTMLHelper Methods**

* HTMLHelper Methods
* Render HTML Form
* Using DropDownList
* Binding Html Helper to Model
* Binding Html Helper to ViewData dictionary
* Using “For” Methods with Typed Model
* Overriding Display Templates
* Overriding Editor Templates
* Writing Custom Helper Extension methods
* Reusing using custom @helper Methods

**Understanding Html Helpers**

An HTML Helper is just a method that returns a string. The string can represent any type of content that you want. HTML Helpers can be used to render standard HTML tags like HTML <input> and <img> tags or to render more complex content such as a <table> of database data.

The ASP.NET MVC framework includes the following set of standard HTML Helpers:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Html.ActionLink() |  | Html.Password() |
|  | Html.BeginForm() |  | Html.RadioButton() |
|  | Html.CheckBox() |  | Html.TextArea() |
|  | Html.DropDownList() |  | Html.TextBox() |
|  | Html.ListBox() |  | Html.Hidden() |

All the above methods also exist in “For” methods. For example we have two methods **Html.CheckBox** and **Html.CheckBoxFor**

Difference between both the methods is, later is used only if view is tightly coupled with model and lambda expression can be used in these methods.

Eg: DropDownListFor used when view is tightly coupled to model otherwise we should use DropDownList method.

**Rendering Links**

@Html.ActionLink("Home", "Index", "Home", new { Id1 = 1 }, new { target = "\_blank" });

Note: The text of link and route values are automatically encoded

To render Image as link:

<a href="@**Url.Action**("Index1")"><img src="delete.gif" alt="delete" /></a>

public class Employee

{

public int Id;

public string Name;

}

1. In Controller, modify the Index method as below, see that Employee Model object is created and passed to View method as argument.

public ActionResult Index()

{

Employee modEmp = new Employee() { Id = 1, Name = "E1" }; return View(modEmp);

}

1. Create a View for the Index and add the following to the View

Hello @Model.Name, Your Id is @Model.Id

Note that as this is now a Loosely Typed View, Intellisense will not work on property Model. Also Model Property is of type dynamic. (**dynamic** WebPageView<**dynamic**>.Model)

1. Now add the following line on top of the View file (Index.chtml) @model MvcApplication1.Models.Employee

**Note that this time the view is tightly coupled to the model and hence the intellisense will start showing the properties of Model object.**

Also Model is strongly typed – Employee WebPageView<Employee>.Model

**Rendering Form and its Elements**

**Option1:**

@{ Html.BeginForm("ActionName", "ControllerName"); }

Name: @Html.TextBox("txtName")

<input type="submit" value="Submit" />

@{ Html.EndForm(); }

**Option2:**

@{

Html.BeginForm("ActionName", "ControllerName");

@:Name:

@Html.TextBox("txtName")

<input type="submit" value="Submit" />

Html.EndForm();

}

**Option3:**

@using (Html.BeginForm("ActionName", "ControllerName"))

{

@:Name:

@Html.TextBox("txtName")

<input type="submit" value="Submit" />

}

Note: Html.BeginForm return MvcForm and this implements IDisposable and Dispose method of this is calling Html.EndForm() which renders “</form>”

@{

Html.BeginForm();

@Html.Label("name","Name:")

@Html.TextBox("name")

<br />

@Html.CheckBox("IsActive")

@Html.Label("IsActive", "Active")

<br />

@Html.DropDownList("DDL1", new List<SelectListItem> {

new SelectListItem() {Text="Item1",Value="1"},

new SelectListItem() {Text="Item2",Value="2",Selected=true}, new SelectListItem() {Text="Item3",Value="3"},

Digging into HTMLHelper Methods



new SelectListItem() {Text="Item4",Value="4"}

}, new { size = "4" });

@Html.ListBox("LST1", new List<SelectListItem> {

new SelectListItem() {Text="Item1",Value="1"},

new SelectListItem() {Text="Item2",Value="2",Selected=true}, new SelectListItem() {Text="Item3",Value="3"}, new SelectListItem() {Text="Item4",Value="4"}

});

<br />

@Html.RadioButton("r1","v1")@:V1

@Html.RadioButton("r1","v2")@:V2

@Html.RadioButton("r1","v3")@:V3

<input type="submit" name="Submit" value="Submit" />

Html.EndForm();

}

**Binding Html Helpers to Model**

1. **Add the following to the Model folder** public class Employee

{

public int Id { get; set; } public string Name{ get; set; } public decimal Salary { get; set; } public bool IsActive { get; set; } public string EmailAddress { get; set; } public DateTime DateOfJoin { get; set; }

}

1. **Add the following to the controller** public ActionResult Index()

{

Employee emp = new Employee() { Id = 1, Name = "E1", Salary = 10000, IsActive = true }; emp.DateOfJoin = DateTime.Now;

emp.EmailAddress = "test@test.com"; return View(emp);

}

[HttpPost()]

public ActionResult Index(Employee e)

{

**ModelState.Clear();** //So that the updated Model is used by Html Helper Methods.

e.Name = "New Name";

return View(e);

}

1. **Add the following to the View**

@using (Html.BeginForm("Index","Home"))

{

@Html.Label("Id","ID: ") @Html.TextBox("Id") <br /> @Html.Label("Name","Name: ") @Html.TextBox("Name")

<br />

@Html.Label("Salary","Salary: ") @Html.TextBox("Salary")

<br />

@Html.CheckBox("IsActive") @Html.Label("IsActive","Is Active:") <br /> @Html.Label("Email","Email: ") @Html.Display("EmailAddress") <br /> @Html.Label("DateOfJoin","Date of Join: ") @Html.TextBox("DateOfJoin")

<br />

<input type="submit" name="btnSumit" value="Submit" />

}

**Note: Html.TextBox first looks in the posted request values and then in the model that you update in your controller. In the posted request values it finds is the old value**.

**Binding DropDownList to Model Property**

1. Add the following to Model Folder public class Department

{

public int DeptId { get; set; } public string DeptName { get; set; }

}

1. Add the property DeptId to Employee class public class Employee

{

//…

public int DeptId { get; set; }

}

1. Initialize TempDate in Index method as shown below public ActionResult Index()

{

List<Department> depts = new List<Department>(); depts.Add(new Department() { DeptName = "D1", DeptId = 1 }); depts.Add(new Department() { DeptName = "D2", DeptId = 2 }); SelectList sl = new SelectList(depts, "DeptId", "DeptName"); TempData["DepartmentList"] = sl;

TempData.Keep();

Employee emp = . . .

}

[HttpPost()]

public ActionResult Index(Employee e)

{

//. . .

TempData.Keep();

e.Name = "New Name";

return View(e);

}

1. Add the following to the View @Html.Label("DeptId", "Department: " ); @Html.DropDownList("DeptId", TempData["DepartmentList"] as SelectList)

**Example Demonstrating “For” Methods with Typed Model**

1. Add the following on top of View and make it as Typed View @model MvcApplication1.Models.Employee
2. Add the following to the View @Html.LabelFor(e=>e.Id, "ID: ") @Html.TextBoxFor(e => e.Id) <br />

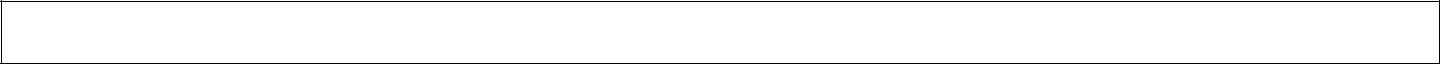
@Html.LabelFor(e=>e.DeptId, "Department: " ); @Html.DropDownListFor(e=>e.DeptId, TempData["DeptId"] as SelectList) <br />

@Html.LabelFor(e=>e.Name, "Name: ")

@Html.TextBoxFor(e => e.Name)

. . .

Note: Lambda Expression is to mention the name of the HTML field and same is two way binded to property of Model



**Binding Html Helpers to ViewData dictionary**

**In Controller:**

public ActionResult Index()

{

ViewData["Id"] = 1;

ViewData["Name"] = "E1";

ViewData["Salary"] = 10000;

ViewData["IsActive"] = true;

ViewData["DeptId"] = 2;

ViewData["DateOfJoin"] = DateTime.Now;

ViewData["EmailAddress"] = "test@test.com";

List<Department> depts = new List<Department>(); depts.Add(new Department() { DeptName = "D1", DeptId = 1 }); depts.Add(new Department() { DeptName = "D2", DeptId = 2 }); SelectList sl = new SelectList(depts, "DeptId", "DeptName", 2); ViewData["DeptId"] = sl;

return View();

}

**In View:**

@using (Html.BeginForm("Index", "Home"))

{

@Html.Label("Id", "ID: ")

@Html.TextBox("Id")

<br />

@Html.Label("DeptId", "Department: " );

@Html.DropDownList("DeptId")

<br />

@Html.Label("Name", "Name: ")

@Html.TextBox("Name")

<br />

@Html.Label("Salary", "Salary: ")

@Html.TextBox("Salary")

<br />

@Html.CheckBox("IsActive")

@Html.Label("IsActive", "Is Active:")

<br />

@Html.Label("Email", "Email: ")

@Html.Display("EmailAddress")

<br />

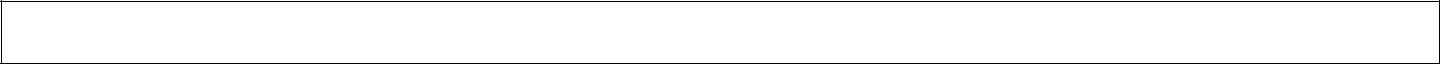
@Html.Label("DateOfJoin", "Date of Join: ")

@Html.TextBox("DateOfJoin")

<br />

<input type="submit" name="btnSumit" value="Submit" />

}



**Display and Editor Method**

These methods generates different HTML markup depending on the **data type of the property** that is being rendered, and according to whether the property is marked with certain attributes.

These method’s renders markup according to the following rules:

1. If the property is typed as a primitive type (integer, string, and so on), the method renders a string that represents the property value.
2. If the property type is Boolean, the method renders an HTML input element for a check box. For example, a Boolean property named Enabled might render markup such as the following:

<input class="check-box" disabled="disabled" type="checkbox" checked="checked" />

1. If a property is annotated with a data type attribute, the attribute specifies the markup that is generated for the property. For example, if the property is marked with the [DataType(DataType.EmailAddress)] attribute, the method generates markup that contains an HTML anchor that is configured with the mailto protocol, as in the following example:

<a href='mailto:test@test.com'>test@test.com</a>

1. If the object contains multiple properties, for each property the method generates a string that consists of markup for the property name and markup for the property value.

**For Example** @Html.DisplayFor(e=>e) will reder: <div class="display-field">1</div> <div class="display-label">DeptId</div> <div class="display-field">2</div> <div class="display-label">Name</div> <div class="display-field">E1</div> <div class="display-label">Salary</div>

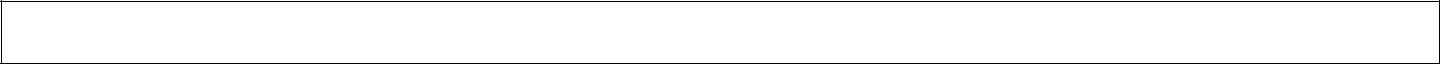
<div class="display-field">10000.00</div> <div class="display-label">IsActive</div>

<div class="display-field"><input checked="checked" class="check-box" disabled="disabled" type="checkbox" /></div>

<div class="display-label">EmailAddress</div>

<div class="display-field"><a href='mailto:test@test.com'>test@test.com</a></div> <div class="display-label">DateOfJoin</div>

<div class="display-field">11/20/2012 6:52:36 PM</div>



**Custom Templates for Display and Editor Methods**

If a template whose name matches the templateName parameter is found in the controller's **DisplayTemplates** folder, that template is used to render the expression. If a template is not found in the controller specific DisplayTemplates folder, the **Views\Shared\DisplayTemplates** is searched for a template that matches the name of the templateName parameter. If no template is found, the default template is used.

[**Overriding Display**](http://buildstarted.com/2010/09/10/overriding-displayfor-and-editorfor-to-create-custom-outputs-for-mvc/) **Template**

1. Create a Model as below:

using System.ComponentModel.DataAnnotations; public class Employee

{

//. . .

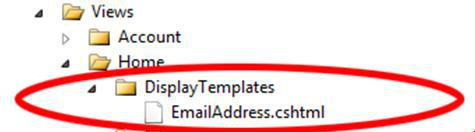
//[DataType(DataType.EmailAddress)]

[UIHint("EmailAddress")]

public string EmailAddress { get; set; }

}

Notes: Display templates are .cshtml partial views that have the same name as the **type** they’re going to override the default templates provided by MVC. To create a Display Template you just create a Folder named “DisplayTemplates” within one of your controller views (or the shared folder)



1. Create an **EmailAddress.cshtml** template that will format an email address with a mailto: link @inherits System.Web.Mvc.WebViewPage<string>

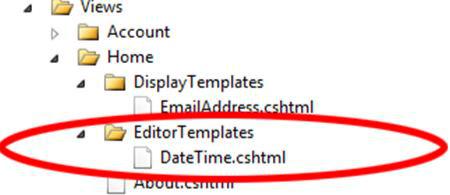
<a href='mailto:@Model'>@Html.DisplayTextFor(m => m)</a>

Note: Both methods, **@Model** and an **Html helper** are used to display the actual value of the model.

1. In View use as below: @Html.Display("EmailAddress")

**Overriding Editor Templates:**

To create an Editor Template you just create a folder named “EditorTemplates” within one of your controller views (or the shared folder) like so:



1. In DateTime.cshtml

@inherits System.Web.Mvc.WebViewPage<System.DateTime> @Html.TextBox("", (Model.ToShortDateString()), new { @class = "datePicker" })

Note: the @class is the html attribute we’re using to assign the datePicker to textbox. Below is the jquery to add the datepicker to our new Textbox with the added class attribute.

1. Add the following to View @section scripts

{

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

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

<script type='text/javascript'>

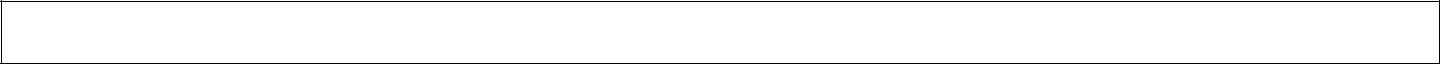
$(document).ready(function () {

$(".datePicker").datepicker();

});

</script>

}



**Reusing using custom @helper Methods**

The @helper syntax within Razor enables you to easily create re-usable helper methods that can encapsulate

output functionality within your view templates. They enable better code reuse, and can also facilitate more

readable code.

**Add following code to View (Specific to the view)**

@helper RenderTable(int tableOf)

{

<table>

@for (int i = 1; i < 10; i++)

{

<tr>

<td>@tableOf</td>

<td>\*</td>

<td>@i</td>

<td>=</td>

<td>@(tableOf\*i)</td>

</tr>

}

</table>

}

Now when ever required we can use this method as shown below

@RenderTable(15)

@RenderTable(25)

@RenderTable(35)

**To reuse Razor Helper in multiple views:**

1. Add **app\_code** folder to the project
2. Add new razor view called MyHtmlHelper.cshtml
3. Add following code to MyHtmlHelper.cshtml @using System.Web.Mvc;

@helper **Script**(string scriptName, UrlHelper url)

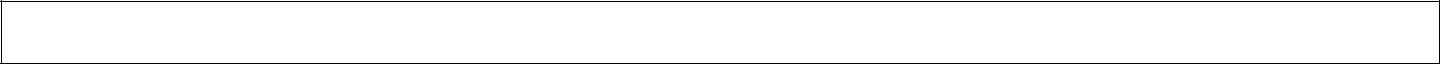
{

<script src="@url.Content("~/Scripts/"+scriptName)" type="text/javascript" />

}

**Note:** Runtime will compile any razor views which it finds inside this folder and make their helpers available asstatic methods in a class whose name is same as .cshtml (MyHtmlHelper.cshtml)

1. Now whenever required can use this method as shown below @Content.MyHtmlHelper ("jquery-1.4.1.min.js", Url) @Content.MyHtmlHelper ("jquery.unobtrusive-ajax.min.js", Url) @Content.MyHtmlHelper ("MicrosoftMvcAjax.js", Url)



**Building Custom Helpers Extension Methods**

HTML Helpers provide a clean way to encapsulate view code so you can keep your views simple and markup focused. There are lots of built in HTML Helpers in the System.Web.Mvc.HtmlHelper class, but one of the best features is that you can easily create your own helpers

using System.Web.Mvc;

namespace MvcApplication1

{

public static class HtmlHelperExtention

{

public static string FormatToCurrency(this HtmlHelper helper, decimal amount)

{

return string.Format("{0:c}", amount);

}

public static MvcHtmlString Image(this HtmlHelper helper, string src, string alt)

{

TagBuilder bulder = new TagBuilder("img");

bulder.MergeAttribute("src", src);

bulder.MergeAttribute("alt", alt);

return MvcHtmlString.Create(bulder.ToString(TagRenderMode.SelfClosing));

}

}

}

**Note: extending the HtmlHelper defined in System.Web.Mvc and not System.Web.WebPages. The using**

**statement is important.**

Our views can make use of this by either importing the namespace with the @using keyword to the views.

@using MvcApplication1

@Html.FormatToCurrency(1020003.00M);

Alternatively, instead of placing **@using** in all views we can place it in web.config

**Adding ref to web.config:**

<system.web.webPages.razor>

<host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />

<pages pageBaseType="System.Web.Mvc.WebViewPage"> <namespaces>

. . .

**<add namespace="MvcApplication1 "/>**

</namespaces>

</pages>

</system.web.webPages.razor>

**Agenda:** Understanding Model Binders

* Html Form behavior
* Model Binder Overview
* DefaultModelBinder
* Binding to Complex Classes
* FormCollection Model Binding
* HttpPostedFileBase Model Binder
* Bind Attribute
* UpdateModel and TryUpdateModel
* Writing Custom Model Binders

**Html Form Behavior**

1. Form tag cannot be nested inside another form tag.
2. Input type’s **button** and **reset** and any **disabled** element name and value pair is not submitted to the server.
3. Input type="button" is used only for writing client side JavaScript code.
4. Input type="checkbox" submits name and value pair only if the checkbox is checked otherwise nothing is submitted. If value is not provided the default value posted is “on”
5. To group radio buttons, **same name** must be given to them. And only the name-value pair of the radio button selected will be submitted to server.
6. For select, name of select and value of the option selected is submitted to server.
7. If multiple options are selected in Listbox (Select tag), then with same name different values (multiple pairs) are submitted to server
8. Input type="image" behaves as submit button whereas <img> tag renders static image.
9. A form can have more than one submit button, but only the name-value pair of the submit button used to post the form is included with the request.
10. If the form has input type="file" to upload a file along with request then its enctype="multipart/form-data" and method="post".

**Model Binders Overview**

Model Binders is responsible for mapping a browser request into an object. Your action methods need data, and the incoming HTTP request carries the data you need and this is embedded into POST-ed form values, and possibly the URL itself.

Model binders allow your controller code to remain cleanly separated from the dirtiness of interrogating the request and its associated environment.

**DefaultModelBinder** magically converts form values and route data into objects.

**Default Model Binder:** It creates following types of objects from the browser request

1. Primitive types such as string, int, decimal, DateTime etc…
2. A class such as Employee or Department…
3. An array such as string or Employee
4. A collection such as IEnumerable<T>, ICollection<T> (When elements like checkbox have same name)

**View :**

@using (Html.BeginForm())

{

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

<input type="checkbox" name="hobbies" value="Football" /><span>Football</span> <input type="checkbox" name="hobbies" value="Cricket" /><span>Cricket</span>

<input type="checkbox" name="hobbies" value="Table Tennis" /><span>Table Tennis</span> <input type="checkbox" name="hobbies" value="Badminton" /><span>Badminton</span> <br />

<input type="submit" name="btnSubmit" value="Submit" />

}

**Controller**:

public class HobbiesController : Controller

{

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(string[] hobbies, string name)

{

string str = "Hobbies of " + name + " are<br>";

foreach (string s in hobbies)

str += s + "<br>";

return Content(str);

}

}

Note: In the above action method parameter can be string[], IEnumerable<string> or List<string>

**Binding to Complex Classes**

This is situation where Model is a class has another class as one of its members. For example: Employee has

Address as its member.

Model Classes

public class Employee

{

public string Name

{ get; set; }

public Address Address

{ get; set; }

}

public class Address

{

public string Street

{ get; set; } public string City { get; set; }

}

**Controller:**

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(Employee em)

{

string str = "Name: " + emp.Name + "<br>";

str += "Street: " + emp.Address.Street + "<br>";

str += "City: " + emp.Address.City + "<br>";

return Content(str);

}

}

**View:**

@using (Html.BeginForm())

{

<label>Name</label><input type="text" name="name" value=" " /><br />

<label>Street</label><input type="text" name="Address.Street" value=" " /><br /> <label>City</label><input type="text" name="Address.City" value=" " /><br /> <input type="submit" name="submit" value="Submit" />

}

**Using FormCollection Model Binding**

It’s an **un-typed** collection of form fields.

[HttpPost]

public ActionResult Index(**FormCollection** fc)

{

string str = "Id=" + fc["Id"] + "<br>";

str += "Name: " + fc["Name"] + "<br>";

str += "Street: " + fc["Address.Street"] + "<br>";

str += "City: " + fc["Address.City"] + "<br>";

return Content(str);

}

**Using the HTTP Posted File Base Model Binder**

This is used for uploading file(s)

**Model:**

public class Employee

{

public int Id { get; set; }

public string Name { get; set; }

public string Photo { get; set; }

}

**View:**

@using (Html.BeginForm("Index", " Home", FormMethod.Post, new { enctype = "multipart/form-data" })) {

<label>Id</label><input type="text" name="Id" value="1" /><br />

<label>Name</label><input type="text" name="Name" value="a1" /><br /> <label>Photo</label><input type="file" name="Photo" value="" /><br /> <input type="submit" name="submit" value="Submit" />

}

**Controller:**

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(HttpPostedFileBase photo, Employee emp)

{

string str = "Id=" + emp.Id + "<br>";

str += "Name: " + emp.Name + "<br>";

string fn = photo.FileName;

photo.**SaveAs**(Server.MapPath("~/uploads/" + fn));

str += fn + " is saved";

return Content(str);

}

}

**Note:**

1. If more than one file upload is used then parameter must be: **IEnumerable<HttpPostedFileBase>**
2. For large files in **web.config**: <httpRuntime maxRequestLength="65536" /> <!--64MB-->

**Using the Bind Attribute**

The form post is matched to a strongly typed object. This can save you time initially but it can also provide a way for a malicious user to exploit vulnerabilities in your code if you're not careful. You can optionally override what values to bind and what not to bind automatically.

Bind attribute is used to control how a model converts a request into an object.

BindAttribute can be used either on model class or parameters of action method only.

Following are its properties

1. **Exclude**: To exclude a comma separated list of properties from binding
2. **Include**: To include a comma separated list of properties from binding
3. **Prefix**: To associate a parameter with a particular form field prefix.

**Example of Exclude Property:** We can provide either to Method Parameter or to Model Class.This is useful when validations are performed on Model using ModalState.IsValid

[HttpPost]

public ActionResult Index([Bind(Exclude = "Id")]Employee emp)

{. . .}

**or**

[Bind(Exclude="Id")]

public class Employee

{. . .}

**Example of Prefix Property:** We won’t need to use the Bind attribute to map prefixes to parameters becausedefault model binder is smart enough to do the correct mapping automatically.

**Model:**

public class Address

{

public string Street

{ get; set; } public string City { get; set; }

}

**View:**

@using (Html.BeginForm())

{

<div>

Billing Address<br />

Street<input type="text" name="Billing.Street" /><br />

City<input type="text" name="Billing.City" /><br />

</div>

<div>

Shipping Address<br />

Street<input type="text" name="Shipping.Street" /><br />

City <input type="text" name="Shipping.City"/><br />

</div>

<input type="submit" name="submit" value="Submit" />

}

**Controller:**

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index([Bind(Prefix="billing")]Address b, Address shipping)

{

string str = "";

str += "Billing: " + b.Street + ", " + b.City + "<br>";

str += "Shipping: " + shipping.Street + ", " + shipping.City + "<br>"; return Content(str);

}

}



**UpdateModel and TryUpdateModel**

**UpdateModel Method** (9 overloaded forms): Updates the specified model instance using values from thevalue provider, a prefix, a list of properties to exclude, and a list of properties to include.

protected internal void UpdateModel<TModel>(

TModel model,

string prefix,

string[] includeProperties,

string[] excludeProperties,

IValueProvider valueProvider

)

**Example of UpdateModel and TryUpdateModel:**

[HttpPost]

public ActionResult Index(FormCollection fc)

{

Address shipping = new Address();

Address billing = new Address();

string str = "";

**UpdateModel**(billing,"billing");//Will throw exception if validation fails

bool b2 = **TryUpdateModel**(shipping, "shipping"); //Return false when validation fails.

str += "Billing: " + billing.Street + ", " + billing.City + "<br>";

str += "Shipping: " + shipping.Street + ", " + shipping.City + "<br>"; return Content(str + " " + " " + b2);

}

* UpdateModel() throws an exception if validation fails.
* **TryUpdateModel** will try to update the model with the given value for Address. If the update failsvalidation then TryUpdateModel will pass update the **ViewData.ModelState** with validation errors and your view will display the validation errors.

**Custom Model Binders**

In situations DefaultModelBinder is not suitable we will have to write Custom Binder.

1. Add the following class to the project: public class Address

{

public string Street

{ get; set; } public string City { get; set; }

}

public class Order

{

public int OrderId { get; set; }

public Address BillingAddress { get; set; }

public Address ShippingAddress { get; set; }

}

1. Add the following to the controller public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(Order order)

{

string str = "";

str += "Billing: " + order.BillingAddress.Street + ", " + order.BillingAddress.City + "<br>";

str += "Shipping: " + order.ShippingAddress.Street + ", " + order.ShippingAddress.City + "<br>"; return Content(Request.Form + " " + str);

}

1. Add the following to the View @using (Html.BeginForm())

{

<div>

Order Id: <input type="text" name="orderId" value="1" /> </div>

<div>

Billing Address<br />

Street<input type="text" name="billingStreet" /><br /> City<input type="text" name="billingCity" /><br />

</div> <div>

Shipping Address<br />

Street<input type="text" name="shippingStreet" /><br /> City <input type="text" name="shippingCity"/><br />

</div>

<input type="submit" name="submit" value="Submit" />

}

1. Add the Custom Model Provider class to Project:

public class CustomOrderBinder : IModelBinder

{

public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)

{

Order order;

if (bindingContext.Model == null)

{

order = new Order();

order.BillingAddress = new Address();

order.ShippingAddress = new Address();

}

else

{

**//Usefull if UpdateModel(order) or TryUpdateModel(order) is used.**

order = (Order)bindingContext.Model;

}

order.OrderId = GetValue<int>(bindingContext, "orderid"); order.BillingAddress.City = GetValue<string>(bindingContext, "billingCity"); order.BillingAddress.Street = GetValue<string>(bindingContext, "billingStreet"); order.ShippingAddress.City = GetValue<string>(bindingContext, "shippingCity"); order.ShippingAddress.Street = GetValue<string>(bindingContext, "shippingStreet"); return order;

}

private T GetValue<T>(ModelBindingContext bindingContext, string key)

{

//At runtime the MVC framework populates the ValueProvider with values it finds in the request’s form, route, and query string collections.

ValueProviderResult valueResult = bindingContext.ValueProvider.GetValue(key); bindingContext.ModelState.SetModelValue(key, valueResult); //\*\*

return (T)valueResult.ConvertTo(typeof(T));

//\*\*

//One of the side-effects of model binding is that binding the model should put model values into ModelState. When an HTML helper sees there is a ModelState error for “Name”, it assumes it will also find the “attempted value” that the user entered. The helper uses attempted values to repopulate inputs and allow the user to fix any errors.

}

}

1. Add the following to Application\_Start in **global.asax**

ModelBinders.Binders.Add(typeof(Order), new CustomOrderBinder());

**OR**

Add the attribute to Order parameter of Action Method. [ModelBinder(typeof(CustomOrderBinder))]

**Agenda: Annotations and Validations**

* Overview of Data Annotations
* Validation Attributes
* How Validation Works
* Using CustomValidationAttribute
* Model Level Validation using IValidatatableObject
* Developing Custom Unobtrusive Client Side Validation
* Applying Annotations to EF Model classes

**Data Annotations and Validation Overview**

* Data annotations are attribute classes in the namespace **System.ComponentModel.DataAnnotations**.
* DataAnnotations provide a really easy way to declaratively add validation rules to objects and properties with minimal code.
* These annotations are available across various Visual Studio projects including

1. ASP.NET MVC,
2. Web Forms,
3. ASP.NET Apps & Web Sites,
4. Dynamic Data
5. Non ASP.NET projects like Silverlight & WPF.
6. For data models like
   * POCOs (plain old CLR objects),
   * EF models,
   * Linq2SQL models, etc...

Basically we can use them in any type of project.

Validating user-input and enforcing business rules/logic are core requirements of most the web applications.

We want to enable this validation to occur on both the server and on the client (via JavaScript).

We’ll implement these rules on our model object – and *not* within our Controller or our View. The benefit of implementing the rules within model object is that this will ensure that the validation will be enforced via any scenario within our application that uses the model object. This will help ensure that we keep our code DRY and avoid repeating rules in multiple places.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **Validation Attributes** |  |  |
|  |  |  |  |  |  |  |
|  | Required |  |  | DataType |  | CreditCard |
|  | Compare |  |  | StringLength |  | FileExtensions |
|  | Range |  |  | MinLength |  | Url |
|  | ReqularExpression | |  | MaxLength |  | EnumDataType |
|  | **CustomValidation** | |  | Email |  | Phone |
| **More Annotation Attributes** | | |  |  |  |  |
|  | |  |  |  |  |  |
| **Attribute Name** | | **Description** |  |  |  |  |
|  | |  | | |  |  |
| DisplayName | | Returns a human-readable name for the data item. | | |  |  |
|  | |  | | | | |
| DisplayFormat | | Returns a composite formatting string, such as {0:2}. This value can be set directly using the | | | | |
|  |  | DisplayFormat attribute or indirectly by other attributes – for example, the DataType.Currency | | | | |
|  |  | value passed to the DataType attribute results in a formatting string that produces two | | | | |
|  |  | decimal places and a currency symbol. | | |  |  |
|  | |  | | | | |
| HiddenInput | | If the HTML element should be hidden – this is set to be true by using the HiddenInput | | | | |
|  |  | attribute with a DisplayValue value of false. | | |  |  |
|  | |  | | | | |
| UIHint | | The name of the template that should be used to render the item. | | | | |
|  |  |  |  |  |  |  |

**Example:** Create an Employee model with Annotations as mentioned below

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

**using System.ComponentModel.DataAnnotations;**

namespace MvcApplication1.Models

{

public class Employee

{

[Required(ErrorMessage = "User Name Required")]

public string UserName { get; set; }

[Required(ErrorMessage = "Password Required")]

[DataType("password")]

public string Password { get; set; }

Annotations and Validation

[Required(ErrorMessage = "DateOfBirth Required")]

[DisplayName("Date of Birth")]

[DataType(DataType.Date)]

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

[Required(ErrorMessage = "Email isRequired")]

[EmailAddress(ErrorMessage = "Please enter valid Email Id")] public string Email { get; set; }

[Required(ErrorMessage = "Rating is Required")]

[Range(1, 10)]

public int Rating { get; set; }

[Required(ErrorMessage = "Phone Number is Required")]

[DisplayName("Phone Number")]

public int PhoneNo { get; set; }

[Required(ErrorMessage = "Comments is Required")]

[DataType(DataType.MultilineText)]

public string Comments { get; set; }

[FileExtensions(Extensions = "png|jpg|jpeg|gif")]

public string Photo { get; set; }

}

}

Once you've setup validation on the model using data annotations, they're automatically consumed by **Html** **Helpers** in views so the helpers can render the proper HTML output.

For client side validation to work, you will need to ensure that these two <SCRIPT> tag references are in your view:

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>

<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

These can be added by writing the following in .cshtml file @Scripts.Render("~/bundles/jqueryval") Example:

3

Annotations and Validations



Create Employee view

@using (Html.BeginForm())

{

@Html.ValidationSummary(true)

<fieldset>

<legend>Employee</legend>

@Html.EditorForModel()

<input type="submit" value="Create" />

</fieldset>

}

OR

@using (Html.BeginForm()) {

@Html.ValidationSummary(true)

<fieldset>

<legend>Employee</legend>

<div class="editor-label">

@Html.LabelFor(model => model.UserName)

</div>

<div class="editor-field">

@Html.EditorFor(model => model.UserName)

**@Html.ValidationMessageFor(model => model.UserName) //Shows ErrorMessage**

**OR**

**Html.ValidateFor(model => model.UserName) //Doesn’t Show ErrorMessage**

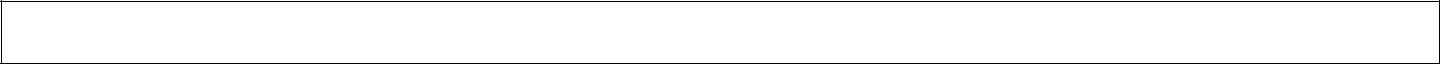
</div>

. . .

</p>

</fieldset>

}



**How Validation Works**

Both client and server side validation work because of a few conventions in your project that match up data annotations and Html Helpers rendered output. Html Helpers in views render HTML elements containing attributes that start with the pattern **data-val-\*.** The data-val-\* attributes contain error messages, regular expressions, ranges, and other validation information that originates in data annotations. That means that this decorated code in the model...

DisplayName("Qty Available")]

[Required(ErrorMessage = "The Qty Available field is required.")] [Range(0,120)]

public int QtyOnHand { get; set; }

...combined with this code in the view...

<div class="editor-field">

@Html.EditorFor(model => model.QtyOnHand)

@Html.ValidationMessageFor(model => model.QtyOnHand)

</div>

...turns into this HTML at runtime...

<div class="editor-field">

<input class="text-box single-line"

data-val="true" data-val-number="The field Qty Available must be a number."

data-val-range="The field Qty Available must be between 0 and 120."

data-val-range-max="120" data-val-range-min="0"

data-val-required="The Qty Available field is required."

id="QtyOnHand" name="QtyOnHand" type="text" value="12" />

<span class="field-validation-valid" data-valmsg-for="QtyOnHand" data-valmsg-replace="true"></span> </div>

The tie-in between the data model annotations and the data-val-\* attributes should be clear after reading the above code, but it's where the client side validation ties in, might not be so obvious. Open the **\Scripts\jquery.validate.unobtrusive.js** file and search for "data-val". Right away you'll see that the JavaScript usesthe data-val-\*, input-\* and field-\* CSS classes to display/hide validation messages on the client. Although you shouldn't modify or need to maintain the built-in .js files; it's worth investigating them to see how things work together in ASP.NET MVC.

function onError(error, inputElement) { // 'this' is the form element

var container = $(this).find("[data-valmsg-for='" + inputElement[0].name + "']"), replace = $.parseJSON(container.attr("data-valmsg-replace")) !== false;

container.removeClass("field-validation-valid").addClass("field-validation-error"); error.data("unobtrusiveContainer", container);

if (replace) {

container.empty();

error.removeClass("input-validation-error").appendTo(container);

}

else {

error.hide();

}

}

Having the onError function tied in by only HTML attributes keeps client side validation unobtrusive, or in other words, out of the way of your view code. Annotations combined with unobtrusive validation make both the view and the output very clean and maintainable. The final result in the browser is fully capable client side validation that falls back to server side validation for browsers without JavaScript enabled. Both way, the same validation happens and the user sees the same error.

**Using CustomValidation Attribute**

1. Add the following class to project public class DesignationValidator

{

public static ValidationResult IsDesignationValid(string designation, ValidationContext context)

{

if (string.IsNullOrWhiteSpace(designation))

return new ValidationResult("Designation cannot be null or white space");

if (designation.ToLower().Equals("senior") || designation.ToLower().Equals("junior")) return **ValidationResult**.**Success**;

return new ValidationResult("Designation can be either senior or junior only");

}

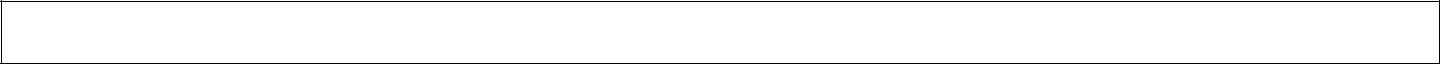
}

Note: The first parameter of the method should be datatype of the property with which we want to attach CustomValidation Attribute.

1. Add the following property to Employee class

[CustomValidation(typeof(DesignationValidator), "IsDesignationValid")] public string Designation { get; set; }

**Note: CustomValidation can be provided for Employee class also but in that case the first parameter data type of the method should be “Employee”.**

**Model Level Validation using IValidatableObject**

The IValidatableObject interface enables you to perform model-level validation, and enables you to provide

validation error messages specific to **the state of the overall model, or between two properties within the**

**model**.

Model has to implement **IValidatableObject** interface and add following method to model class

public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)

{

ValidationResult vr = null;

if (Designation.ToLower() == "senior" && Rating < 5)

vr = new ValidationResult("Invalid designation based on rating"); return new List<ValidationResult>() { vr };

}

Note: This particular method is only called after all other validations have passed successfully.

Add a post action method in the controller

[HttpPost]

public ActionResult Employee(Employee emp)

{

If (**ModelState**.**IsValid**)

return View();

return View(emp);

}

To turn off client side validation in view

@{ Html.**EnableClientValidation**(false); }

**Developing Custom Unobtrusive Client Side Validation**

**Step1: Add a class to project:**

public class DateRangeAttribute : ValidationAttribute, IClientValidatable

{

private const string DefaultErrorMessage = "'{0}' must be a date between {1:d} and {2:d}";

public DateTime Min { get; set; }

public DateTime Max { get; set; }

private static DateTime ParseDate(string dateValue)

{

return DateTime.ParseExact(dateValue, "MM/dd/yyyy", System.Globalization.CultureInfo.InvariantCulture);

}

//Constructor with two parameters

public DateRangeAttribute(string min, string max)

* base(DefaultErrorMessage)

{

Min = ParseDate(min);

Max = ParseDate(max);

}

public override bool IsValid(object value)

{

if (value == null || !(value is DateTime))

{ return true; }

DateTime dateValue = (DateTime)value;

return Min <= dateValue && dateValue <= Max;

}

public override string FormatErrorMessage(string name)

{

return String.Format(ErrorMessageString, name, Min, Max);

}

//Member of IClientValidatable for Server Side Validation.

public IEnumerable<ModelClientValidationRule> GetClientValidationRules(ModelMetadata metadata, ControllerContext context)

{

return new[] { new ModelClientValidationDateRangeRule(FormatErrorMessage(metadata.GetDisplayName()), Min, Max) };

}

}

public class ModelClientValidationDateRangeRule : ModelClientValidationRule

{

public ModelClientValidationDateRangeRule(string errorMessage, DateTime min, DateTime max)

{

ErrorMessage = errorMessage;

ValidationType = "daterange";

ValidationParameters["min"] = min.ToString("MM/dd/yyyy");

ValidationParameters["max"] = max.ToString("MM/dd/yyyy");

}

}

Note: The method GetValidationRules returns an array of ModelClientValidationRule instances. Each of these instances represents metadata for a validation rule that is written in JavaScript and will be run in the client. This is purely metadata at this point and the array will get converted into JSON and emitted in the client so that client validation can hook up all the correct rules.

**Step2:** Facilitate Client Side Validation

Server-side, the validation parameters are written to the rendered HTML as attributes on the form inputs. These

attributes are then picked up by some client-side helpers that add the appropriate client-side validation. The

RangeDateValidator.js script below contains both the custom validation plugin for jQuery and the plugin for the

unobtrusive validation adapters (each section is called out by comments in the script.

Add the following script to View

<script type="text/javascript">

(function ($) {

var parseDate = function (str) {

var m = str.match(/^(\d{1,2})\/(\d{1,2})\/(\d{4})$/);

return (m) ? new Date(m[3], m[1] - 1, m[2]) : null;

};

//Validator function

$.validator.addMethod('daterange',

function (value, element, param) {

if (!value)

return false;

//To Convert string to date for comparison in date format.

var min = parseDate(param.min);

var max = parseDate(param.max);

var current = parseDate(value);

if (min == null || max == null || current == null) {

return false;

}

return (current >= min && current <= max);

});

//The adapter to support ASP.NET MVC unobtrusive validation

$.validator.unobtrusive.adapters.add('daterange', ['min', 'max'], function (options) {

var params = {

min: options.params.min,

max: options.params.max

};

options.rules['daterange'] = params;

if (options.message) {

options.messages['daterange'] = options.message;

}

});

} (jQuery)); </script>

Note: You can see that the validator name (“daterange”) matches on both client and server, as do the parameters (“min” and “max”).

Step 3: Use the attribute in Model class:

[DateRange("12/31/2011", "12/31/2013", ErrorMessage = "'{0}' must be a date between {1:d} and {2:d}.")] [DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode=true)]

public DateTime RetirementDate { get; set; }

Ste 4: Include in View:

<div class="editor-label">

@Html.LabelFor(model => model.RetirementDate)

</div>

<div class="editor-field">

@Html.EditorFor(model => model.RetirementDate)

@Html.ValidationMessageFor(model => model.RetirementDate)

</div>

**Using Data Validation Annotators with MetaData class**.

public class UserProfileMetadata

{

[Required(ErrorMessage = "FirstName is required")]

public string FirstName { get; set; }

[Required(ErrorMessage = "LastName is required")]

public string LastName { get; set; }

[Required(ErrorMessage = "UserName is required")]

public string UserName { get; set; }

[Required(ErrorMessage = "Password is required")]

[DataType(DataType.Password)]

public string Password { get; set; }

[Email(ErrorMessage = "Please enter a valid Email Address.")] public string EmailId { get; set; }

}

**[MetadataType(typeof(UserProfileMetadata))]**

public partial class UserProfile

{ . . . }

|  |  |
| --- | --- |
| **Agenda:** Using Entity Framework |  |

* Using Entity Framework to Perform CRUD operations.
* Building and using BO class.
* Writing Generic BO class.

For Code Samples please refer to Source code of attached projects.

**Agenda: Authentication and Authorization**

* Authentication and Authorization
* Types of Authentication
* Windows Authentication Implementation
* Forms Authentication Implementation
* Roles Implementation
* AntiForgeryToken Implementation

**Authentication and Authorization**

1. Authentication is an act where one has to prove what they claim themselves to be.
2. Authorization is act of either allowing or denying authenticated users.

**Types of Authentication:**

1. Windows Authentication
2. Forms Authentication

**Windows Authentication**

* It relies on IIS to provide authenticated users.
* Called as Integrated Authentication
* Used in Intranet Applications
* Allows implementation of an impersonation scheme.

**Forms Authentication**

* Forms authentication is used in public websites and is recommended if the application needs to collect its own user credentials at logon time through HTML forms.
* All the unauthorized requests are redirected to the logon page using HTTP client-side redirection.
* Forms authentication provider may implement custom logic for validating username and password against identity store.
* If the application authenticates the request, the system issues a ticket in the form of a cookie that contains a key for reestablishing the identity for subsequent requests.
* Internet
* Would require SSL because Username and password otherwise will be submitted in plain text format.

**Implementing Windows Authentication**

1. Change authentication mode in configuration file <authentication **mode="Windows"**>

**<!--**<forms loginUrl="~/Account/Login" timeout="2880" />**-->** </authentication>

1. Create an Application in IIS using Internet Service Manager
2. Enable Basic Authentication

**Implementing Forms Authentication**

1. **In Web.Config add the following** <authentication mode="Forms">

<forms loginUrl="~/Account/Login" timeout="2880" /> </authentication>

1. **Add the LoginModel to the project**

public class LoginModel : IValidatableObject

{

[Required]

[Display(Name = "User name")] public string Username { get; set; } [Required] [DataType(DataType.Password)] public string Password { get; set; } [Display(Name = "Remember me?")] public bool RememberMe { get; set; }

public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)

{

ValidationResult vr = null; if (Username == Password)

vr = ValidationResult.Success; else

vr = new ValidationResult("Invalid username or password"); return new List<ValidationResult> { vr };

}

}

1. **Add the AccountController as below** public class AccountController : Controller

{

public ActionResult **Login**()

{

LoginModel user = new LoginModel();

return View(user);

}

[HttpPost]

public ActionResult **Login**(LoginModel user)

{

if (ModelState.IsValid)

{

FormsAuthentication.RedirectFromLoginPage(user.Username, false);

}

return View(user);

}

}

1. **Add the View (Login) as below**

@model EFCrudOperationDemoApp.Models.LoginModel @{

ViewBag.Title = "Login";

}

@using (Html.BeginForm())

{

****@Html.ValidationSummary(true) <fieldset style="width: 200px;"> <legend>Login Here</legend>

<div class="editor-label">

****@Html.LabelFor(model => model.Username) </div>

<div class="editor-field"> @Html.EditorFor(model => model.Username) @Html.ValidationMessageFor(model => model.Username)

</div>

<div class="editor-label"> @Html.LabelFor(model => model.Password)

</div>

<div class="editor-field">

@Html.EditorFor(model => model.Password)

@Html.ValidationMessageFor(model => model.Password)

</div>

<div class="editor-label">

@Html.EditorFor(model => model.RememberMe)

@Html.LabelFor(model => model.RememberMe)

</div>

<p>

<input type="submit" value="Login" />

</p>

</fieldset>

}

1. In the Controller provide [Authorize] attribute for all action methods which must be secured.

**To Implement Roles Based Security**

1. **To Global.asax add the following**

protected void Application\_AuthenticateRequest(object sender, EventArgs args)

{

if (Request.IsAuthenticated)

{

string[] roles = null ;

switch (User.Identity.Name)

{

case "u1":

roles = new string[] { "Create", "Edit", "Delete", "View" }; break;

case "u2":

roles = new string[] { "Edit", "Delete", "View" }; break;

case "u3":

roles = new string[] { "View" }; break;

}

Context.User = new System.Security.Principal.GenericPrincipal(User.Identity, roles);

}

}

1. **Modify the EmployeeController as below** public class EmployeeController : Controller

{

public ActionResult Index() { … }

**[Authorize(Roles="View")]**

public ActionResult Details(int id = 0)

{ … }

**[Authorize(Roles="Create")]**

public ActionResult Create()

{ … }

[HttpPost]

**[Authorize(Roles = "Create")]**

public ActionResult Create(Employee employee)

{ … }

**[Authorize(Roles = "Edit")]**

public ActionResult Edit(int id = 0)

{ … }

[HttpPost]

**[Authorize(Roles = "Edit")]**

public ActionResult Edit(Employee employee)

{ … }

**[Authorize(Roles = "Delete")]**

public ActionResult Delete(int id = 0)

{ … }

**[Authorize(Roles = "Delete")]**

[HttpPost, ActionName("Delete")]

public ActionResult DeleteConfirmed(int id)

{ … }

}

**Agenda: URL Routing**

* Overview
* Mapping URL to Controller Action Method
* Applying Constraints on Route parameters
* Resolving Namespace Ambiguity
* Ignoring Routes



**URL Routing Overview**

The ASP.NET MVC framework uses the ASP.NET routing engine, which provides flexibility for mapping URLs to controller classes. You can define routing rules that the ASP.NET MVC framework uses in order to evaluate incoming URLs and to select the appropriate controller. You can also have the routing engine automatically parse variables that are defined in the URL, and have the ASP.NET MVC framework pass the values to the controller action methods as parameters.

**Global URL Routing Defaults:**

Routes are initialized in the **Application\_Start** method of the Global.asax file. The following example shows a typical Global.asax file that class RourteConfig.RegisterRoutes which includes default routing logic.

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

// . . .

**RouteConfig.RegisterRoutes(RouteTable.Routes);**

// . . .

}

}

public class RouteConfig

{

public static void **RegisterRoutes**(RouteCollection routes)

{

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

routes.**MapRoute**(

name: "Default",

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

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional } );

}

}

**Implementing Routing for Employee Application:**

Add following route methods to the Global.asax file

routes.MapRoute(

"Create", // Route name

"Create", // URL with parameters

new { controller = "Employee", action = "Create" } // Parameter defaults

);

routes.MapRoute(

"Edit", // Route name

"Edit/{Id}", // URL with parameters

new { controller = "Employee", action = "Edit" } // Parameter defaults

);

routes.MapRoute(

"Delete", // Route name

"Delete/{id}", // URL with parameters

new { controller = "Employee", action = "Delete" } // Parameter defaults

);

**Note: All the entries must be above "Default" route because otherwise it will take precedence and others will not be useful.**

**Applying Constraints on Route parameters**

**routes.MapRoute(**

name: "Default",

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

defaults: new { controller = "Home", action = "Index", name="test", id = UrlParameter.Optional },

**constraints: new { name=**"[a-z]{5}", **id = @"\d{1,8}" }**

);

The above route puts a constraint on ID that it should be between 1 and 8 digits only.

**Example 2: Overloaded version of MapRoute**

routes.MapRoute("Default ",

"{controller}/{action}/{id}",

new RouteValueDictionary { //Default Values

controller = "Home",

action = "Index",

name="test",

id = UrlParameter.Optional },

new RouteValueDictionary { //Contraints

{ "name", "[a-z]{5}" }, { "id", @"\d{4}" } });

**Resolving Namespace Ambiguity**

routes.MapRoute(

name: "Default",

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

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

**namespaces: new[] { "MyDemoApp.Controllers" }**

);

//OR

**ControllerBuilder.Current.DefaultNamespaces.Add("MyDemoApp.Controllers");**

**Ignoring Route URL’s**

**If routes.IgnoreRoute("Home/Index/123"); is added to RegisterRoutes method the url:** [**http://localhost:49165/Home/Index/123**](http://localhost:49165/Home/Index/123)

**will render HTTP Error 404.0 - Not Found**

**Agenda: Working with Areas**

* Understanding Areas
* Adding Areas
* Registering Areas
* Linking between Areas

**Understanding Areas**

* The MVC pattern separates the model (data) logic of an application from its presentation logic and business logic. In ASP.NET MVC, this logical separation is also implemented physically in the project structure, where controllers and views are kept in folders that use naming conventions to define relationships. This structure supports the needs of most Web applications.
* However, some applications can have a large number of controllers, and each controller can be associated with several views. For these types of applications, the default ASP.NET MVC project structure can become complicated to maintain.
* To accommodate large projects, ASP.NET MVC lets you partition Web applications into smaller units that are referred to as areas. Areas provide a way to separate a large MVC Web application into smaller functional groupings. An area is effectively an MVC structure inside an application. An application could contain several MVC structures (areas).
* For example, a single large application might be divided into areas that represent the Blog, HelpDesk, and Shopping. Each area represents a separate function of the overall application.
* Note: With areas in place, each module replicates the MVC folder structure. For example, each module will have its own Models, Views and Controllers folder. You can then have HomeController class in Blog module as well as HelpDesk module. So, in the above example, essentially you will have four MVC structures - one for the main application and one each for the three modules (Blog, HelpDesk and Shopping).

**Adding Area**

In Solution Explorer, right-click the project name, click Add, and then click Area.

In Area Name, type Admin and then click Add.

An Areas folder is added to the project. The Areas folder contains a folder structure that allows each child area to have its own models, views, and controllers.

[**Adding Area-Specific Controllers:**](javascript:void(0))

1. In Solution Explorer, right-click the Controllers subfolder for the Admin area, click Add, and then click Controller.
2. Name the controller HomeController and then click Add.

**Registering Areas**

When you add an area to a project, a route for the area is defined in an **AreaRegistration** file. The route sends requests to the area based on the request URL. To register routes for areas, you add code to the Global.asax file that can automatically find the area routes in the AreaRegistration file. Following line will be present in Application\_Start in Global.asax

**AreaRegistration.RegisterAllAreas();**

This code calls the route registration methods for each child area.

**Linking between Areas**

ASP.NET MVC Views often need to link to some action method residing in controller classes. Unless otherwise specified, action methods and controllers are assumed to be from the current area. In case you wish to link to an action method belonging to some other area, here is how you would do just that:

**Using ActionLink:**

@Html.ActionLink("Admin Area", "Index", "Home", **new { area = "Admin" }**, null)

**Using RedirectToAction:**

RedirectToAction("Index", "Home", new { Area = "Admin" });

**Agenda: AJAX and JQuery**

* Ajax implementation using JQuery
* Using Ajax.BeginForm Helper method
* JQuery UI Controls
* JQuery Templates

**AJAX implementation using JQuery**

**Ajax:** The jQuery library has a full suite of AJAX capabilities. The functions and methods therein allow us to loaddata from the server without a browser page refresh.

**Example: Fetching Server Time.**

Add the following to HomeController

[OutputCache(Duration=0,NoStore=true)]

public string GetServerTime()

{

System.Threading.Thread.Sleep(2000);

return DateTime.Now.ToLongTimeString();

}

Add the following in head section of \_Layout.cshtml (Master Page)

<script src="/Scripts/jquery.unobtrusive-ajax.min.js" type="text/javascript"></script>

Add following code to View

@Ajax.ActionLink("Click here to get Server Time", "GetServerTime", "Home", new AjaxOptions

{

UpdateTargetId = "idServerTime",

InsertionMode = InsertionMode.Replace,

HttpMethod = "Get",

LoadingElementId = "ProgressSectionId"

})

<div id="idServerTime"></div>

<div id="ProgressSectionId" style="border:none;display: none;margin:auto;">

Please Wait...It's loading...

</div>

Note: LoadingElementId can be replaced with

OnFailure = "AjaxFailure",

OnBegin = "AjaxBegin",

OnComplete = "AjaxComplete",

And

<script type="text/javascript">

var showProgress = false;

function AjaxBegin() {

showProgress = true;

setTimeout("if (showProgress) { $('#ProgressSectionId').show(); }", 800);

}

function AjaxComplete() {

showProgress = false;

$("#ProgressSectionId").hide();

}

function AjaxFailure(ajaxContext) {

var response = ajaxContext.responseText;

alert("Error Code [" + ajaxContext.ErrorCode + "] " + response);

}

</script>

**Using Ajax.BeginForm Helper Method**

Add following code to Index.cshtml

<div>

@using (Ajax.BeginForm("SayHello", "Home", new AjaxOptions { HttpMethod = "Get", InsertionMode = InsertionMode.Replace, UpdateTargetId = "msgResult" })) {

<input type="text" name="name" />

<input type="submit" value="Submit" />

}

</div>

<div id="msgResult"></div>

Add **SayHello** action method in HomeController

[OutputCache(Duration = 0, NoStore = true)]

public ActionResult SayHello(string name)

{

return Content("Hello " + name);

}

**Using JQueryUI**

**jQuery UI** provides abstractions for low-level interaction and animation, advanced effects and high-level,themeable widgets, built on top of the jQuery JavaScript Library, that you can use to build highly interactive web applications.

The following script is required to support JQueryUI

<script src="/Scripts/jquery-ui-1.8.20.js"></script>

**Example: AutoComplete search using JQuery**

Write a “Quicksearch” action method in employee controller

[OutputCache(Duration = 0, NoStore = true)]

public ActionResult GetFilteredList(string term)

{

var result = new List<object>() { new { label = term + "1" }, new { label = term + "2" }, new { label = term + "3" }, new { label = term + "4" } };

return Json(result, JsonRequestBehavior.AllowGet);

}

**Note: The variables “term” and “label” must be named like mentioned only.**

Add following Jquery to View file:

<form >

<input type="text" name="txt1" data-autocomplete="@Url.Action("GetFilteredList", "Home")" />

</form>

@section scripts

{

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

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

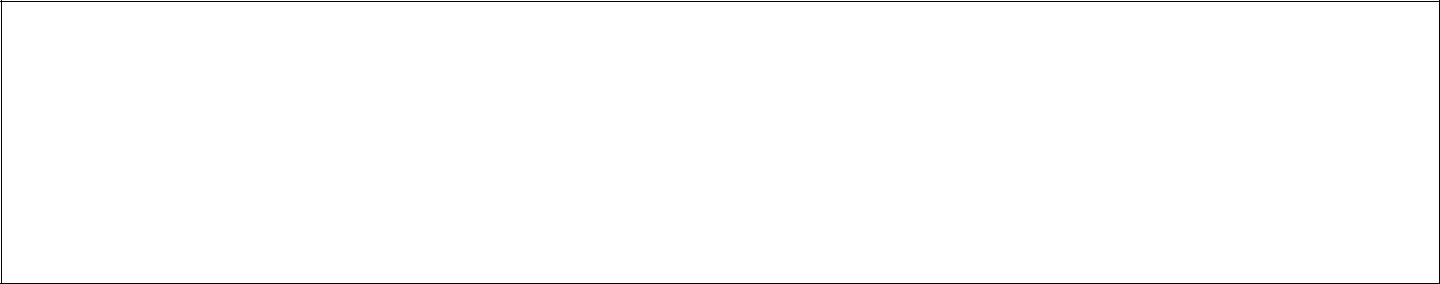
<script type="text/javascript">

$(document).ready(function () {

$(":input[data-autocomplete]").each(function () {

3

AJAX and JQuery



$(this).autocomplete({ source: $(this).attr("data-autocomplete") })

});

})

</script>

}

**jQuery Templates**

**jQuery Templates are used to use Json result in views**

Download a new jQuery template from new get(jquery-temp.js) and add it to the script folder. Like a html template, jQuery template is a mixture of data and place holders.

**Step1:** Goto ManageNuGet PackagesSearch Jquery Templates and Double Click on Jquery TemplatesThis will add jQuery-tmpl.js file to the scripts folder of the project

**Step2:** Write this action method in Home controller

[OutputCache(Duration = 0, NoStore = true)]

public ActionResult GetEmployees(string term)

{

var result = new List<object>() { new { EmpName = "E1", EmpSalary = 10000 }, new { EmpName = "E2", EmpSalary = 20000 }, new { EmpName = "E3", EmpSalary = 30000 } };

return Json(result, JsonRequestBehavior.AllowGet);

}

**Write a form which gets the Json result:**

**Step 3:** Add following code to View

<form method="get" action="@Url.Action("GetEmployees", "Home")" id="empForm"> <input type="submit" value="Get Employees" />

</form>

<table>

<thead>

<th>EmpName

</th>

<th>EmpSalary

</th>

</thead>

<tbody id="empResult">

</tbody>

</table>

<script id="empTemplate" type="text/x-jquery-tmpl">

<tr>

<td>${EmpName}

</td>

<td>${EmpSalary}

</td>

</tr>

</script>

**Step4:** Append the following Script section in View.

<script src="/Scripts/jQuery.tmpl.js"></script>

<script type="text/javascript">

$("#empForm").submit(function () {

$.getJSON($(this).attr("action"), //url to get the json from

$(this).serialize(),

function (data) {

var result = $("#empTemplate").tmpl(data);

$("#empResult").empty().append(result);

});

return false;

});

</script>

**Agenda: Bundling and Minification**

* Why Bundling and Minification
* What is Bundling
* What is Minification
* Impact of Bundling and Minification
* Implementing Bundling and Minification
* Creating and Using Bundles
* Using Wildcard to Select Files
* Using a CDN while Bundling

**Why Bundling and Minification**

**What is Bundling**

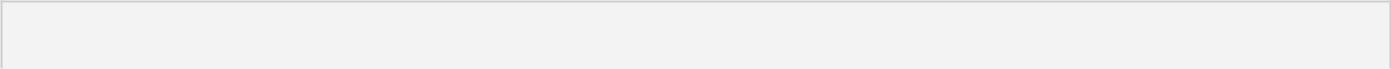
Bundling is a new feature in ASP.NET 4.5 that makes it easy to combine or bundle multiple files into a single file. You can create CSS, JavaScript and other bundles. Fewer files means fewer HTTP requests and that can improve page load performance.

**What is Minification**

Minification performs a variety of different code optimizations to scripts or css, such as removing unnecessary white space and comments and shortening variable names to one character. Consider the following JavaScript function.

Note: Unless EnableOptimizations is true or the debug attribute in the compilation Element in the Web.config file is set to false, files will not be bundled or minified. Additionally, the .min version of files will not be used instead the full debug versions will be selected.

**Open the *App\_Start\BundleConfig.cs* file.**

****

public static void RegisterBundles(BundleCollection bundles)



{



bundles.Add(new ScriptBundle("~/bundles/jquery").Include( "~/Scripts/jquery-{version}.js"));

// ...

}

The preceding code creates a new JavaScript bundle named *~/bundles/jquery* that includes all the appropriate (that is debug or minified but not .*vsdoc*) files in the *Scripts* folder that match the wild card string "~/Scripts/jquery-{version}.js".

For ASP.NET MVC 4, this means with a debug configuration, the file *jquery-1.7.1.js* will be added to the bundle.

In a release configuration, *jquery-1.7.1.min.js* will be added.

The {version} wild card matching shown above is used to automatically create a jQuery bundle with the appropriate version of jQuery in your *Scripts* folder.

Using a wild card provides the following benefits:

* Allows you to use NuGet to update to a newer jQuery version without changing the preceding bundling code or jQuery references in your view pages.
* Automatically selects the full version for debug configurations and the ".min" version for release builds.

**Creating and Using a Bundles**

The [Bundle](http://msdn.microsoft.com/en-us/library/system.web.optimization.bundle(v=VS.110).aspx) class Include method takes an array of strings, where each string is a virtual path to resource. The following code from the RegisterBundles method in the *App\_Start\BundleConfig.cs* file shows how multiple files are added to a bundle:

bundles.Add(new StyleBundle("~/Content/themes/base/css").Include( "~/Content/themes/base/jquery.ui.core.css", "~/Content/themes/base/jquery.ui.resizable.css", "~/Content/themes/base/jquery.ui.selectable.css", "~/Content/themes/base/jquery.ui.accordion.css", "~/Content/themes/base/jquery.ui.autocomplete.css", "~/Content/themes/base/jquery.ui.button.css", "~/Content/themes/base/jquery.ui.dialog.css", "~/Content/themes/base/jquery.ui.slider.css", "~/Content/themes/base/jquery.ui.tabs.css",

* Introduction
* Web API Routing
* Developing Sample application
* Calling Web API with Javascript and jQuery
* Creating a Web API that Supports CRUD Operations using EF



**Introduction to Web API**

* ASP.NET Web API is a framework for building and consuming HTTP services that can reach a broad range of clients including browsers, phones and tablets.
* ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.
* We can use either XML or JSON as output. JSON is good for mobile apps with slow connections, for example. You can call an API from jQuery and better utilize the client's machine and browser.
* ASP.NET Web API takes the best features from WCF Web API and merges them with the best features from MVC. WCF needs lot of configurations, URI Templates, Contracts and endpoints. Service implementation and consumption is not simple.

**Example:**

1. Create a project using ASP.NET MVC 4 Web Application temple
2. VS2012 -> New Project -> ASP.NET MVC 4 Web Application
3. Web API is one of the built in project template. Click OK
4. ValuesController inherits **ApiController** and has some simple methods to returns some sample data.
5. Edit App\_Start/WebApiConfig.cs:

**Web API Routes:**

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

config.Routes.MapHttpRoute( name: "DefaultApi",

routeTemplate: "api/{controller}/{id}", **//Note that action method name is missing**

defaults: new { id = RouteParameter.Optional }

);

}

}

1. Let’s just modify the methods to have better test data.

public class ValuesController : ApiController

{

Web API

private List<string> list = new List<string> { "Item 1", "Item 2", "Item 3", "Item 4", "Item 5" }; // GET api/values

public IEnumerable<string> GetList()

{

return list;

}

// GET api/values/5

public string GetItem(int id)

{

return list[id];

}

// POST api/values

public List<string> Post(string value)

{

list.Add(value);

return list;

}

// PUT api/values/5

public void Put(int id, string value)

{

}

// DELETE api/values/5

public List<string> DeleteItem(int id)

{

list.Remove(list.Find((i => i.ToString().Contains(id.ToString())))); return list;

}

WEB API is REST Complaint, so it typically consists of Get(), Put(), Post(), Delete() methods.

|  |  |
| --- | --- |
| **Method** | **URL Structure** |
|  |  |
| Get() | api/Values |
|  |  |
| GetItem(int i) | api/Values/i |
|  |  |
| Post (i) | api/Values/i with Post method |
|  |  |
| Delete(i) | api/Values/i with Delete method. |
|  |  |

1. **Testing in browser:** Type the following URL in browser and view the output in both IE and FF.
   * http://localhost:2201/api/values/1
   * <http://localhost:2201/api/values>

2

Web API



**Note:** The reason for the difference is that Internet Explorer and Firefox send different Accept headers, so theweb API sends different content types in the response.



**Calling the Web API with Javascript and jQuery**

1. **Add the following to Index View of Home Controller:**

<div class="main-content">

<div>

<h1>All Items</h1>

<ul id="items" />

</div>

<div>

<label for="indItem">INDEX:</label>

<input type="text" id="indItem" size="5" />

<input type="button" value="Search" onclick="find();" /> <p id="item" />

</div>

</div>

@section scripts

{

<script type="text/javascript">

$(document).ready(function () {

* Send an AJAX request $.getJSON("/api/values/", function (data) {
  + On success, 'data' contains a list of Items. $.each(data, function (key, val) {
* Format the text to display. var str = val;
* Add a list item.

$('<li/>', { text: str })

.appendTo($('#items'));

});

});

});

function find() {

var ind = $('#indItem').val();

alert(ind)

3

Web API

$.getJSON("/api/values/" + ind,

function (data) {

var str = data;//.Name + ': $' + data.Price;

$('#item').text(str);

})

}

function deletevalue() {

var ind = $('#indItem').val();

$.ajax("/api/values/" + ind,

{

type: "DELETE",

contentType: "application/json",

dataType: "JSON",

success: function (data) {

alert('Deleted');

}

});

} </script>

}

**Creating a Web API that Supports CRUD Operations using EF**

1. Create a MVC 4 project with Web API template
2. Add Ado.NET Entity Data Model
3. Add Employee Table to DemoDb.Edmx file.
4. Add controller **EmployeeController,** Template= API Controller with read/write actions, using Entity

Framework

Following are the list of methods which are auto generated:

1. **To Get all Employees**

// GET api/Emp

**IEnumerable<Employee> GetEmployees()**

The method name starts with "Get", so by convention it maps to GET requests. Also, because the method has no parameters, it maps to a URI that does not contain an *"id"* segment in the path.

**b)** **To Get details of particular employee**

// GET api/Emp/5

**Employee GetEmployee(int id)**

This method name also starts with "Get", but the method has a parameter named *id*. This parameter is mapped to the "id" segment of the URI path. The ASP.NET Web API framework automatically converts the ID to the correct data type (**int**) for the parameter.

1. **To Add a new Employee**

4

Web API



// POST api/Emp

**HttpResponseMessage PostEmployee(Employee employee)**

To handle POST requests, we define a method whose name starts with "Post...". The method takes a parameter of type Employee. By default, parameters with complex types are deserialized from the request body. Therefore, we expect the client to send us a serialized representation of an employee object, using either XML or JSON for the serialization.

**Protocol Requirements:**

1. **Response code:** By default, the Web API framework sets the response status code to 200(OK). But according to the HTTP/1.1 protocol, when a POST request results in the creation of a resource, the server should reply with status 201 (Created).
2. **Location:** When the server creates a resource, it should include the URI of the new resource

in the Location header of the response.

**Note**: Notice that the method return type is now **HttpResponseMessage**. By returning anHttpResponseMessage instead of a Employee, we can control the details of the HTTP response message, including the status code and the Location header.

The **CreateResponse** method creates an **HttpResponseMessage** and automatically writes a serialized representation of the Employee object into the body of the response message.

**d)** **To update an Employee**

// PUT api/Emp/5

HttpResponseMessage PutEmployee(int id, Employee employee)

The method name starts with "Put...", so Web API matches it to PUT requests. The method takes two parameters, the Emp ID and the updated employee. The *id* parameter is taken from the URI path, and the *employee* parameter is deserialized from the request body. By default, the ASP.NET Web API framework takes simple parameter types from the route and complex types from the request body.

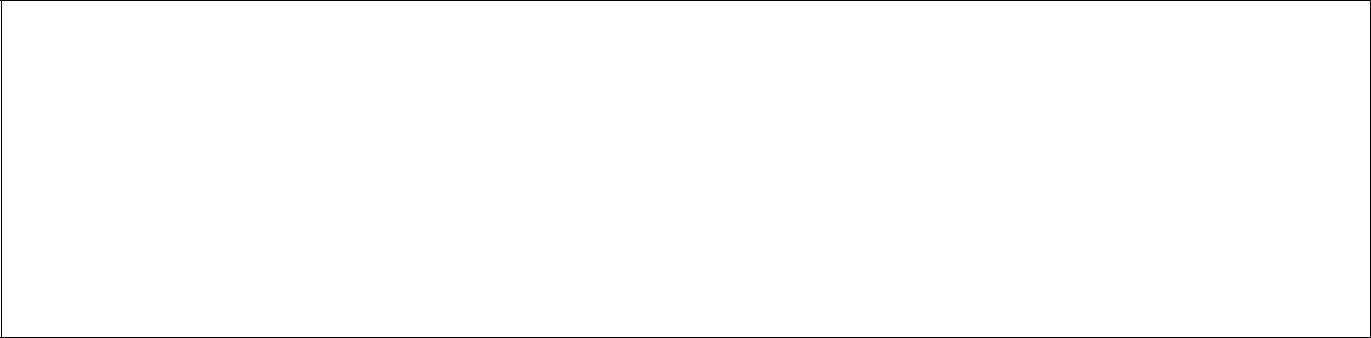
**e) To Delete an Employee**

// DELETE api/Emp/5

HttpResponseMessage DeleteEmployee(int id)

If a DELETE request succeeds, it can return status 200 (OK) with an entity-body that describes the status; status 202 (Accepted) if the deletion is still pending; or status 204 (No Content) with no entity body. In this case, the DeleteEmployee method has a void return type, so ASP.NET Web API automatically translates this into status code 204 (No Content).

1. **Add the following to Index View of Home Controller**

****

<div>

<div>

<h1>All Employees</h1>

<ul id="items" />

</div>

<div>

5

Web API

<label for="empid">Id:</label>

<input type="text" id="empid" />

<input type="button" value="Get Employee Details" onclick="**GetEmployeeDetails**();" /> <input type="button" value="Delete" onclick="**DeleteEmployee**();" /> <p id="item" />

</div>

<h1>Add Employee</h1>

<div>

<label for="name">Name:</label>

<input type="text" id="name" />

<label for="salary">Salary:</label>

<input type="text" id="salary" />

<label for="dateOfJoin">DateOfJoin:</label>

<input type="text" id="dateOfJoin" />

<input type="button" value="Add Employee" onclick="**AddEmployee**();" /> </div>

</div>

@section scripts

{

<script type="text/javascript">

$(document).**ready**(function () {

GetAllEmployees()

});

function **GetAllEmployees**() {

$.getJSON("/api/Employee/",

function (data) {

* On success, 'data' contains a list of Employees. $('#items').text("");

$.each(data, function (key, emp) {

* + Format the text to display.

var str = emp.EmpId + ", " + emp.EmpName + ", " + emp.Salary + ", " + emp.IsActive; // Add a list item.

$('<li/>', { text: str }).appendTo($('#items'));

});

});

}

6

Web API

function **GetEmployeeDetails**() {

var id = $('#empid').val();

$.getJSON("/api/Employee/" + id,

function (emp) {

var str = emp.EmpId + ", " + emp.EmpName + ", " + emp.Salary + ", " + emp.IsActive; $('#item').text(str);

})

}

function **AddEmployee**() {

$.ajax({

type: "POST",

contentType: "application/json;charset=utf-8",

url: "/api/Employees",

data: JSON.stringify

({

EmpId: 0,

EmpName: $("#name ").val(),

Salary: $("#salary").val(),

DateOfJoin: $("#dateOfJoin").val(),

IsActive: true,

DeptId: 1

}),

dataType: "json",

success: function (data) {

GetAllEmployees();

}

});

}

function **DeleteEmployee**() {

var id = $('#empid').val();

$.ajax({

type: "DELETE",

contentType: "application/json;charset=utf-8",

url: "/api/Employee/" + id,

dataType: "json",

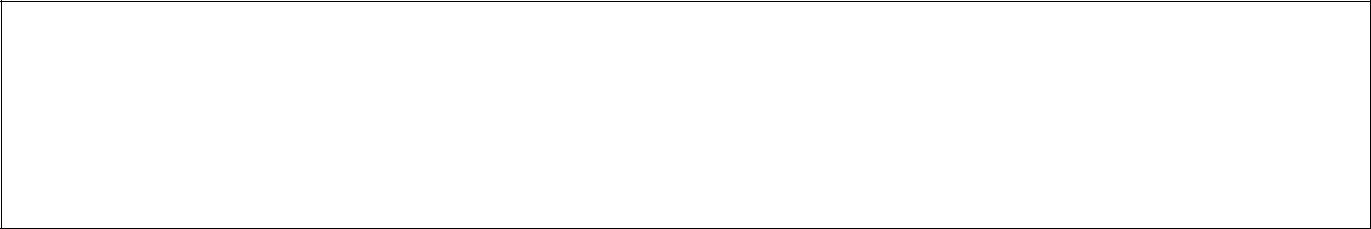
success: function (data) {

GetAllEmployees();

}

7

Web API



});

}

</script>

}

1



Understanding Model Binders



7

Understanding Model Binders



8



